# GEF（Graphical Editing Framework，图形编辑框架）指南

文档版本：1.1
2007 年 10 月 17 日：首版发布
2008 年 4 月 26 日：网络更新
2011 年 7 月 19 日：首版翻译

Epitech students

Jean-Charles MAMMANA *(jc.mammana [at] gmail.com)*

Romain MESON *(romain.meson [at] gmail.com)*

Jonathan GRAMAIN *(jonathan.gramain [at] gmail.com)*

Made during a work period at INRIA - Institut National de Recherche en Informatique et Automatique, Rocquencourt (78), in 2007

In cooperation with ENSMP (Ecole des Mines de Paris)

Translated by WBH *(wangbohan1987 [at] gmail.com)*

# 目录

# 0. 导论（Introduction）

GEF（图形编辑框架）是一个 Java 技术，是由 IBM 开发的 Eclipse 框架的一部分。它为开发者提供图形化建模的完全解决方案，并可以用于连接如 EMF（Eclipse 建模框架）或 GMF（图形建模框架）的其他技术，从而在应用开发中获得抽象层。

# 1. 创建 RCP 插件（RCP Plug-in）

开始创建一个 RCP（富客户端平台）的 Eclipse 插件来创建一个使用 GEF 的应用。运行 Eclipse，点击 File—New—Project。

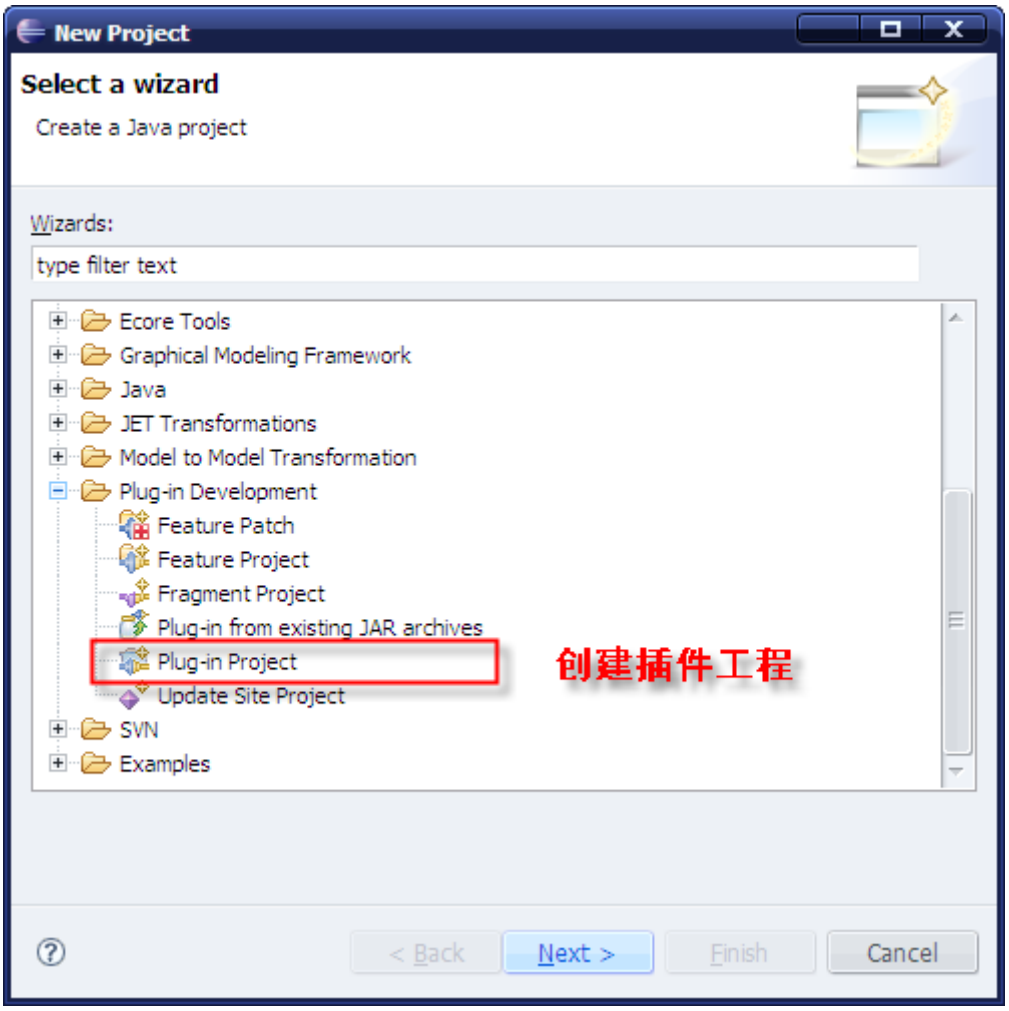

图1 创建 Plug-in 工程

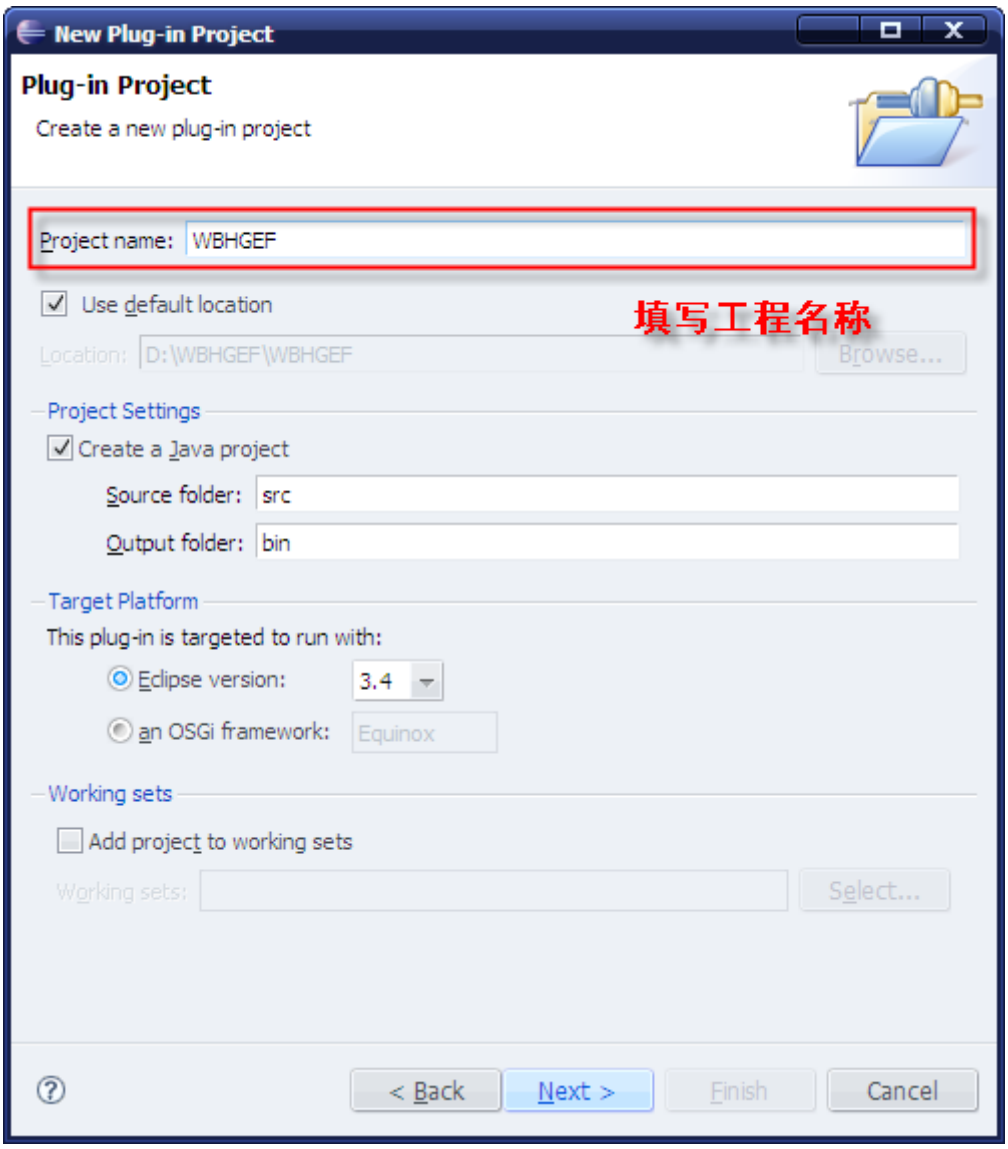


图2 填写工程名称

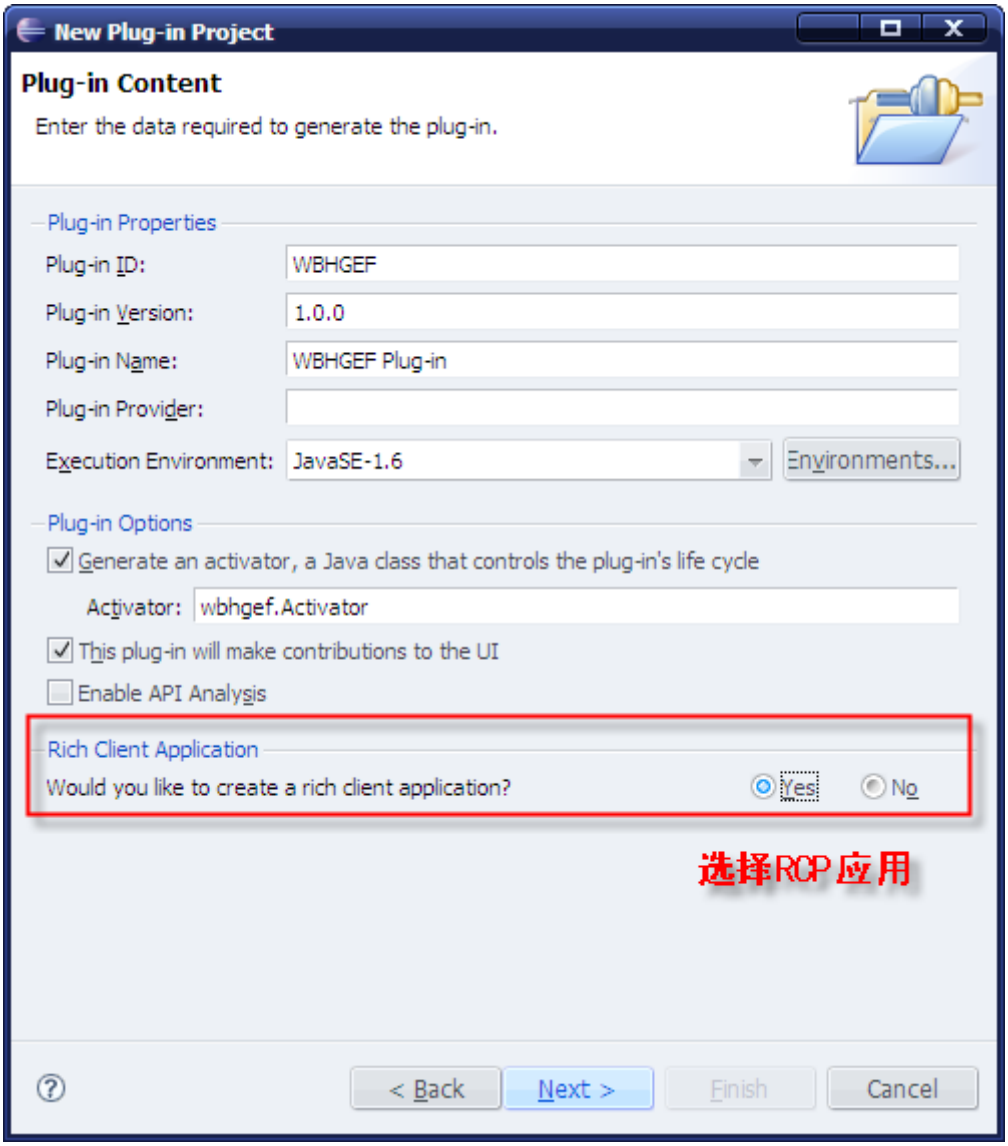


图3 选择创建 RCP 应用

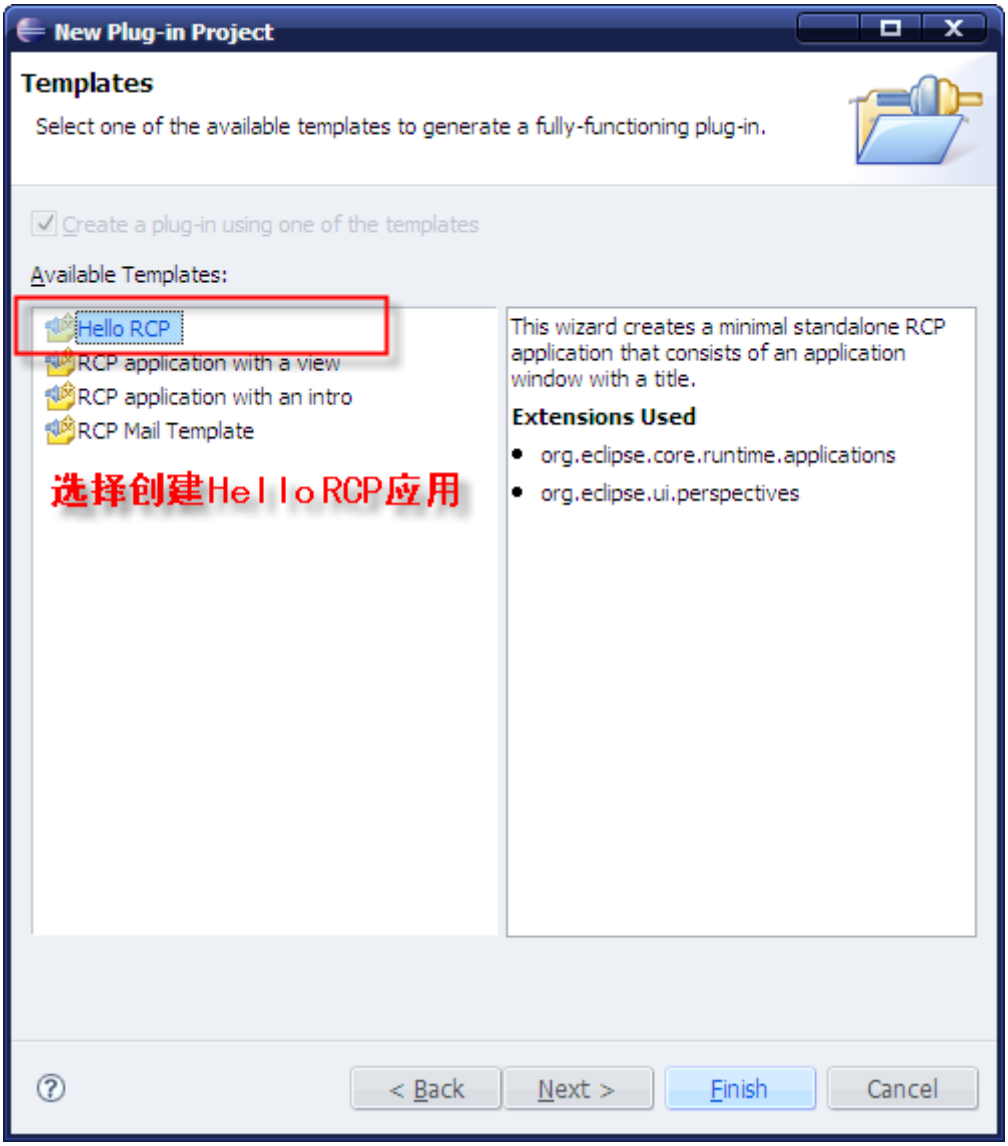


图4 选择创建 Hello RCP 应用

需要为工程的依赖（dependencies）列表添加 GEF 插件。编辑 Dependencies 表下的 plugin.xml，点击 Add，选择 org.eclipse.gef 插件。

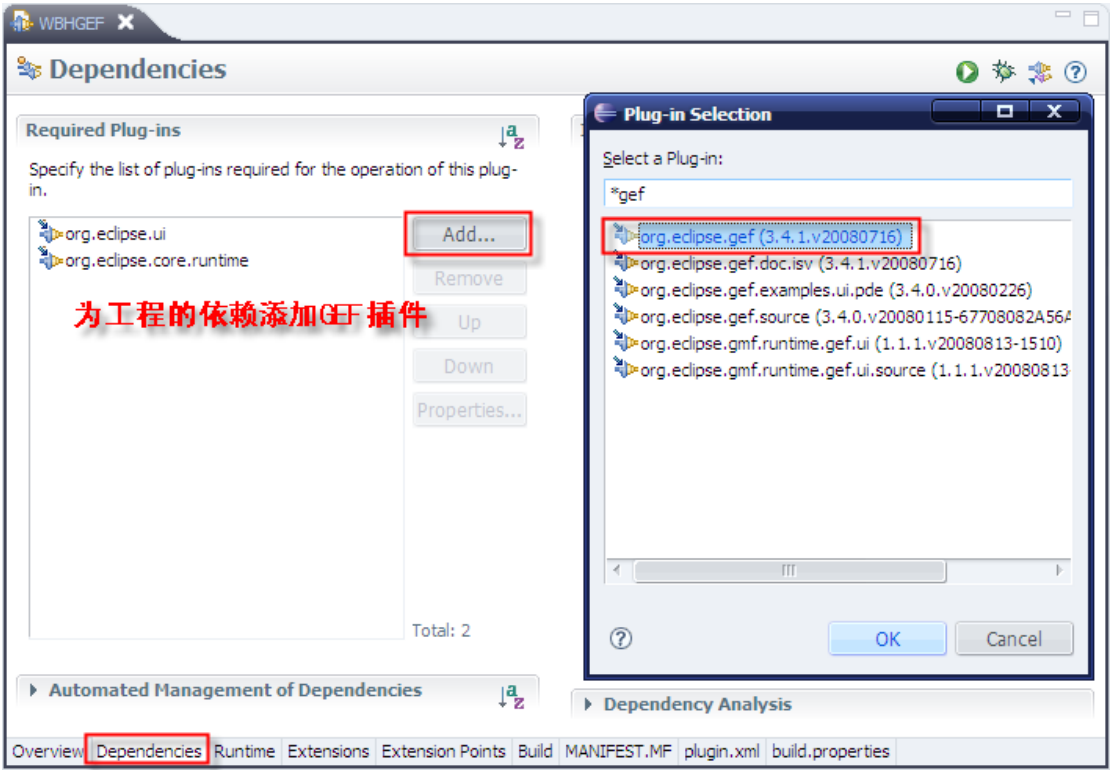


图5 为工程的依赖添加 org.eclipse.gef 的插件

现在在包中创建一个继承 org.eclipse.gef.ui.parts.GraphicalEditor 的类，这个类用于定义我们自己的图形编辑器（Graphical Editor）。这是我们工作的一类起点。搜先，EditorPart 需要整合到插件中。

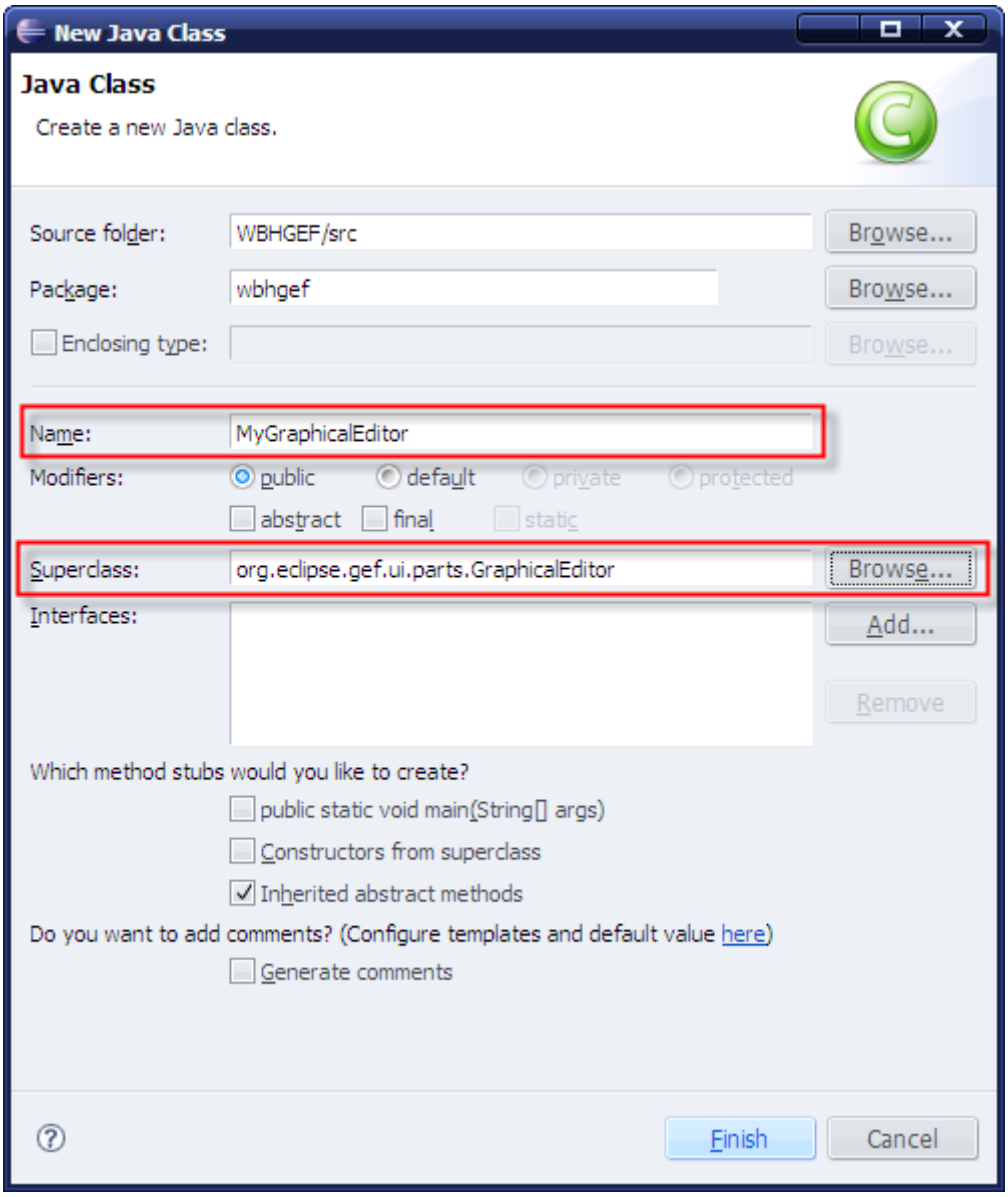


图6 添加自定义 GraphicalEditor 继承 gef 的 GraphicalEditor

添加 ID 告诉 Eclipse 我们想使用一个 extension。然后创建一个构造器去定义我们要使用的 EditDomain。为 `MyGraphicalEditor` 类添加的具体添加代码如下：

```
public static final String ID = "WBHGEF.mygraphicaleditor";

public MyGraphicalEditor() {
    setEditDomain(new DefaultEditDomain(this));
}
```

需要建立另一个类 `MyEditorInput`，实现 ***org.eclipse.ui.IEditorInput*** 接口，类的描述如下：

```
package wbhgef;
```

```
import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.ui.IEditorInput;
import org.eclipse.ui.IPersistableElement;

public class MyEditorInput implements IEditorInput {

    public String name = null;

    public MyEditorInput(String name) {
        this.name = name;
    }

    @Override
    public boolean exists() {
        // TODO Auto-generated method stub
        return (this.name != null);
    }

    public boolean equals(Object o) {
        if(!(o instanceof MyEditorInput))
            return false;
        return ((MyEditorInput)o).getName().equals(getName());
    }

    @Override
    public ImageDescriptor getImageDescriptor() {
        // TODO Auto-generated method stub
        return ImageDescriptor.getMissingImageDescriptor();
    }

    @Override
    public String getName() {
        // TODO Auto-generated method stub
        return this.name;
    }

    @Override
    public IPersistableElement getPersistable() {
        // TODO Auto-generated method stub
        return null;
    }
```

```
    @Override
    public String getToolTipText() {
        // TODO Auto-generated method stub
        return this.name;
    }

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

编辑 Extensions 表中的 plugin.xml，并点击 Add 添加一个 extension，然后选择 *org.eclipse.ui.editor*。需要修改 id 为我们自定义 GraphicalEditor 中所设置的 ID，并修改名称，最后，选择 extensions 所扩展的类。

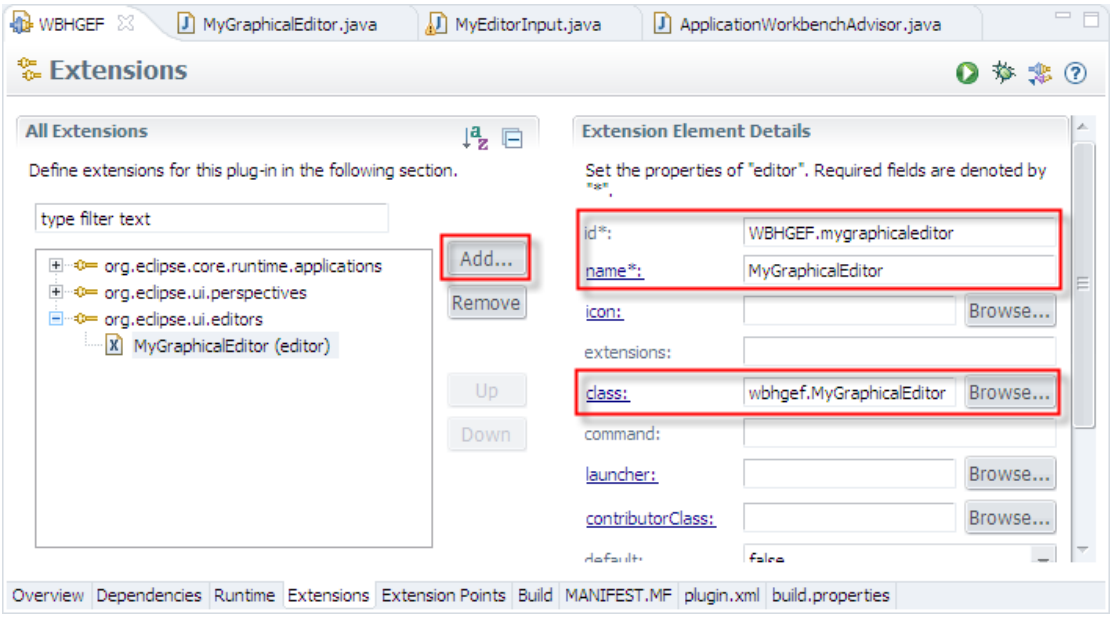


图7 添加 Extension 并修改 id 和 name 及 extensions

现在，剩余的工作就是在应用的启动中运行，为此，我们将加载 *ApplicationWorkbenchAdvisor* 中的 postStartup()方法。添加如下方法：

```
    @Override
    public void postStartup() {
        try {
            IWorkbenchPage page =

    PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePag
e();
```

```
                page.openEditor(new MyEditorInput("WBHGEF"),
MyGraphicalEditor.ID, false);
            } catch (Exception e) {
                e.printStackTrace();
            }
    }
```

现在拥有了一个可运行的 RCP 插件，整合一个准备运作 GEF 的 Editor。运行效果如下图。


图8 扩展自定义 GraphicalEditor 的空白 RCP 运行效果

这个阶段的代码为 WBHGEF01。

# 2. 建立模型（Model）

现在我们有了建立 GEF 图形的基础，我们将要创建基本的对象模型，并在后面展现出来。我们建立一个新的包叫做 wbhgef.model。

下面，在这个包中创建一个类 Node，用作我们所有模型元素的父类。这个类包含所有派生类都要用到的基本属性。代码如下。

```
package wbhgef.model;

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

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

public class Node {

    private String name;
    private Rectangle layout;
    private List<Node> children;
    private Node parent;

    public Node() {
        this.name = "Unknown";
        this.layout = new Rectangle(10,10,100,100);
        this.children = new ArrayList<Node>();
        this.parent = null;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void setLayout(Rectangle layout) {
        this.layout = layout;
    }

    public Rectangle getLayout() {
        return this.layout;
    }

    public boolean addChild(Node child) {
        child.setParent(this);
        return this.children.add(child);
    }

    public boolean removeChild(Node child) {
        return this.children.remove(child);
    }

    public List<Node> getChildrenArray() {
```

```
        return this.children;
    }

    public void setParent(Node parent) {
        this.parent = parent;
    }

    public Node getParent() {
        return this.parent;
    }
}
```

下面创建一个派生自 Node 的类，并且它在分类（hierarchy）里是最顶级的类。如类 company：它包含几个部分并且每个部门里都包含几个雇员。

下面是 Enterprise 类、Service 类和 Employee 类。

Enterprise 类：

```
package wbhgef.model;

public class Enterprise extends Node {

    private String address;
    private int capital;

    public void setAddress(String address) {
        this.address = address;
    }

    public void setCapital(int capital) {
        this.capital = capital;
    }

    public String getAddress() {
        return this.address;
    }

    public int getCapital() {
        return this.capital;
    }
}
```

Service 类：

```
package wbhgef.model;
```

```
public class Service extends Node {

    private int etage;

    public void setEtage(int etage) {
        this.etage = etage;
    }

    public int getEtage() {
        return this.etage;
    }
}
```

Employee 类：

```
package wbhgef.model;

public class Employee extends Node {

    private String prenom;

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    public String getPrenom() {
        return this.prenom;
    }
}
```

每个模型中的对象需要关联一个 Figure（Draw2D）和一个 EditPart。

我们由三个模型类，所以需要三个 Figure 类和三个 EditPart（一个 EditPart 是一个对象，用于连接模型对象和它的可视化表示）。

现在创建关联于 Company（即 Enterprise）的 Figure 和 EditPart 的派生类从而显示它。（我们将图形类放置到叫做 wbhgef.figure 的包中，将 EditPart 放置于包 wbhgef.editpart）

代码如下。

EnterpriseFigure 类：

```
package wbhgef.figure;

import org.eclipse.draw2d.ColorConstants;
```

```
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.Label;
import org.eclipse.draw2d.LineBorder;
import org.eclipse.draw2d.XYLayout;
import org.eclipse.draw2d.geometry.Rectangle;

public class EnterpriseFigure extends Figure {

    private Label labelName = new Label();
    private Label labelAddress = new Label();
    private Label labelCapital = new Label();
    private XYLayout layout;

    public EnterpriseFigure() {
        layout = new XYLayout();
        setLayoutManager(layout);

        labelName.setForegroundColor(ColorConstants.blue);
        add(labelName);
        setConstraint(labelName, new Rectangle(5,5,-1,-1));

        labelAddress.setForegroundColor(ColorConstants.lightBlue);
        add(labelAddress);
        setConstraint(labelAddress, new Rectangle(5,17,-1,-1));

        labelCapital.setForegroundColor(ColorConstants.lightBlue);
        add(labelCapital);
        setConstraint(labelCapital, new Rectangle(5,30,-1,-1));

        setForegroundColor(ColorConstants.black);
        setBorder(new LineBorder(5));

    }

    public void setLayout(Rectangle rect) {
        setBounds(rect);
    }

    public void setName(String text) {
        labelName.setText(text);
    }
```

```
    public void setAddress(String text) {
        labelAddress.setText(text);
    }

    public void setCapital(int capital) {
        labelCapital.setText("Capital:" + capital);
    }

}
```

EnterpriseEditPart 类：

```
package wbhgef.editpart;

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

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

import wbhgef.figure.EnterpriseFigure;
import wbhgef.model.Enterprise;
import wbhgef.model.Node;

public class EnterprisePart extends AbstractGraphicalEditPart {

    @Override
    protected IFigure createFigure() {
        // TODO Auto-generated method stub
        IFigure figure = new EnterpriseFigure();
        return figure;
    }

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

    protected void refreshVisuals() {
        EnterpriseFigure figure = (EnterpriseFigure)getFigure();
        Enterprise model = (Enterprise)getModel();

        figure.setName(model.getName());
```

```
        figure.setAddress(model.getAddress());
        figure.setCapital(model.getCapital());
    }

    public List<Node> getModelChildren() {
        return new ArrayList<Node>();
    }

}
```

现在只差它和员工的连接了。

需要建立工厂来管理 EditParts。工厂类用于处理适当的对象（EditPart）创建，基于任何我们想获得的对象类。工厂如下：

```
package wbhgef.editpart;

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

import wbhgef.model.Enterprise;

public class AppEditPartFactory implements EditPartFactory {

    @Override
    public EditPart createEditPart(EditPart context, Object model) {
        // TODO Auto-generated method stub
        AbstractGraphicalEditPart part = null;

        if(model instanceof Enterprise) {
            part = new EnterprisePart();
        }

        part.setModel(model);
        return part;
    }

}
```

在先前指南中 MyGraphicalEditor 类，我们已经重载了 *configureGraphicalViewer()*方法来通知 editor 我们要使用它的工厂。我们还创建了方法用于处理创建对象模型。最后，我们使用了 *initializeGraphicalViewer()*方法在 editor 中载入了对象模型。

在 MyGraphicalEditor 中添加如下代码：

```
protected void configureGraphicalViewer() {
    super.configureGraphicalViewer();
    GraphicalViewer viewer = getGraphicalViewer();
    viewer.setEditPartFactory(new AppEditPartFactory());
}

@Override
protected void initializeGraphicalViewer() {
    // TODO Auto-generated method stub
    GraphicalViewer viewer = getGraphicalViewer();
    viewer.setContents(CreateEnterprise());

    }

public Enterprise CreateEnterprise() {
    Enterprise enterprise = new Enterprise();

    enterprise.setName("Weapon Workshop");
    enterprise.setAddress("Rd ChangAn No.1");
    enterprise.setCapital(8000000);

    return enterprise;
    }
```

运行可得结果：

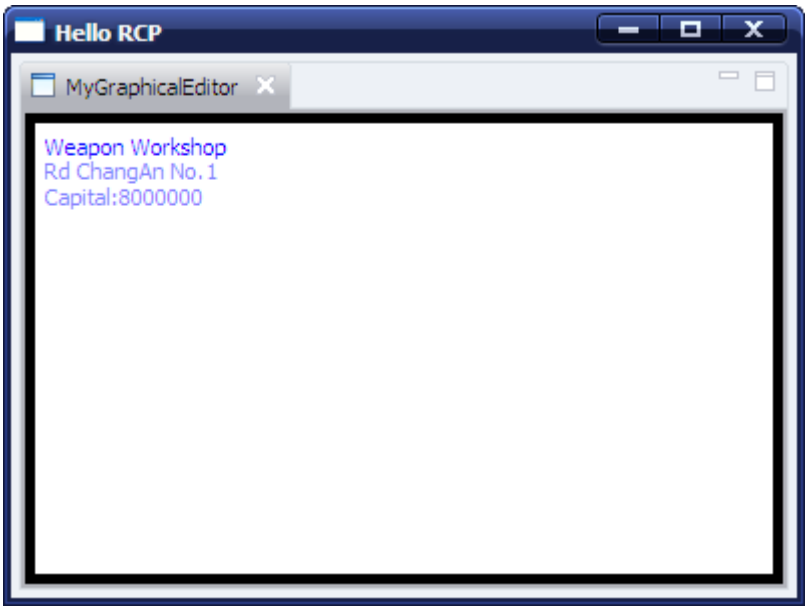


图9 添加 Model、EditPart、Figure 并进行连接的 RCP 效果

以上阶段为代码 WBHGEF02。

我们为 Service 和 Employee 重复上述操作，编写它们的 EditPart 和 Figure。代码如下，类似于 EnterpriseFigure 和 EnterprisePart：

ServiceFigure 类：

```
package wbhgef.figure;

import org.eclipse.draw2d.ColorConstants;
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.Label;
import org.eclipse.draw2d.LineBorder;
import org.eclipse.draw2d.ToolbarLayout;
import org.eclipse.draw2d.XYLayout;
import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.swt.graphics.Color;

public class ServiceFigure extends Figure {

    private Label labelName = new Label();
    private Label labelEtage = new Label();

    public ServiceFigure() {
        XYLayout layout = new XYLayout();
        setLayoutManager(layout);

        labelName.setForegroundColor(ColorConstants.darkGray);
        add(labelName, ToolbarLayout.ALIGN_CENTER);
        setConstraint(labelName, new Rectangle(5,17,-1,-1));

        labelEtage.setForegroundColor(ColorConstants.black);
        add(labelEtage, ToolbarLayout.ALIGN_CENTER);
        setConstraint(labelEtage, new Rectangle(5,5,-1,-1));

        setForegroundColor(new Color(null,
                (new Double(Math.random()*128)).intValue(),
                (new Double(Math.random()*128)).intValue(),
                (new Double(Math.random()*128)).intValue()));
        setBackgroundColor(new Color(null,
                (new Double(Math.random()*128)).intValue() + 128,
                (new Double(Math.random()*128)).intValue() + 128,
                (new Double(Math.random()*128)).intValue() + 128));
```

```
        setBorder(new LineBorder(1));
        setOpaque(true);
    }

    public void setName(String text) {
        labelName.setText(text);
    }

    public void setEtage(int etage) {
        labelEtage.setText("Etage:" + etage);
    }

    public void setLayout(Rectangle rect) {
        getParent().setConstraint(this, rect);
    }

}
```

ServicePart 类如下：

```
package wbhgef.editpart;

import java.util.List;

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

import wbhgef.figure.ServiceFigure;
import wbhgef.model.Node;
import wbhgef.model.Service;

public class ServicePart extends AbstractGraphicalEditPart {

    @Override
    protected IFigure createFigure() {
        // TODO Auto-generated method stub
        IFigure figure = new ServiceFigure();
        return figure;
    }

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

```
    }

    protected void refreshVisuals() {
        ServiceFigure figure = (ServiceFigure)getFigure();
        Service model = (Service)getModel();

        figure.setName(model.getName());
        figure.setEtage(model.getEtage());
        figure.setLayout(model.getLayout());
    }

    public List<Node> getModelChildren() {
        return ((Service)getModel()).getChildrenArray();
    }

}
```

注意，*getModelChildren()*方法刚结束，需要在 EnterpriseParty 类中重载如下：

```
    public List<Node> getModelChildren() {
//      return new ArrayList<Node>();
        return ((Enterprise)getModel()).getChildrenArray();
    }
```

接下来是 Employee。

```
package wbhgef.figure;

import org.eclipse.draw2d.ColorConstants;
import org.eclipse.draw2d.Figure;
import org.eclipse.draw2d.Label;
import org.eclipse.draw2d.LineBorder;
import org.eclipse.draw2d.ToolbarLayout;
import org.eclipse.draw2d.geometry.Rectangle;

public class EmployeeFigure extends Figure {
    private Label labelName = new Label();
    private Label labelFirstName = new Label();

    public EmployeeFigure() {
        ToolbarLayout layout = new ToolbarLayout();
        setLayoutManager(layout);
        labelFirstName.setForegroundColor(ColorConstants.black);
        add(labelFirstName, ToolbarLayout.ALIGN_CENTER);
```

```
        labelName.setForegroundColor(ColorConstants.darkGray);
        add(labelName, ToolbarLayout.ALIGN_CENTER);
        setForegroundColor(ColorConstants.darkGray);
        setBackgroundColor(ColorConstants.lightGray);
        setBorder(new LineBorder(1)); setOpaque(true);
    }

    public void setName(String text) {
        labelName.setText(text);
    }

    public void setFirstName(String text) {
        labelFirstName.setText(text);
    }

    public void setLayout(Rectangle rect) {
        getParent().setConstraint(this, rect);
    }

}
```

EmployeePart 类：

```
package wbhgef.editpart;

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

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

import wbhgef.figure.EmployeeFigure;
import wbhgef.model.Employee;
import wbhgef.model.Node;

public class EmployeePart extends AbstractGraphicalEditPart {

    @Override
    protected IFigure createFigure() {
        // TODO Auto-generated method stub
        IFigure figure = new EmployeeFigure();
        return figure;
    }
```

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

    protected void refreshVisuals(){
        EmployeeFigure figure = (EmployeeFigure)getFigure();
        Employee model = (Employee)getModel();

        figure.setName(model.getName());
        figure.setFirstName(model.getPrenom());
        figure.setLayout(model.getLayout());
    }

    public List<Node> getModelChildren() {
        return new ArrayList<Node>();
    }

}
```

注意对于这个类，*getModelChildren()*返回空表，这是正常的，因为雇员是分类中最低的一级，因此它没有 child。

现在需要修改工厂从而管理新的 EditParts。

AppEditPartFactory 类修改为：

```
public class AppEditPartFactory implements EditPartFactory {

    @Override
    public EditPart createEditPart(EditPart context, Object model) {
        // TODO Auto-generated method stub
        AbstractGraphicalEditPart part = null;

        if(model instanceof Enterprise) {
            part = new EnterprisePart();
        } else if(model instanceof Service) {
            part = new ServicePart();
        } else if(model instanceof Employee) {
            part = new EmployeePart();
        }

        part.setModel(model);
```

```
        return part;
    }
}
```

下面将部门和雇员移居至公司中。

仍是在 *MyGraphicalEditor* 的 *CreateEntreprise()*方法中，我们添加部门和雇员。注意每个元素的坐标和尺寸需要详述出(坐标相对坐标，以父元素的坐标为 0.0)。

*CreateEntreprise()*方法修改代码如下：

```
public Enterprise CreateEnterprise() {

    Enterprise enterprise = new Enterprise();

    enterprise.setName("同福客栈");
    enterprise.setAddress("西绒线胡同七号");
    enterprise.setCapital(8000000);

        Service service_QianTang = new Service();
        service_QianTang.setName("前堂");
        service_QianTang.setEtage(2);
        service_QianTang.setLayout(new Rectangle(30,50,250,150));

            Employee empolyee_1 = new Employee();
            empolyee_1.setName("掌柜");
            empolyee_1.setPrenom("佟");
            empolyee_1.setLayout(new Rectangle(25,40,60,40));
            service_QianTang.addChild(empolyee_1);

            Employee empolyee_2 = new Employee();
            empolyee_2.setName("展堂");
            empolyee_2.setPrenom("白");
            empolyee_2.setLayout(new Rectangle(100,60,60,40));
            service_QianTang.addChild(empolyee_2);

            Employee empolyee_3 = new Employee();
            empolyee_3.setName("秀才");
            empolyee_3.setPrenom("吕");
            empolyee_3.setLayout(new Rectangle(180,90,60,40));
            service_QianTang.addChild(empolyee_3);

        enterprise.addChild(service_QianTang);

        Service service_HouChu = new Service();
```

```
        service_HouChu.setName("后厨");
        service_HouChu.setEtage(1);
        service_HouChu.setLayout(new Rectangle(220,230,250,150));

            Employee employee_4 = new Employee();
            employee_4.setName("大嘴");
            employee_4.setPrenom("李");
            employee_4.setLayout(new Rectangle(40,70,60,40));
            service_HouChu.addChild(employee_4);

            Employee employee_5 = new Employee();
            employee_5.setName("芙蓉");
            employee_5.setPrenom("郭");
            employee_5.setLayout(new Rectangle(170,100,60,40));
            service_HouChu.addChild(employee_5);

        enterprise.addChild(service_HouChu);

    return enterprise;
}
```

添加子类后的运行效果如下图：

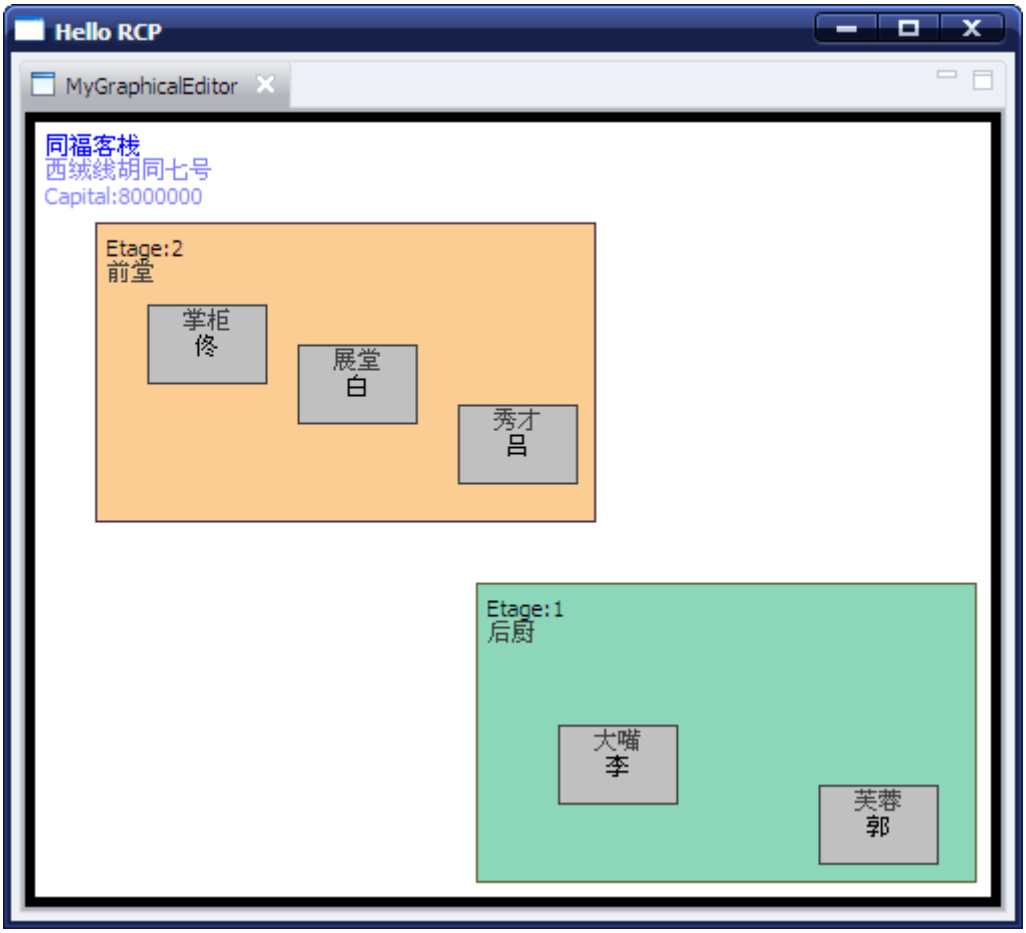


图10 添加 Service 及 Employee 后的 Model、EditPart、Figure 并进行连接的 RCP 效果

至此阶段代码为 WBHGEF03。

# 3. 与图形（Figure）的交互

在第三部分，我们开始与图形进行交互：对一个盒子进行选择、移动、放大缩小。

GEF 提供一个对象模型的视觉展现。在上述简单的例子里，我们已经将 Enterprise 设置为顶端，并让其下层为 Service，Service 的下层为 Employee。

GEF 基于 Model – View – Controller（模型——视图——控制器，MVC）架构。事实上，它可以叫做 Model – Controller – View（模型——控制器——视图），这样我们可以用控制器区分开模型（Enterprise）和视图（图形中的盒子）。控制器是作为中介被引用的。它促使视图依赖模型，并且根据对图形的动作修改模型（Model）。

这个指南中的 Java 包深入展现了依赖于 MVC 模型中的角色的类。

在这个部分，我们还将见到如何和图形（Figure，Graph）交互，控制器（Controller）如何执行命令（Command），Command 如何修改模型，以及模型如何通知视图它发生改变从而使视图更新。请别走开，事实上它比看起来的要简单。

我们向可以移动 Enterprise 中的 Service，并移动 Service 中的 Employee。为此，需要为 Employee 和 Service 创建一个 Command。（这些类创建在 wbhgef.command 包中）

作为准备工作，我们创建一个小的抽象桩（abstract stub）从而避免一些后续的问题。

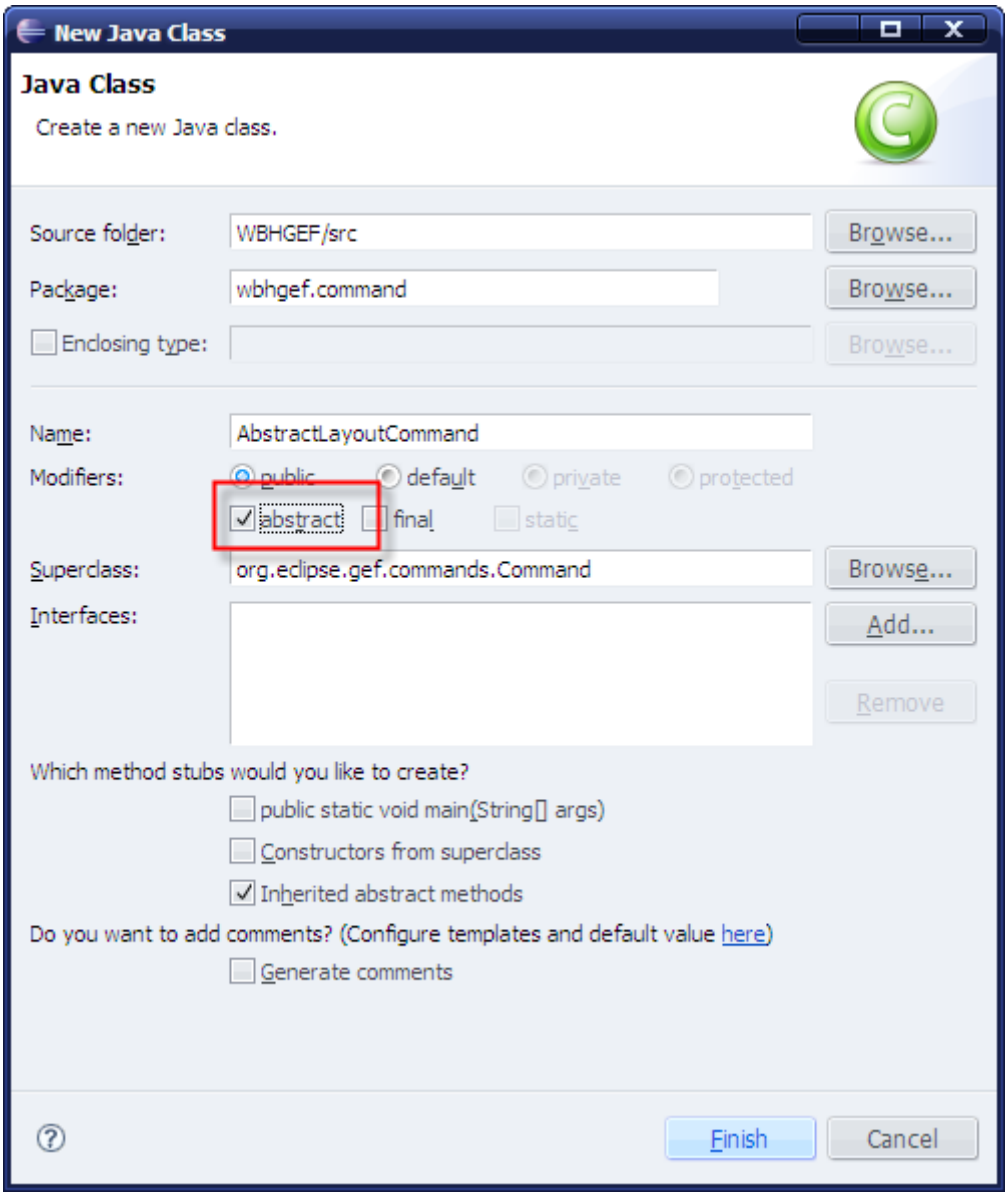


图11 建立 AbstractLayoutCommand 抽象类

AbstractLayoutCommand 代码如下：

```
package wbhgef.command;
```

```
import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.gef.commands.Command;

public abstract class AbstractLayoutCommand extends Command {
    public abstract void setConstraint(Rectangle rect);
    public abstract void setModel(Object model);
}
```

下面我们为 Employee 和 Service 写两个继承自这个抽象类的类：

Employee 改变布局的 Command 类的代码为：

```
package wbhgef.command;

import org.eclipse.draw2d.geometry.Rectangle;

import wbhgef.model.Employee;

public class EmployeeChangeLayoutCommand extends AbstractLayoutCommand
{

    private Employee model;
    private Rectangle layout;

    public void execute() {
        model.setLayout(layout);
    }

    @Override
    public void setConstraint(Rectangle rect) {
        // TODO Auto-generated method stub
        this.layout = rect;
    }

    @Override
    public void setModel(Object model) {
        // TODO Auto-generated method stub
        this.model = (Employee)model;
    }

}
```

Service 改变布局的 Command 类的代码为：

```
package wbhgef.command;
```

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

import wbhgef.model.Service;

public class ServiceChangeLayoutCommand extends AbstractLayoutCommand {

    private Service model;
    private Rectangle layout;

    public void execute() {
        model.setLayout(layout);
    }

    @Override
    public void setConstraint(Rectangle rect) {
        // TODO Auto-generated method stub
        this.layout = rect;
    }

    @Override
    public void setModel(Object model) {
        // TODO Auto-generated method stub
        this.model = (Service)model;
    }
}
```

这两个 Command 将会被 EditPolicy 调用，后面会将 EditPolicy 用于对象的 EditPart。

我们创建如下类在包 wbh.editpolicy 中。

AppEditLayoutPolicy 类代码如下：

```
package wbhgef.editpolicy;

import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.gef.EditPart;
import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.XYLayoutEditPolicy;
import org.eclipse.gef.requests.CreateRequest;

import wbhgef.command.AbstractLayoutCommand;
import wbhgef.command.EmployeeChangeLayoutCommand;
```

```
import wbhgef.command.ServiceChangeLayoutCommand;
import wbhgef.editpart.EmployeePart;
import wbhgef.editpart.ServicePart;

public class AppEditLayoutPolicy extends XYLayoutEditPolicy {

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

        AbstractLayoutCommand command = null;

        if(child instanceof EmployeePart) {
            command = new EmployeeChangeLayoutCommand();
        } else if(child instanceof ServicePart) {
            command = new ServiceChangeLayoutCommand();
        }

        command.setModel(child.getModel());
        command.setConstraint((Rectangle)constraint);
        return command;
    }

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

}
```

现在，我们需要为这个 EditPart 应用一些我们想要的 EditPolicy。

在 EnterprisePart 和 ServicePart 类中，为 *createEditPolicies()*方法添加一行如下代码：

```
protected void createEditPolicies() {
    // TODO Auto-generated method stub
    installEditPolicy(EditPolicy.LAYOUT_ROLE, new
AppEditLayoutPolicy());
}
```

下面运行一试：

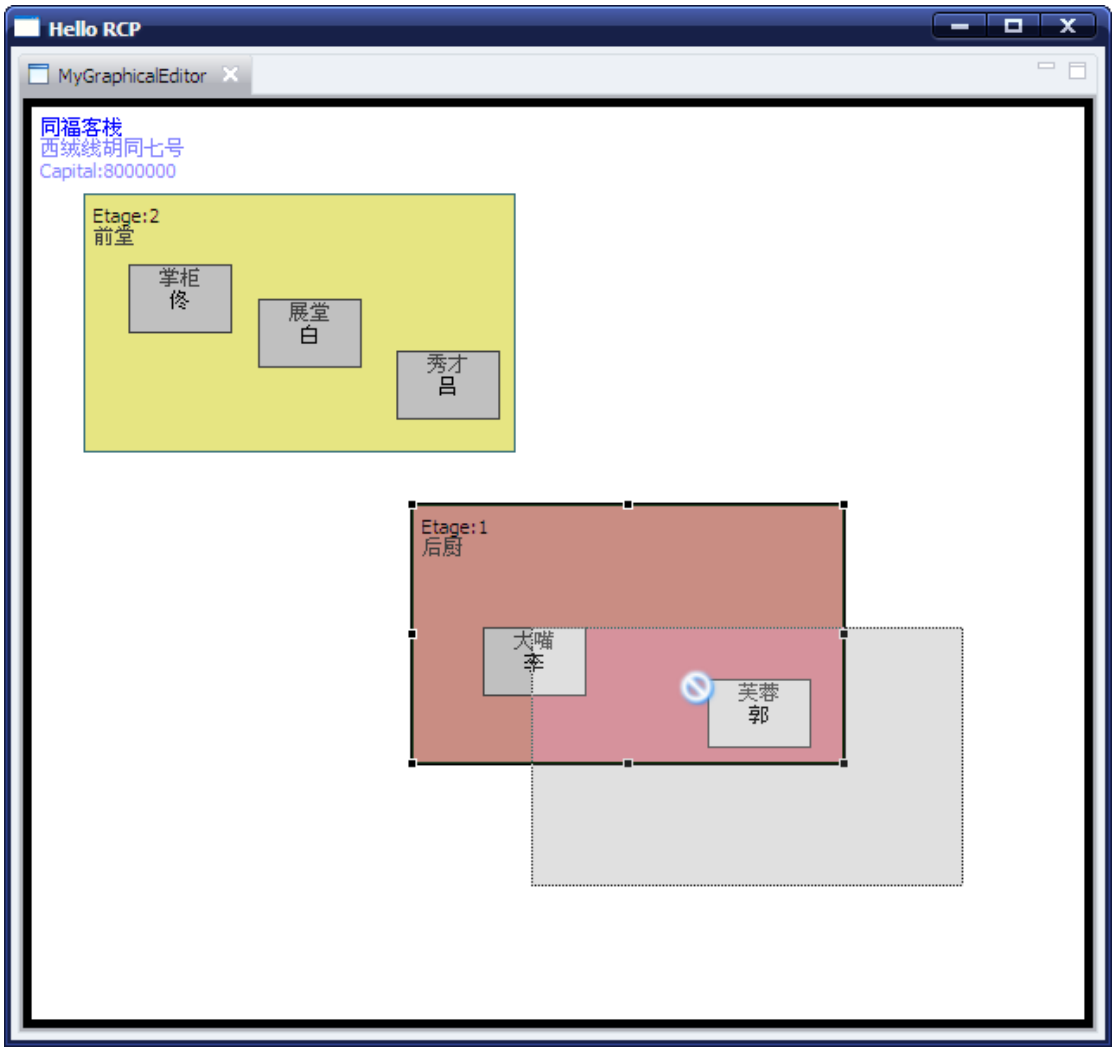


图12 为 EditPart 添加了布局更改的 EditPolicy 的效果

现在可以选择一个图形，甚至用 Ctrl 选择多个，但是移动和改变大小时，会有问题。

它们并没有移动到想要的位置，而是返回了原来的位置，why?

流程还没有结束：View 更新了 Model，即是说，此时由于鼠标移动图形的位置，模型的位置信息已经改变，但是，Model 并未更新 View，即是说，模型的变化并未更新所显示的视图。(看起来很怪诞)

需要做的是给我们的 EditPart 安置 Listener（监听器），这样它们就可以检查到模型的变化，并及时更新视图。

首先修改 Node 类使得属性变化时可以触发事件。

Node 类代码修改如下：

```
public class Node {

    private PropertyChangeSupport listeners;
    public static final String PROPERTY_LAYOUT = "NodeLayout";
     (……)
```

```
    public Node() {
        (……)
        this.listeners = new PropertyChangeSupport(this);
    }

    (……)
    public void addPropertyChangeListener(PropertyChangeListener
listener) {
        listeners.addPropertyChangeListener(listener);
    }

    public PropertyChangeSupport getListeners() {
        return listeners;
    }

    public void removePropertyChangeListener(PropertyChangeListener
listener) {
        listeners.removePropertyChangeListener(listener);
    }

}
```

我们修改 *setLayout()*方法从而当它被调用时，可以 fire（传递）出属性。

*setLayout()*方法代码如下：

```
    public void setLayout(Rectangle newLayout) {
//      this.layout = layout;
        Rectangle oldLayout = this.layout;
        this.layout = newLayout;
        getListeners().firePropertyChange(PROPERTY_LAYOUT, oldLayout,
newLayout);
    }
```

现在，当模型元素移动时，它将触发一个事件。但是目前没有接收这个事件的。

我们写一个抽象类处理事件管理。然后让 EditPart 继承这个类。

创建一个 AppAbstractEditPart 的抽象类，继承 AbstractGraphicalEditPart 类，实现 PropertyChangeListener 接口，如下图：

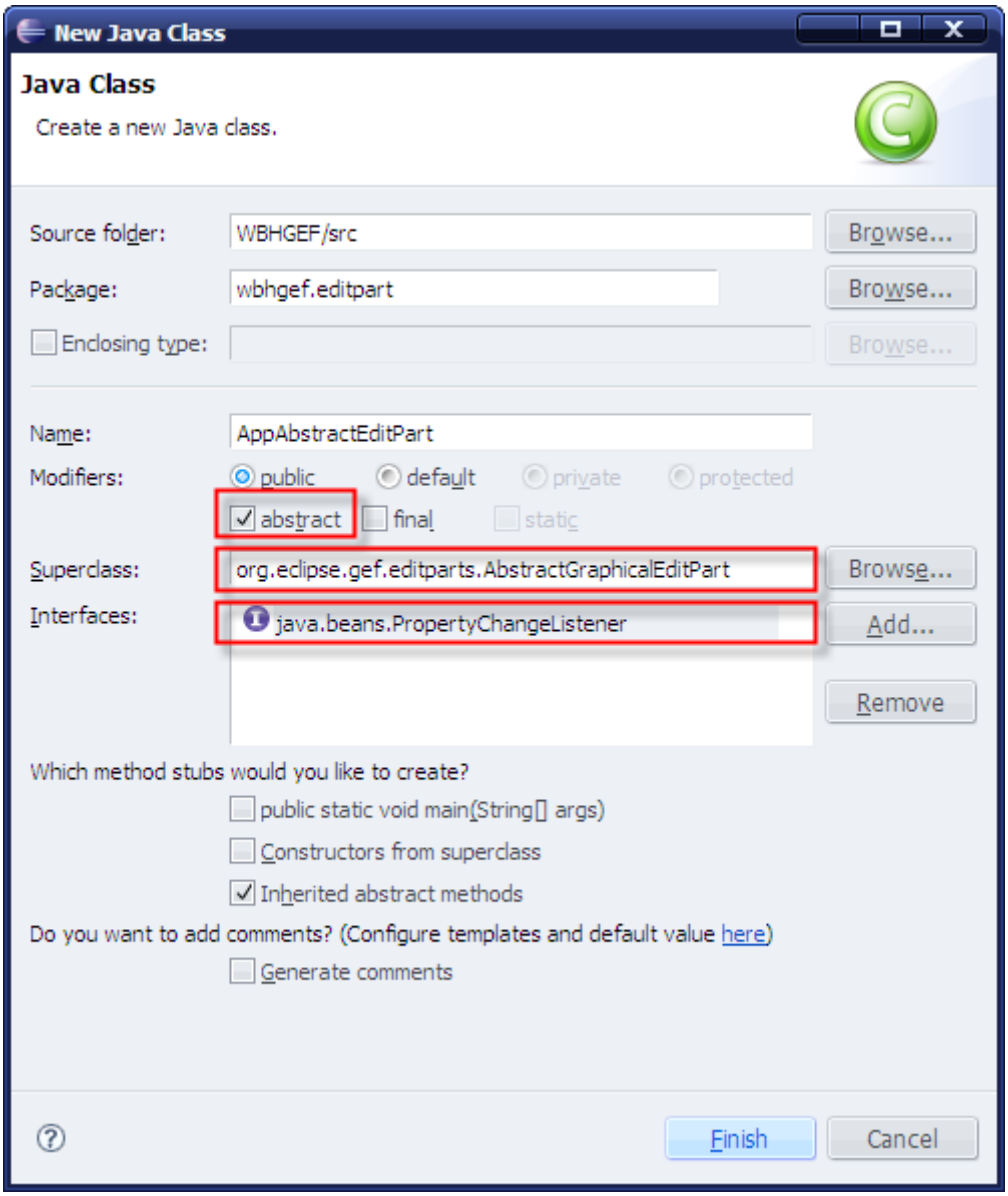


图13 创建 AppAbstractEditPart 抽象类

代码如下：

```
package wbhgef.editpart;

//import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;

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

import wbhgef.model.Node;

public abstract class AppAbstractEditPart extends
```

```
AbstractGraphicalEditPart
        implements PropertyChangeListener {

//  @Override
//  protected IFigure createFigure() {
//      // TODO Auto-generated method stub
//      return null;
//  }
//
//  @Override
//  protected void createEditPolicies() {
//      // TODO Auto-generated method stub
//
//  }
//
//  @Override
//  public void propertyChange(PropertyChangeEvent evt) {
//      // TODO Auto-generated method stub
//
//  }

    public void activate() {
        super.activate();
        ((Node)getModel()).addPropertyChangeListener(this);
    }

    public void deactivate() {
        super.deactivate();
        ((Node)getModel()).removePropertyChangeListener(this);
    }

}
```

然后，为 EmployeeParty、ServicePart 和 EnterprisePart 的类宣言修改代码如下：

```
public class XXXXPart extends AppAbstractEditPart {
    (……)
}
```

并将以下方法附加于三个类中：

```
@Override
public void propertyChange(PropertyChangeEvent evt) {
```

```
        // TODO Auto-generated method stub
        if(evt.getPropertyName().equals(Node.PROPERTY_LAYOUT))
refreshVisuals();
    }
```

这个方法会接收最终由 *firePropertyChange()*触发的事件。

如果事件匹配一个 PROPERTY_LAYOUT 类型的属性，那么我们刷新这个发生改变的模型元素的显示。

运行程序并移动 Service 或者 Employee，如下图：

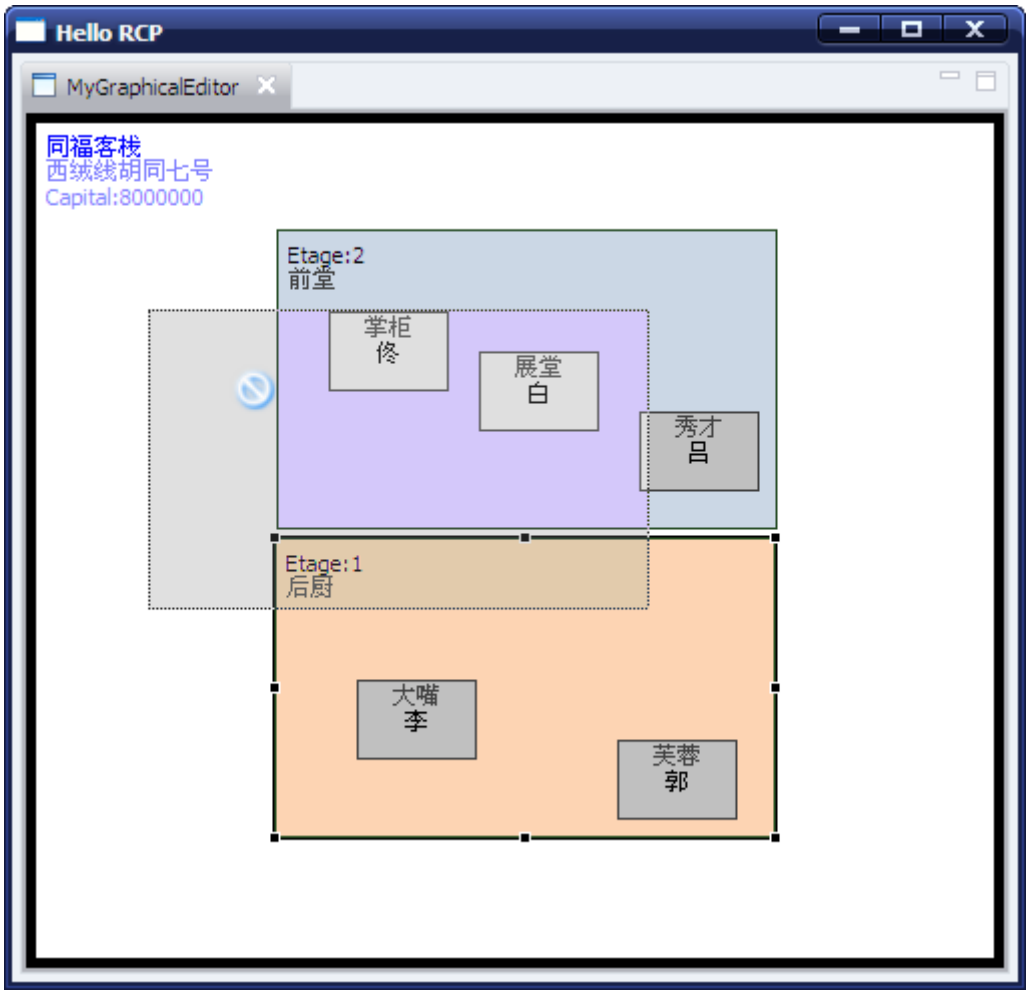


图14 移动 Service 前

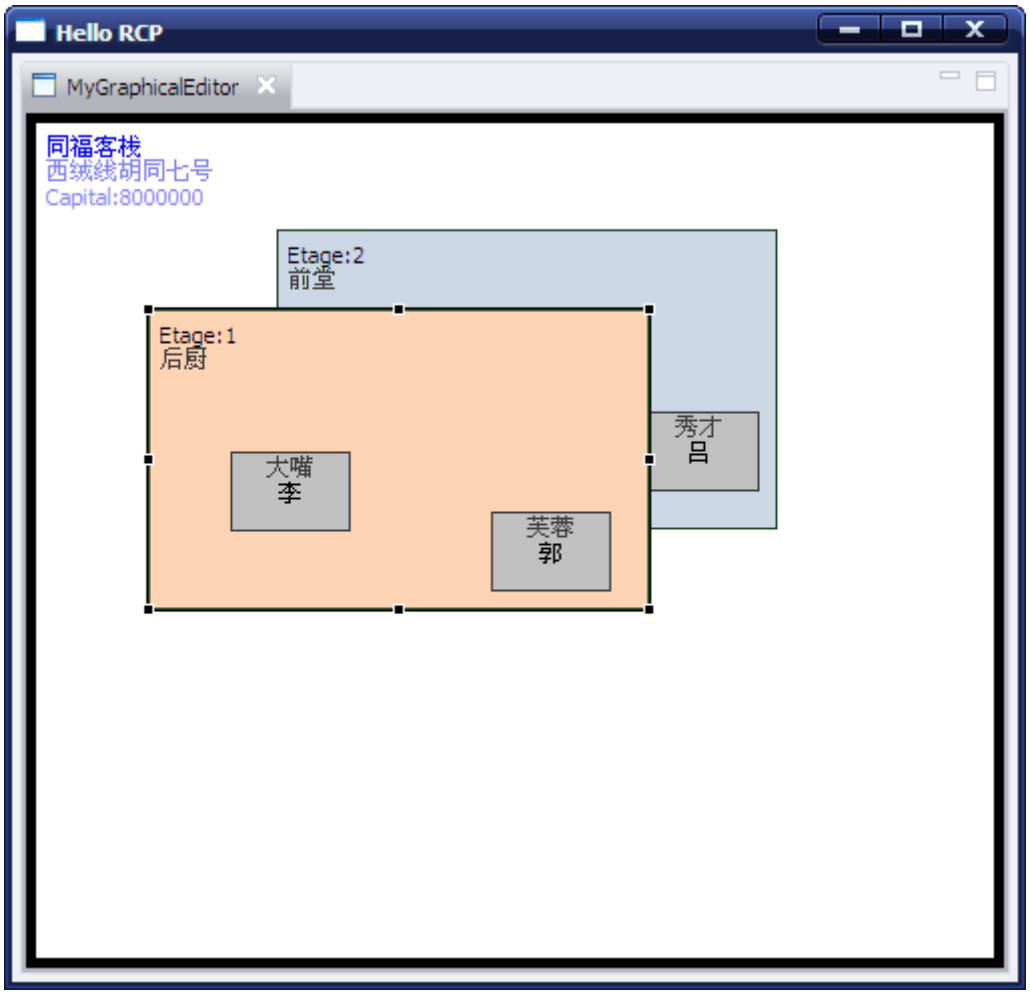


图15 Service 的移动反应到图形上

这个阶段的代码为 WBHGEF04。

# 4. 撤销/回复（Undo/Redo）

我们已经可以和图形交互，现在我们要添加管理功能，如 undo/redo，完全由 GEF 处理，同样也将元素移走。首先，为 editor 添加一个工具栏（toolbar），它包含要实现的操作按钮。

为了给 editor 添加一个 toolbar，我们需要先给它添加一个 Contributor（助推器），命名为 **MyGraphicalEditorActionBarContributor**，继承自 **ActionBarContributor**.。

代码如下：

```
package wbhgef;
```

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

public class MyGraphicalEditorActionBarContributor extends
ActionBarContributor {

    @Override
    protected void buildActions() {
        // TODO Auto-generated method stub

    }

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

    }

}
```

现在编辑 plugin.xml，在 Extensions 表，为 editor 的 **contributorClass** 添加 class path，如下图：

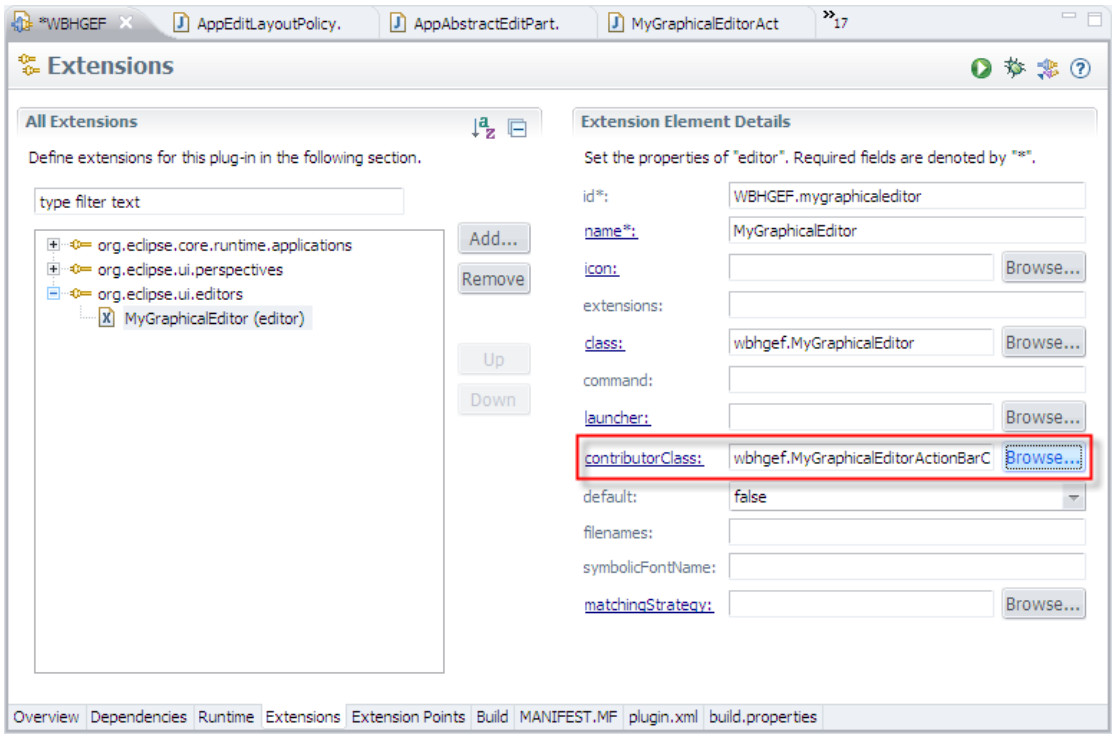


图16 为 editor 的 contributorClass 添加 class path 到自定义的 EditorActionBarContributor

余下的工作是显示这个空白的 toolbar。在 **ApplicationWorkbenchWindowAdvisor** 类的 *preWindowOpen()*方法中添加如下代码：

```
        configurer.setShowCoolBar(true);
```

现在添加 editor 中对 undo/redo 的支持。

在 **MyGraphicalEditorActionBarContributor** 类中，添加如下代码：

```
package wbhgef;

import org.eclipse.gef.ui.actions.ActionBarContributor;
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 MyGraphicalEditorActionBarContributor extends
ActionBarContributor {

    @Override
    protected void buildActions() {
        // TODO Auto-generated method stub
        addRetargetAction(new UndoRetargetAction());
        addRetargetAction(new RedoRetargetAction());

    }

    @Override
    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()));
    }

}
```

这样做的目的是在 bar 中添加 undo/redo 控制并配置它们相关的 action（由 GEF 处理）。

现在做的所有事情是配置感兴趣的 Command，如 **ServiceChangeLayoutCommand** 和 **EmployeeChangeLayoutCommand**：

ServiceChangeLayoutCommand 类代码修改如下：

```
public class ServiceChangeLayoutCommand extends AbstractLayoutCommand {
```

```
    (……)
    private Rectangle oldLayout;

    (……)

    @Override
    public void setModel(Object model) {
        // TODO Auto-generated method stub
        this.model = (Service)model;
        this.oldLayout = ((Service)model).getLayout();
    }

    public void undo() {
        this.model.setLayout(this.oldLayout);
    }

}
```

EmployeeChangeLayoutCommand 类代码修改如下：

```
public class EmployeeChangeLayoutCommand extends AbstractLayoutCommand
{

    (……)
    private Rectangle oldLayout;

    (……)

    @Override
    public void setModel(Object model) {
        // TODO Auto-generated method stub
        this.model = (Employee)model;
        this.oldLayout = ((Employee)model).getLayout();
    }

    public void undo() {
        this.model.setLayout(this.oldLayout);
    }

}
```

效果如下图：

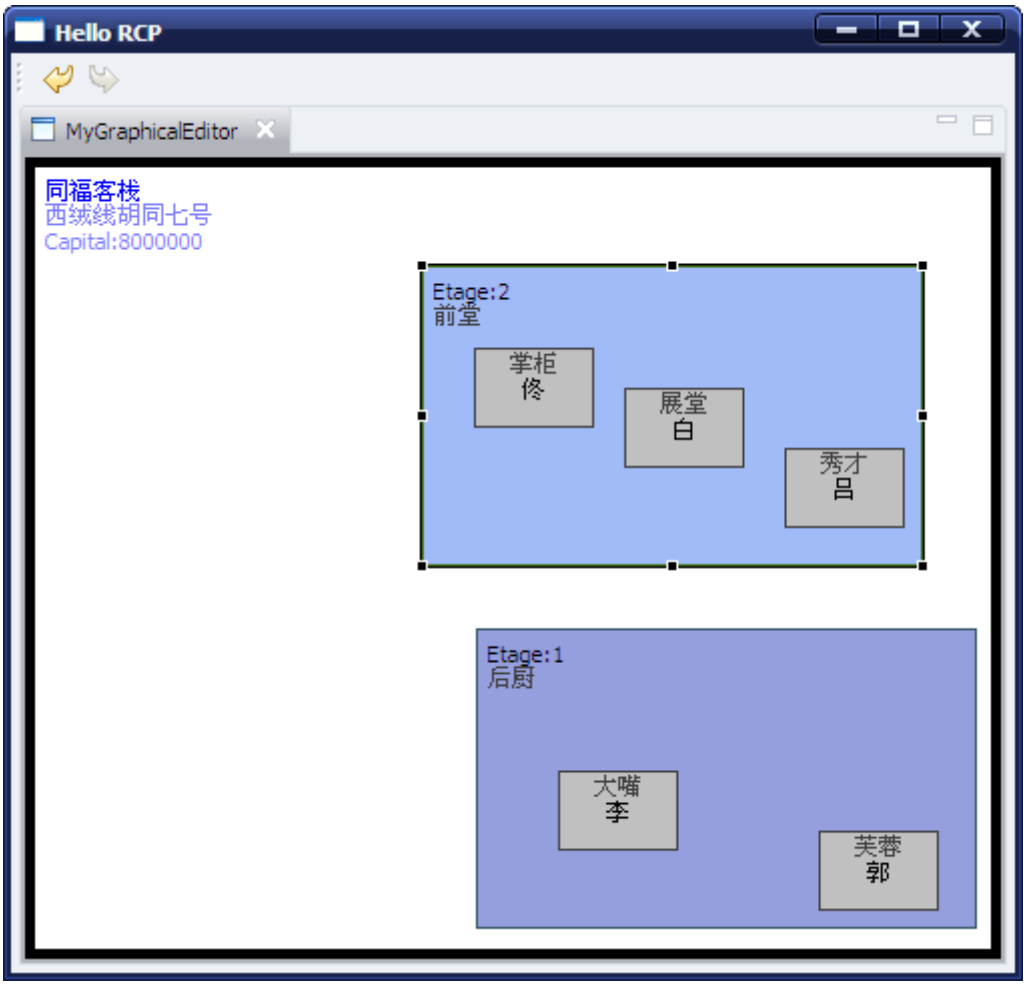


图17 添加并实现 undo/redo 箭头的 toolbar

此处设置完 undo 后，似乎是 GEF 自动记录并形成 redo 操作。

现在添加对删除的支持，我们希望可以删除整个 Service，或者只是删除一个 Employee。大体原则和 undo/redo 一样：

- 添加一个 toolbar 控制
- 创建相应的 command
- 创建一个使用这个 command 的 policy
- 在 ServicePart 和 EmployeePart 中应用这个 Policy
- 在 Node 的 addChild()和 removeChild()方法中触发事件，然后通过 Enterprise 和 Service 的 EditPart 中的 refreshChildren()方法更新视图

在 **MyGraphicalEditorActionBarContributor** 添加 action 和删除按钮。

```
public class MyGraphicalEditorActionBarContributor extends
ActionBarContributor {

    @Override
```

```
    protected void buildActions() {
        // TODO Auto-generated method stub
         (……)
        addRetargetAction(new DeleteRetargetAction());
    }
     (……)
    public void contributeToToolBar(IToolBarManager toolBarManager) {
         (……)
        toolBarManager.add(getAction(ActionFactory.DELETE.getId()));
    }
}
```

command 也很简单，代码如下：

```
package wbhgef.command;

import org.eclipse.gef.commands.Command;

import wbhgef.model.Node;

public class DeleteCommand extends Command {
    private Node model;
    private Node parentModel;

    public void execute() {
        this.parentModel.removeChild(model);
    }

    public void setModel() {
        this.model = (Node)model;
    }

    public void setParentModel(Object model) {
        parentModel = (Node)model;
    }

    public void undo() {
        this.parentModel.addChild(model);
    }

}
```

下面创建相关联的 policy：（疑问？何时创建 policy？为何 undo/redo 不用？

而 EditLayout 和 Delete 需要？）

```
package wbhgef.editpolicy;

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

import wbhgef.command.DeleteCommand;

public class AppDeletePolicy extends ComponentEditPolicy {

    protected Command createDeleteCommand(GroupRequest deleteRequest) {
        DeleteCommand command = new DeleteCommand();
        command.setModel(getHost().getModel());
        command.setParentModel(getHost().getParent().getModel());
        return command;
    }
}
```

可见方法同 undo/redo。

将 policy 安装到 ServicePart 和 EmployeePart 中：

```
    protected void createEditPolicies() {
        // TODO Auto-generated method stub
        installEditPolicy(EditPolicy.COMPONENT_ROLE, new
AppDeletePolicy());
    }
```

我们在 Node 类中创建一个新的属性，然后在 *addChild()*和 *removeChild()*中为这个属性的变化触发一个事件，代码如下：

```
public class Node {

    (……)
    public static final String PROPERTY_ADD = "NodeAddChild";
    public static final String PROPERTY_REMOVE = "NodeRemoveChild";

    (……)
    public boolean addChild(Node child) {
//      child.setParent(this);
//      return this.children.add(child);
        boolean b = this.children.add(child);
        if(b) {
            child.setParent(this);
```

```
            getListeners().firePropertyChange(PROPERTY_ADD, null,
child);
        }
        return b;
    }

    public boolean removeChild(Node child) {
//      return this.children.remove(child);
        boolean b = this.children.remove(child);
        if(b) {
            getListeners().firePropertyChange(PROPERTY_REMOVE, child,
null);
        }
        return b;
    }
```

最后的工作就是当模型发生变化时，刷新图形，在 EnterprisePart 和 ServicePart 中分别刷新 child：

EnterprisePart 中：

```
public class EnterprisePart extends AppAbstractEditPart {
     (……)
    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        // TODO Auto-generated method stub
         (……)
        if(evt.getPropertyName().equals(Node.PROPERTY_ADD))
refreshChildren();
        if(evt.getPropertyName().equals(Node.PROPERTY_REMOVE))
refreshChildren();
    }

}
```

ServicePart中：

```
public class ServicePart extends AppAbstractEditPart {
     (……)
    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        // TODO Auto-generated method stub
         (……)
        if(evt.getPropertyName().equals(Node.PROPERTY_ADD))
refreshChildren();
```

```
        if(evt.getPropertyName().equals(Node.PROPERTY_REMOVE))
refreshChildren();
    }

}
```

运行效果如下图：

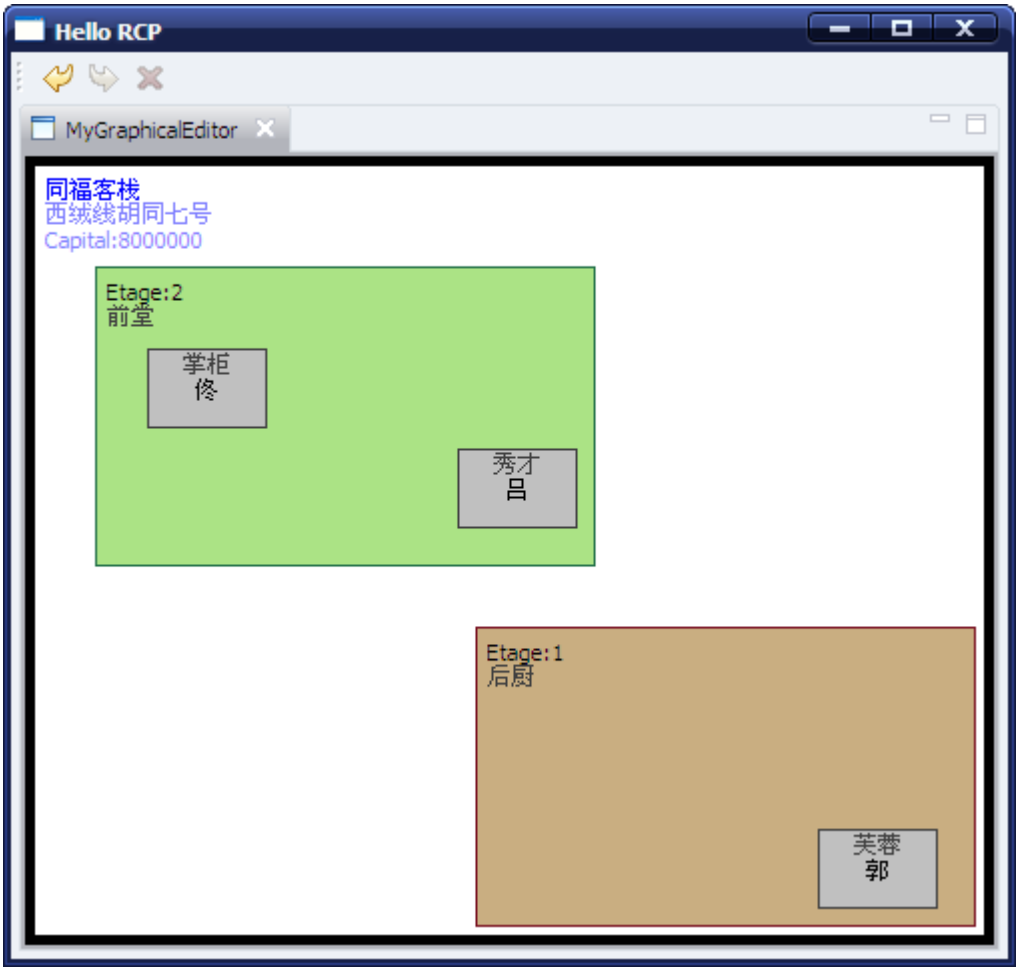


图18 添加并实现 delete 按钮的 toolbar

疑问：命令栈也自动生成？可多步 undo/redo？

此阶段代码为 WBHGEF05。

# 5. 缩放（Zoom）和快捷键（Keyboard Shortcut）

在这个指南中，我们将添加几个简单的小功能，如缩放图形，然后将把快捷键和 action 关联起来。

与 delete 和 undo/redo 一样，缩放（Zoom）是一个 action，因此将给 toolbar 添加 action。

这即是说，缩放是图形对它自己的一个 action，而不像之前的是对模型的 action。因此需要对 **MyGraphicalEditor** 进行修改。

现在开始。

我们进入 *configureGraphicalViewer()*方法，了解如何请求使用一个整合 Zoom 功能的 EditPart，以及如何将它添加到图形，和最终如何创建 Zoom 功能。

**MyGraphicalEditor** 修改如下：

```
public class MyGraphicalEditor extends GraphicalEditor {
    (……)
   protected void configureGraphicalViewer() {

       double[] zoomLevels;
       ArrayList<String> zoomContributions;
        (……)
       ScalableRootEditPart rootEditPart = new ScalableRootEditPart();
       viewer.setRootEditPart(rootEditPart);

       ZoomManager manager = rootEditPart.getZoomManager();
       getActionRegistry().registerAction(new ZoomInAction(manager));
       getActionRegistry().registerAction(new
ZoomOutAction(manager));

       zoomLevels = new double[] {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);

       zoomContributions = new ArrayList<String>();
       zoomContributions.add(ZoomManager.FIT_ALL);
       zoomContributions.add(ZoomManager.FIT_HEIGHT);
       zoomContributions.add(ZoomManager.FIT_WIDTH);
       manager.setZoomLevelContributions(zoomContributions);
```

```
    }

    public Object getAdapter(Class type) {
        if(type == ZoomManager.class) {
            return
((ScalableRootEditPart)getGraphicalViewer().getRootEditPart()).getZoo
mManager();
        } else {
            return super.getAdapter(type);
        }
    }
     (……)
}
```

现在只需给 toolbar 添加 action，MyGraphicalEditorActionBarContributor 代码如下：

```
public class MyGraphicalEditorActionBarContributor extends
ActionBarContributor {

    @Override
    protected void buildActions() {
        // TODO Auto-generated method stub
         (……)
        addRetargetAction(new ZoomInRetargetAction());
        addRetargetAction(new ZoomOutRetargetAction());
    }
     (……)

    public void contributeToToolBar(IToolBarManager toolBarManager) {
         (……)
        toolBarManager.add(new Separator());
        toolBarManager.add(getAction(GEFActionConstants.ZOOM_IN));
        toolBarManager.add(getAction(GEFActionConstants.ZOOM_OUT));
        toolBarManager.add(new ZoomComboContributionItem(getPage()));
    }
}
```

运行效果如下图：

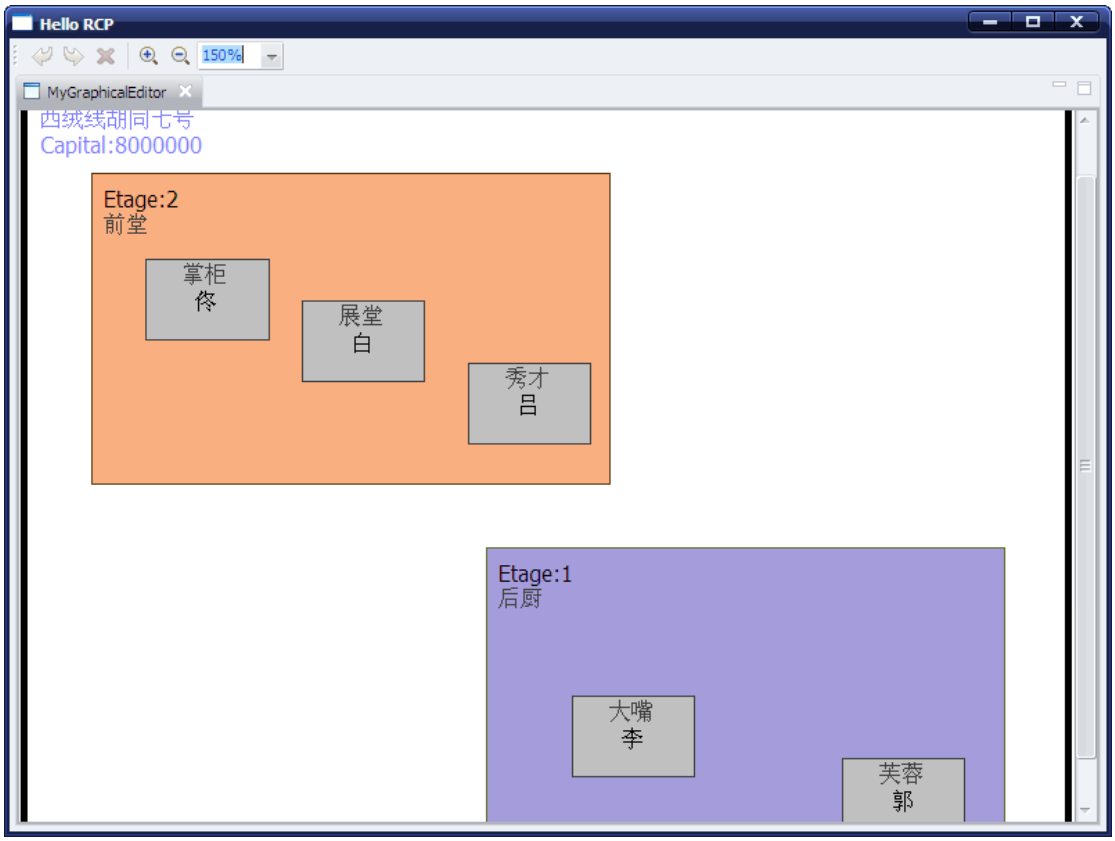


图19 添加并实现 zoom 按钮的 toolbar

此处所有键盘快捷键失常，需要建立按键“+”、“-”到缩放、“del”到删除的映射。

很简单：

所需要的仅是一个“key handler”，添加所有需要的关联到透视图的 action，并通过给 viewer 添加 key handler 来完成这项工作。

修改 MyGraphicalEditor 代码如下“

```
public class MyGraphicalEditor extends GraphicalEditor {

    (……)
   protected void configureGraphicalViewer() {

        (……)
       KeyHandler keyHandler = new KeyHandler();
        (……)
       keyHandler.put(
              KeyStroke.getPressed(SWT.DEL, 127, 0),

   getActionRegistry().getAction(ActionFactory.DELETE.getId()));
       keyHandler.put(
```

```
                KeyStroke.getPressed('+',SWT.KEYPAD_ADD, 0),

    getActionRegistry().getAction(GEFActionConstants.ZOOM_IN));
        keyHandler.put(
                KeyStroke.getPressed('-',SWT.KEYPAD_SUBTRACT, 0),

    getActionRegistry().getAction(GEFActionConstants.ZOOM_OUT));

        viewer.setProperty(
                MouseWheelHandler.KeyGenerator.getKey(SWT.NONE),
                MouseWheelZoomHandler.SINGLETON);

        viewer.setKeyHandler(keyHandler);
    }

     (……)
}
```

运行后可使用快捷键。

此阶段代码为 WBHGEF06。

# 6. 大纲（Outline）

GEF 提供为 editor 提供一些视图类型，到现在我们已经给 editor 装了一些可爱的小盒子。难道不应该装一个 view 将图显示成树状（tree）？

不用说第二次。

事实上这个特定的 Eclipse 视图叫做“大纲”。

首相在 RCP plug-in 中添加这个视图。

透视图（Perspective）类定义了我们 plug-in 中的视图（view），我们给它添加大纲视图。

在 Perspective 类中修改代码如下：

```
public class Perspective implements IPerspectiveFactory {

    public void createInitialLayout(IPageLayout layout) {
        String editorArea = layout.getEditorArea();
        layout.setEditorAreaVisible(true);
        layout.addStandaloneView(IPageLayout.ID_OUTLINE, true,
```

```
IPageLayout.LEFT, 0.3f, editorArea);
    }
}
```

运行到这个程度，可以看见一个视图出现在图形的左边，如下图：

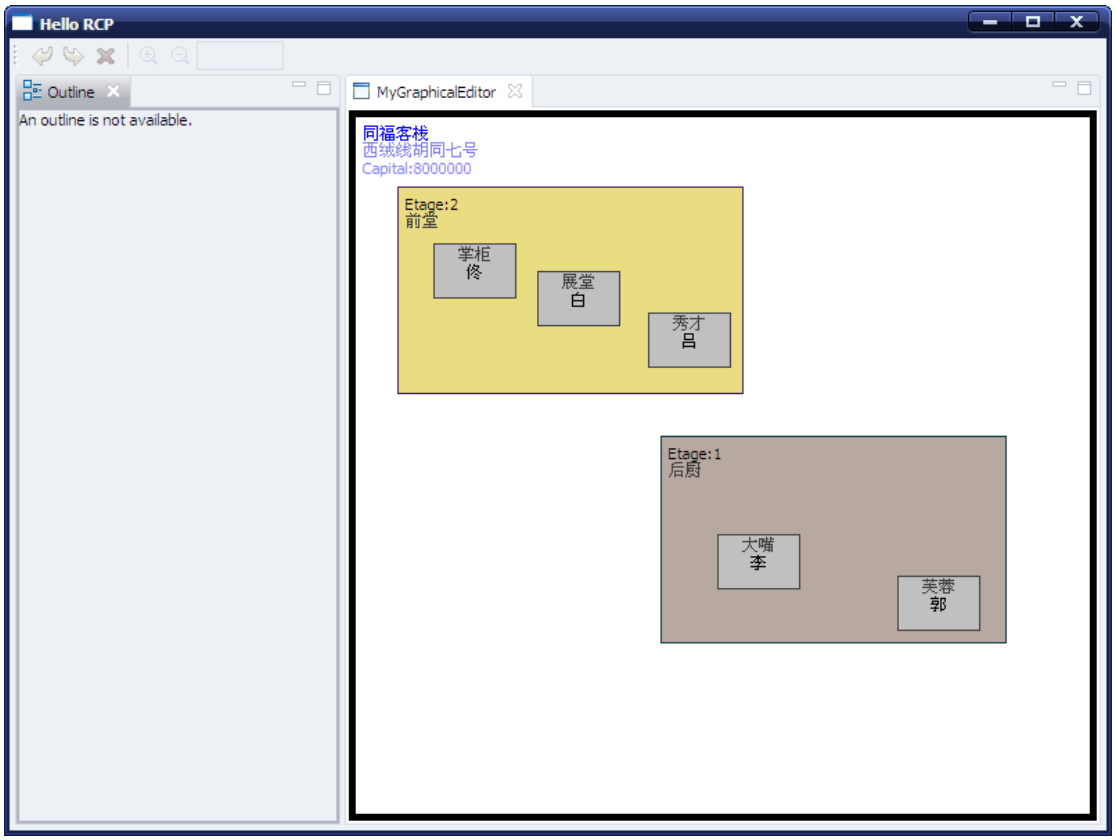


图20 添加 outline 的 view

大纲视图有它自己的 EditPart，向对于图形，它可以有不同形式的展现方式。此处不作讨论。

我们给这个视图创建所有必须的 EditPart，并且在 wbhgef.part.tree 包中关联工厂。

现在从 EditPart 将要扩展的基抽象类开始。

建立 AppAbstractTreeEditPart 类，扩展 AbstractTreeEditPart 类，实现 PropertyChangeListener 接口，代码如下：

```
package wbhgef.editpart.tree;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;

import org.eclipse.gef.editparts.AbstractTreeEditPart;

import wbhgef.model.Node;
```

```
public abstract class AppAbstractTreeEditPart extends
AbstractTreeEditPart
        implements PropertyChangeListener {

    public void activate() {
        super.activate();
        ((Node)getModel()).addPropertyChangeListener(this);
    }

    public void deactivate() {
        ((Node)getModel()).removePropertyChangeListener(this);
        super.deactivate();
    }

    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        // TODO Auto-generated method stub

    }

}
```

然后是 EditPart。我们注意到它们和图形框架一样。

EntrepriseTreeEditPart 代码如下：

```
package wbhgef.editpart.tree;

import java.beans.PropertyChangeEvent;
import java.util.List;

import wbhgef.model.Enterprise;
import wbhgef.model.Node;

public class EnterpriseTreeEditPart extends AppAbstractTreeEditPart {

    protected List<Node> getModelChildren() {
        return ((Enterprise)getModel()).getChildrenArray();
    }

    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        // TODO Auto-generated method stub
```

```
        if(evt.getPropertyName().equals(Node.PROPERTY_ADD))
refreshChildren();
        if(evt.getPropertyName().equals(Node.PROPERTY_REMOVE))
refreshChildren();
    }

}
```

ServiceTreeEditPart 代码如下：

```
package wbhgef.editpart.tree;

import java.beans.PropertyChangeEvent;
import java.util.List;

import org.eclipse.gef.EditPolicy;
import org.eclipse.ui.ISharedImages;
import org.eclipse.ui.PlatformUI;

import wbhgef.editpolicy.AppDeletePolicy;
import wbhgef.model.Node;
import wbhgef.model.Service;

public class ServiceTreeEditPart extends AppAbstractTreeEditPart {

    protected List<Node> getModelChildren() {
        return ((Service)getModel()).getChildrenArray();
    }

    @Override
    protected void createEditPolicies() {
        installEditPolicy(EditPolicy.COMPONENT_ROLE, new
AppDeletePolicy());
    }

    public void refreshVisuals() {
        Service model = (Service)getModel();
        setWidgetText(model.getName());

    setWidgetImage(PlatformUI.getWorkbench().getSharedImages().getIma
ge(ISharedImages.IMG_OBJ_ELEMENT));
    }
```

```
    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        // TODO Auto-generated method stub
        if(evt.getPropertyName().equals(Node.PROPERTY_ADD))
refreshChildren();
        if(evt.getPropertyName().equals(Node.PROPERTY_REMOVE))
refreshChildren();
    }

}
```

EmployeeTreeEditPart 代码如下：

```
package wbhgef.editpart.tree;

import java.beans.PropertyChangeEvent;
import java.util.List;

import org.eclipse.gef.EditPolicy;
import org.eclipse.ui.ISharedImages;
import org.eclipse.ui.PlatformUI;

import wbhgef.editpolicy.AppDeletePolicy;
import wbhgef.model.Employee;
import wbhgef.model.Node;

public class EmployeeTreeEditPart extends AppAbstractTreeEditPart {

    protected List<Node> getModelChildren() {
        return ((Employee)getModel()).getChildrenArray();
    }

    @Override
    protected void createEditPolicies() {
        installEditPolicy(EditPolicy.COMPONENT_ROLE, new
AppDeletePolicy());
    }

    public void refreshVisuals() {
        Employee model = (Employee)getModel();
        setWidgetText(model.getName() + " " + model.getPrenom());

    setWidgetImage(PlatformUI.getWorkbench().getSharedImages().getIma
```

```
ge(
                ISharedImages.IMG_DEF_VIEW));
    }

    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        // TODO Auto-generated method stub
        if(evt.getPropertyName().equals(Node.PROPERTY_ADD))
refreshChildren();
        if(evt.getPropertyName().equals(Node.PROPERTY_REMOVE))
refreshChildren();
    }

}
```

然后是 AppTreeEditPartFactory 工厂，代码如下：

```
package wbhgef.editpart.tree;

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

import wbhgef.model.Employee;
import wbhgef.model.Enterprise;
import wbhgef.model.Service;

public class AppTreeEditPartFactory implements EditPartFactory {

    @Override
    public EditPart createEditPart(EditPart context, Object model) {
        // TODO Auto-generated method stub
        EditPart part = null;

        if(model instanceof Enterprise) {
            part = new EnterpriseTreeEditPart();
        } else if(model instanceof Service) {
            part = new ServiceTreeEditPart();
        } else if(model instanceof Employee) {
            part = new EmployeeTreeEditPart();
        }

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

```
        }

        return part;
    }

}
```

现在在 **MyGraphicalEditor** 类中创建一个嵌套类，用于处理大纲视图。我们将执行其变化，从而使模型和 key handler 可用，代码如下：

```
public class MyGraphicalEditor extends GraphicalEditor {
     (……)
    private Enterprise model;
    private KeyHandler keyHandler;

     (……)
    protected void configureGraphicalViewer() {

         (……)
        KeyHandler keyHandler = new KeyHandler();
         (……)
        keyHandler.put(
                KeyStroke.getPressed(SWT.DEL, 127, 0),

    getActionRegistry().getAction(ActionFactory.DELETE.getId()));
        keyHandler.put(
                KeyStroke.getPressed('+',SWT.KEYPAD_ADD, 0),

    getActionRegistry().getAction(GEFActionConstants.ZOOM_IN));
        keyHandler.put(
                KeyStroke.getPressed('-',SWT.KEYPAD_SUBTRACT, 0),

    getActionRegistry().getAction(GEFActionConstants.ZOOM_OUT));

        viewer.setProperty(
                MouseWheelHandler.KeyGenerator.getKey(SWT.NONE),
                MouseWheelZoomHandler.SINGLETON);

        viewer.setKeyHandler(keyHandler);
    }
     (……)
    @Override
    protected void initializeGraphicalViewer() {
```

```
        // TODO Auto-generated method stub
        GraphicalViewer viewer = getGraphicalViewer();
        model = CreateEnterprise();
        viewer.setContents(CreateEnterprise());

    }
     (……)

    protected class OutlinePage extends ContentOutlinePage {

        private SashForm sash;

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

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

            getViewer().createControl(sash);

            getViewer().setEditDomain(getEditDomain());
            getViewer().setEditPartFactory(new
AppTreeEditPartFactory());
            getViewer().setContents(model);

            getSelectionSynchronizer().addViewer(getViewer());
        }

        public void init(IPageSite pageSite) {
            super.init(pageSite);

            IActionBars bars = getSite().getActionBars();
            bars.setGlobalActionHandler(ActionFactory.UNDO.getId(),

    getActionRegistry().getAction(ActionFactory.UNDO.getId()));
            bars.setGlobalActionHandler(ActionFactory.REDO.getId(),

    getActionRegistry().getAction(ActionFactory.REDO.getId()));
            bars.setGlobalActionHandler(ActionFactory.DELETE.getId(),

    getActionRegistry().getAction(ActionFactory.DELETE.getId()));
```

```
            bars.updateActionBars();

            getViewer().setKeyHandler(keyHandler);
        }

        public Control getControl() {
            return sash;
        }

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

}
```

注意，可能需手动添加如下导入包：

```
import org.eclipse.ui.views.contentoutline.IContentOutlinePage;
```

在 plugin.xml 中，需要添加对 org.eclipse.ui.views 插件的依赖，如下图：

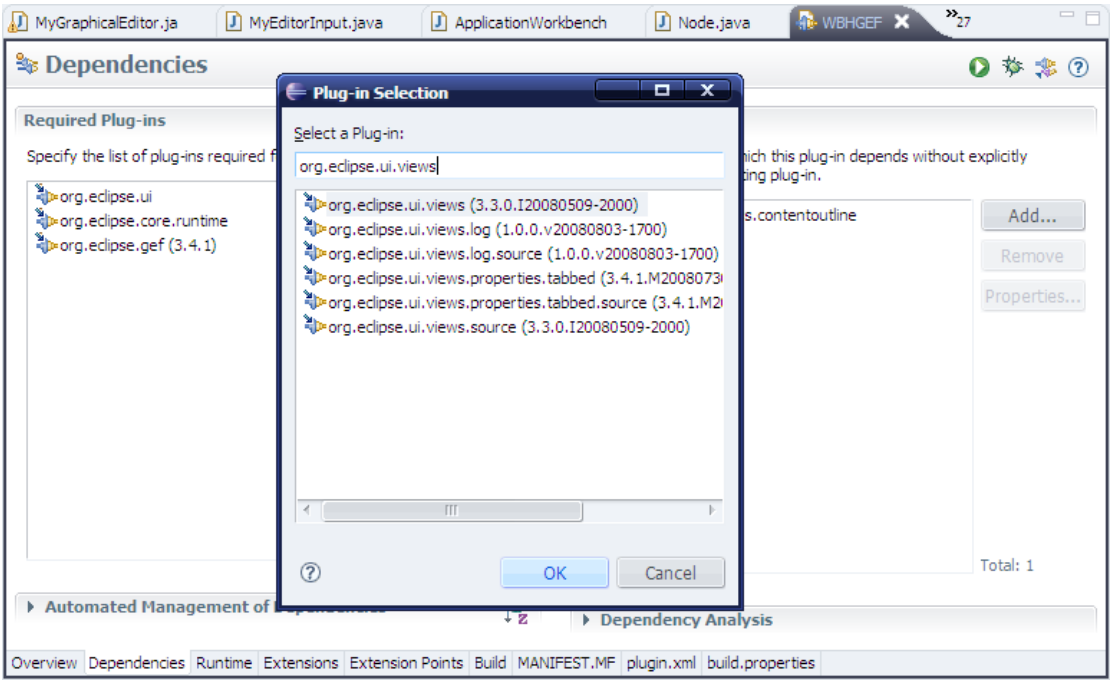


图21 添加对 org.eclipse.ui.views 插件的依赖

然后，嵌套的类在需要时就可以被调用。getAdapter()方法是一个检验，代码如下：

```
public class MyGraphicalEditor extends GraphicalEditor {
```

```
    (……)
    public Object getAdapter(Class type) {
        if(type == ZoomManager.class) {
            return
((ScalableRootEditPart)getGraphicalViewer().getRootEditPart()).getZoo
mManager();
//      } else {
//          return super.getAdapter(type);
        }
        if(type == IContentOutlinePage.class) {
            return new OutlinePage();
        }
        return super.getAdapter(type);
    }
    (……)
}
```

运行效果如下图：

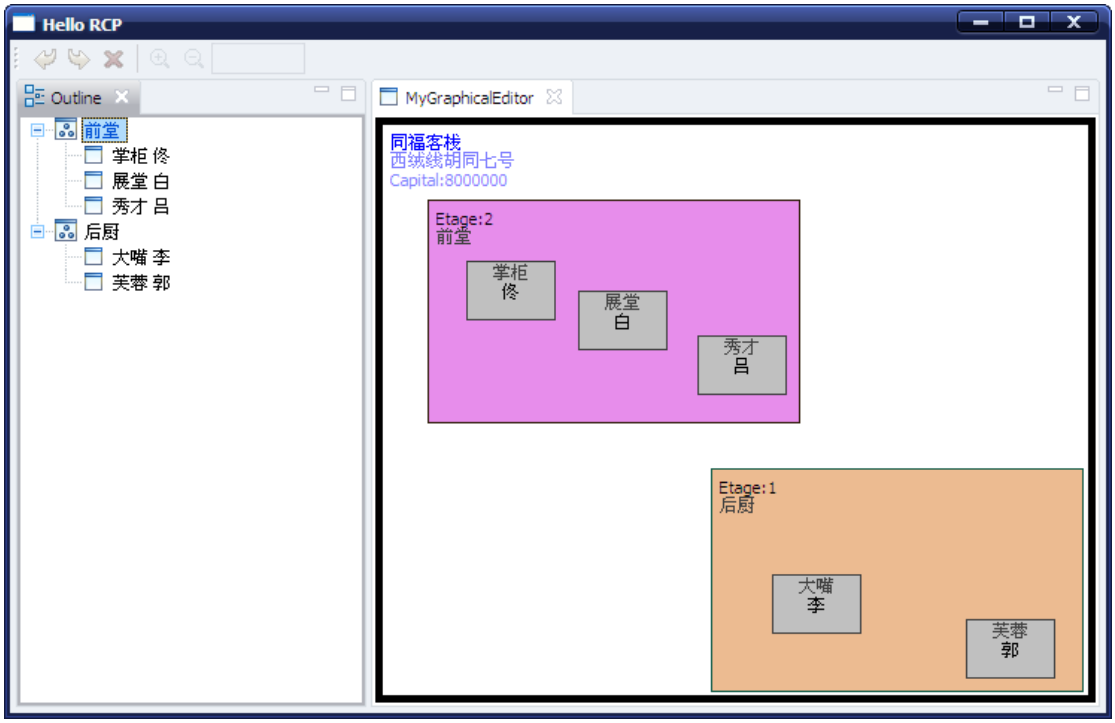


图22 添加树状视图效果

以上阶段代码为 WBHGEF07。

# 7. 缩小（鸟瞰，BirdView）视图

这部分我们将看到如何在大纲视图中添加图形的缩小视图。在使用缩放（zoom）功能时这将十分有用。实现机制依赖于 **MyGraphicalEditor** 类中大纲视图中的嵌套类。代码如下：

```
    protected class OutlinePage extends ContentOutlinePage {
        private ScrollableThumbnail thumbnail;
        private DisposeListener disposeListener;
         (……)
        public void createControl(Composite parent) {
             (……)
            Canvas canvas = new Canvas(sash, SWT.BORDER);
            LightweightSystem lws = new LightweightSystem(canvas);

            thumbnail = new ScrollableThumbnail(

    (Viewport)((ScalableRootEditPart)getGraphicalViewer()
                          .getRootEditPart()).getFigure());

    thumbnail.setSource(((ScalableRootEditPart)getGraphicalViewer()

    .getRootEditPart()).getLayer(LayerConstants.PRINTABLE_LAYERS));

            lws.setContents(thumbnail);

            disposeListener = new DisposeListener() {
                @Override
                public void widgetDisposed(DisposeEvent e) {
                    if(thumbnail != null) {
                        thumbnail.deactivate();
                        thumbnail = null;
                    }
                }
            };

    getGraphicalViewer().getControl().addDisposeListener(disposeListe
ner);
        }

         (……)
```

```
        public void dispose() {
            getSelectionSynchronizer().removeViewer(getViewer());
            if(getGraphicalViewer().getControl() != null
                    && !getGraphicalViewer().getControl().isDisposed())
{

    getGraphicalViewer().getControl().removeDisposeListener(disposeLi
stener);
            }
            super.dispose();
        }
    }
}
```

运行效果如下：

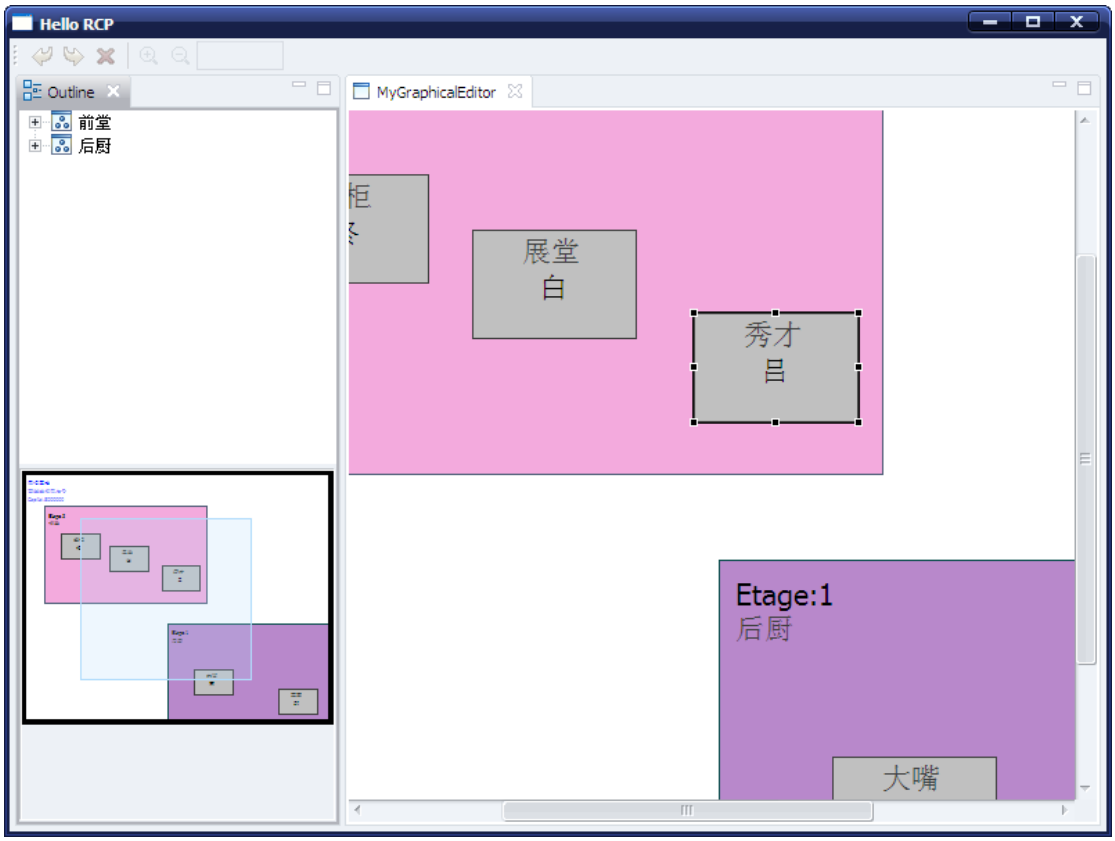


图23 添加鸟瞰视图效果

# 8. 环境菜单（Context Menu）

在这部分指南中，我们添加一个绑定于鼠标右键的环境菜单，首先添加菜单类。

这个类继承自 **ContextMenuProvider**，并且建立的方式大致和 toolbar 类似。

我们为菜单添加 delete 和 undo/redo，代码如下：

```
package wbhgef;

import org.eclipse.gef.ContextMenuProvider;
import org.eclipse.gef.EditPartViewer;
import org.eclipse.gef.ui.actions.ActionRegistry;
import org.eclipse.gef.ui.actions.GEFActionConstants;
import org.eclipse.jface.action.IAction;
import org.eclipse.jface.action.IMenuManager;
import org.eclipse.ui.actions.ActionFactory;

public class AppContextMenuProvider extends ContextMenuProvider {

    private ActionRegistry actionRegestry;

    public AppContextMenuProvider(EditPartViewer viewer, ActionRegistry
registry) {
        super(viewer);
        // TODO Auto-generated constructor stub
        setActionRegistry(registry);
    }

    @Override
    public void buildContextMenu(IMenuManager menu) {
        // TODO Auto-generated method stub
        IAction action;

        GEFActionConstants.addStandardActionGroups(menu);

        action =
getActionRegistry().getAction(ActionFactory.DELETE.getId());
        menu.appendToGroup(GEFActionConstants.GROUP_EDIT, action);

    }
```

```
    private ActionRegistry getActionRegistry() {
        return actionRegestry;
    }

    private void setActionRegistry(ActionRegistry registry) {
        actionRegestry = registry;
    }

}
```

现在将菜单应用在想应用的地方，将其放置于图形和树状视图，代码如下：

```
public class MyGraphicalEditor extends GraphicalEditor {
     (……)
    protected void configureGraphicalViewer() {
         (……)
       ContextMenuProvider provider = new AppContextMenuProvider(viewer,
getActionRegistry());
        viewer.setContextMenu(provider);
    }
     (……)
    protected class OutlinePage extends ContentOutlinePage {
         (……)
        public void init(IPageSite pageSite) {
             (……)
            ContextMenuProvider provider = new
AppContextMenuProvider(getViewer(), getActionRegistry());
            getViewer().setContextMenu(provider);
        }
         (……)
    }
     (……)
}
```

运行效果如下：

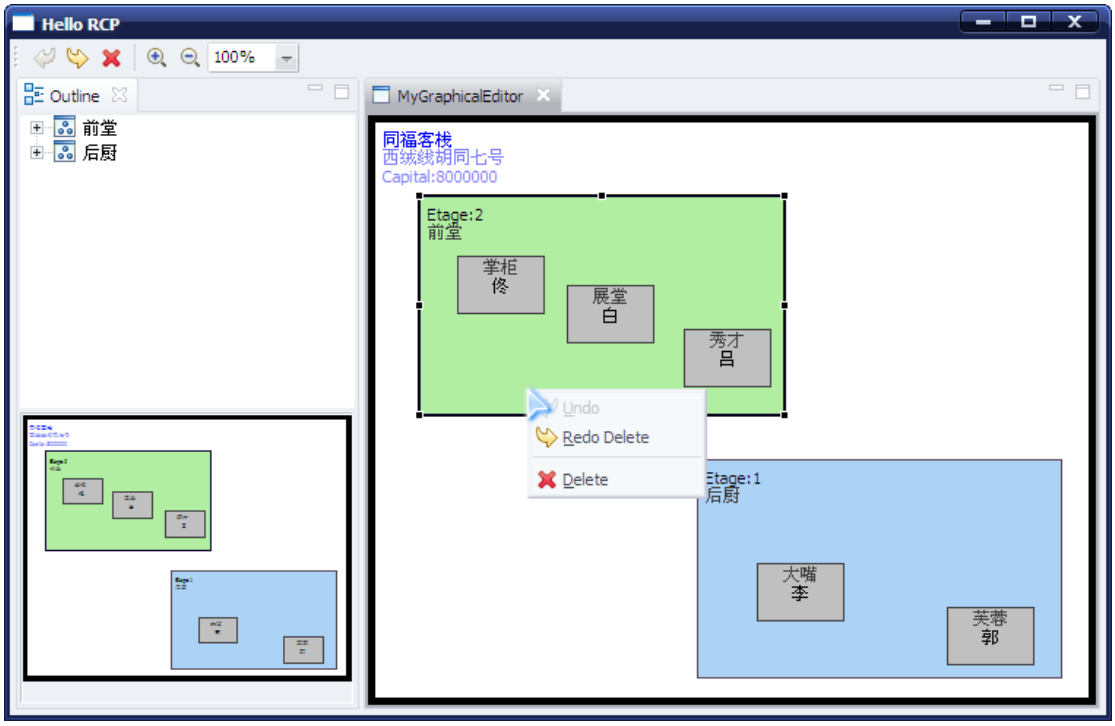


图24 添加环境菜单效果

此阶段代码为 WBHGEF08。

# 9. 创建用户自定义操作（Custom Action）

前面已经看到我们可以为一个 EditPart 添加一些 EditPolicy，但是这仅是使用已经存在的 action，并且由 GEF 处理（如组件的移除）。

我现在创建自己的 action 并将其添加至一个 EditPolicy 中，正如为应用添加环境菜单。比如说允许重命名 Service。

为此，我们将建立如下元素：

- 向导（Wizard），用于向用户请求新名称
- Command，用于执行重命名，并将其整合至 undo/redo 系统
- Action，运行向导并创建 request
- EditPolicy，从重命名 request 生成 command，这将最终由其他编辑操作完成

**创建 Wizard**

这并不是纯粹的 GEF，但对 Eclipse 是极有用的工具，并且很定制化，这就是为何对于向用户请求新名称这类事务组成的简单操作，我们仍要对其创建一个

Wizard。这个 Wizard 将值包含标签，和一个让用户输入名称的文本编辑区域，同时要显示旧名称。这个 Wizard 事实上是 IWizardPage 的一个容器，每个 page 代表向导过程中的一步。我们的 Wizard 值包含一个 page，这个 page 由继承 WizardPage 的类实现。

RenameWizard 代码如下：

```
package wbhgef;

import org.eclipse.jface.wizard.Wizard;
import org.eclipse.jface.wizard.WizardPage;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;

public class RenameWizard extends Wizard {

    private class RenamePage extends WizardPage {
        public Text nameText;

        public RenamePage(String pageName) {
            super(pageName);
            setTitle("Rename");
            setDescription("Rename a component");
        }

        @Override
        public void createControl(Composite parent) {
            // TODO Auto-generated method stub
            Composite composite = new Composite(parent, SWT.NONE);

            Label label = new Label(composite, SWT.NONE);
            label.setText("Rename to:");

            nameText = new Text(composite, SWT.NONE);
            nameText.setText(oldName);

            RowLayout I = new RowLayout();
            composite.setLayout(I);
            setControl(composite);
        }
    }
```

```
    private String oldName;
    private String newName;

    public RenameWizard(String oldName) {
        this.oldName = oldName;
        this.newName = null;

        addPage(new RenamePage("MyRenamePage"));
    }

    @Override
    public boolean performFinish() {
        // TODO Auto-generated method stub
        RenamePage page = (RenamePage)getPage("MyRenamePage");
        if(page.nameText.getText().isEmpty()) {
            page.setErrorMessage("名称不可为空！");
            return false;
        }
        newName = page.nameText.getText();
        return true;
    }

    public String getRenameValue() {
        return newName;
    }
}
```

当用户按 Finish 按钮时会调用 performFinish()方法。如果返回 true，则向导成功完成，否则它仍停滞在等待用户修改错误的状态。

**创建 Command**

现在返回 GEF。我们创建 command 来执行重命名 action。正如对 DeleteCommand 的实现，我们为这个 command 实现 execute()方法和 undo()方法。我们在 command 域中备份新旧名称这样就可以在 undo/redo 过程中恢复它们。

RenameCommand 代码如下：

```
package wbhgef.command;

import org.eclipse.gef.commands.Command;

import wbhgef.model.Node;
```

```
public class RenameCommand extends Command {

    private Node model;
    private String oldName;
    private String newName;

    public void execute() {
        this.oldName = model.getName();
        this.model.setName(newName);
    }

    public void setModel(Object model) {
        this.model = (Node)model;
    }

    public void setNewName(String newName) {
        this.newName = newName;
    }

    public void undo() {
        this.model.setName(oldName);
    }

}
```

**创建 Action**

现在我们可以执行重命名操作并请求用户命名新名称，需要创建 action 来运行 wizard 并创建 command。创建的 action 名称叫做 RenameAction。在它执行期间，它将创建一个具有 rename 类型的 GEF 的 Request（为这个场合创建的新类型），然后我们用它的 extend data（扩展数据，对象到对象间的一个映射）来保存 request 中的新名称（我们选择 newName 作为键，它的值是新名称的字符串）。rename 类型将被我们的 EditPolicy 辨识，后面可以看到。最后，我们调用最先选择的 EditPart 的 getCommand()方法，来请求 GEF 处理由 EditPolicy 产生的 request，并返回生成的 command（由许多 command 链组成，但是此处只有一个 rename command）。

RenameAction 代码如下：

```
package wbhgef.action;

import java.util.HashMap;
import java.util.List;

import org.eclipse.gef.EditPart;
```

```
import org.eclipse.gef.Request;
import org.eclipse.gef.commands.Command;
import org.eclipse.gef.ui.actions.SelectionAction;
import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.jface.wizard.WizardDialog;
import org.eclipse.ui.IWorkbenchPart;
import org.eclipse.ui.actions.ActionFactory;
import org.eclipse.ui.plugin.AbstractUIPlugin;

import wbhgef.RenameWizard;
import wbhgef.model.Node;

public class RenameAction extends SelectionAction {

    public RenameAction(IWorkbenchPart part) {
        super(part);
        setLazyEnablementCalculation(false);
    }

    protected void init() {
        setText("Rename...");
        setToolTipText("Rename");

        setId(ActionFactory.RENAME.getId());

        ImageDescriptor icon =
AbstractUIPlugin.imageDescriptorFromPlugin("WBHGEF",
"icons/rename.gif");
        if(icon != null) {
            setImageDescriptor(icon);
        }
        setEnabled(false);
    }

    @Override
    protected boolean calculateEnabled() {
        // TODO Auto-generated method stub
        Command cmd = createRenameCommand("");
        if(cmd == null) {
            return false;
        }
        return true;
```

```
    }

    public Command createRenameCommand(String name) {
        Request renameReq = new Request("rename");

        HashMap<String, String> reqData = new HashMap<String, String> ();
        reqData.put("newName", name);
        renameReq.setExtendedData(reqData);

        EditPart object = (EditPart)getSelectedObjects().get(0);
        Command cmd = object.getCommand(renameReq);
        return cmd;
    }

    public void run() {
        Node node = getSelectedNode();
        RenameWizard wizard = new RenameWizard(node.getName());
        WizardDialog dialog = new
WizardDialog(getWorkbenchPart().getSite().getShell(), wizard);

        dialog.create();
        dialog.getShell().setSize(400, 320);

        dialog.setTitle("Rename wizard");
        dialog.setMessage("Rename");
        if(dialog.open() == WizardDialog.OK) {
            String name = wizard.getRenameValue();
            execute(createRenameCommand(name));
        }
    }

//  Helper
    private Node getSelectedNode() {
        List objects = getSelectedObjects();
        if(objects.isEmpty()) {
            return null;
        }
        if(!(objects.get(0) instanceof EditPart)) {
            return null;
        }
        EditPart part = (EditPart)objects.get(0);
        return (Node)part.getModel();
```

```
    }

}
```

别忘了当创建 action 时，需要在 **MyGraphicalEditor** 的 ActionRegistry 中注册它，代码如下：

```
    public void createActions() {
        super.createActions();

        ActionRegistry registry = getActionRegistry();
        IAction action = new RenameAction(this);
        registry.registerAction(action);
        getSelectionActions().add(action.getId());
    }
```

然后在它的 Editor 的环境菜单中详述其快捷方式（通过用它的 Eclipse 标识在 AppContextMenuProvider 中检索全局 action），代码如下：

```
    public void buildContextMenu(IMenuManager menu) {
        // TODO Auto-generated method stub
        IAction action;
         (……)

        action =
getActionRegistry().getAction(ActionFactory.RENAME.getId());
        menu.appendToGroup(GEFActionConstants.GROUP_EDIT, action);

    }
```

然后在应用程序的全局菜单（即 MyGraphicalEditorActionBarContributor）中，添加如下代码：

```
    public void contributeToMenu(IMenuManager menuManager) {

    }
```

**创建 EditPolicy**

我们的由执行 rename 处理的 command，以及可以创建 request 去在 EditPolicy 中检索 command 的 action。剩下的就是添加 EditPolicy 用于理解新的 request，并创建重命名的 command。我们创建一个新的 EditPolicy 命名为 AppRenamePolicy，它将直接源于 AbstractEditPolicy，因为我们将定义自己的 getCommand()方法。

AppRenamePolicy 代码如下：

```
package wbhgef.editpolicy;
```

```
import org.eclipse.gef.Request;
import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.AbstractEditPolicy;

import wbhgef.command.RenameCommand;

public class AppRenamePolicy extends AbstractEditPolicy {

    public Command getCommand(Request request) {
        if(request.getType().equals("rename")) {
            return createRenameCommand(request);
        }
        return null;
    }

    protected Command createRenameCommand(Request renameRequest) {
        RenameCommand command = new RenameCommand();
        command.setModel(getHost().getModel());

    command.setNewName((String)renameRequest.getExtendedData().get("n
ewName"));
        return command;
    }

}
```

这个类让我们想起了 **org.eclipse.gef.editpolicies.ComponentEditPolicy** 的创建一个重命名 command 的方法：验证 request 类型，然后它建立相应的 command（使用 extended data 来读取新名称）。

**将新的 EditPolicy 与 EditPart 关联**

我们已经详述 EditPart 使其有特权去重命名。现在将这个新的 EditPolicy 安装与 ServicePart 和 ServiceTreePart（在相应的文件中添加一行来提供图形和树图中的重命名服务）。代码如下：

```
    protected void createEditPolicies() {
        // TODO Auto-generated method stub
         (……)
        installEditPolicy(EditPolicy.NODE_ROLE, new AppRenamePolicy());
    }
```

**最后，激活属性来更新视图**

重命名已经可用，但视图并不知道它们需要更新自己了。为此，我们需要在

Node 中创建一个新属性来触发随时的新名称变化，并且相关的 EditPart 的 propertyChange()方法将监听新属性的变化并自动更新它们自己。

Node 修改代码如下：

```
    public static final String PROPERTY_RENAME = "NodeRename";
     (……)
    public void setName(String name) {
        String oldName = this.name;
        this.name = name;
        getListeners().firePropertyChange(PROPERTY_RENAME, oldName,
this.name);
    }
```

在 ServicePart 和 ServiceTreePart 中修改代码：

```
    public void propertyChange(PropertyChangeEvent evt) {
         (……)
        if(evt.getPropertyName().equals(Node.PROPERTY_RENAME))
refreshVisuals();
    }
```

运行效果如下：

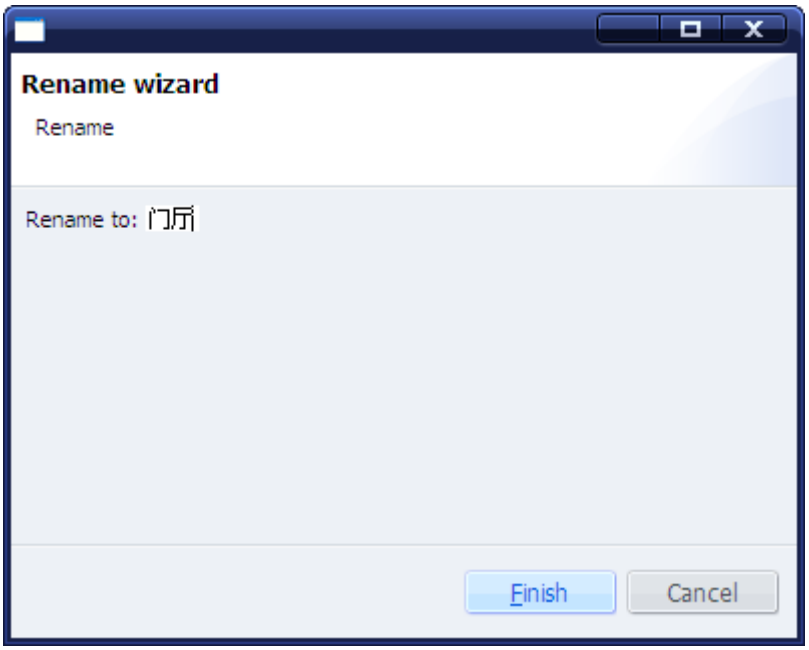


图25 修改 Service 名称

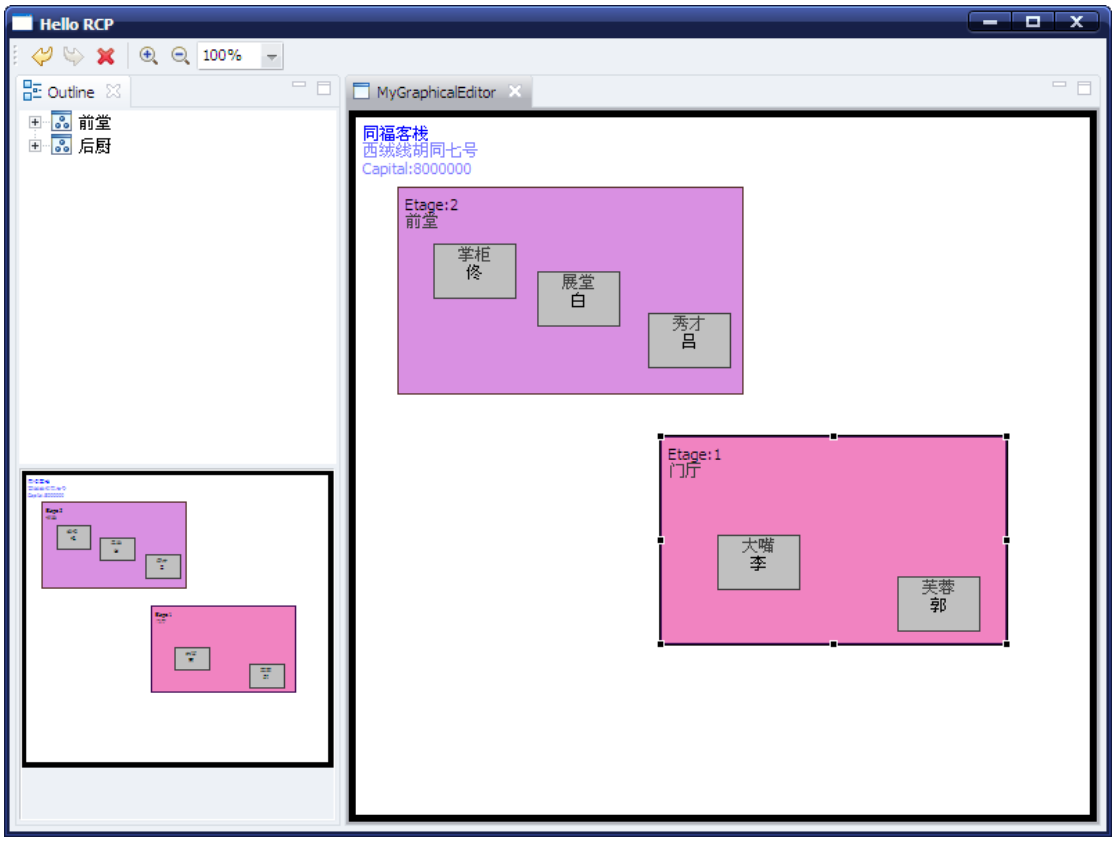


图26 Service 名称修改完成

此阶段代码为 WBHGEF09。

# 10. 属性页（Property Sheet）

在这节指南中，我们将尝试实现属性窗口，来展现并直接修改模型属性，并令之前指南中的修改方法废弃。

部分代码创建时过分讲究但确实很简明并有逻辑。

**在模型中添加 Service 的颜色**

首先，在进入主题之前，我们先添加一个功能，当选中一个 Service 的时候，修改其为随机颜色。

在先前版本的指南中，Service 图形的颜色选择是随机的，即是说我们无法控制它。我们在模型域中保存 Service 的颜色，并且创建连接器读取并修改它，然后当发生修改时触发颜色属性变化来通知视图它们需要更新了。

在 Service 的图形中，修改构造器并不决定背景颜色，以及边界颜色到黑色。

ServiceFigure 代码如下：

```
    public ServiceFigure() {
        XYLayout layout = new XYLayout();
        setLayoutManager(layout);

        Service service = new Service();

        labelName.setForegroundColor(ColorConstants.darkGray);
        add(labelName, ToolbarLayout.ALIGN_CENTER);
        setConstraint(labelName, new Rectangle(5,17,-1,-1));

        labelEtage.setForegroundColor(ColorConstants.black);
        add(labelEtage, ToolbarLayout.ALIGN_CENTER);
        setConstraint(labelEtage, new Rectangle(5,5,-1,-1));

//      setForegroundColor(new Color(null,
//              (new Double(Math.random()*128)).intValue(),
//              (new Double(Math.random()*128)).intValue(),
//              (new Double(Math.random()*128)).intValue()));
//      setBackgroundColor(new Color(null,
//              (new Double(Math.random()*128)).intValue() + 128,
//              (new Double(Math.random()*128)).intValue() + 128,
//              (new Double(Math.random()*128)).intValue() + 128));

        setForegroundColor(ColorConstants.black);

        setBackgroundColor(service.getColor());

        setBorder(new LineBorder(1));
        setOpaque(true);
    }
```

在 Service 的模型中，添加颜色属性，仍在开始时初始化为随机颜色，代码如下：

```
public class Service extends Node {

    (……)
    private Color color;

    (……)
    private Color createRandomColor() {
        return new Color(null,
                (new Double(Math.random() * 128)).intValue() + 128,
```

```
                (new Double(Math.random() * 128)).intValue() + 128,
                (new Double(Math.random() * 128)).intValue() + 128);
    }

    public Service() {
        this.color = createRandomColor();
    }

    public Color getColor() {
        return color;
    }

    public void setColor(Color color) {
        Color oldColor = this.color;
        this.color = color;

        getListeners().firePropertyChange(PROPERTY_COLOR, oldColor,
color);
    }
}
```

**定义一个属性源**

Eclipse 平台使用标准属性页来自动反馈选择的变化，并更新所选对象属性的显示。我们的选择已经直接由模型对象展现出来，那么我们将把它们定义为属性源。

属性源可以用两种方式定义：

- 可以直接由模型对象中的 IPropertySource 接口实现
- 或由模型对象中的 IAdaptable 接口实现，并返回一个在 getAdapter()方法中实现 IPropertySource 的对象，此时 IPropertySource 类型作为方法的参数传递。这个技术具有使模型和属性源分离得更好的优点，而且仅需要在 Node 的中实现的一次，所有源于它的对象，即继承 Node 的类将有这些属性（即，Enterprise、Service 和 Employee）。

Node 的代码修改如下：

```
public class Node implements IAdaptable{

    (……)
    private IPropertySource propertySource = null;
    (……)
    @Override
    public Object getAdapter(Class adapter) {
```

```
        // TODO Auto-generated method stub
        if(adapter == IPropertySource.class) {
            if(propertySource == null) {
                propertySource = new NodePropertySource(this);
            }
            return propertySource;
        }
        return null;
    }

}
```

这个代码无法编译，因为 NodePropertySource 丢失，现在来创建它，并实现 IPropertySource。我们将它的描述剪切为几步来对每个方法详细阐述。在此之前，我们先创建唯一的静态域的属性标识符，在模型中进行定义。

在 Service 中定义的静态域如下：

```
public static final String PROPERTY_COLOR = "ServiceColor";
public static final String PROPERTY_FLOOR = "ServiceFloor";
```

在 Enterprise 中定义如下：

```
public static final String PROPERTY_CAPITAL = "EnterpriseCapital";
```

在 Employee 中定义如下：

```
public static final String PROPERTY_FIRSTNAME = "EmployeePrenom";
```

通过这个方法，在不同 EditPart 中更新属性监听者，这样视图可以在任何可编辑属性法身个变化时自己更新。

在 EnterprisePart 中，我们添加 PROPERTY_RENAME 和 PROPERTY_CAPITAL 的处理，代码如下：

```
    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        // TODO Auto-generated method stub
         (……)
        if(evt.getPropertyName().equals(Node.PROPERTY_RENAME))
refreshVisuals();
        if(evt.getPropertyName().equals(Enterprise.PROPERTY_CAPITAL))
refreshVisuals();
    }
```

在所有 EditPart 和 TreeEditPart 中做类似修改。

修改 EnterpriseTreeEditPart 代码如下：

```
@Override
public void propertyChange(PropertyChangeEvent evt) {
    // TODO Auto-generated method stub
     (……)
    if(evt.getPropertyName().equals(Node.PROPERTY_RENAME))
refreshVisuals();
    if(evt.getPropertyName().equals(Enterprise.PROPERTY_CAPITAL))
refreshVisuals();
}
```

ServicePart 代码修改如下：

```
@Override
public void propertyChange(PropertyChangeEvent evt) {
    // TODO Auto-generated method stub
     (……)
    if(evt.getPropertyName().equals(Service.PROPERTY_COLOR))
refreshVisuals();
    if(evt.getPropertyName().equals(Service.PROPERTY_FLOOR))
refreshVisuals();
}
```

ServiceTreeEditPart 代码修改如下：

```
@Override
public void propertyChange(PropertyChangeEvent evt) {
    // TODO Auto-generated method stub
     (……)
    if(evt.getPropertyName().equals(Service.PROPERTY_COLOR))
refreshVisuals();
    if(evt.getPropertyName().equals(Service.PROPERTY_FLOOR))
refreshVisuals();
}
```

EmployeePart 代码修改如下：

```
@Override
public void propertyChange(PropertyChangeEvent evt) {
    // TODO Auto-generated method stub
     (……)
    if(evt.getPropertyName().equals(Employee.PROPERTY_FIRSTNAME))
refreshVisuals();
}
```

EmployeeTreeEditPart 代码修改如下：

```
    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        // TODO Auto-generated method stub
         (……)
        if(evt.getPropertyName().equals(Employee.PROPERTY_FIRSTNAME))
refreshVisuals();
    }
```

现在返回 NodePropertySource 类。

首先，储存模型对象从而在后面引用它。

NodePropertySource 类代码如下：

```
public class NodePropertySource implements IPropertySource {

    private Node node;

    public NodePropertySource(Node node) {
        this.node = node;
    }

    @Override
    public Object getEditableValue() {
        // TODO Auto-generated method stub
        return null;
    }
     (……)
}
```

在这个指南中，我们选择在一个单一类中实现所有模型属性，因此对一个属性是否可用的不同测试，也对其它模型元素进行了验证，但确实可能对每个模型对象创建不同类并重复实现这些方法，或者在所有拥有属性的模型中直接实现 IPropertySource。

下面的方法返回 IPropertyDescriptor，每个代表模型的一个属性。如果属性是只读的，那么一个简单的 PropertyDescriptor 就足够了；派生类的角色给属性页返回一个适合编辑这个单元的 CellEditor。就像我们不想编辑 employee 的姓名时，我们返回一个 PropertyDescriptor。

继续编辑 NodePropertySource 代码如下：

```
    @Override
    public Object getEditableValue() {
        // TODO Auto-generated method stub
        return null;
```

```
    }

    @Override
    public IPropertyDescriptor[] getPropertyDescriptors() {
        // TODO Auto-generated method stub
        ArrayList<IPropertyDescriptor> properties = new
ArrayList<IPropertyDescriptor>();

        if(node instanceof Employee) {
            properties.add(new PropertyDescriptor(Node.PROPERTY_RENAME,
"Name"));
        } else {
            properties.add(new
TextPropertyDescriptor(Node.PROPERTY_RENAME, "Name"));
        }

        if(node instanceof Service) {
            properties.add(new
ColorPropertyDescriptor(Service.PROPERTY_COLOR, "Color"));
            properties.add(new
TextPropertyDescriptor(Service.PROPERTY_FLOOR, "Etage"));
        } else if (node instanceof Enterprise) {
            properties.add(new
TextPropertyDescriptor(Enterprise.PROPERTY_CAPITAL, "Capital"));
        } else if (node instanceof Employee) {
            properties.add(new
PropertyDescriptor(Employee.PROPERTY_FIRSTNAME, "Prenom"));
        }
        return properties.toArray(new IPropertyDescriptor[0]);
    }
```

重获取属性值依赖于它的标识符，代码如下：

```
    @Override
    public Object getPropertyValue(Object id) {
        // TODO Auto-generated method stub
        if(id.equals(Node.PROPERTY_RENAME)) {
            return node.getName();
        }
        if(id.equals(Service.PROPERTY_COLOR)) {
            return ((Service)node).getColor().getRGB();
        }
        if(id.equals(Service.PROPERTY_FLOOR)) {
```

```
            return Integer.toString(((Service)node).getEtage());
        }
        if(id.equals(Enterprise.PROPERTY_CAPITAL)) {
            return Integer.toString(((Enterprise)node).getCapital());
        }
        if(id.equals(Employee.PROPERTY_FIRSTNAME)) {
            return (((Employee)node).getPrenom());
        }
        return null;
    }

    @Override
    public boolean isPropertySet(Object id) {
        // TODO Auto-generated method stub
        return false;
    }

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

    }
```

在编辑后记录属性值，代码如下：

```
    @Override
    public void setPropertyValue(Object id, Object value) {
        // TODO Auto-generated method stub
        if(id.equals(Node.PROPERTY_RENAME)) {
            node.setName((String)value);
        } else if(id.equals(Service.PROPERTY_COLOR)) {
            Color newColor = new Color(null, (RGB)value);
            ((Service)node).setColor(newColor);
        } else if(id.equals(Service.PROPERTY_FLOOR)) {
            try {
                Integer floor = Integer.parseInt((String)value);
            } catch (NumberFormatException e) {}
        } else if(id.equals(Enterprise.PROPERTY_CAPITAL)) {
            try {
                Integer capital = Integer.parseInt((String)value);
                ((Enterprise)node).setCapital(capital);
            } catch (NumberFormatException e) {}
        }
```

```
    }
```

**属性页的整合**

属性源已经准备给出模型对象的属性。但仍未将它整合至 Eclipse 透视图的属性页。

事实上用两步进行：首先，我们为这个属性页定义一个占位符，然后当用户双击 GEF 对象时，申请显示它。

下面详述窗口的放置：在默认透视图定义，即 perspective.java 中修改：

```
package wbhgef;

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

public class Perspective implements IPerspectiveFactory {

   private static final String ID_TABS_FOLDER = "PropertySheet";

   public void createInitialLayout(IPageLayout layout) {
       String editorArea = layout.getEditorArea();
       layout.setEditorAreaVisible(true);
//     layout.addStandaloneView(IPageLayout.ID_OUTLINE, true,
IPageLayout.LEFT, 0.3f, editorArea);
       IFolderLayout tabs = layout.createFolder(ID_TABS_FOLDER,
IPageLayout.LEFT, 0.3f, editorArea);
       tabs.addView(IPageLayout.ID_OUTLINE);
       tabs.addPlaceholder(IPageLayout.ID_PROP_SHEET);
   }
}
```

在先前版本的指南中，大纲直接附加于主窗口，现在我们直接用 IFolderLayout（tab 表）附加于这两个窗口，大纲仍在 startup 中显示。

现在我们将双击操作关联到 EditPart 来打开焦点的属性页，在这部分指南我们又回到 GEF。

对 GraphicalEditPart 双击事实上生成了一个 REQ_OPEN 类型的 Request。这个请求没有转化为 EditPolicy，而是被 EditPart 中的 performRequest()方法处理。

AppAbstractEditPart（处理图形元素的双击）代码如下：

```
   public void performRequest(Request req) {
       if(req.getType().equals(RequestConstants.REQ_OPEN)) {
           try {
               IWorkbenchPage page =
```

```
    PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePag
e();
                page.showView(IPageLayout.ID_PROP_SHEET);
            } catch (PartInitException e) {
                e.printStackTrace();
            }
        }
    }
```

这对继承自 AbstractGraphicalEditPart 的 EditPart 已经足够，因为它返回一个 DragTracker（拖拽跟踪器）来处理 getDragTracker()方法的选项。因为这个方法在 AbstractTreeEditPart（树状视图使用）返回空，我们需要更进一步实现它来处理树图中的双击事件。

在 AppAbstractTreeEditPart 代码中修改如下：

```
    public DragTracker getDragTracker(Request req) {
        return new SelectEditPartTracker(this);
    }

    public void performRequest(Request req) {
        if(req.getType().equals(RequestConstants.REQ_OPEN)) {
            try {
                IWorkbenchPage page =

    PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePag
e();
                page.showView(IPageLayout.ID_PROP_SHEET);
            } catch(PartInitException e) {
                e.printStackTrace();
            }
        }
    }
```

运行效果如下图：

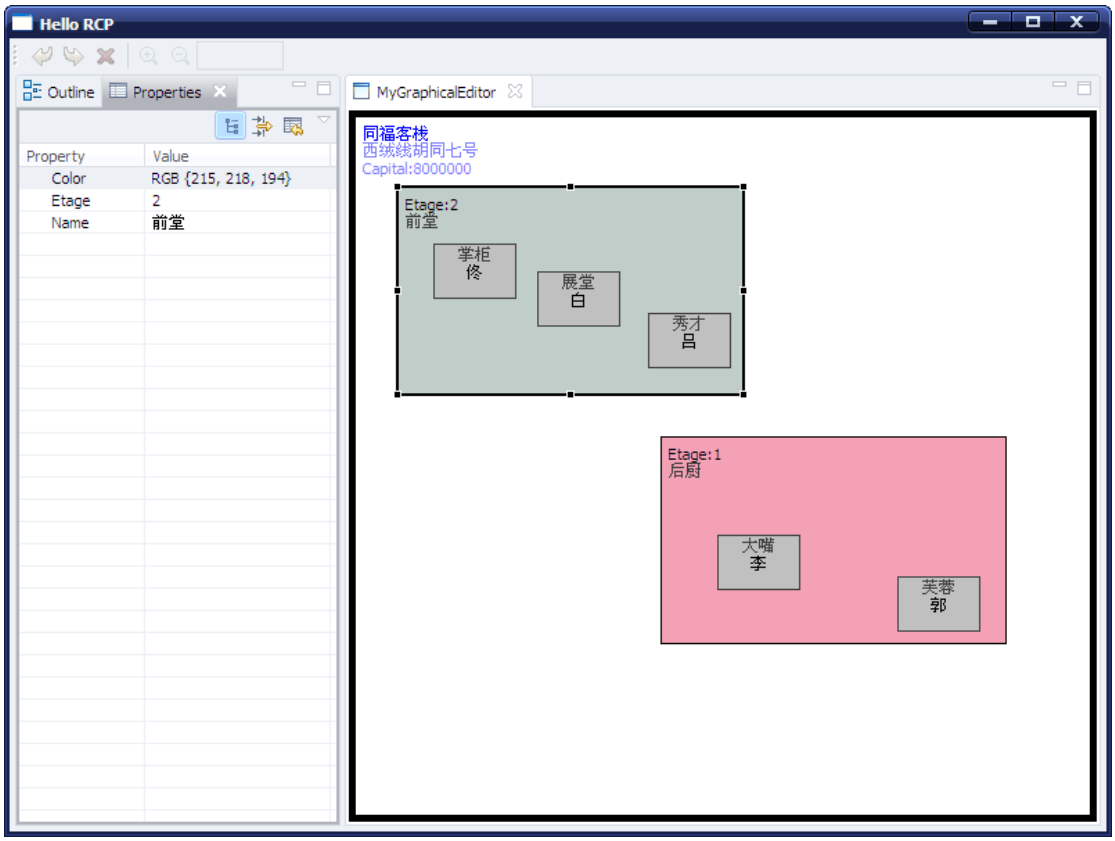


图27 具有属性页并可将属性修改传递给图形的效果

注：此阶段代码还有部分需完善，内容如下：

1．Property 视图在程序启动时并未自动打开，需双击图形视图或大纲视图中某一对象才打开；在 Perspective 的 Design 下有 Property 页显示，不知为何。

2．Service 相对完善，其中：

Name 可在属性页中修改，并直接反馈于图形及大纲；

Color 及 Etage 可在属性页中修改，但图形及大纲无反馈；

3．Employee 的姓和名皆不可修改；

4．Enterprise：

Capital 无反馈，Name 有；

在 Name 修改激活后，Capital 也改变；

Address 在属性页中无。

此阶段代码为 WBHGEF10。

# 11. 添加新的图形元素

在下部分指南中，我们将集中处理图形排版和元素添加。在这一部分，我们将创建 palette，添加一些工具来做基本编辑并插入 service 和 employee。

**插入 Palette**

首先要做的显然是为 editor 添加 palette，为此，我们敬爱那个修改 editor（MyGraphicalEditor）所继承的类：将其从 GraphicalEditor 转换为 GraphicalEditorWithPalette（对更需要互动的 Palette 甚至可以是继承 GraphicalEditorWithFlyoutPalette）。

修改 MyGraphicalEditor 代码如下：

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette {
（……）
}
```

新继承的类需要我们实现 getPaletteRoot()方法。这个方法必须返回一个 PaletteRoot 类型的对象，包含 palette 树。它通常由几个 Palette 组构成，这些组可以分解图形化地分解（Palette 分解器），每个组包含整个不同的类型。在开始之前，我们将添加最基础的调色板工具，选择工具和框选工具。

MyGraphicalEditor 的 getPaletteRoot()方法代码如下：

```
    @Override
    protected PaletteRoot getPaletteRoot() {
        // TODO Auto-generated method stub
        PaletteRoot root = new PaletteRoot();

        PaletteGroup manipGroup = new PaletteGroup("编辑对象工具");
        root.add(manipGroup);

        SelectionToolEntry selectionToolEntry = new
SelectionToolEntry();
        manipGroup.add(selectionToolEntry);
        manipGroup.add(new MarqueeToolEntry());

        root.setDefaultEntry(selectionToolEntry);
        return root;
    }
```

运行效果如图，已可使用框选工具。

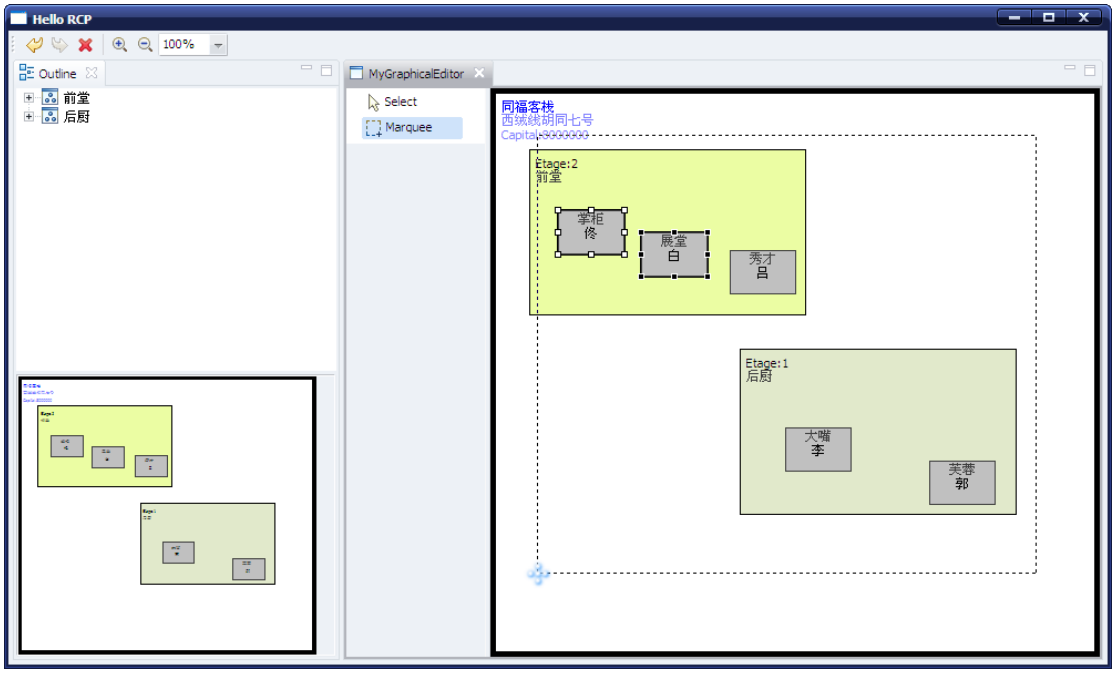


图28 添加 Palette 并可使用框选工具

框选工具用于选择属于用户定义区域的多个元素。很显然这对于不只用 ctrl 键而大批量的移动、删除实体或和其他实体组（此处即 service 或 employee）交互很有用。

### 添加一个 service

现在已有 palette，为它填充一些东西更好，如添加创建一个 service 的可能。为此，我们将首先创建一个继承自 CreationFactory 的类。工厂类用于创建依赖于环境新的对象实体。后面我们可以见到，用这个类树的好处是，调用方法是透明的并且由 GEF 内部处理的。所以让我们创建工厂，命名为 NodeCreationFactory（一个类属的一般名称，因为后面我们将修改它处理创建 employee），继承自 CreationFactory。

代码如下：

```
package wbhgef.command;

import org.eclipse.gef.requests.CreationFactory;

import wbhgef.model.Service;

public class NodeCreationFactory implements CreationFactory {

    private Class<?> template;

    public NodeCreationFactory(Class<?> t) {
        this.template = t;
```

```
    }

    @Override
    public Object getNewObject() {
        // TODO Auto-generated method stub
        if(template == null) {
            return null;
        }
        if(template == Service.class) {
            Service srv = new Service();
            srv.setEtage(42);
            srv.setName("客房");
            return srv;
        }
        return null;
    }

    @Override
    public Object getObjectType() {
        // TODO Auto-generated method stub
        return template;
    }

}
```

易见，我们可以改进新对象的生成，尤其是关于楼层的名称和详述（我们更关心概念而不是结果）。现在我们的类甚至可以从删除中生成对象，我们将创建使用 CreationFactory 的指令。这个指令需要记住有用的信息（如，新创建的 service 以及它属于的 Enterprise，并添加进去），当然还有对 undo/redo 的整合（在命令栈中）。这个命令（叫做 ServiceCreateCommand）继承自 Command。

ServiceCreateCommand 代码如下：

```
package wbhgef.command;

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

import wbhgef.model.Enterprise;
import wbhgef.model.Service;

public class ServiceCreateCommand extends Command {
    private Enterprise en;
    private Service srv;
```

```
public ServiceCreateCommand() {
    super();
    en = null;
    srv = null;
}

public void setService(Object s) {
    if(s instanceof Service) {
        this.srv = (Service)s;
    }
}

public void setEnterprise(Object e ) {
    if(e instanceof Enterprise) {
        this.en = (Enterprise)e;
    }
}

public void setLayout(Rectangle r) {
    if(srv == null) {
        return ;
    }
    srv.setLayout(r);
}

@Override
public boolean canExecute() {
    if(srv == null || en == null) {
        return false;
    }
    return true;
}

@Override
public void execute() {
    en.addChild(srv);
}

@Override
public boolean canUndo() {
    if(en == null || srv == null) {
```

```
            return false;
        }
        return en.contains(srv);
    }

    @Override
    public void undo() {
        en.removeChild(srv);
    }

}
```

其次，使其感觉更加完备，比如，只有当 Enterprise 的资产大于 10000 时，才允许在其中加入新的部门（当然只是例子）。现在只用这些零星代码步伐编译，需要在 Node 类中加入方法检查节点在其父模型中存在。

Node 类代码修改如下：

```
public class Node implements IAdaptable{

    (……)
    public boolean contains(Node child) {
        return children.contains(child);
    }
}
```

如你所知，在 GEF 中，没有 EditPolicy 的话，Command 的什么也不是。现在关注的是，在 AppEditLayoutPolicy 类里，在我们关注的 EditPolicy 中整合 Command 的生成。

AppEditLayoutPolicy 代码修改如下：

```
public class AppEditLayoutPolicy extends XYLayoutEditPolicy {

    (……)

    @Override
    protected Command getCreateCommand(CreateRequest request) {
        // TODO Auto-generated method stub
        if(request.getType() == REQ_CREATE && getHost() instanceof
EnterprisePart) {
            ServiceCreateCommand cmd = new ServiceCreateCommand();
            cmd.setEnterprise(getHost().getModel());
            cmd.setService(request.getNewObject());

            Rectangle constraint = (Rectangle)getConstraintFor(request);
```

```
            constraint.x = (constraint.x < 0) ? 0 : constraint.x;
            constraint.y = (constraint.y < 0) ? 0 : constraint.y;
            constraint.width = (constraint.width <= 0) ?
                    ServiceFigure.SERVICE_FIGURE_DEFWIDTH :
constraint.width;
            constraint.height = (constraint.height <= 0) ?
                    ServiceFigure.SERVICE_FIGURE_DEFHEIGHT :
constraint.height;
            cmd.setLayout(constraint);

            return cmd;
        }
        return null;
    }

}
```

此处，如果 Request 类型是 Creation（因为开始在 getCreationCommand()中创建，测试不是很必须）并且如果 EditPart 是 EnterprisePart 时，我们创建 command，将其填充有用的信息。另外，最需要关注两个尺寸常量需要在 ServiceFigure 中添加（相关的 service 以及它们的显示，所以需要加在这里）。

ServiceFigure 代码修改如下：

```
public class ServiceFigure extends Figure {

    public static final int SERVICE_FIGURE_DEFWIDTH = 250;
    public static final int SERVICE_FIGURE_DEFHEIGHT = 150;
     (……)
}
```

现在我们不要检验刚修改的由 EditPart 使用的 EditPolicy，在此我们想要可以放置一个新元素。确实需要关注 EnterprisePart，我们不允许创建的 Service 覆盖住其他的 Service（至少精确地讲，左上角应是另一个 Service 的空间）。本例中已经完成，需要做的是为 palette 添加入口。

在 MyGraphicalEditor 中修改代码如下：

```
    @Override
    protected PaletteRoot getPaletteRoot() {
        // TODO Auto-generated method stub
         (……)
        PaletteSeparator sep2 = new PaletteSeparator();
        root.add(sep2);

        PaletteGroup instGroup = new PaletteGroup("创建元素工具");
```

```
        root.add(instGroup);

        instGroup.add(new CreationToolEntry("Service", "创建一个service
类",
                new NodeCreationFactory(Service.class), null, null));
        (……)
    }
```

这样我们便在 palette 中定义了一个新的入口（将其放置于新的 palette 组），以如下顺序给出：名称、描述、用于生成对象的工厂实例，然后是图像描述来关联大、小各自的图标。现在来测试一下，运行效果如下：

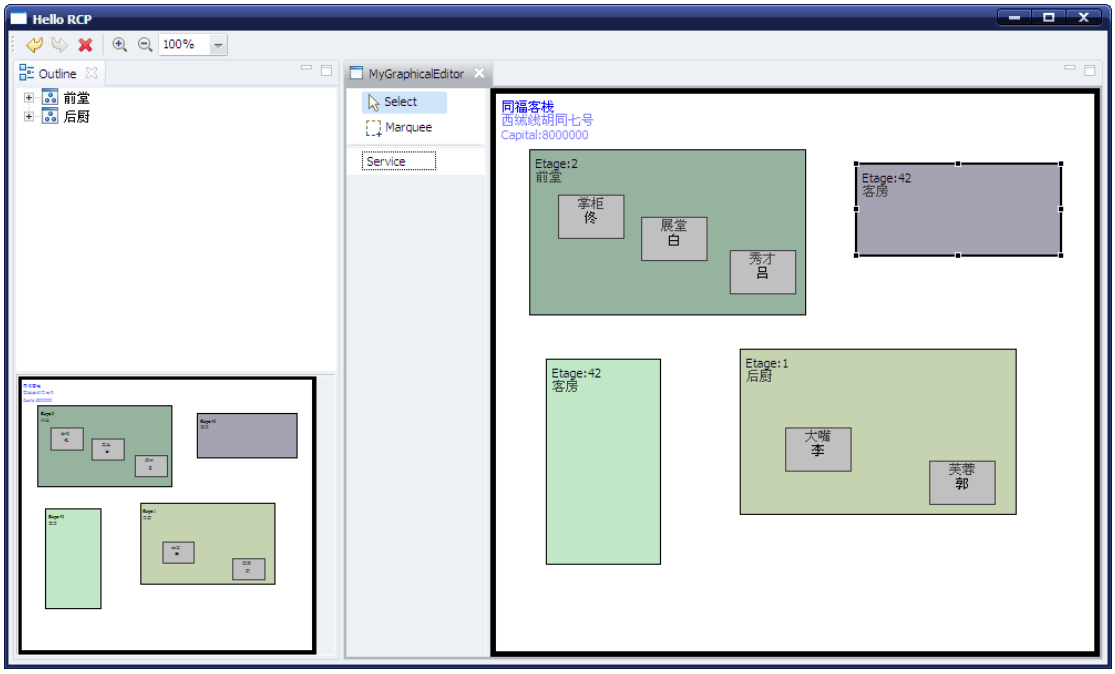


图29 新建 Service

**在 Service 中添加一个 Employee**

现在可以在 Enterprise 中添加一个新的 Service，需要在 Service 中能够添加 Employee。全面地重复前面的工作。

首先修改 CreationFactory 这样它便可以产生 Employee 对象。NodeCreationFactory 代码修改如下：

```
    @Override
    public Object getNewObject() {
        // TODO Auto-generated method stub
        (……)
        else if (template == Employee.class) {
            Employee emp = new Employee();
            emp.setPrenom("祝");
```

```
            emp.setName("无双");
            return emp;
        }
        return null;
    }
```

现在定义一个新 Command，叫做 EmployeeCreateCommand，相当于在特定的 Service 中添加一个 Employee，代码如下：

```
package wbhgef.command;

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

import wbhgef.model.Employee;
import wbhgef.model.Service;

public class EmployeeCreateCommand extends Command {

    private Service srv;
    private Employee emp;

    public EmployeeCreateCommand() {
        super();
        srv = null;
        emp = null;
    }

    public void setService(Object s) {
        if(s instanceof Service) {
            this.srv = (Service)s;
        }
    }

    public void setEmployee(Object e) {
        if(e instanceof Employee) {
            this.emp = (Employee)e;
        }
    }

    public void setLayout(Rectangle r) {
        if(emp == null) return;
```

```
        emp.setLayout(r);
    }

    @Override
    public boolean canExecute() {
        if(srv == null || emp == null)
            return false;
        return true;
    }

    @Override
    public void execute() {
        srv.addChild(emp);
    }

    @Override
    public boolean canUndo() {
        if(srv == null || emp == null)
            return false;
        return srv.contains(emp);
    }

    @Override
    public void undo() {
        srv.removeChild(emp);
    }

}
```

这个概念没有发生太多变化，我们记住有用的信息来实现 undo/redo 命令，并在特定的 Service 中添加 Employee。在下面这个文本修改后，将建立新的 Command。AppEditLayoutPolicy 代码修改如下：

```
public class AppEditLayoutPolicy extends XYLayoutEditPolicy {

……

    @Override
    protected Command getCreateCommand(CreateRequest request) {
        // TODO Auto-generated method stub
        if(request.getType() == REQ_CREATE && getHost() instanceof
EnterprisePart) {
            ……
```

```
        }

        else if (request.getType() == REQ_CREATE && getHost() instanceof
ServicePart) {
            EmployeeCreateCommand cmd = new EmployeeCreateCommand();

            cmd.setService(getHost().getModel());
            cmd.setEmployee(request.getNewObject());

            Rectangle constraint = (Rectangle)getConstraintFor(request);
            constraint.x = (constraint.x < 0) ? 0 : constraint.x;
            constraint.y = (constraint.y < 0) ? 0 : constraint.y;
            constraint.width = (constraint.width <=0 ) ?
                    EmployeeFigure.EMPLOYE_FIGURE_DEFWIDTH :
constraint.width;
            constraint.height = (constraint.height <=0 ) ?
                    EmployeeFigure.EMPLOYE_FIGURE_DEFHEIGHT :
constraint.height;
            cmd.setLayout(constraint);
            return cmd;
        }

        return null;
    }

}
```

再次，我们在 EmployeeFigure 中设置默认值使其整体同质化，我们还在创建图形时使用它们。在 EmployeeFigure 类中设置新建 Employee 时默认图形大小，修改代码如下：

```
public class EmployeeFigure extends Figure {
    ……

    @Override
    protected PaletteRoot getPaletteRoot() {
        // TODO Auto-generated method stub
        ……
        PaletteGroup instGroup = new PaletteGroup("创建元素工具");
        root.add(instGroup);

//      instGroup.add(new CreationToolEntry("Service", "创建一个service
类",
//              new NodeCreationFactory(Service.class), null, null));
```

```
        instGroup.add(new CreationToolEntry("Service", "创建一个service
类",
                new NodeCreationFactory(Service.class),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                        Activator.PLUGIN_ID, "icons/service.gif"),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                        Activator.PLUGIN_ID, "icons/usergroups.gif")));

        instGroup.add(new CreationToolEntry("Employee", "创建一个
employee类",
                new NodeCreationFactory(Employee.class),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                        Activator.PLUGIN_ID, "icons/employee.gif"),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                        Activator.PLUGIN_ID, "icons/member.gif")));

        return root;
    }

}
```

目前一切顺利，只差 palette 的入口还没有，它使用下面漂亮的代码。

MyGraphicalEditor 类修改如下：

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette {

    ......
    @Override
    protected PaletteRoot getPaletteRoot() {
        // TODO Auto-generated method stub
        PaletteRoot root = new PaletteRoot();

        PaletteGroup manipGroup = new PaletteGroup("编辑对象工具");
        root.add(manipGroup);

        SelectionToolEntry selectionToolEntry = new
SelectionToolEntry();
        manipGroup.add(selectionToolEntry);
        manipGroup.add(new MarqueeToolEntry());

        root.setDefaultEntry(selectionToolEntry);
```

```
        PaletteSeparator sep2 = new PaletteSeparator();
        root.add(sep2);

        PaletteGroup instGroup = new PaletteGroup("创建元素工具");
        root.add(instGroup);

//      instGroup.add(new CreationToolEntry("Service", "创建一个service
类",
//              new NodeCreationFactory(Service.class), null, null));

        instGroup.add(new CreationToolEntry("Service", "创建一个service
类",
                new NodeCreationFactory(Service.class),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                        Activator.PLUGIN_ID, "icons/service.gif"),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                        Activator.PLUGIN_ID, "icons/usergroups.gif")));

        instGroup.add(new CreationToolEntry("Employee", "创建一个
employee类",
                new NodeCreationFactory(Employee.class),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                        Activator.PLUGIN_ID, "icons/employee.gif"),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                        Activator.PLUGIN_ID, "icons/member.gif")));

        return root;
    }
    }
```

顺便，我们给这两个入口添加一些图标：一个是创建 Service 的入口，一个是创建 Employee 的入口。我们用插件标识符检索 *ImageDescriptor* 并将里面的图标关联起来。如果更喜好交互性的调色板（*FlyoutPalette*），将可以取得一个很好的配置方法，尤其是使用大版本的图标。

所有这些已经完成，我们现在可以测试整个系统。

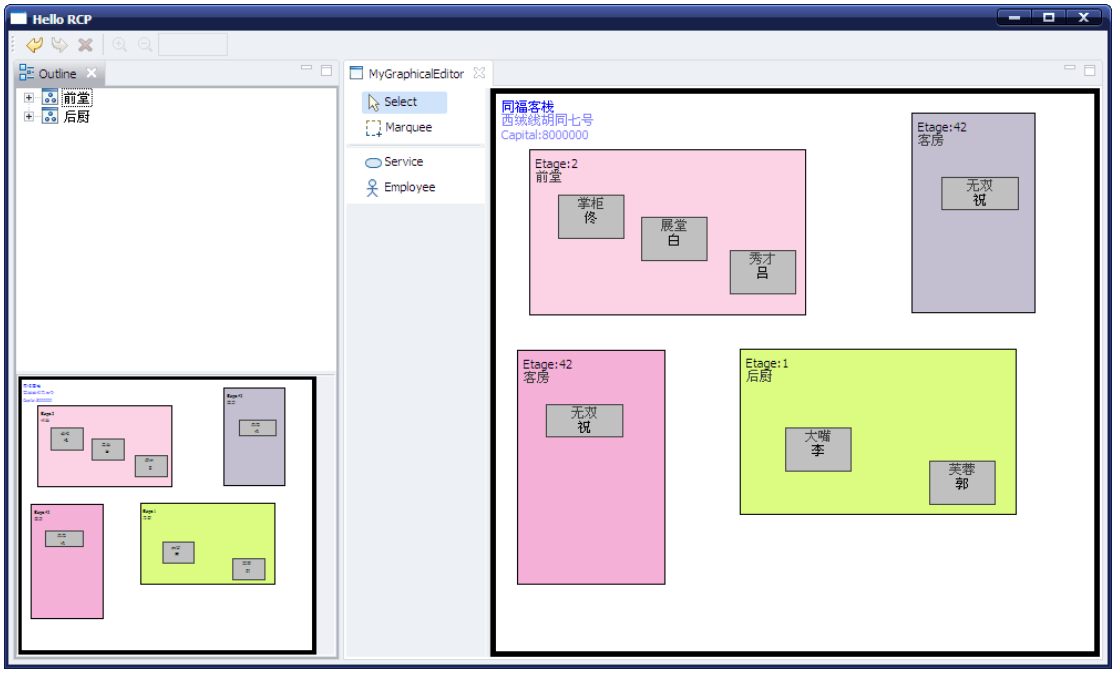


图30 新建 Employee

此阶段代码为 WBHGEF11。

# 12. 拖放（Drag and Drop, DnD）

我们已经可以通过点击和绘制来创建图形化的 Service 和 Employee 实体，但这并不是用户友好及实用的方式。我们现在整合一个在每个称为现代的界面中都必备的功能：拖放（Drag and Drop, DnD）。

几乎所有的需要修改的东西都发生在我们的 Editor（MyGraphicalEditor）中。我们需要做的是，从调色板中，选择（drag）一个将放（drop）到 editor 中的元素。唯一需要做的事情是为 palette 添加一个 *TemplateTransfertDragSourceListener* 类型的的 listener，并为 editor 添加一个 *TemplateTransferDropTargetListener* 类型的 listener。命名已经不言自明：它们是转移模板（部件从源元素 x 移动到目标元素 y，这称为转移），一个是源（拖），另一个是目标（放）。

修改 MyGraphicalEditor 代码如下：

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette {
    ……
        @Override
    protected void initializeGraphicalViewer() {
        // TODO Auto-generated method stub
```

```
        GraphicalViewer viewer = getGraphicalViewer();
        model = CreateEnterprise();
        //viewer.setContents(CreateEnterprise());
        viewer.setContents(model);
        viewer.addDropTargetListener(new
MyTemplateTransferDropTargetListener(viewer));

    }

    @Override
    protected void initializePaletteViewer() {
        super.initializePaletteViewer();
        getPaletteViewer().addDragSourceListener(
                new
TemplateTransferDragSourceListener(getPaletteViewer()));
    }
    ……

}
```

你将注意到我们为直接使用 *TemplateTransferDropTargetListener*，而是使用我们自己派生的版本。确实，drop 的概念完全依赖于模型，所以它的一般类的版本不能生成满足我们的特定需求，类功能将会被重载。

新建 Listener 包，MyTemplateTransferDropTargetListener 类代码如下：

```
package wbhgef.listener;

import org.eclipse.gef.EditPartViewer;
import org.eclipse.gef.dnd.TemplateTransferDropTargetListener;
import org.eclipse.gef.requests.CreationFactory;

import wbhgef.command.NodeCreationFactory;

public class MyTemplateTransferDropTargetListener extends
        TemplateTransferDropTargetListener {

    public MyTemplateTransferDropTargetListener(EditPartViewer viewer)
{
        super(viewer);
        // TODO Auto-generated constructor stub
    }

    @Override
```

```
    protected CreationFactory getFactory(Object template) {
        return new NodeCreationFactory((Class<?>)template);
    }

}
```

唯一关注的部分是工厂中的代码片段，在这里我们可以在请求时返回一个实例。在这个指南的范围里，我们基础地使用我们想用的“类”这个对象。然而，正如我们刚才所见，这个对象被当作模板使用。所以在此建议使用一个对象（独享的实例）的模板版本来完成此前我们所做的。我们在这里没这么做是因为我们的对象非常简单并且不需要模型对象（模板）中建立任何专用信息。因此我们需要考虑采用你的 listener，尤其是你的 CreationFactory。

现在仍需做一件事：为这个类型的交互采用 palette 入口。这有一类 CreationToolEntry 不仅支持我们初始化的创建，还支持 drag-and-drop：*CombinedTemplateCreationEntry*。

MyGraphicalEditor 类代码修改如下：

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette {

    ……
    @Override
    protected PaletteRoot getPaletteRoot() {
        // TODO Auto-generated method stub
        PaletteRoot root = new PaletteRoot();

        PaletteGroup manipGroup = new PaletteGroup("编辑对象工具");
        root.add(manipGroup);

        SelectionToolEntry selectionToolEntry = new
SelectionToolEntry();
        manipGroup.add(selectionToolEntry);
        manipGroup.add(new MarqueeToolEntry());

        root.setDefaultEntry(selectionToolEntry);

        PaletteSeparator sep2 = new PaletteSeparator();
        root.add(sep2);

        PaletteGroup instGroup = new PaletteGroup("创建元素工具");
        root.add(instGroup);
```

```
//      instGroup.add(new CreationToolEntry("Service", "创建一个service
类",
//              new NodeCreationFactory(Service.class), null, null));

//      instGroup.add(new CreationToolEntry("Service", "创建一个service
类",
        instGroup.add(new CombinedTemplateCreationEntry("Service", "创
建一个service类",
                new NodeCreationFactory(Service.class),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                        Activator.PLUGIN_ID, "icons/service.gif"),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                        Activator.PLUGIN_ID, "icons/usergroups.gif")));

//      instGroup.add(new CreationToolEntry("Employee", "创建一个
employee类",
        instGroup.add(new CombinedTemplateCreationEntry("Employee", "创
建一个employee类",
                new NodeCreationFactory(Employee.class),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                        Activator.PLUGIN_ID, "icons/employee.gif"),
                AbstractUIPlugin.imageDescriptorFromPlugin(
                        Activator.PLUGIN_ID, "icons/member.gif")));

        return root;
    }
```

这两个构造型的区别显而易见：我们谈到的模板对象在工厂前插入。在这个实例中，由于前述原因，我们插入对象类。

运行效果如下。!!! 此处拖放较快则为禁止拖放图标，较慢时拖放图形也未更新，需修改。

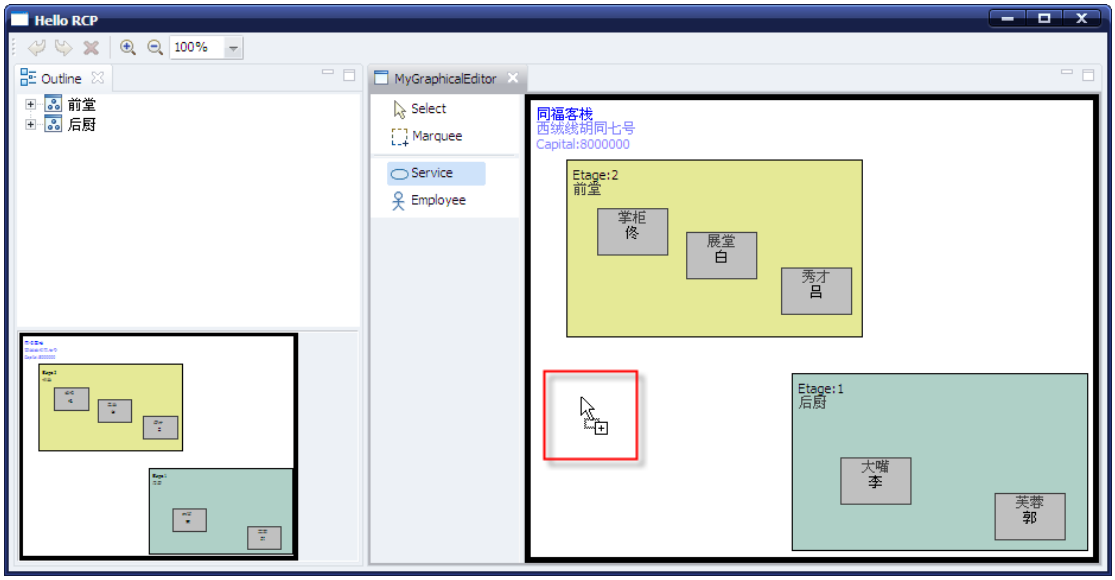


图31 拖放

此阶段代码为 WBHGEF12。

# 13. 剪切和粘贴（Cut and Paste）

在已经看到如何整合拖放功能后，现在我们来关注另一个现今每个体面的界面都必需的功能：cut/paste 功能。这是继 palette 插入和拖放后的第三个功能也是最后一个功能，可以提供新对象的图形创建。

和主要的 GEF 功能相反，剪切和粘贴已经完成，正如名称中提示到的，分为两步。第一步，我们将创建代码 copy 的 command，当环境的 action 合适时将会建立这样的 command。第二部将为 paste 重复上述步骤。最后将把这些 action 整合到应用中。

现在从 command 开始。它当然是继承子 Command 类，并在环境可用时在一个 ArrayList 元素中进行记录从而复制。在我们的案例中，我们要让 employee 和 service 都可以复制。

新建 CopyNodeCommand 类，代码如下：

```
package wbhgef.command;

import java.util.ArrayList;
import java.util.Iterator;
```

```
import org.eclipse.gef.commands.Command;
import org.eclipse.gef.ui.actions.Clipboard;

import wbhgef.model.Employee;
import wbhgef.model.Node;
import wbhgef.model.Service;

public class CopyNodeCommand extends Command {

    private ArrayList<Node> list = new ArrayList<Node>();

    public boolean addElement(Node node) {
        if (!list.contains(node)) {
            return list.add(node);
        }
        return false;
    }

    @Override
    public boolean canExecute() {
        if (list == null || list.isEmpty())
            return false;
        Iterator<Node> it = list.iterator();
        while (it.hasNext()) {
            if (!isCopyableNode(it.next()))
                return false;
        }
        return true;
    }

    @Override
    public void execute() {
        if(canExecute())
            Clipboard.getDefault().setContents(list);
    }

    @Override
    public boolean canUndo() {
        return false;
    }
```

```
    public boolean isCopyableNode(Node node) {
        if(node instanceof Service || node instanceof Employee)
            return true;
        return false;
    }

}
```

如上所示，我们应用 Selection（ArrayList）于 GEF 通过静态方法 *Clipboard.getDefault().setContents(list)*的传递内部机制。稍后，将看到如何检索这些信息。

我们已经有了 command，现在需要定义 action 并整合到应用中。如果一切顺利的话，它将建立这个 command。我们尝试监护 action 到最简形式，那么就在 command 中抽象它的复杂度。

在 action 包中新建 CopyNodeAction，代码如下：

```
package wbhgef.action;

import java.util.Iterator;
import java.util.List;

import org.eclipse.gef.EditPart;
import org.eclipse.gef.commands.Command;
import org.eclipse.gef.ui.actions.SelectionAction;
import org.eclipse.ui.ISharedImages;
import org.eclipse.ui.IWorkbenchPart;
import org.eclipse.ui.PlatformUI;
import org.eclipse.ui.actions.ActionFactory;

import wbhgef.command.CopyNodeCommand;
import wbhgef.model.Node;

public class CopyNodeAction extends SelectionAction {

    public CopyNodeAction(IWorkbenchPart part) {
        super(part);
        // TODO Auto-generated constructor stub
        setLazyEnablementCalculation(true);
    }

    @Override
```

```
    protected void init() {
        super.init();
        ISharedImages sharedImages =
PlatformUI.getWorkbench().getSharedImages();
        setText("Copy");
        setId(ActionFactory.COPY.getId());

        setHoverImageDescriptor(sharedImages.getImageDescriptor(
                ISharedImages.IMG_TOOL_COPY));
        setImageDescriptor(sharedImages.getImageDescriptor(
                ISharedImages.IMG_TOOL_COPY));
        setDisabledImageDescriptor(sharedImages.getImageDescriptor(
                ISharedImages.IMG_TOOL_COPY));
        setEnabled(false);
    }

    private Command createCopyCommand(List<Object> selectedObjects) {
        if(selectedObjects == null || selectedObjects.isEmpty()) {
            return null;
        }

        CopyNodeCommand cmd = new CopyNodeCommand();
        Iterator<Object> it = selectedObjects.iterator();
        while(it.hasNext()) {
            EditPart ep = (EditPart)it.next();
            Node node = (Node)ep.getModel();
            if(!cmd.isCopyableNode(node))
                return null;
            cmd.addElement(node);
        }
        return cmd;
    }

    @Override
    protected boolean calculateEnabled() {
        // TODO Auto-generated method stub
        Command cmd = createCopyCommand(getSelectedObjects());
        if(cmd == null)
            return false;
        return cmd.canExecute();
    }
```

```
    @Override
    public void run() {
        Command cmd = createCopyCommand(getSelectedObjects());
        if(cmd != null && cmd.canExecute()) {
            cmd.execute();
        }
    }

}
```

在第一步中，值得在 *Init()*方法中注意的是，我们将载入 Eclipse 中可用的图标。和在 toolbar 中显示的一样，这些也将显示在环境菜单中（并且，如果已经定义电话，也可以显示在应用的全局菜单中）。

第二步，注意 *run()*方法，我们直接调用这个 command 的 *execute()*方法，而不是将这个 command 作为参数传递时的 *execute()*方法。这是为了避免在 GEF 指令栈中整合 copy；确实，我们不考虑实现一个复制操作的 undo（但是对 paste 的 command 就不是这样）。

复制令人高兴，但是为了粘贴我们刚才复制的东西，才是目的所在。我们将以同样的方式处理。我们创建 command 及一个相关的将会在需要且合适时创建这个 command 的 action。

在 command 包中仙剑 PasteNodeCommand 类，代码如下：

```
package wbhgef.command;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;

import org.eclipse.gef.commands.Command;
import org.eclipse.gef.ui.actions.Clipboard;

import wbhgef.model.Employee;
import wbhgef.model.Node;
import wbhgef.model.Service;

public class PasteNodeCommand extends Command {

    private HashMap<Node, Node> list = new HashMap<Node, Node>();

    @Override
```

```
    public boolean canExecute() {
        ArrayList<Node> bList =
(ArrayList<Node>)Clipboard.getDefault().getContents();
        if(bList == null || bList.isEmpty())
            return false;

        Iterator<Node> it = bList.iterator();
        while(it.hasNext()) {
            Node node = (Node)it.next();
            if(isPastableNode(node)) {
                list.put(node, null);
            }
        }
        return true;
    }

    @Override
    public void execute() {
        if(!canExecute())
            return ;

        Iterator<Node> it = list.keySet().iterator();
        while(it.hasNext()) {
            Node node = (Node)it.next();
            try {
                if(node instanceof Service) {
                    Service srv = (Service)node;
                    Service clone = (Service)srv.clone();
                    list.put(node, clone);
                }
                else if(node instanceof Employee) {
                    Employee emp = (Employee)node;
                    Employee clone = (Employee)emp.clone();
                    list.put(node, clone);
                }
            }
            catch (CloneNotSupportedException e) {
                e.printStackTrace();
            }
        }
        redo();
    }
```

```
    @Override
    public void redo() {
        Iterator<Node> it = list.values().iterator();
        while(it.hasNext()) {
            Node node = it.next();
            if(isPastableNode(node)) {
                node.getParent().addChild(node);
            }
        }
    }

    @Override
    public boolean canUndo() {
        return !(list.isEmpty());
    }

    @Override
    public void undo() {
        Iterator<Node> it = list.values().iterator();
        while(it.hasNext()) {
            Node node = it.next();
            if(isPastableNode(node)) {
                node.getParent().removeChild(node);
            }
        }
    }

    public boolean isPastableNode(Node node) {
        if(node instanceof Service || node instanceof Employee)
            return true;
        return false;
    }

}
```

我们检索 Clipboard 中我们在 copy 的 command 中定义过的内容。下一步，一个用于储存的 HashMap，源对象作为 key，从 key 中复制形成的作为 value。然后我们使用这个值来检索其父亲（同样的 key，因为 *clone()*方法保持其亲属关系，见下）。新建的对象需要保存，这样剪切和粘贴可以整合到 GEF 的 undo/redo 设备中。来做下面的工作。

在 action 包中新建 PasteNodeAction，代码如下：

```
package wbhgef.action;

import org.eclipse.gef.commands.Command;
import org.eclipse.gef.ui.actions.SelectionAction;
import org.eclipse.ui.ISharedImages;
import org.eclipse.ui.IWorkbenchPart;
import org.eclipse.ui.PlatformUI;
import org.eclipse.ui.actions.ActionFactory;

import wbhgef.command.PasteNodeCommand;

public class PasteNodeAction extends SelectionAction {

    public PasteNodeAction(IWorkbenchPart part) {
        super(part);
        // TODO Auto-generated constructor stub
        setLazyEnablementCalculation(true);
    }

    protected void init()
    {
        super.init();
        ISharedImages sharedImages =
PlatformUI.getWorkbench().getSharedImages();
        setText("Paste");
        setId(ActionFactory.PASTE.getId());

        setHoverImageDescriptor(sharedImages.getImageDescriptor(
                ISharedImages.IMG_TOOL_PASTE));
        setImageDescriptor(sharedImages.getImageDescriptor(
                ISharedImages.IMG_TOOL_PASTE));
        setDisabledImageDescriptor(sharedImages.getImageDescriptor(
                ISharedImages.IMG_TOOL_PASTE));
        setEnabled(false);
    }

    private Command createPasteCommand() {
        return new PasteNodeCommand();
    }

    @Override
```

```
    protected boolean calculateEnabled() {
        // TODO Auto-generated method stub
        Command command = createPasteCommand();
        return command != null && command.canExecute();
    }

    @Override
    public void run() {
        Command command = createPasteCommand();
        if(command != null && command.canExecute())
            execute(command);
    }

}
```

此处没有激发任何事情。我们创建建立这个 command 的 action。然而让我们先来看以下细节。如果你仔细观察，应该已经注意到 *clone()*方法。这是方法是一般的并且需要进行个性化从而可以返回一个新对象（一个新实例），很类似于 *calling* 实例。两个 *clone()*方法应该由同样的父亲，它将验证 paste 指令将会正确运作。

在 Employee 类中添加 clone()方法，代码如下：

```
public class Employee extends Node {

    ……
    @Override
    public Object clone() throws CloneNotSupportedException {
        Employee emp = new Employee();
        emp.setName(this.getName());
        emp.setParent(this.getParent());
        emp.setPrenom(this.prenom);
        emp.setLayout(new Rectangle(getLayout().x+10, getLayout().y+10,
                getLayout().width,getLayout().height));
        return emp;
    }
}
```

在 Service 的实例中，我们需要 *clone()*方法返回一个新的 Service，并和其模型一样拥有同样的 employees；这样我们还需要复制这个对象的子元素。对复制 employee 的布局有些繁琐，这个可以保持它们在 Service 内同样准确的布局。

在 Service 类中添加 clone()方法，代码如下：

```
public class Service extends Node {

    ……
    @Override
    public Object clone() throws CloneNotSupportedException {
        Service srv = new Service();
        srv.setColor(this.color);
        srv.setEtage(this.etage);
        srv.setName(this.getName());
        srv.setParent(this.getParent());
        srv.setLayout(new Rectangle(
                getLayout().x+10, getLayout().y+10,
                getLayout().width, getLayout().height));

        Iterator<Node> it = this.getChildrenArray().iterator();
        while(it.hasNext()) {
            Node node = it.next();
            if(node instanceof Employee) {
                Employee child = (Employee)node;
                Node clone = (Node)child.clone();
                srv.addChild(clone);
                clone.setLayout(child.getLayout());
            }
        }
        return srv;
    }
}
```

现在我们的 action（及其各自的 command）都已生效，我们需要将其整合进应用。首先，我们 *MyGraphicalEditorActionBarContributor* 的的 toolbar 中添加 action。

修改 MyGraphicalEditorActionBarContributor 代码如下：

```
public class MyGraphicalEditorActionBarContributor extends
ActionBarContributor {

    @Override
    protected void buildActions() {
        // TODO Auto-generated method stub

        IWorkbenchWindow iww = getPage().getWorkbenchWindow();
```

```
    addRetargetAction((RetargetAction)ActionFactory.COPY.create(iww))
;

    addRetargetAction((RetargetAction)ActionFactory.PASTE.create(iww)
);
        ……
    }

    ……
    public void contributeToToolBar(IToolBarManager toolBarManager) {
        ……
        toolBarManager.add(getAction(ActionFactory.COPY.getId()));
        toolBarManager.add(getAction(ActionFactory.PASTE.getId()));
    }

    ……
}
```

然后，这些 action 需要在 editor 控制器中添加，这个当然像下面一样，添加在 *MyGraphicalEditor* 中。

修改 *MyGraphicalEditor* 代码如下：

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette {

    ……
    protected class OutlinePage extends ContentOutlinePage {
        ……
        public void createControl(Composite parent) {
            ……
            IActionBars bars = getSite().getActionBars();
            ActionRegistry ar = getActionRegistry();
            ……
            bars.setGlobalActionHandler(ActionFactory.COPY.getId(),
                    ar.getAction(ActionFactory.COPY.getId()));
            bars.setGlobalActionHandler(ActionFactory.PASTE.getId(),
                    ar.getAction(ActionFactory.PASTE.getId()));
            ……
        }
        ……
    }
    ……
    public void createActions() {
```

```
        ……

        action = new CopyNodeAction(this);
        registry.registerAction(action);
        getSelectionActions().add(action.getId());

        action = new PasteNodeAction(this);
        registry.registerAction(action);
        getSelectionActions().add(action.getId());

    }
    ……
}
```

别走开，尚未完成。最后一点要做的是，不确定键盘快捷键可以使用。为此，我们在 *ApplicationActionBarAdvisor* 中注册这些 action。

修改代码 *ApplicationActionBarAdvisor*.如下：

```
public class ApplicationActionBarAdvisor extends ActionBarAdvisor {

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

    protected void makeActions(IWorkbenchWindow window) {
    makeAction(window, ActionFactory.UNDO);
    makeAction(window, ActionFactory.REDO);
    makeAction(window, ActionFactory.COPY);
    makeAction(window, ActionFactory.PASTE);
    }

    protected void fillMenuBar(IMenuManager menuBar) {
    }

    protected IWorkbenchAction makeAction(IWorkbenchWindow window,
ActionFactory af) {
    IWorkbenchAction action = af.create(window);
    register(action);
    return action;
    }
```

```
}
```

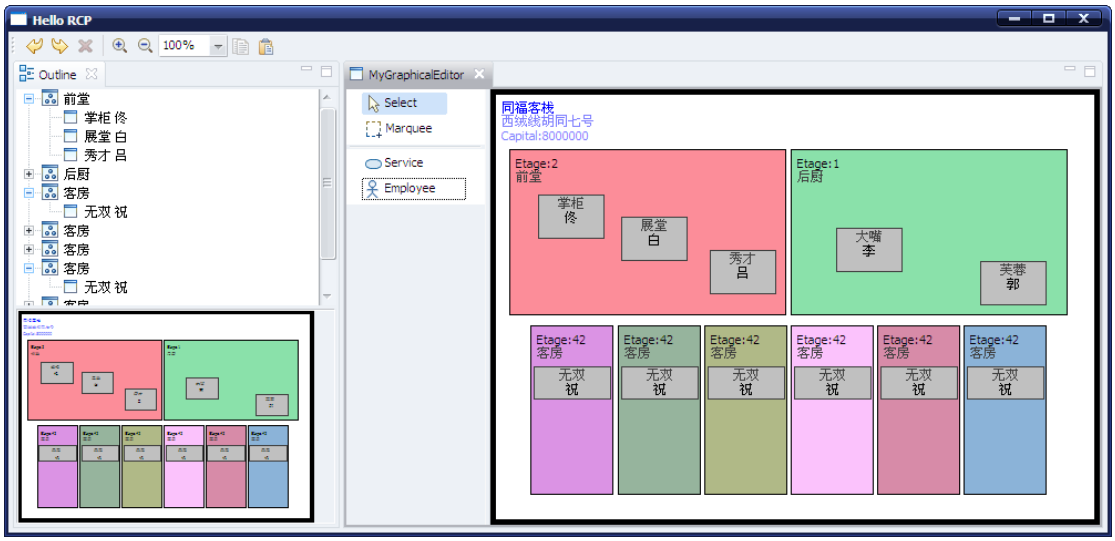


图32 复制和粘贴

此阶段代码为 WBHGEF13。

# 14. 总结（Conclusion）

有了这个指南，你可以看到 GEF 中最常用和最惯用的功能，并看到 Eclipse 一些有用的部分（RCP，属性页，action）。

# 15. 参考书目（References）

GEF website:
http://download.eclipse.org/tools/gef
GEF in the Eclipse wiki:
http://wiki.eclipse.org/index.php/Graphical_Editing_Framework

# 16. 译者后记（PostScripts）

07 年第首版发布，08 年中网络更新版本，至 10 年中才见到，11 年中方始翻译，至 7 月完成。期间依指南进行编码，收获甚多。工程建立、框架简介，以及功能的逐步完善皆详尽介绍，代码完整，依指南步骤可原本复原。对初步了解 Eclipse 及 GEF 框架有所帮助。

然而，此版翻译仍有待改进之处，如文字为直译，未加润色；通知机制只完成部分演示，并非所有元素支持属性页修改及相应反馈；环境菜单只实现部分按钮等。另，如关系连线的建立、锚点的设置、单击模型名称进行修改、属性页添加下拉菜单等，在此文中也尚未提及，还需结合其他资料再进行整合。

望能更新更加完善版本，实现更多功能。

# 17. 连线（Connection）

本章开始无资料，为网络搜寻材料自行整理，故结构有不合理之处多包涵。

关于连线的基本须知：

1．GEF 中，一个连接线模型要成功创建，则这个连线模型必须有源和目的。默认情况下点击一个源，再任意点击一个点，无连线。

2．GEF 中，一个连接线模型即使已经被附着到了源和目的上，也不一定会被显示出来。这就要提到 GEF 框架中 AbstractGraphicalEditPart 类的两个方法：getModelSourceConnections() 和 getModelTargetConnections() 。和 getModelChildren()方法获得模型的子元素一样，这两个方法就是用来返回结点上的源连接线和目的连接线。所以其实我们真正要保证的是这两个方法返回的结果正确。

3．GEF 中，一个模型对象要支持连接，它的 EditPart 必须实现一个接口：NodeEditPart。实现这个接口主要就是要实现它的四个方法用来确定连接线的锚点。一般来说简单的返回一个 ChopboxAnchor 对象就可以。当然也可以自己实现自定义的锚点算法。

**定义结点对象（修改 Model）**

首先对要安装连线功能的模型作为结点对象，进行上述步骤。为记录结点对象的源和目的，加上两个 List 以及相应方法来储存和操作这些连接线。此处以 Service 为例子。

修改 Service 类代码如下：

```
public class Service extends Node {
```

```
    ……
    private List inputs = new ArrayList();
    private List outputs = new ArrayList();
    ……

    public List getInputs() {
        return inputs;
    }

    public List getOutputs() {
        return outputs;
    }

    public void addIn(AbstractConnectionModel model) {
        if(!inputs.contains(model)) {
            inputs.add(model);
        }
    }

    public void addOut(AbstractConnectionModel model) {
        if(!outputs.contains(model)) {
            outputs.add(model);
        }
    }

    public void removeInput(AbstractConnectionModel model) {
        if(inputs.contains(model)) {
            inputs.remove(model);
        }
    }

    public void removeOut(AbstractConnectionModel model) {
        if(outputs.contains(model)) {
            outputs.remove(model);
        }
    }
}
```

先假设所有的连线模型都继承子 AbstractConnectionModel。

**修改节点的 EditPart**

让模型相应的 EditPart 实现 GEF 中 NodeEditPart 类，并完成下述方法。

修改 ServiceEditPart 代码如下：

```
public class ServicePart extends AppAbstractEditPart implements
NodeEditPart{

    ……

    @Override
    public ConnectionAnchor getSourceConnectionAnchor(
            ConnectionEditPart connection) {
        // TODO Auto-generated method stub
        return new ChopboxAnchor(getFigure());
    }

    @Override
    public ConnectionAnchor getSourceConnectionAnchor(Request request)
{
        // TODO Auto-generated method stub
        return new ChopboxAnchor(getFigure());
    }

    @Override
    public ConnectionAnchor getTargetConnectionAnchor(
            ConnectionEditPart connection) {
        // TODO Auto-generated method stub
        return new ChopboxAnchor(getFigure());
    }

    @Override
    public ConnectionAnchor getTargetConnectionAnchor(Request request)
{
        // TODO Auto-generated method stub
        return new ChopboxAnchor(getFigure());
    }

    @Override
    protected List getModelSourceConnections() {
        return ((Service)getModel()).getOutputs();
    }

    @Override
    protected List getModelTargetConnections() {
        return ((Service)getModel()).getInputs();
```

```
    }

}
```

下面为连接线添加 MVC 结构。由于连接线在 GEF 中也被视为模型，所以也遵从 MVC 结构，需要创建相应的模型（model）、视图（View）即控制器（Controller，即 EditPart）。注意，连接线的 EditPart 需要继承 AbstractConnectionEditPart。

### 创建模型

由于连接线多种多样，如有无箭头，有无路由点。对其中共同的性质：每个连接线只能有一个源、一个目的，可以令连接线模型继承一个抽象类 AbstractConnectionModel。

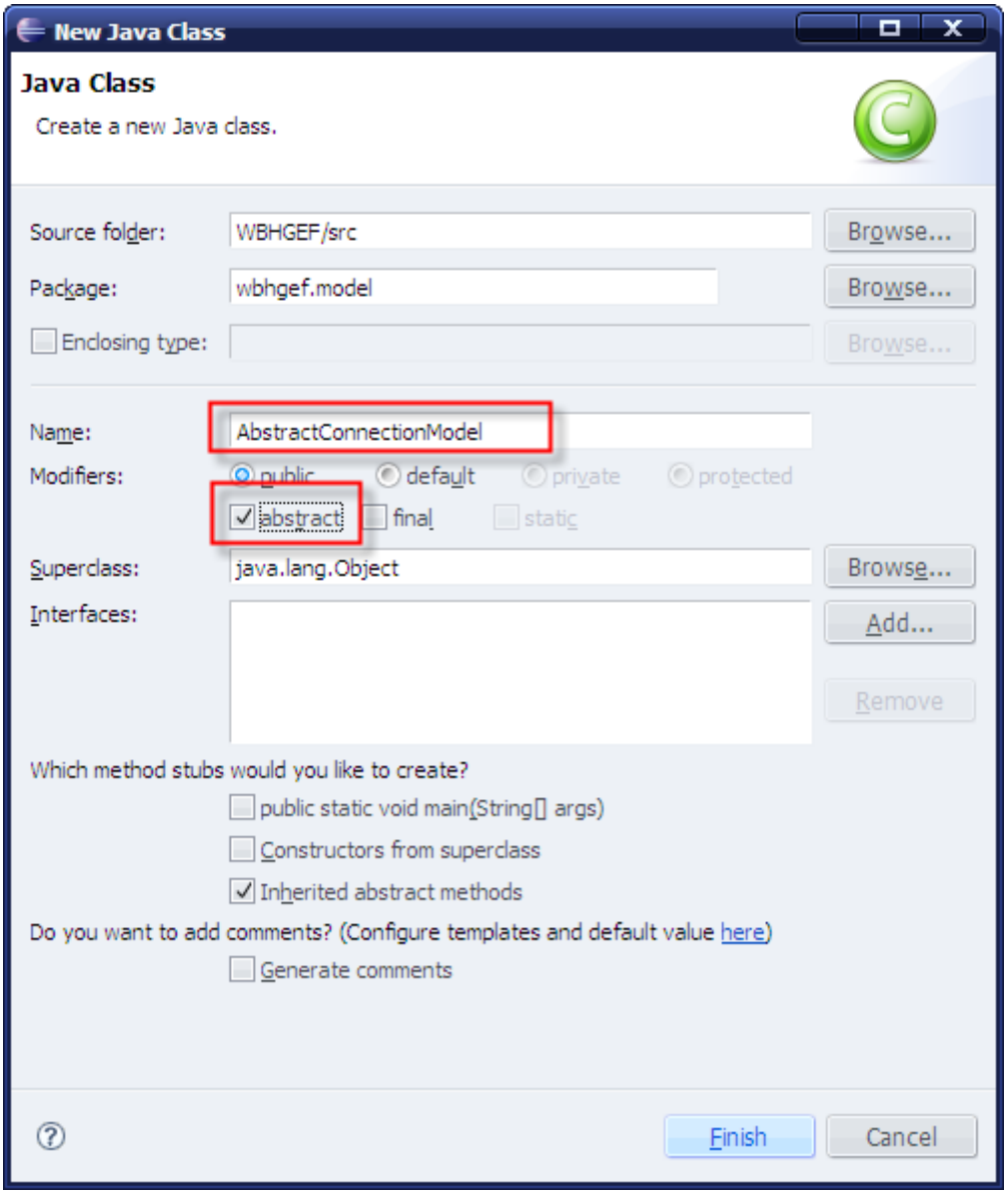

图33 新建连线抽象模型

在 model 包中新建抽象类 AbstractConnectionModel，代码如下：

```
public abstract class AbstractConnectionModel {

    private Service src;
    private Service target;

    public Service getSrc() {
        return src;
    }

    public void setSrc(Service src) {
        this.src = src;
    }

    public Service getTarget() {
        return target;
    }

    public void setTarget(Service target) {
        this.target = target;
    }

    public void attachSource() {
        if(!src.getOutputs().contains(this)) {
            src.addOut(this);
        }
    }

    public void attachTarget() {
        if(!target.getInputs().contains(this)) {
            target.addInput(this);
        }
    }

    public void deattachSource() {
        if(src.getOutputs().contains(this)) {
            src.removeOut(this);
        }
    }
```

```
    public void deattachTarget() {
        if(target.getInputs().contains(this)) {
            target.removeInput(this);
        }
    }

}
```

再新建一个 PlainConnectionModel 类，继承这个类，内容为空，表示一个普通的连接线。

**创建 EditPart**

在 EditPart 包中新建一个 PlainConnectionEditPart 类，继承 GEF 中的 AbstractConnectionEditPart，内容暂时空。

代码如下：

```
public class PlainConnectionEditPart extends AbstractConnectionEditPart
{

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

    }

}
```

**创建视图**

暂时不需，连接线会有一个默认的视图。

**关联 MVC**

在 EditPartFactory 内追加关联，

```
public class AppEditPartFactory implements EditPartFactory {

    @Override
    public EditPart createEditPart(EditPart context, Object model) {
        // TODO Auto-generated method stub
        AbstractGraphicalEditPart part = null;

        if(model instanceof Enterprise) {
            part = new EnterprisePart();
        } else if(model instanceof Service) {
```

```
            part = new ServicePart();
        } else if(model instanceof Employee) {
            part = new EmployeePart();
        }

        //关联连线的MVC
        if(model instanceof PlainConnectionModel) {
            PlainConnectionEditPart editPart = new
PlainConnectionEditPart();
            editPart.setModel(model);
            return editPart;
        }

        part.setModel(model);
        return part;
    }

}
```

先测试下刚才写过的代码，在 MyGraphicalEditor 中修改 CreateEnterprise()方法，添加一些连接对象。

修改 MyGraphicalEditor 代码如下：

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette {
    ……
    public Enterprise CreateEnterprise() {
        ……
            PlainConnectionModel connection =  new
PlainConnectionModel();
            connection.setSrc(service_QianTang);
            connection.setTarget(service_HouChu);
            connection.attachSource();
            connection.attachTarget();
    }
    ……
}
```

运行效果如下：

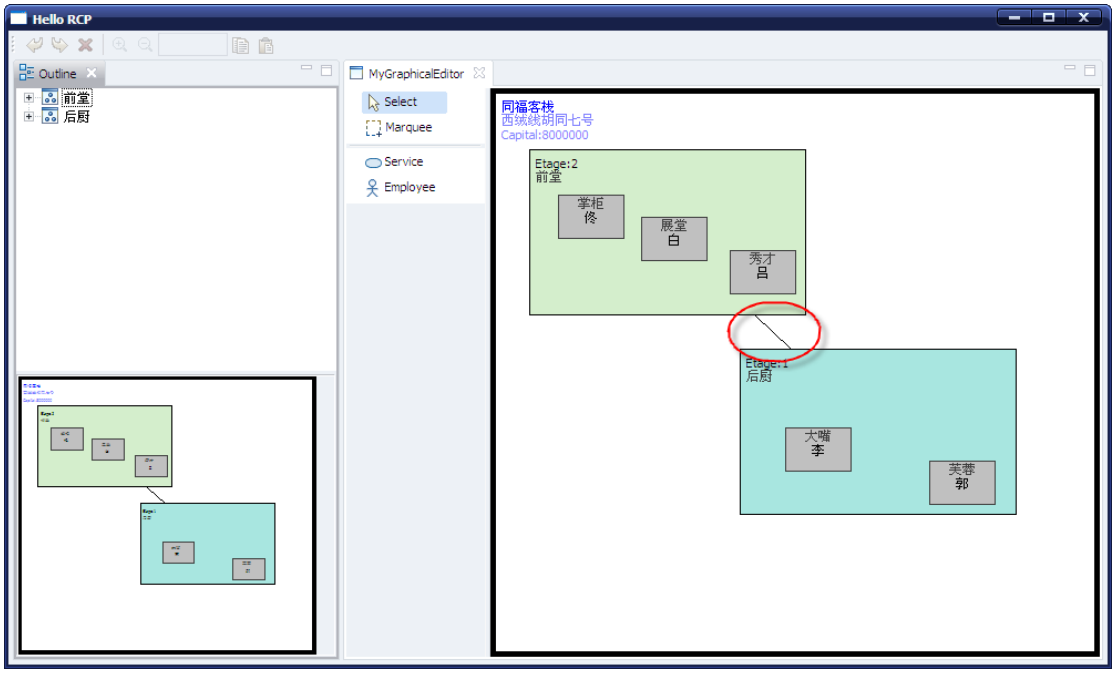


图34 添加普通连线

此阶段代码为 WBHGEF14。

下面开始动态创建连线。

修改 MyGraphicalEditor 代码如下：

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette {
    ……
    @Override
    protected PaletteRoot getPaletteRoot() {
        ……
        //连线工具栏
        PaletteGroup connectionGroup = new PaletteGroup("创建连线工具");
        root.add(connectionGroup);

        ConnectionCreationToolEntry plainConnection = new
ConnectionCreationToolEntry(
                "直线", "普通的直联连线", new SimpleFactory(
                        PlainConnectionModel.class),

    Activator.getImageDescriptor("icons/plainline.gif"),

    Activator.getImageDescriptor("icons/plainline.gif"));

        connectionGroup.add(plainConnection);
```

```
        PaletteSeparator sep3 = new PaletteSeparator();
        root.add(sep2);
        ……
    }
    ……
}
```

为了实现新建模型连接，需要在模型结点上安装 GraphicalNodeEditPolicy。

修改 ServicePart 类，安装新的图形结点 EditPolicy，代码如下：

```
public class ServicePart extends AppAbstractEditPart implements
NodeEditPart{
    ……
    @Override
    protected void createEditPolicies() {
        // TODO Auto-generated method stub
        installEditPolicy(EditPolicy.LAYOUT_ROLE, new
AppEditLayoutPolicy());
        installEditPolicy(EditPolicy.COMPONENT_ROLE, new
AppDeletePolicy());
        installEditPolicy(EditPolicy.NODE_ROLE, new AppRenamePolicy());
        installEditPolicy(EditPolicy.GRAPHICAL_NODE_ROLE, new
ServiceGraphicalNodeEditPolicy());
    }
    ……
}
```

其中 ServiceGraphicalNodeEditPolicy 是我们自己设置的 Policy 实现类，后面再做添加，先完成 Command。

要实现一个连接，至少需要三个对象：连接线、源结点、目标结点。

此时 Command 添加能创建，不能显示，因为没有刷新。增加刷新事件箭头即可。

在 Command 包中新建 AddConnectionCommand 类，代码如下：

```
public class AddConnectionCommand extends Command {

    private AbstractConnectionModel connection;

    public AddConnectionCommand(AbstractConnectionModel connection) {
        super();
```

```
        this.connection = connection;
    }

    public void setSrc(Service src) {
        connection.setSrc(src);
    }

    public void setTarget(Service target) {
        connection.setTarget(target);
    }

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

    @Override
    public void execute() {
        connection.attachSource();
        connection.attachTarget();
    }

    @Override
    public void undo() {
        connection.deattachSource();
        connection.deattachTarget();
    }

}
```

然后实现 Policy，分为两步：完成源结点的连接，和完成目标结点的连接。

在 EditPolicy 包中新建 ServiceGraphicalNodeEditPolicy 类，继承自 GraphicalNodeEditPolicy 类，暂不用实现 EditPolicy 接口。

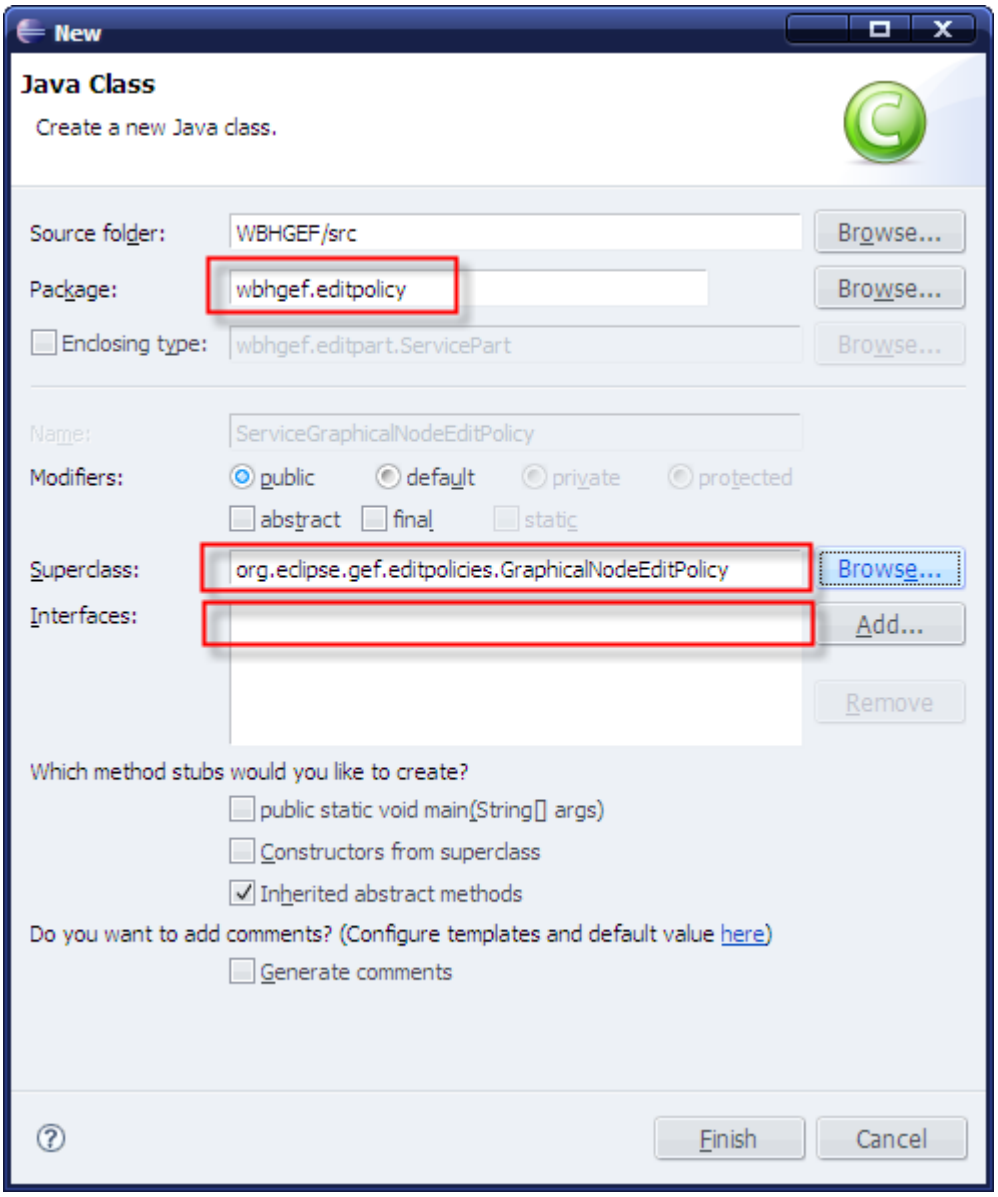


图35 新建 ServiceGraphicalNodeEditPolicy 类

代码如下：

```
public class ServiceGraphicalNodeEditPolicy extends
GraphicalNodeEditPolicy {

    @Override
    protected Command getConnectionCompleteCommand(
            CreateConnectionRequest request) {
        // TODO Auto-generated method stub
        AddConnectionCommand command =
(AddConnectionCommand)request.getStartCommand();
        command.setTarget((Service)getHost().getModel());
        return command;
```

```
    }

    @Override
    protected Command
getConnectionCreateCommand(CreateConnectionRequest request) {
        // TODO Auto-generated method stub
        AddConnectionCommand command = new
AddConnectionCommand((AbstractConnectionModel)
                request.getNewObject());
        command.setSrc((Service)getHost().getModel());
        request.setStartCommand(command);
        return command;
    }

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

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

}
```

修改 Service 模型中相关的四个方法，使得源及目标添加移除事件可以传递。

修改 Service 类代码如下：

```
public class Service extends Node {

    ……
    public static final String P_SOURCE = "p_source";
    public static final String P_TARGET = "p_target";
    ……
    public void addInput(AbstractConnectionModel model) {
        if(!inputs.contains(model)) {
            inputs.add(model);
```

```
            firePropertyChange(P_TARGET, null, model);
        }
    }

    public void addOut(AbstractConnectionModel model) {
        if(!outputs.contains(model)) {
            outputs.add(model);
            firePropertyChange(P_SOURCE, null, model);
        }
    }

    public void removeInput(AbstractConnectionModel model) {
        if(inputs.contains(model)) {
            inputs.remove(model);
            firePropertyChange(P_TARGET, model, null);
        }
    }

    public void removeOut(AbstractConnectionModel model) {
        if(outputs.contains(model)) {
            outputs.remove(model);
            firePropertyChange(P_SOURCE, model, null);
        }
    }
}
```

还需在 Node 类中添加如下代码，用于传递事件：

```
public class Node implements IAdaptable{
    ……
    public void firePropertyChange(String propertyName, Object oldValue,
Object newValue) {
        listeners.firePropertyChange(propertyName, oldValue, newValue);
    }
}
```

再修改 ServiceEditPart 类中的 propertyChange()方法，代码如下：

```
public class ServicePart extends AppAbstractEditPart implements
NodeEditPart{
    ……
    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        ……
```

```
        if(evt.getPropertyName().equals(Service.P_SOURCE))
refreshSourceConnections();
        if(evt.getPropertyName().equals(Service.P_TARGET))
refreshTargetConnections();
    }
    ……
}
```

运行效果如下图：

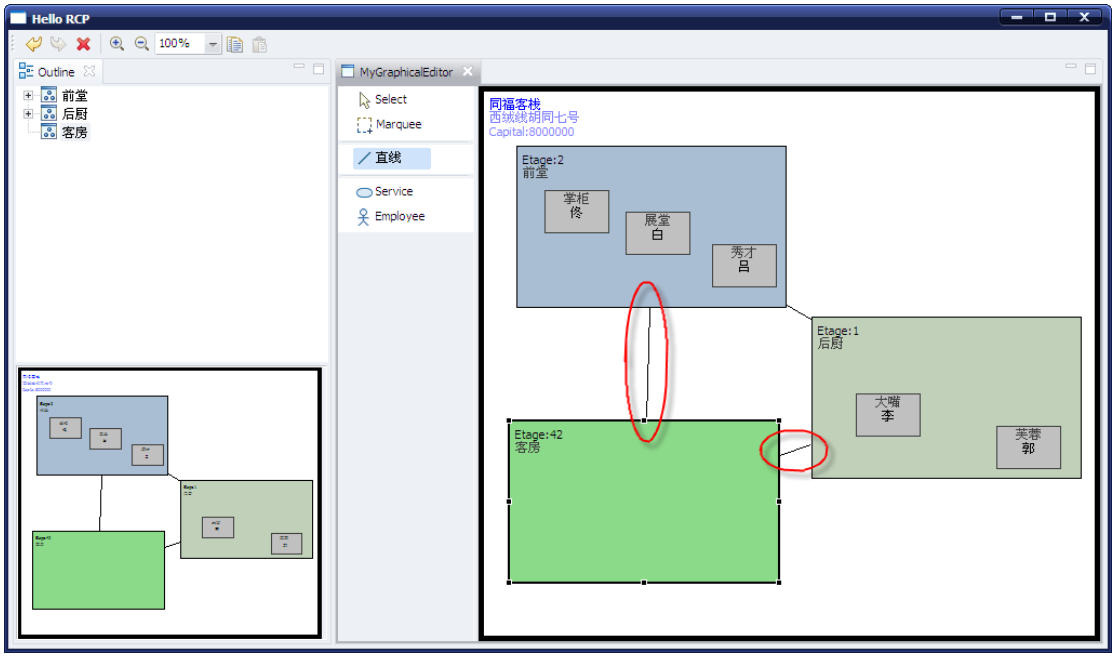


图36 新建连线图形更新

此阶段代码为 WBHGEF15。

现在添加连线的选中与重定向功能。

因为所有连线都需要可以被选中，故这个 EditPolicy 安装在 AbstractConnectionModel 对应的 EditPart 上。

新建 AbstractConnectionEditPart，继承 GEF 的 AbstractConnectionEditPart 类，代码如下：

```
public class AbstractConnectionEditPart extends
        org.eclipse.gef.editparts.AbstractConnectionEditPart {

    @Override
    protected void createEditPolicies() {
        // TODO Auto-generated method stub
        installEditPolicy(EditPolicy.CONNECTION_ROLE,
```

```
                new ConnectionEndpointEditPolicy());

    }

}
```

再令其他的 connection 的 EditPart 继承它。

删除 PlainConnectionEditPart 中的如下代码即可：

```
import org.eclipse.gef.editparts.AbstractConnectionEditPart;
```

此时普通直线的 EditPart 继承的便是同一个包（EditPart 包）内我们自己的抽象连线 EditPart。

此时还需要把 PlainConnectionEditPart 类中的 createEditPolicies()方法注释掉，因其调用覆盖了我们刚才在抽象类中安装的 EditPolicy。

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

这样以后所有连线即会调用抽象类中的可选中连线的 EditPolicy。

运行效果如下：

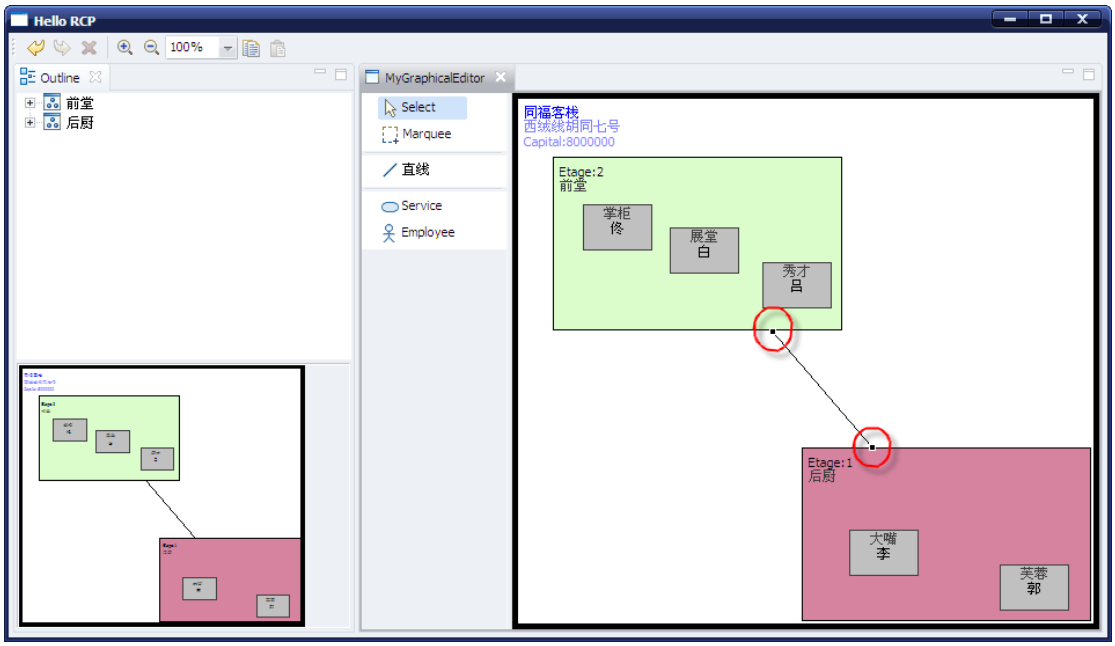


图37 连线可选中

下面添加重定向。

在 ServiceGraphicalNodeEditPolicy 类继承 GraphicalNodeEditPolicy 类时，还有两个方法未完成，即 getReconnectSourceCommand() 和 getReconnectTargetCommand()，用于实现重连。

写重连用的 Command 即可。

在此回顾一下连接的过程：建立一个连接对象（即连线），给对象设置源和目标，最后把连线附着到源和目标上。

对于重连，如果是重新定向源，则首先离开原来的源，再把目标源设置为新的源，最后附着到这个新的源。大致代码如下：

```
connection.deattachSource();
connection.setSrc(newSrc);
connection.attachSource();
```

重定向目标结点过程类似。

在 Command 包中新建 ReconnectSourceCommand 类，代码如下：

```
public class ReconnectSourceCommand extends Command {

    private AbstractConnectionModel connection;

    private Service src;
    private Service old;

    public ReconnectSourceCommand(AbstractConnectionModel connection,
            Service newSource) {
        super();
        this.connection = connection;
        this.src = newSource;
    }

    @Override
    public void execute() {
        old = connection.getSrc();
        connection.deattachSource();
        connection.setSrc(src);
        connection.attachSource();
    }
```

```
    @Override
    public void undo() {
        src = connection.getSrc();
        connection.deattachSource();
        connection.setSrc(old);
        connection.attachSource();
    }

}
```

并新建 ReconnectTargetCommand 类，代码如下：

```
public class ReconnectTargetCommand extends Command {

    private AbstractConnectionModel connection;

    private HelloWorldModel target;

    private HelloWorldModel old;

    public ReconnectTargetCommand(AbstractConnectionModel connection,
HelloWorldModel newTarget) {
        super();
        this.connection = connection;
        this.target = newTarget;
    }

    @Override
    public void execute() {
        old = connection.getTarget();
        connection.deattachTarget();
        connection.setTarget(target);
        connection.attachTarget();
    }

    @Override
    public void undo() {
        target = connection.getTarget();
        connection.deattachTarget();
        connection.setTarget(old);
        connection.attachSource();
    }
```

```
}
```

最后补上先前 ServiceGraphicalNodeEditPolicy 类中继承的未完成的重连源及目标的方法：

修改 ServiceGraphicalNodeEditPolicy 类代码如下：

```
public class ServiceGraphicalNodeEditPolicy extends
GraphicalNodeEditPolicy {

    ……

    @Override
    protected Command getReconnectSourceCommand(ReconnectRequest
request) {
        // TODO Auto-generated method stub
        ReconnectSourceCommand command = new ReconnectSourceCommand(

    (AbstractConnectionModel)request.getConnectionEditPart().getModel
(),
                (Service)getHost().getModel());
        return command;
    }

    @Override
    protected Command getReconnectTargetCommand(ReconnectRequest
request) {
        // TODO Auto-generated method stub
        ReconnectTargetCommand command = new ReconnectTargetCommand(

    (AbstractConnectionModel)request.getConnectionEditPart().getModel
(),
                (Service)getHost().getModel());
        return command;
    }

}
```

运行效果如下：

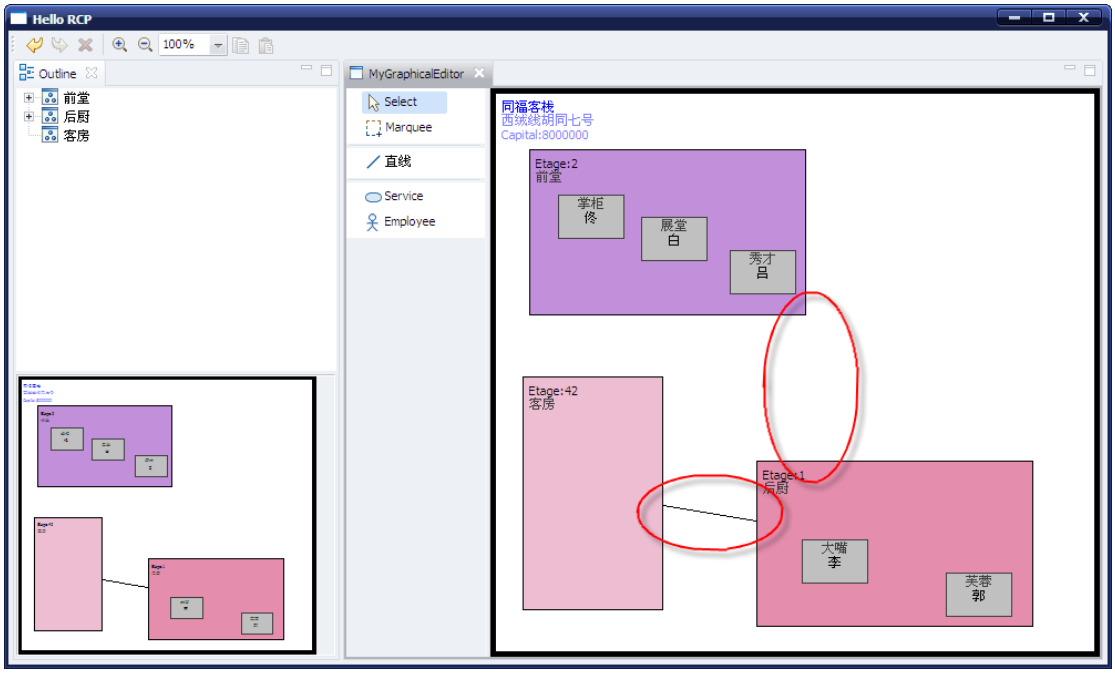


图38 连线可重连

此时若结点删除，连线仍在，需要进行修改使得结点删除时，与其连接的线也一并删除。

修改删除结点的 Command，即 DeleteCommand 类代码：

```
public class DeleteCommand extends Command {
    ……
    private List<AbstractConnectionModel> inputs =
        new ArrayList<AbstractConnectionModel>();
    private List<AbstractConnectionModel> outs =
        new ArrayList<AbstractConnectionModel>();

    public void execute() {
        this.parentModel.removeChild(model);
        if(model instanceof Service) {
            inputs.addAll(((Service) model).getInputs());
            outs.addAll(((Service) model).getOutputs());
            for(AbstractConnectionModel connection : inputs) {
                connection.deattachSource();
                connection.deattachTarget();
            }
            for(AbstractConnectionModel connection : outs) {
                connection.deattachSource();
                connection.deattachTarget();
            }
```

```
        }
    }
    ……
    public void undo() {
        this.parentModel.addChild(model);
        if(model instanceof Service) {
            inputs.addAll(((Service) model).getInputs());
            outs.addAll(((Service) model).getOutputs());
            for(AbstractConnectionModel connection : inputs) {
                connection.attachSource();
                connection.attachTarget();
            }
            for(AbstractConnectionModel connection : outs) {
                connection.attachSource();
                connection.attachTarget();
            }
        }
    }

}
```

即，先在删除时储存所有连线，删除结点时，获取结点的所有连线，在删除执行（execute()）时（先判断删除模型类是否为 Service，此处暂只支持 Service 的连线删除）进行 deattach()操作（图形自动消失），并在回复操作时进行 attach()操作。此段为自行编写，未检验其他功能是否仍正确，只实现结点的删除与恢复时，其连线一同删除与恢复。

现在考虑连接的删除。由于 Service 或 Employee 使用的删除是由父类删除相应子元素，而连线在本例中尚未建立模型信息（也可能已经建立，而未将其归属于 Enterprise 模型的子类，且在大纲视图中未显示），所以无法按此例删除。

解决方案最好为选择连线后可点击 toolbar 中删除按钮，或键盘 del 直接删除。

考虑到将来连线也包含属性可以进行扩展，应为其建立属于底板（Enterprise）的模型，视同于 Service 等模型进行类似操作。

此处又涉及到将连线归为哪类元素的子类的问题，如，同样是在 Enterprise 元素内的 Service 和 Service 的 Connection，Service 属于 Enterprise，而 Connection 连接 Service 时，可以属于 Enterprise（MagicDraw 方案），即对所有 Connection 进行统一管理，也可属于所连的 Service（此问题较大，若 Connection 连接两元素，而只属于一个 Service，连线不属于的 Service 删除时，要通知到连线较麻烦。尚未见有建模工具在两 Service 下进行同样拷贝）。

删除的问题暂且搁置，后面再作讨论。

下面建立带箭头的连线。

建立箭头连接线模型 ArrowConnectionModel 类，继承 AbstractConnectionModel 类，模型为空即可。

```
public class ArrowConnectionModel extends AbstractConnectionModel {

}
```

再建立 ArrowConnectionEditPart 类，继承自 AbstractConnectionEditPart 类，因不能使用缺省连线，故还需修改 createFigure()方法，代码如下：

```
public class ArrowConnectionEditPart extends AbstractConnectionEditPart
{

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

    }

    @Override
    protected IFigure createFigure() {
        PolylineConnection connection = (PolylineConnection)
super.createFigure();
        PolygonDecoration decoration = new PolygonDecoration();
        connection.setTargetDecoration(decoration);
        return connection;
    }

}
```

修改 AppEditPartFactory 类中代码，添加这一对模型与控制器：

```
public class AppEditPartFactory implements EditPartFactory {

    @Override
    public EditPart createEditPart(EditPart context, Object model) {
        ……
        if(model instanceof ArrowConnectionModel) {
            ArrowConnectionEditPart editPart = new
ArrowConnectionEditPart();
```

```
            editPart.setModel(model);
            return editPart;
        }

        part.setModel(model);
        return part;
    }

}
```

然后在调色板上追加创建工具，修改 MyGraphicalEditor 的 getPaletteRoot() 方法，代码如下：

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette {
    ……
    @Override
    protected PaletteRoot getPaletteRoot() {
        ……
                ConnectionCreationToolEntry arrowConnection = new
ConnectionCreationToolEntry(
                "箭头", "带箭头的直线", new SimpleFactory(
                        ArrowConnectionModel.class),
                        Activator.getImageDescriptor("icons/arrow.gif"),

    Activator.getImageDescriptor("icons/arrow.gif"));

        connectionGroup.add(arrowConnection);
    }
    ……
}
```

运行效果如下：

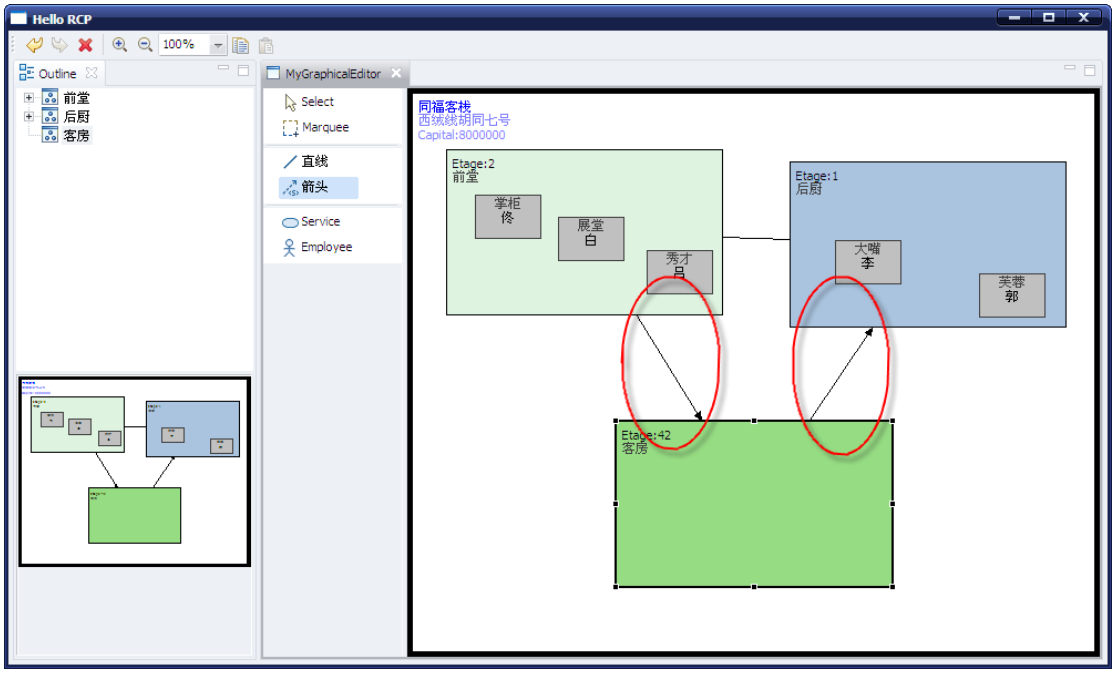


图39 带箭头的连线

下面为连线增加路由点。

需要为 PolylineConnection 设置新的 ConnectionRouter，因缺省的是 ConnectionRouter.NULL。可以找一个 ConnectionRouter 的子类。此处设为 BendpointConnectionRouter。支持路由点即由此决定。

下面看其如何取得并设置路由点：

首先在方法 setConstraint(Connection connection, Object constraint)的实现里，以 connection 为简直，将所有 connection 的 constraint 放到一个 Map 中去；

然后看方法：route(Connection conn)，用来具体设置路由点，它先以 connection 为键值，从 Map 里取出所有 constraint，把它强制为 List，最后从 List 中取出的对象强制为 BendPoint 对象。

可见，在 BendpointConnectionRouter 为连接路由的情况下，可以如下设置路由点：

```
List list = new ArrayList();
list.add(new AbsoluteBendpoint(10,30));
connection.setRoutingConstraint(list);
```

其中，AbsoluteBendpoint 为 BendPoint 类的一个子类。

首先修改模型，添加属性记录连接线的路由点。修改 `AbstractConnectionModel` 类，使所有线都支持路由点，并令其继承 Node，从

而在后面实现事件通知。

```
public abstract class AbstractConnectionModel extends Node{

    ……

    public static final String P_CONSTRAINT = "p_constraint";

    private List<Bendpoint> constraints = new ArrayList<Bendpoint>();

    public List<Bendpoint> getConstraints() {
        return constraints;
    }

    public void addConstraint(int index, Bendpoint bendpoint) {
        if(!constraints.contains(bendpoint)) {
            constraints.add(index, bendpoint);
            firePropertyChange(P_CONSTRAINT, null, bendpoint);
        }
    }

    public void setConstraint(int index, Bendpoint bendpoint) {
        if(!constraints.contains(bendpoint)) {
            constraints.set(index, bendpoint);
            firePropertyChange(P_CONSTRAINT, null, bendpoint);
        }
    }

    public void removeConstraint(int index) {
        Bendpoint bendpoint = constraints.get(index);
        constraints.remove(index);
        firePropertyChange(P_CONSTRAINT, bendpoint, null);
    }
    ……
}
```

注意，为保证操作后的路由点显示正确，需在操作时保证操作节点的顺序。因此，上面三个处理方法都是基于 index 实现的。

下面实现 Policy 和 Command。

将 BendpointEditPolicy 安装在 AbstractConnectionModelEditPart，代码如下：

```
public class AbstractConnectionEditPart extends
        org.eclipse.gef.editparts.AbstractConnectionEditPart {
```

```
    @Override
    protected void createEditPolicies() {
        // TODO Auto-generated method stub
        ……
        installEditPolicy(EditPolicy.CONNECTION_BENDPOINTS_ROLE,
                new ConnectionBendpointEditPolicy());
    }
}
```

并新建 ConnectionBendpointEditPolicy，来实现我们自己的 Policy，代码如下：

```
public class CreateBendPointCommand extends Command {

    private AbstractConnectionModel model;
    private AbsoluteBendpoint point;

    private int index;

    public CreateBendPointCommand(AbstractConnectionModel model, Point
point, int index) {
        super();
        this.model = model;
        this.point = new AbsoluteBendpoint(point);
        this.index = index;
    }

    @Override
    public void execute() {
        model.addConstraint(index, point);
    }

    @Override
    public void undo() {
        model.removeConstraint(index);
    }
}
```

新建 RemoveBendPointCommand 类如下：

```
public class RemoveBendPointCommand extends Command {

    private AbstractConnectionModel model;
    private AbsoluteBendpoint point;
```

```
    private int index;

    public RemoveBendPointCommand(AbstractConnectionModel model, Point
point, int index) {
        super();
        this.model = model;
        this.point = new AbsoluteBendpoint(point);
        this.index = index;
    }

    @Override
    public void execute() {
        model.removeConstraint(index);
    }

    @Override
    public void undo() {
        model.addConstraint(index, point);
    }

}
```

新建 MoveBendPointCommand 类如下：

```
public class MoveBendPointCommand extends Command {

    private AbstractConnectionModel model;
    private AbsoluteBendpoint point;

    private int index;

  public MoveBendPointCommand(AbstractConnectionModel model, int index,
Point point) {
        super();
        this.model = model;
        this.index = index;
        this.point = new AbsoluteBendpoint(point);
    }

    @Override
    public void execute() {
        model.removeConstraint(index);
```

```
        model.addConstraint(index, point);
    }

    @Override
    public void undo() {
        model.removeConstraint(index);
        model.addConstraint(index, point);
    }

}
```

然后完成 Policy，新建 ConnectionBendpointEditPolicy 代码如下：

```
public class ConnectionBendpointEditPolicy extends BendpointEditPolicy
{

    @Override
    protected Command getCreateBendpointCommand(BendpointRequest
request) {
        // TODO Auto-generated method stub
        CreateBendPointCommand command = new CreateBendPointCommand(

    (AbstractConnectionModel)request.getSource().getModel(),
                request.getLocation(), request.getIndex());
        return command;
    }

    @Override
    protected Command getDeleteBendpointCommand(BendpointRequest
request) {
        // TODO Auto-generated method stub
        RemoveBendPointCommand command = new RemoveBendPointCommand(

    (AbstractConnectionModel)request.getSource().getModel(),
                request.getLocation(), request.getIndex());
        return command;
    }

    @Override
    protected Command getMoveBendpointCommand(BendpointRequest request)
{
        // TODO Auto-generated method stub
        MoveBendPointCommand command = new MoveBendPointCommand(
```

```
    (AbstractConnectionModel)request.getSource().getModel(),
              request.getIndex(), request.getLocation());
        return command;
    }

}
```

为支持路由，需要给连接线设置一个 ConnectionRouter，在我们自己的 AbstractConnectionEditPart 中添加 createFigure()方法：

```
public class AbstractConnectionEditPart extends
        org.eclipse.gef.editparts.AbstractConnectionEditPart {

    ……
    @Override
    protected IFigure createFigure() {
        PolylineConnection connection = new PolylineConnection();
        connection.setConnectionRouter(new
BendpointConnectionRouter());
        return connection;
    }


}
```

此时已可拖拽出路由点，但放开后又恢复，因为虽然完成了写了监听，但没有完成时间的监听与处理，继续修改 AbstractConnectionEditPart，实现 PropertyChangeListener 监听代码如下：

```
    public void propertyChange(PropertyChangeEvent evt) {

    if(evt.getPropertyName().equals(AbstractConnectionModel.P_CONSTRA
INT)) {
            refreshConnectionRoutes();
        }
    }
```

并添加相应的 refreshConnectionRoutes()方法：

```
    private void refreshConnectionRoutes() {
        // TODO Auto-generated method stub
        List<Bendpoint> constraints =
((AbstractConnectionModel)getModel()).getContraints();
```

```
    ((PolylineConnection)getFigure()).setRoutingConstraint(constraint
s);
    }
```

此时仍未进入 propertyChange()方法，因为未加入监听，添加如下代码并令 AbstractConnectionEditPart 实现 PropertyChangeListener 接口：

```
    @Override
    public void activate() {
        super.activate();

    ((AbstractConnectionModel)getModel()).addPropertyChangeListener(t
his);
    }

    @Override
    public void deactivate() {

    ((AbstractConnectionModel)getModel()).removePropertyChangeListene
r(this);
    }
```

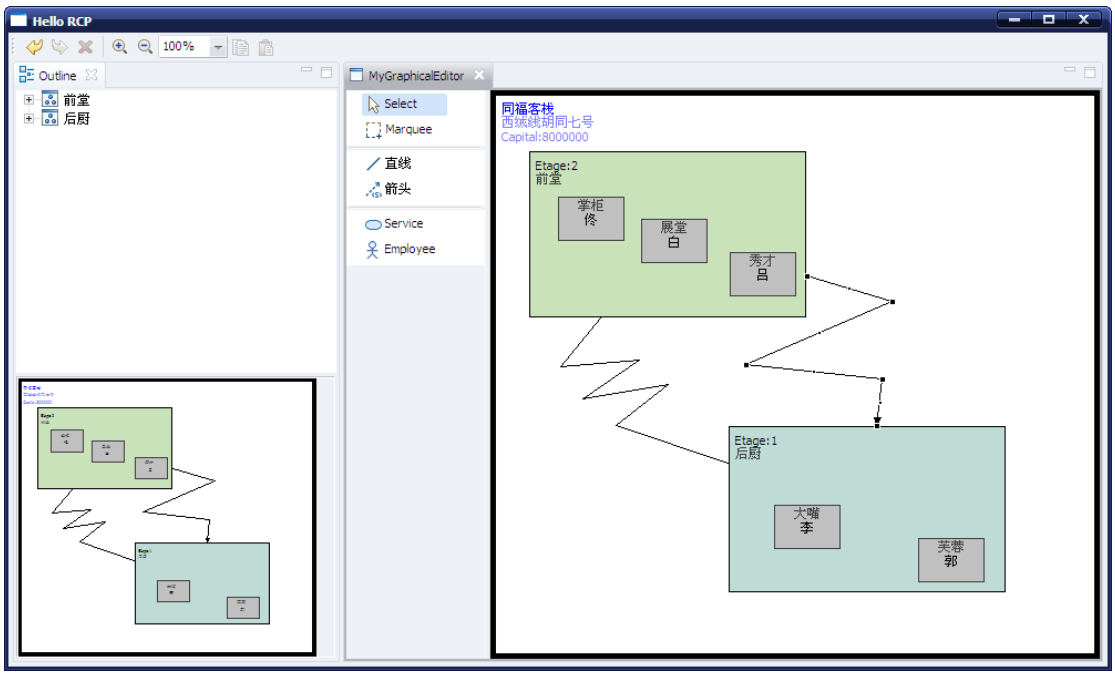


图40 添加连线的路由点

此阶段代码为 WBHGEF16。

# 18. 直接编辑（Direct Edit）

**直接编辑的支持**

想要支持直接编辑功能，也需要安装 Policy，写 Command。

写一个子类，NodeDirectEditPolicy，继承 EditPolicy 中的 DirectEditPolicy，然后将其安装于 NodeEditPart 中。

新建 NodeDirectEditPolicy 代码如下：

```
public class NodeDirectEditPolicy extends DirectEditPolicy {

    @Override
    protected Command getDirectEditCommand(DirectEditRequest request) {
        // TODO Auto-generated method stub
        Text text = (Text)request.getCellEditor().getControl();
        DirectEditNodeCommand command = new DirectEditNodeCommand(
                (Node)getHost().getModel(), text.getText().trim());
        return command;
    }

    @Override
    protected void showCurrentEditValue(DirectEditRequest request) {
        // TODO Auto-generated method stub

    }
}
```

编辑完成相应 Command，主要需要：当前编辑的 Node 对象，及修改后的值，故新建 DirectEditNodeCommand 代码如下：

```
public class DirectEditNodeCommand extends Command {

    private Node model;
    private String text;
    private String old;

    public DirectEditNodeCommand(Node model, String text) {
```

```
        super();
        this.model = model;
        this.text = text;
    }

    @Override
    public void execute() {
        old = model.getName();
        model.setName(text);
    }

    @Override
    public void undo() {
        text = model.getName();
        model.setName(old);
    }

}
```

后面再讲 ShowCurrentEditValue()方法。

此处假设编辑的控件是 text 框，具体由选择的 CellEditor 决定，此处选择 TextCellEditor。在做树及表编辑时也将用到。

通常 GEF 上的编辑在单击鼠标时可以显示，但此处我们在 Policy 中把 CellEditor 的 Control 强制转换为 Text，并没有通知 GEF。

此外，也并未告诉 GEF 这个 Text 的位置、大小等。

这样，则要引入另一个类：DirectoryEditManager 类。

DirectoryEditManager 类是一个抽象类，需要给出具体实现。在实现我们的 DirectoryEditManager 类前，首先看下其构造方法。

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

其中：

@param source the source edit part：要编辑的模型的 EditPart

@param editorType the cell editor type：编辑单元类，如 TextCellEditor.class

@param locator the locator：决定编辑控件的大小和位置，需要自己实现

下面实现 CellEditorLocator 类，新建 tool 包，新建 NodeCellEditorLocator 类代码如下：

```
public class NodeCellEditorLocator implements CellEditorLocator {
```

```
    private Node model;

    public NodeCellEditorLocator(Node model) {
        super();
        this.model = model;
    }

    @Override
    public void relocate(CellEditor celleditor) {
        // TODO Auto-generated method stub
        Text text = (Text)celleditor.getControl();
        Rectangle constraints = model.getLayout();
        int a = 0;
        int b = 0;
        if(model.getParent() != null) {
            a =  model.getParent().getLayout().x;
            b = model.getParent().getLayout().y;
        }

        text.setBounds(constraints.x + a, constraints.y + b,
                constraints.width, constraints.height);
    }
}
```

此处增加构造方法，传入编辑的模型，让编辑控件的大小刚好等于编辑模型的图形大小，且位置正好覆盖原位置。

下面实现 DirectEditManager 类：

```
public class NodeDirectEditManager extends DirectEditManager {

    public NodeDirectEditManager(GraphicalEditPart source, Class
editorType,
            CellEditorLocator locator) {
        super(source, editorType, locator);
        // TODO Auto-generated constructor stub
    }

    @Override
    protected void initCellEditor() {
        // TODO Auto-generated method stub
        Text text = (Text)getCellEditor().getControl();
```

```
        Node model = (Node)getEditPart().getModel();
        text.setText(model.getName());
    }
}
```

这便完成让编辑控件的初始值为模型的 text 值，下面决定其放置哪里。

### 放置 EditManager

在 GEF 的 EditPart 中，使用 performRequest()方法处理此事，将 EditManager 放于此。

修改 AppAbstractEditPart 代码如下：

```
public abstract class AppAbstractEditPart extends
AbstractGraphicalEditPart
        implements PropertyChangeListener {
    ……
    public void performRequest(Request req) {
        ……
        if(req.getType().equals(RequestConstants.REQ_DIRECT_EDIT)) {
            NodeDirectEditManager manager = new NodeDirectEditManager(
                    this, TextCellEditor.class,
                    new NodeCellEditorLocator((Node)getModel()));
            manager.show();
        } else {
            super.performRequest(req);
        }
    }

}
```

运行效果如下：

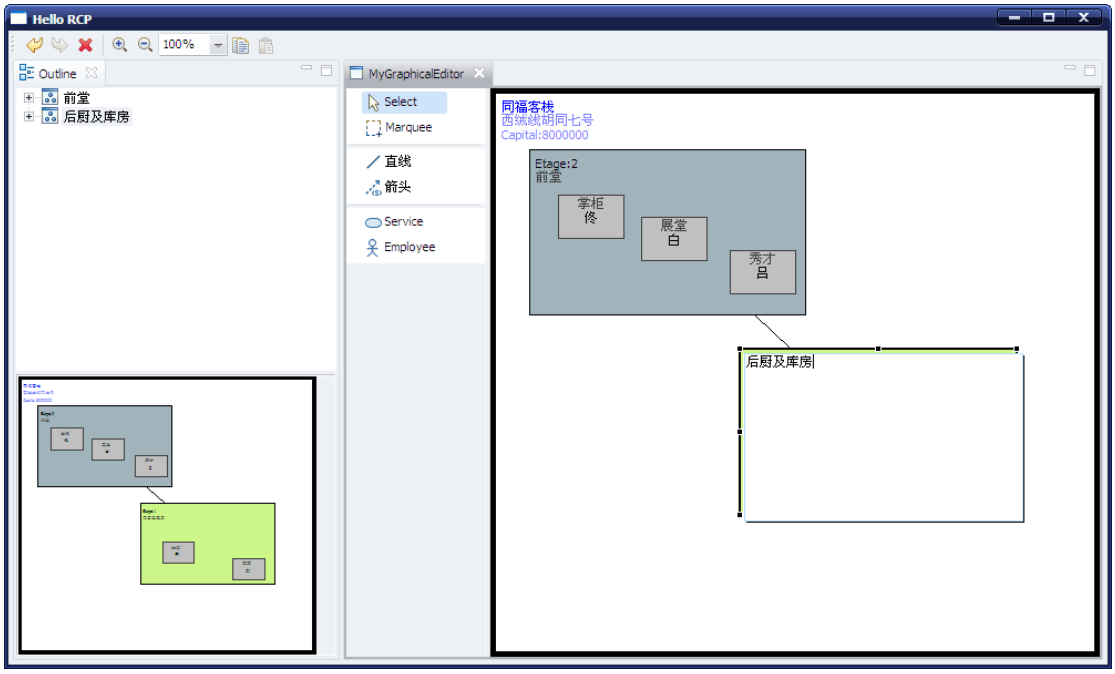


图41 直接编辑效果

此时已在相应的 EditPart 中的 propertyChange()方法中添加名称改变时的通知方法：

```
        if(evt.getPropertyName().equals(Node.PROPERTY_RENAME))
refreshVisuals();
```

但改名后仍无反应，因为之前我们忘记安装最先提到的直接编辑的EditPolicy了，即，在需要修改的模型的相应的EditPart中，修改

```
    @Override
    protected void createEditPolicies() {
        // TODO Auto-generated method stub
        ……
        installEditPolicy(EditPolicy.DIRECT_EDIT_ROLE, new
NodeDirectEditPolicy());
    }
```

此时 Employee 仍不支持名称变更，在属性页内修改也是一样（因为之前我们并未为其添加改名功能，右键环境菜单可见，故此处未刷新，但由于继承抽象类中单击改名方法，故单击时可出现改名框），故先不安装此 EditPolicy。

### ComboboxCellEditor

下面尝试在修改时提供下拉菜单，修改 TextCellEditor 为 ComboboxCellEditor。

手绣修改 NodeDirectEditManager 类，在构造函数内增加一个参数，用于初始化 Combo 元素，修改如下：

```
public class NodeDirectEditManager extends DirectEditManager {

    private String[] items;

    public NodeDirectEditManager(GraphicalEditPart source, Class
editorType,
            CellEditorLocator locator, String[] items) {
        super(source, editorType, locator);
        // TODO Auto-generated constructor stub
        this.items = items;
    }

    @Override
    protected void initCellEditor() {
        // TODO Auto-generated method stub
//      Text text = (Text)getCellEditor().getControl();
        CCombo text = (CCombo)getCellEditor().getControl();
        Node model = (Node)getEditPart().getModel();
        text.setText(model.getName());
    }

    @Override
    protected CellEditor createCellEditorOn(Composite composite) {
        try {
            Constructor constructor =
                ComboBoxCellEditor.class.getConstructor(
                        new Class[] {Composite.class, String[].class});
            return(CellEditor)constructor.newInstance(new Object[] {
                    composite, items});
        } catch (Exception e) {
            return null;
        }
    }

    @Override
    protected boolean isDirty() {

    return !((CCombo)getCellEditor().getControl()).getText().equals(
                ((Node)getEditPart().getModel()).getName());
    }
```

```
}
```

后面两个方法暂未理解。

第一个方法需要重写，如果不写的话，编辑控件无法出来，因其缺省会查询 CellEditor 中是否由一个带 Composite 对象的构造方法，若没有，则返回 null，故若不重写，因 ComboboxCellEditor 无此构造方法，返回 null。

第二个方法需要重写因为 isDirty()总认为是 false，故在 commit()方法中就直接跳出了。

下面修改 AppAbstractEditPart 类，修改 performRequest()方法，即修改编辑器类型，再在 EditManager 中追加一个参数，代码如下：

```
public abstract class AppAbstractEditPart extends
AbstractGraphicalEditPart
        implements PropertyChangeListener {
    ……
    public void performRequest(Request req) {
        ……
        if(req.getType().equals(RequestConstants.REQ_DIRECT_EDIT)) {
            NodeDirectEditManager manager = new NodeDirectEditManager(
                    this, TextCellEditor.class,
//                  new NodeCellEditorLocator((Node)getModel()));
                    new NodeCellEditorLocator((Node)getModel()),
                    new String[]{"刑捕头","燕小六"});
            manager.show();
        } else {
            super.performRequest(req);
        }
    }

}
```

然后修改 NodeCellEditorLocator 类，需要注意的是，ComboboxCellEditor 中的控件是 CCombo，而不是 Combo 等。

```
public class NodeCellEditorLocator implements CellEditorLocator {
    ……
    @Override
    public void relocate(CellEditor celleditor) {
        // TODO Auto-generated method stub
        //Text text = (Text)celleditor.getControl();
        CCombo text = (CCombo)celleditor.getControl();
        ……
```

```
    }
}
```

并修改 NodeDirectEditPolicy 的控件类型：

```
public class NodeDirectEditPolicy extends DirectEditPolicy {

    @Override
    protected Command getDirectEditCommand(DirectEditRequest request) {
        // TODO Auto-generated method stub
//      Text text = (Text)request.getCellEditor().getControl();
        CCombo text = (CCombo)request.getCellEditor().getControl();
        ……
    }
}
```

运行效果如下：

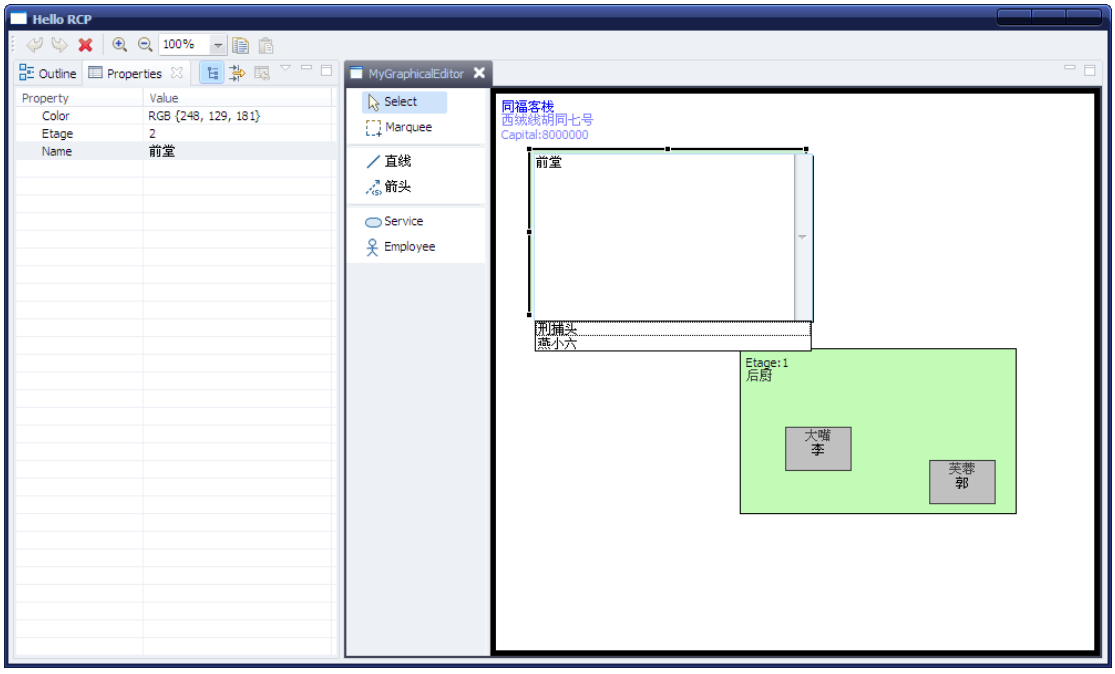


图42 下拉菜单直接编辑效果

对 Perspective 类修改如下，进行简单美化：

```
public class Perspective implements IPerspectiveFactory {

    private static final String ID_TABS_FOLDER = "PropertySheet";

    public void createInitialLayout(IPageLayout layout) {
        String editorArea = layout.getEditorArea();
```

```
//      layout.setEditorAreaVisible(true);
////        layout.addStandaloneView(IPageLayout.ID_OUTLINE, true,
IPageLayout.LEFT, 0.3f, editorArea);
//      IFolderLayout tabs = layout.createFolder(ID_TABS_FOLDER,
IPageLayout.LEFT, 0.2f, editorArea);
//      tabs.addView(IPageLayout.ID_OUTLINE);
//      tabs.addPlaceholder(IPageLayout.ID_PROP_SHEET);
        layout.addView("org.eclipse.ui.views.ContentOutline",
                IPageLayout.LEFT, 0.2f, layout.getEditorArea());
        layout.addView("org.eclipse.ui.views.PropertySheet",
                IPageLayout.BOTTOM, 0.8f, editorArea);
    }
}
```

运行效果如下：

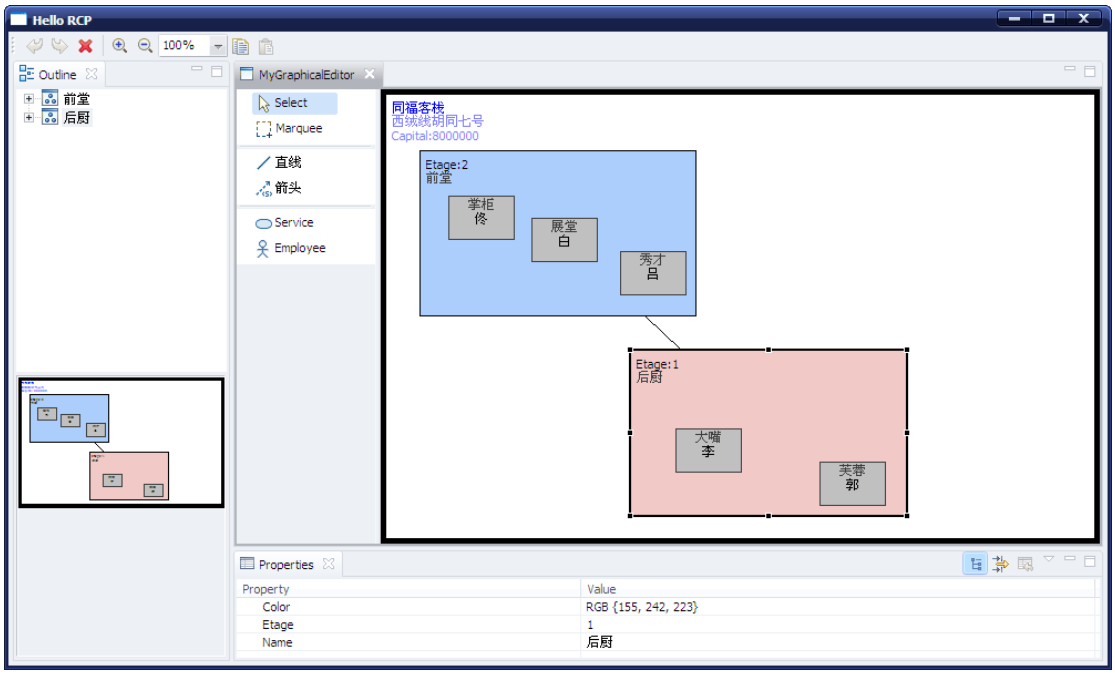


图43 界面美化效果

此阶段代码为 WBHGEF17。

# 19. 变更标记（Dirty）

通常 Eclipse 中 Editor 的内容发生变化时，会有 dirty 标记。目前我们的程序，在模型发生修改时关闭，会提示是否保存，即编辑器已经知道改动。而没有 dirty 标记是因为我们没有通知它。

Eclipse 的 Editor 都是通过 isDirty()方法的返回值判断是否为脏，且 Editor 不是一致轮训 isDirty()方法，而是要事件来触发。又因为所有的操作通过 Command 来完成，且从 Command 堆栈可有 editor 获得。综上，isDirty()返回值由命令栈决定是否为空。不用在 Command 执行和撤销时发出通知，而是使用 GEF 中的 commandStackChanged(EventObject event)方法，在命令栈变化时调用。

从其默认实现可见：

```
public void commandStackChanged(EventObject event) {
    updateActions(stackActions);
}
```

虽然脏标记没有表示出来，但 action 的可用状态总是正确的。

如下图，commandStackChanged()方法被调用的位置可见，在每个由 Command 执行的地方，都发出其事件机制。

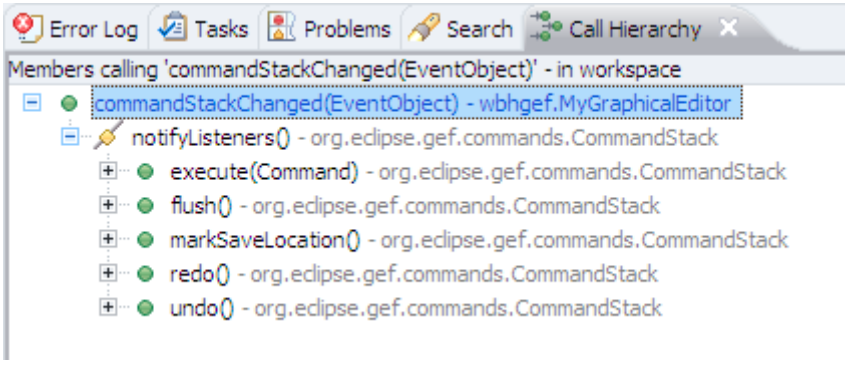


图44 commandStackChanged()方法被调用的位置

在 MyGraphicalEditor 类最后添加如下代码：

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette {
    ……
    @Override
    public boolean isDirty() {
        return getCommandStack().isDirty();
    }

    @Override
```

```
    public void commandStackChanged(EventObject event) {
        super.commandStackChanged(event);
        firePropertyChange(PROP_DIRTY);
    }

}
```

运行效果如下，属性页变化也可反应：

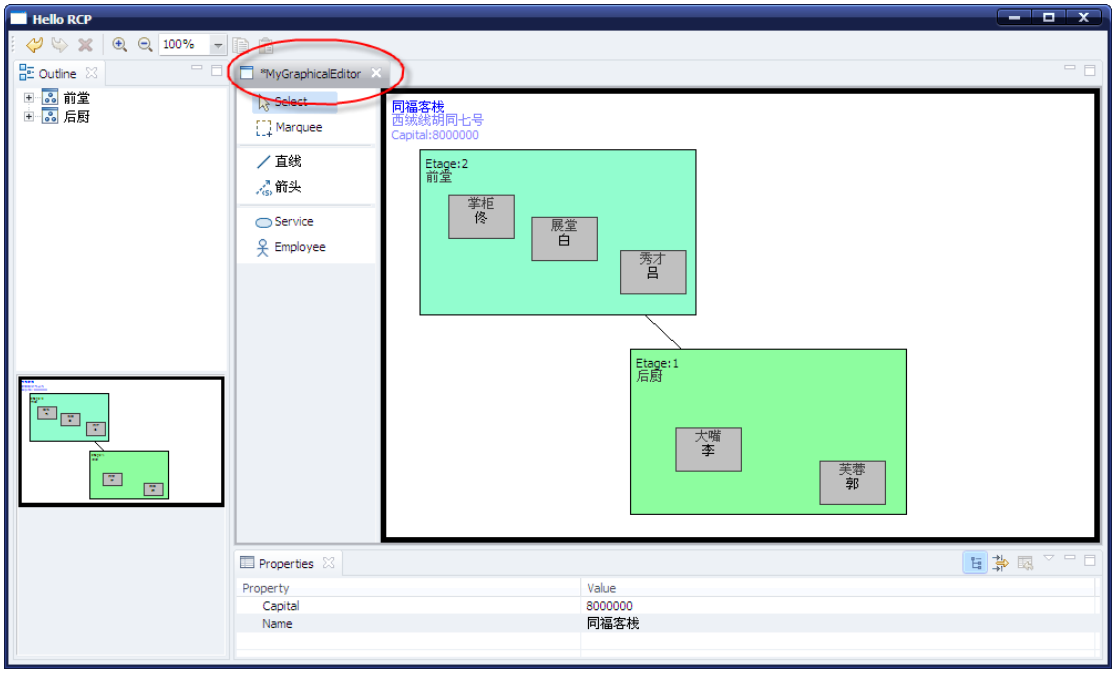


图45 带有 Dirty 标记的编辑区