GEF (Graphical Editing Framework)

Tutorial

Document version: 1.1

October 17, 2007: Initial release

April 26, 2008: Website link updated

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)

SUMMARY

Summary 3
Introduction 4
Part 1: Creation of a RCP plug-in 5
Part 2: Creation of the model 12
Part 3: First interaction with the graph 24
Part 4: Undo/Redo 30
Part 5: Zoom and keyboard shortcuts 36
Part 6: Outline 39
Part 7: Miniature view 45
Part 8: Context menu 47
Part 9: Creating a custom action 49
Wizard creation 49
Command creation 50
Action creation 51
EditPolicy creation 53
Associate the new EditPolicy with the EditParts 53
Last, property activation to update views 54
Part 10: Property Sheet 56
Adding the color of the services in the model *56*
Define a property source 57
Integration of the property sheet *61*
Part 11: Adding new graphical elements 63
Palette insertion 63
Adding a service 64
Adding an employee in a service 69
Part 12 : Drag and drop 72
Part 13 : Cut and paste 74
Conclusion 82
References 82

Introduction

GEF (Graphical Editing Framework) is a Java technology, it is a part of the Eclipse framework developed by IBM. It gives developers a full solution for graphical modeling of a java object model, and it can be used in conjunction with other technologies such as EMF (Eclipse Modeling Framework) or GMF (Graphical Modeling Framework), to be able to gain abstraction levels in application development.

We created this tutorial because we encountered a lot of problems during the development of a GEF application, and moreover, the main documentation we found about GEF on the Internet and in books (like the IBM Redbook) was more about GEF theoretical concepts rather than practical application about how to use GEF classes. That is the main motivation we had in writing this tutorial, that means to be very pragmatic and useful for the programmer, hopefully.

If you prefer a French version, you can access it at this address:
http://www.psykokwak.com/blog/index.php/tag/gef

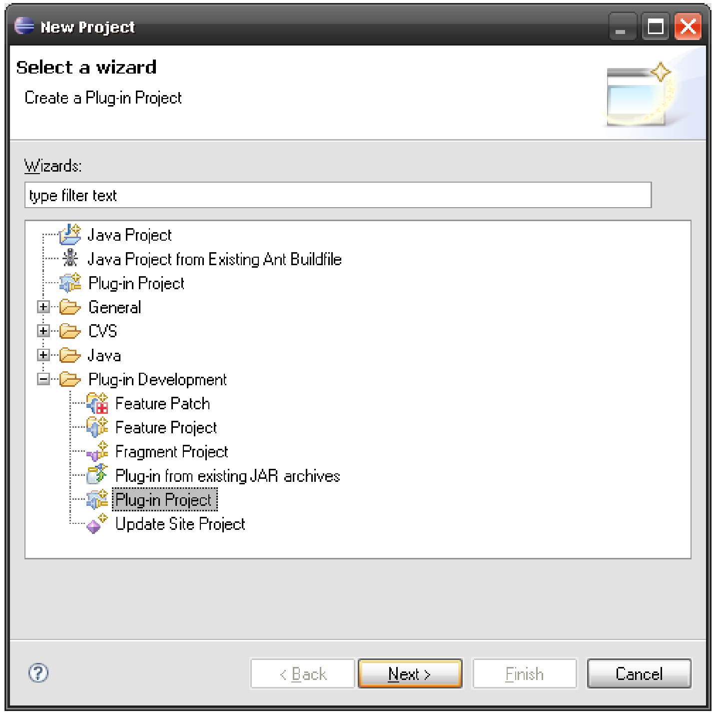
You can download all source codes at the end of each french chapters on the website.

Part 1: Creation of a RCP plug-in

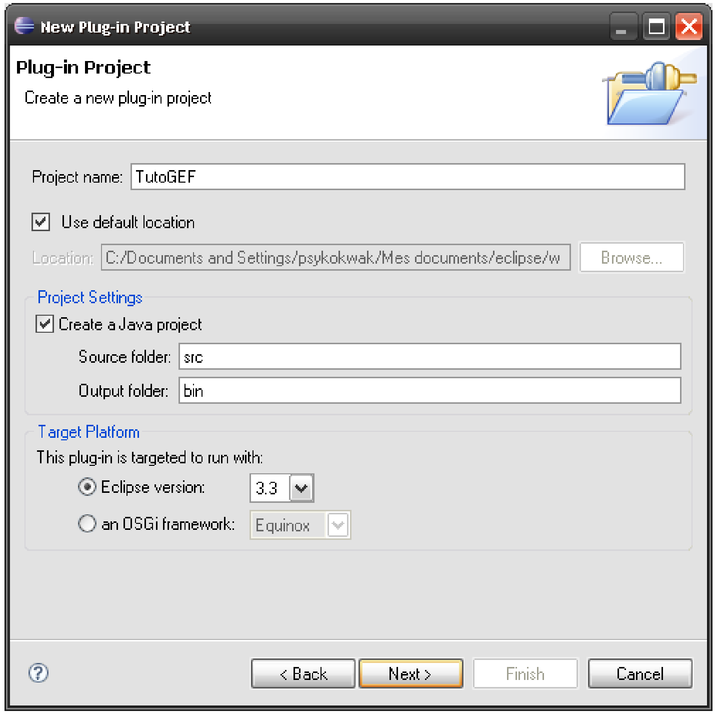
As I had myself huge difficulties to fiddle with GEF for the needs of an Eclipse application, I decided to put together my knowledge on this subject HERE in order that others can benefit from it.

Beforehand, you need to have GEF installed on your system: http://download.eclipse.org/tools/gef/downloads/

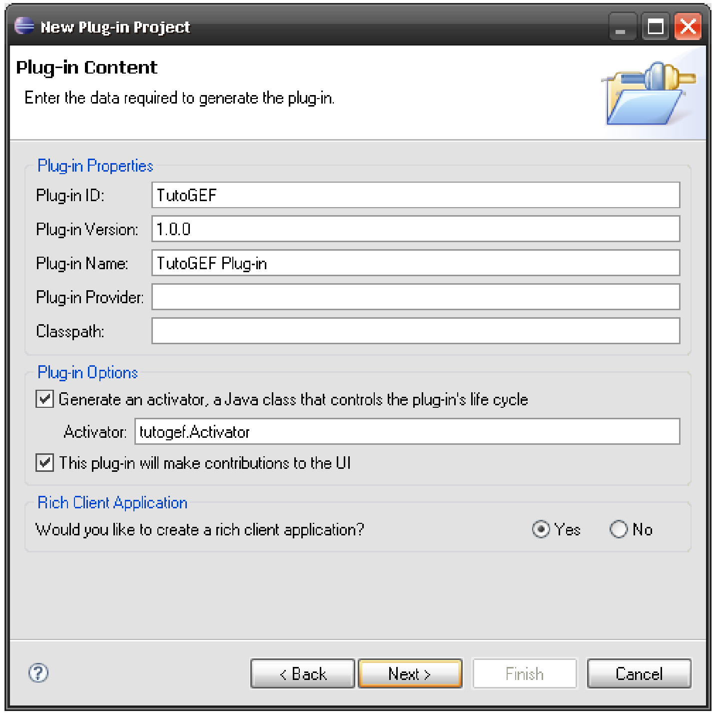
We're going to start on a basis of an RCP (Rich Client Platform) Eclipse plug-in to create an application that uses GEF. Launch Eclipse, then File – New – Project.



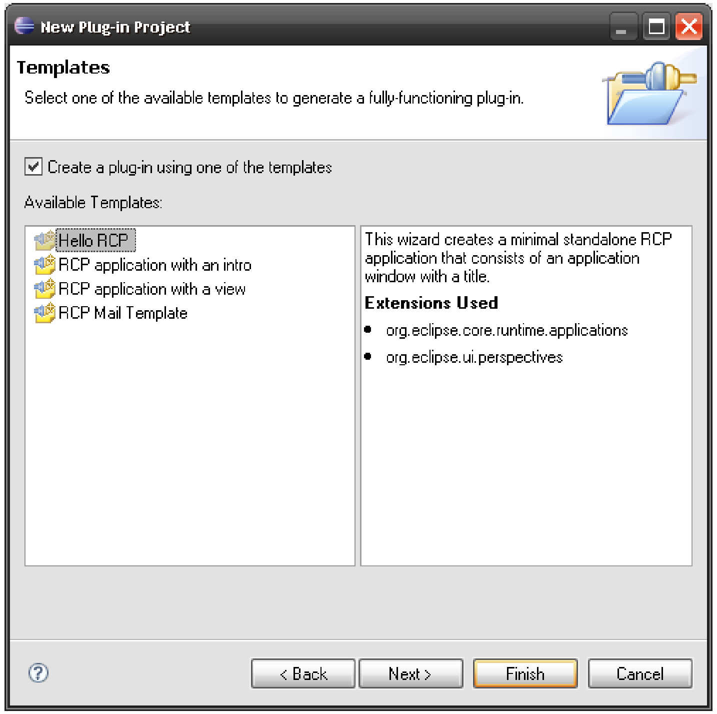
Select "*Plug-in Project*".



Give it a name.



We would like to create a "*Rich Client Application*".



You should choose a template, let's start with "Hello RCP" which is the most basic template.

A popup window may ask you if you wish to open the perspective associated with this project type. Answer "yes". *A perspective represents the way graphical elements are laid out under Eclipse.*

We need to add GEF plug-ins in the list of dependencies of our project. Edit the file "plugin.xml" in the "Dependancies" tab, by clicking "Add".

Now, let's create a new class in the package which inherits from ***org.eclipse.gef.ui.parts.GraphicalEditor.*** This class will define our Graphical Editor. It is a sort of starting point of our work. But first, the EditorPart needs to be integrated into the plug-in…

Add an ID to tell Eclipse that we want to use this class as an extension. Create then a constructor in which we define the EditDomain we want to use.

```
public class MyGraphicalEditor extends GraphicalEditor {

	public static final String ID = "tutogef.mygraphicaleditor";

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

Another class needs to be created, which implements the ***org.eclipse.ui.IEditorInput*** interface. This class describes the EditorPart.

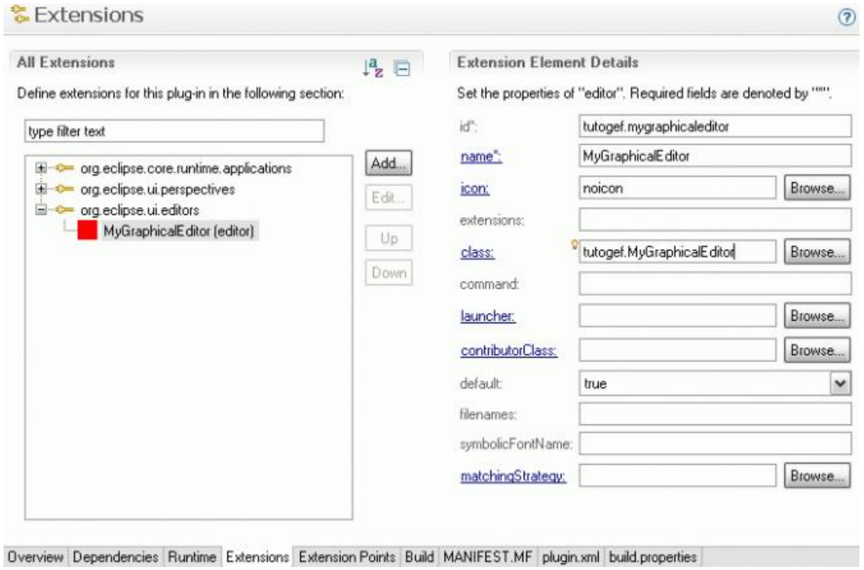
```
package tutogef;

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() {
		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() {
		return ImageDescriptor.getMissingImageDescriptor();
	}

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

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



```
	@Override
	public String getToolTipText() {
		return this.name;
	}

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

Edit “plugin.xml”, in the “Extensions” tab, and add an extension by clicking on “Add”, then choose “*org.eclipse.ui.editor”.*

We set some of the properties as in the screenshot below.



Now, all that is remaining is launching the editor at application start-up. For that, we will overload the *postStartup()* method in *ApplicationWorkbenchAdvisor*:

```
public class ApplicationWorkbenchAdvisor extends WorkbenchAdvisor {
	(...)
	@Override
	public void postStartup() {

		try {
			IWorkbenchPage page =
PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage();
			page.openEditor(new MyEditorInput("TutoGEF"), MyGraphicalEditor.ID, false);
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

You now have a functional RCP plug-in, integrating an Editor ready to welcome GEF.

Part 2: Creation of the model

Now that we have the base to build a GEF graph, we're going to create a basic object model, to be able to display it later. We create a new package that we call "*tutogef.model*".

Next, we write a class (in this package) that will be used as the parent class for all of our model elements. This class will contain the basic properties that all derived classes will be able to use.

```
package tutogef.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;
	}
```

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

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

Next, we write a class derived from *Node*, and that will be the topmost class in the hierarchy. Take the example of a company: it contains several services and each of these employs several persons.

Here are the *Enterprise*, *Service* and *Employe* classes:

```java
package tutogef.model;

public class Entreprise 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;
	}
}
```



```java
package tutogef.model;

public class Service extends Node {
	private int etage;

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

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

```java
package tutogef.model;

public class Employe extends Node {
	private String prenom;

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

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

Each object in the model needs to be associated with a *Figure* (*draw2d*) and an *EditPart*.

We have three classes for the model; we thus need three classes for figures, and three for the EditParts… (*an EditPart is an object that will link its model object and its visual representation*).

Let’s create our *Figure* and *EditPart* derived classes associated to the company (Entreprise) to be able to display it. (We place the figure classes in a package called *tutogef.figure*, and the EditPart ones in *tutogef.editpart*)

```java
package tutogef.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 EntrepriseFigure extends Figure {
	private Label labelName = new Label();
	private Label labelAddress = new Label();
	private Label labelCapital = new Label();
	private XYLayout layout;
```

```
	public EntrepriseFigure() {
		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);
	}
}
```

```
package tutogef.part;

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

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

import tutogef.figure.EntrepriseFigure;
import tutogef.model.Entreprise;
import tutogef.model.Node;


public class EntreprisePart extends AbstractGraphicalEditPart{

	@Override
	protected IFigure createFigure() {
		IFigure figure = new EntrepriseFigure();
		return figure;
	}

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



```
	}

	protected void refreshVisuals(){
		EntrepriseFigure figure = (EntrepriseFigure)getFigure();
		Entreprise model = (Entreprise)getModel();

		figure.setName(model.getName());
		figure.setAddress(model.getAddress());
		figure.setCapital(model.getCapital());
	}

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

Now all that is missing is the “linking” between all this stuff.

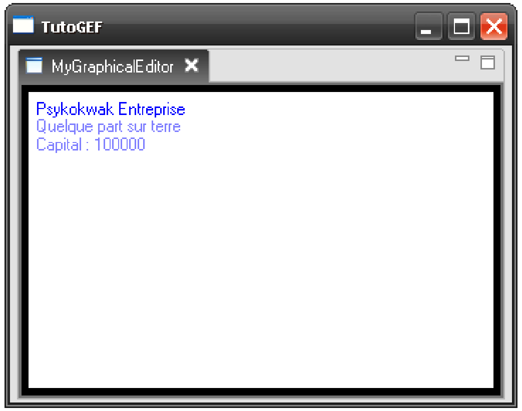
A Factory is needed to manage the EditParts. A Factory is a class that will handle appropriate object creation (the EditPart) depending on what we want to obtain, no matter what is the object class. (*see Design Patterns*). For now, our factory looks like that:

```
package tutogef.part;

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

import tutogef.model.Entreprise;

public class AppEditPartFactory implements EditPartFactory {

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

		if (model instanceof Entreprise) {
			part = new EntreprisePart();
		}

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

In the *MyGraphicalEditor* class from the previous tutorial, we have to overload the *configureGraphicalViewer()* method to tell the editor that we want to use its factory. We also create a method that will handle the creation of the object model. Last, we load the object model in the editor with the *initializeGraphicalViewer()* method.

```
public class MyGraphicalEditor extends GraphicalEditor {
        public Entreprise CreateEntreprise(){
                Entreprise psyEntreprise = new Entreprise();

                psyEntreprise.setName("Psykokwak Entreprise");
                psyEntreprise.setAddress("Quelque part sur terre");
                psyEntreprise.setCapital(100000);

                return psyEntreprise;
        }
        (...)
        protected void configureGraphicalViewer() {
                 super.configureGraphicalViewer();
                 GraphicalViewer viewer = getGraphicalViewer();
                 viewer.setEditPartFactory(new AppEditPartFactory());
        }
        (...)
        protected void initializeGraphicalViewer() {
                GraphicalViewer viewer = getGraphicalViewer();
            viewer.setContents(CreateEntreprise());
        }
}
```

That's it, you may execute… You should obtain this result:



We have to repeat the previous operation for the *Service* and *Employe* classes. Writing its EditPart and its Figure… Here are the *ServiceFigure* and *ServicePart* classes. They look much like the *EntrepriseFigure* and *EntreprisePart* classes…

```
package tutogef.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));

                /** Just for Fun :) **/
                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);
        }
}
```

```
package tutogef.part;

import java.util.List;

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

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

public class ServicePart extends AbstractGraphicalEditPart {

	@Override
	protected IFigure createFigure() {
		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();
	}
}
```

Note the *getModelChildren()* method just over, it needs to be overloaded in the *EntreprisePart* class like this one.

```
public class EntreprisePart extends AbstractGraphicalEditPart {
	(...)
	public List<Node> getModelChildren() {
	    return ((Entreprise)getModel()).getChildrenArray();
	}
}
```


And we start again for Employe…

```
package tutogef.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 EmployeFigure extends Figure {

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

	public EmployeFigure() {
		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);
	}
}
```

```java
package tutogef.part;

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

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

import tutogef.figure.EmployeFigure;
import tutogef.model.Employe;
import tutogef.model.Node;

public class EmployePart extends AbstractGraphicalEditPart{

	@Override
	protected IFigure createFigure() {
		IFigure figure = new EmployeFigure();
		return figure;
	}

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

	}

	protected void refreshVisuals(){
		EmployeFigure figure = (EmployeFigure)getFigure();
		Employe model = (Employe)getModel();

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

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

}
```

(Note that for this class, *getModelChildren()* returns an empty list. It is normal because *Personne* is the lowest class in the hierarchy. It thus has no child.)

Now the factory needs to be modified so that it can manage our new EditParts.

```java
public class AppEditPartFactory implements EditPartFactory {

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

		if (model instanceof Entreprise) {
			part = new EntreprisePart();
		} else if (model instanceof Service) {
			part = new ServicePart();
		} else if (model instanceof Employe) {
			part = new EmployePart();
		}

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

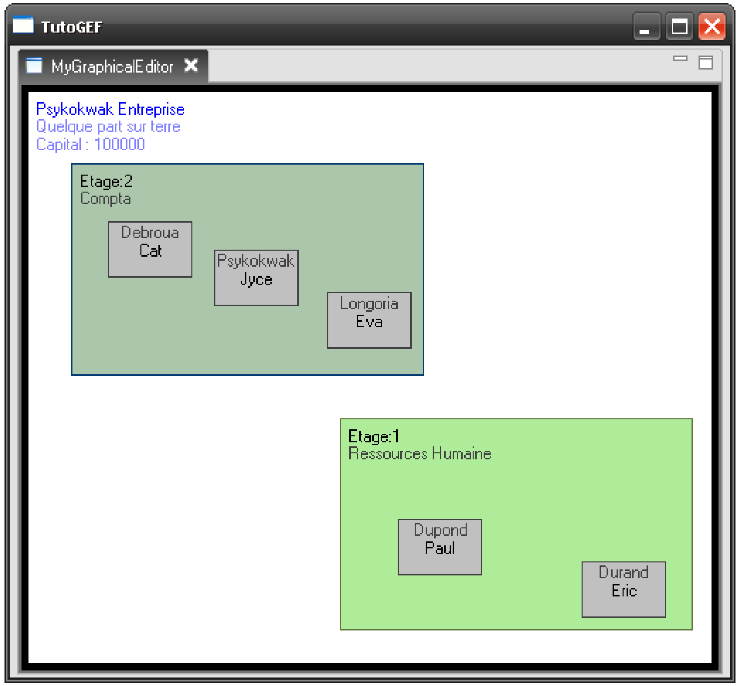
Let’s populate our company with services and employees.

Still in the *CreateEntreprise()* method of the *MyGraphicalEditor* class, we’re going to add services and employees. Note that coordinates and size of each element must be specified. (*coordinates are relative to 0.0 coordinate of the parent)*

```java
   public Entreprise CreateEntreprise(){
		Entreprise psyEntreprise = new Entreprise();

		psyEntreprise.setName("Psykokwak Entreprise");
		psyEntreprise.setAddress("Quelque part sur terre");
		psyEntreprise.setCapital(100000);

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

				Employe employeCat = new Employe();
				employeCat.setName("Debroua");
				employeCat.setPrenom("Cat");
				employeCat.setLayout(new Rectangle(25, 40, 60, 40));
				comptaService.addChild(employeCat);

				Employe employeJyce = new Employe();
				employeJyce.setName("Psykokwak");
				employeJyce.setPrenom("Jyce");
				employeJyce.setLayout(new Rectangle(100, 60, 60, 40));
				comptaService.addChild(employeJyce);

				Employe employeEva = new Employe();
				employeEva.setName("Longoria");
				employeEva.setPrenom("Eva");
				employeEva.setLayout(new Rectangle(180, 90, 60, 40));
				comptaService.addChild(employeEva);
			psyEntreprise.addChild(comptaService);

			Service rhService = new Service();
			rhService.setName("Ressources Humaine");
```

```
            rhService.setEtage(1);
            rhService.setLayout(new Rectangle(220, 230, 250, 150));

                Employe employePaul = new Employe();
                employePaul.setName("Dupond");
                employePaul.setPrenom("Paul");
                employePaul.setLayout(new Rectangle(40, 70, 60, 40));
                rhService.addChild(employePaul);

                Employe employeEric = new Employe();
                employeEric.setName("Durand");
                employeEric.setPrenom("Eric");
                employeEric.setLayout(new Rectangle(170, 100, 60, 40));
                rhService.addChild(employeEric);

        psyEntreprise.addChild(rhService);

        return psyEntreprise;
    }
```

You can now launch the program. You should obtain this (colors are changing at each start):



Part 3: First interaction with the graph

In this third part, we're going to interact with the graph: select a box, move it, resize it…

GEF provides a visual representation of an object model. In our example (simple), we have on top our *Entreprise* which contains a list of *Services* which in turn contain *Employes.*

GEF is based on a **Model – View – Controller (MVC)** architecture. In fact, it could also be called "*Model – Controller – View*"… meaning we distinct between our model (our company) and our view (our graph with its boxes…). The controller is the referee in the middle! It drives the view depending on the model and it modifies the model depending on actions carried on the view…

Java packages in this tutorial indeed represent the classes depending on their role in the MVC model.

The GEF wiki (http://wiki.eclipse.org/index.php/GEF_Description) explains in more detail how GEF works (*that being said, you have all my acknowledgments if you manage to understand everything in one go…)*

In this tutorial part, We will see how to interact with the graph, how the controller executes commands, how commands modify the model and how the model informs the view that it has changed so that the view can update itself. *Please don't run out of here, it is simpler than it seems…*

We want to be able to move our services in the company and the employees in their service. For this, it is necessary to create a *Command* for Employe and one for Service. (Classes are created in a new package, *tutogef.commands*)

Beforehand, we create a little abstract stub class in order to avoid some problems later:

```
package tutogef.commands;

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);
}
```

Then we write the two classes inheriting from our abstract class for Employe and Service (they are our commands).

```
package tutogef.commands;

import org.eclipse.draw2d.geometry.Rectangle;

import tutogef.model.Employe;

public class EmployeChangeLayoutCommand extends AbstractLayoutCommand {
	private Employe model;
	private Rectangle layout;

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

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

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

```
package tutogef.commands;

import org.eclipse.draw2d.geometry.Rectangle;

import tutogef.model.Service;

public class ServiceChangeLayoutCommand extends AbstractLayoutCommand {
	private Service model;
	private Rectangle layout;

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

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

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


These commands will be called by an EditPolicy which will later be applied to our objects' EditParts.

We create this class as shown in another package that we call *tutogef.editpolicies*:

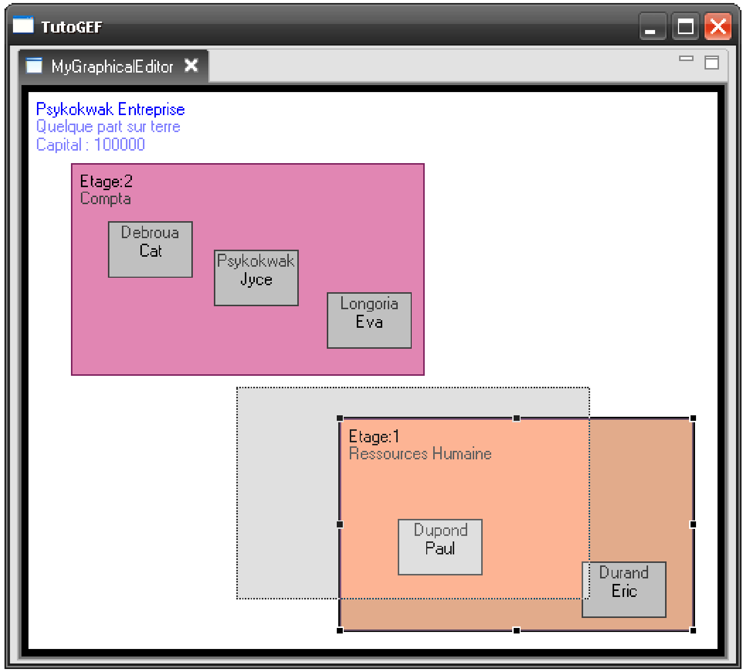
```
package tutogef.editpolicies;

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 tutogef.commands.AbstractLayoutCommand;
import tutogef.commands.EmployeChangeLayoutCommand;
import tutogef.commands.ServiceChangeLayoutCommand;
import tutogef.part.EmployePart;
import tutogef.part.ServicePart;

public class AppEditLayoutPolicy extends XYLayoutEditPolicy {

	@Override
	protected Command createChangeConstraintCommand(EditPart child, Object constraint) {

		AbstractLayoutCommand command = null;

		if (child instanceof EmployePart) {
			command = new EmployeChangeLayoutCommand();
		} 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;
	}
}
```

Now, we need to apply this EditPolicy to the EditParts we're interested in.

In the **EntreprisePart** and **ServicePart** classes, a line has to be added in the *createEditPolicies()* method:

```
	protected void createEditPolicies() {
		installEditPolicy(EditPolicy.LAYOUT_ROLE, new AppEditLayoutPolicy());
	}
```

Then launch the program, just for fun.



It is possible to select a box (even several with CTRL key), and to move and resize it. But?!!

They don't want to stay in place and return to their original position… Why?

The loop is not yet finished: the view updates the model, but the model doesn't update the view (*seems weird said that way)*…

What is needed is Listeners put on our EditParts so that they are able to detect when the model changes and update the view accordingly.

Let's first start by modifying the Node class to activate events when properties are changed.

```
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);
	}
}
```

Then we modify the *setLayout()* method so that it "fires" a property when it's called.

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

From now, when a model element is moved, it triggers an event. But for now, no one will receive it.

We'll write an abstract class that will handle events management. Then we'll make our EditParts derive from this class.

```
package tutogef.part;

import java.beans.PropertyChangeListener;
import org.eclipse.gef.editparts.AbstractGraphicalEditPart;

import tutogef.model.Node;

public abstract class AppAbstractEditPart extends AbstractGraphicalEditPart implements
PropertyChangeListener {
	public void activate() {
		super.activate();
		((Node) getModel()).addPropertyChangeListener(this);
	}

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

Then, for the *EmployePart, ServicePart and EntreprisePart* classes, class declaration is changed for:

```
//Remplacez XXXXPart par le nom de la class.
public class XXXXPart extends AppAbstractEditPart{
(...)
}
```

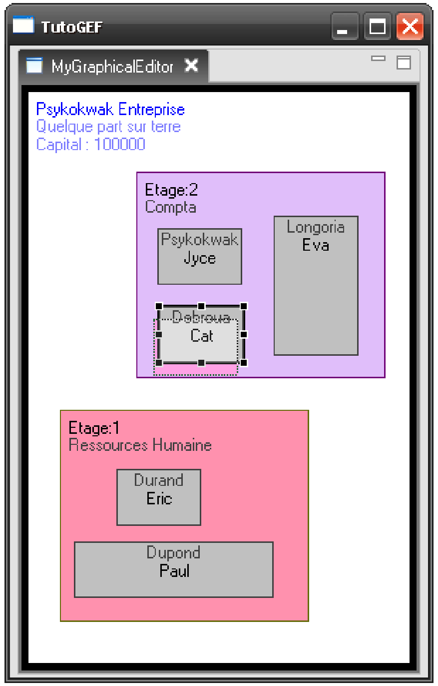
Also, this method has to be appended to these three classes:

```
@Override
public void propertyChange(PropertyChangeEvent evt) {
    if (evt.getPropertyName().equals(Node.PROPERTY_LAYOUT)) refreshVisuals();
}
```

This method will receive the event last triggered by *firePropertyChange()*.

If the event matches a property of type PROPERTY_LAYOUT, then we refresh the display of the model element that has changed.

Launch the program and move a service or an employee.



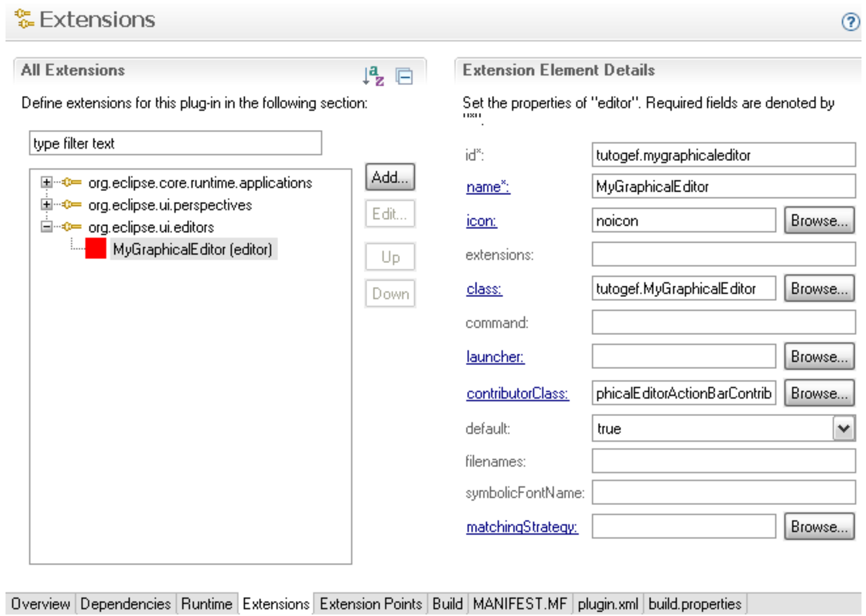
Part 4: Undo/Redo

Now that we're able to interact with the graph, we're going to add management functionalities, like the undo/redo, handled fully by GEF, as well as component removal. At first, a toolbar will be added in the editor, that will contain action buttons to be implemented.

To add a toolbar to an editor, we need to add a *Contributor* to it. We're going to create a class that will be called **MyGraphicalEditorActionBarContributor** that will be derived from **ActionBarContributor**.

```
package tutogef;

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

public class MyGraphicalEditorActionBarContributor extends ActionBarContributor {

    @Override
    protected void buildActions() {
    }

    @Override
    protected void declareGlobalActionKeys() {
    }
}
```

Next, we edit "plugin.xml". In the Extensions tab, add the class path in **contributorClass** for our editor.



Now, all that is remaining is to display the (empty) toolbar. For that, in the *preWindowOpen()* method of the **ApplicationWorkbenchWindowAdvisor** class, we have to add (at the end):

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

We're going to add undo/redo support in the editor.

In the **MyGraphicalEditorActionBarContributor** class, we add this:

```
public class MyGraphicalEditorActionBarContributor extends ActionBarContributor {

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

    (...)

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

The goal of this is to add undo/redo controls in the bar and configure their associated action (handled by GEF).

All we have to do now is to configure the commands of interest, ie. **ServiceChangeLayoutCommand** and **EmployeChangeLayoutCommand**:

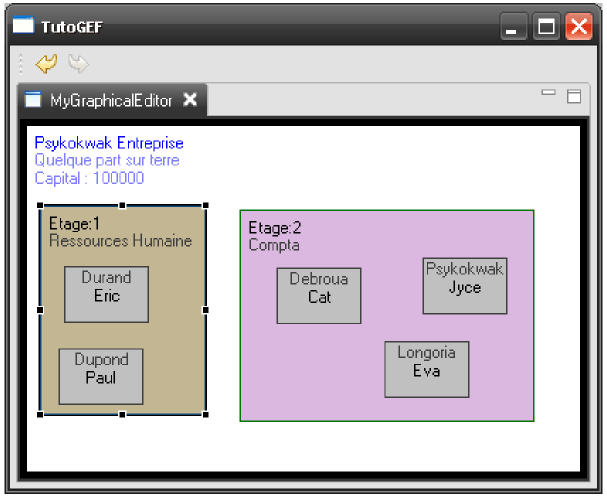
```
public class EmployeChangeLayoutCommand extends AbstractLayoutCommand {
        (...)
        private Rectangle oldLayout;

        (...)

        public void setModel(Object model) {
                this.model = (Employe)model;
                // On sauvegarde l'ancien layout avant de le changer.
                this.oldLayout = ((Employe)model).getLayout();
        }

        // Méthode hérité appelé lors de l'action undo.
        public void undo() {
                this.model.setLayout(this.oldLayout);
        }
}
```

```
public class ServiceChangeLayoutCommand extends AbstractLayoutCommand {
        (...)
        private Rectangle oldLayout;

        (...)

        public void setModel(Object model) {
                this.model = (Service)model;
                // On sauvegarde l'ancien layout avant de le changer.
```



```
                this.oldLayout = ((Service)model).getLayout();
        }

        // Méthode hérité appelé lors de l'action undo.
        public void undo() {
                this.model.setLayout(this.oldLayout);
        }
}
```

Launch the program. You should have the two undo/redo arrows.



Let's now add delete support. We wish to be able to delete a complete service, or just an employee. The principle is roughly the same than for undo/redo:

- Add the toolbar control
- Create the command
- Create a new policy using the new command
- Apply this policy in **ServicePart** and **EmployePart**.
- Trigger an event in *addChild()* and *removeChild()* in **Node**, then update the view by calling *refreshChildren()* in the EditPart of *Entreprise* and *Service*.

In **MyGraphicalEditorActionBarContributor**, we add the action and the "delete" button:

```
public class MyGraphicalEditorActionBarContributor extends ActionBarContributor {

	@Override
	protected void buildActions() {
		(...)
		addRetargetAction(new DeleteRetargetAction());
	}

	(...)

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

The command is all that simple, here is the code:

```
package tutogef.commands;

import org.eclipse.gef.commands.Command;

import tutogef.model.Node;

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

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

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

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

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

Then we create the associated policy:

```
package tutogef.editpolicies;

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

import tutogef.commands.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;
	}
}
```

You have noticed that the principle is the same than for undo/redo…

We install this policy inside **ServicePart** and **EmployePart**:

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

We create a new property in the **Node** class, then we trigger an event for this property change in *addChild()* and *removeChild()*:

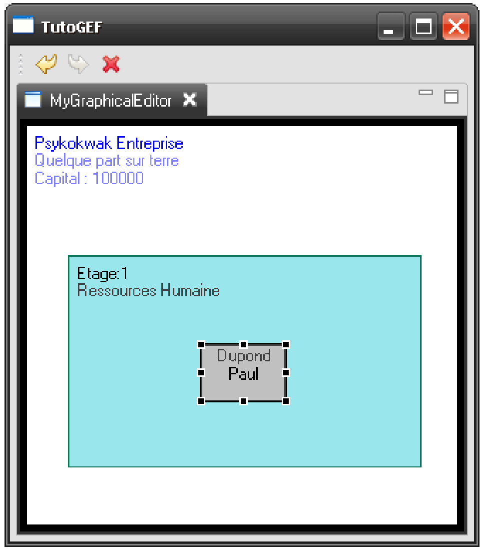
```
public class Node {
	(...)
	public static final String PROPERTY_ADD = "NodeAddChild";
	public static final String PROPERTY_REMOVE = "NodeRemoveChild";
	(...)

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

All that needs to be done is refresh the view when the model changes:

```
public class EntreprisePart extends AppAbstractEditPart{
	(...)
	@Override
	public void propertyChange(PropertyChangeEvent evt) {
	    if (evt.getPropertyName().equals(Node.PROPERTY_ADD)) refreshChildren();
	    if (evt.getPropertyName().equals(Node.PROPERTY_REMOVE)) refreshChildren();
	}
}
```

```
public class ServicePart extends AppAbstractEditPart {
	(...)
	@Override
	public void propertyChange(PropertyChangeEvent evt) {
	    (...)
	    if (evt.getPropertyName().equals(Node.PROPERTY_ADD)) refreshChildren();
	    if (evt.getPropertyName().equals(Node.PROPERTY_REMOVE)) refreshChildren();
	}
```

Well done. You can launch the program and sit back in your chair contemplating the result…



Part 5: Zoom and keyboard shortcuts

In this tutorial, we're going to add a few simple and little things like zooming the graph, then we will associate key presses to these actions.

Zooming is an action, like delete and undo/redo. We will thus have to add the action to the toolbar.

That being said, zooming is an action on the graph itself, and not on the model like previous actions. It is therefore **MyGraphicalEditor** that will be modified.

Let's start here.

We go in the *configureGraphicalViewer()* method, see how to ask for using an EditPart integrating zoom functions, how to add it to the graph, and finally how to create the zoom functions (more or less).

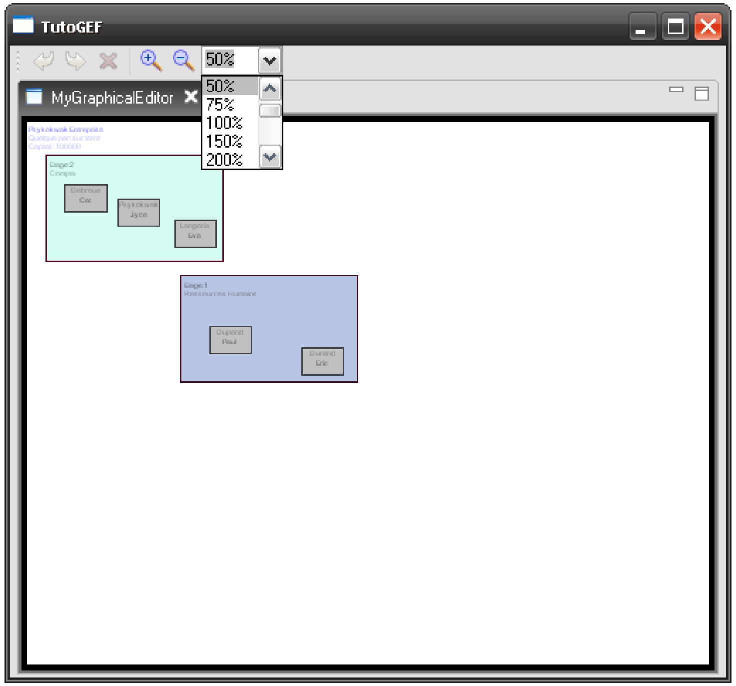
```
public class MyGraphicalEditor extends GraphicalEditor {
	(...)
	protected void configureGraphicalViewer() {
		double[] zoomLevels;
		ArrayList<String> zoomContributions;
		 (...)
		 ScalableRootEditPart rootEditPart = new ScalableRootEditPart();
		 viewer.setRootEditPart(rootEditPart);

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

		 // La liste des zooms possible. 1 = 100%
		 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);

		 // On ajoute certains zooms prédéfinis
		 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()).getZoomManager();
		else
			return super.getAdapter(type);
	}
}
```

We now only have to add our action in the toolbar.

```
public class MyGraphicalEditorActionBarContributor extends ActionBarContributor {
```

```
    @Override
    protected void buildActions() {
        (...)
        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()));
    }
}
```

Well done.



All keyboard freaks out there like to use keyboard shortcuts. They will be pleased to hear that we think about them by mapping keys + and – for zooming, as well as **del** for removal.

So simple:

All that is needed is a “key handler”, add to it all wanted associations with their respective actions, and finish by adding this key handler with our viewer.


```
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));

        // On peut meme zoomer avec la molette de la souris.
        viewer.setProperty(
                MouseWheelHandler.KeyGenerator.getKey(SWT.NONE),
                MouseWheelZoomHandler.SINGLETON);

        viewer.setKeyHandler(keyHandler);

    }

    (...)
}
```

Launch the application and test the keyboard shortcuts.

Part 6: Outline

GEF provides several view types for an editor. Up until now, we had our editor with our little cute boxes. Wouldn't it be pleasing to add a view to it used to show the graph as a tree?

No need to tell it twice…

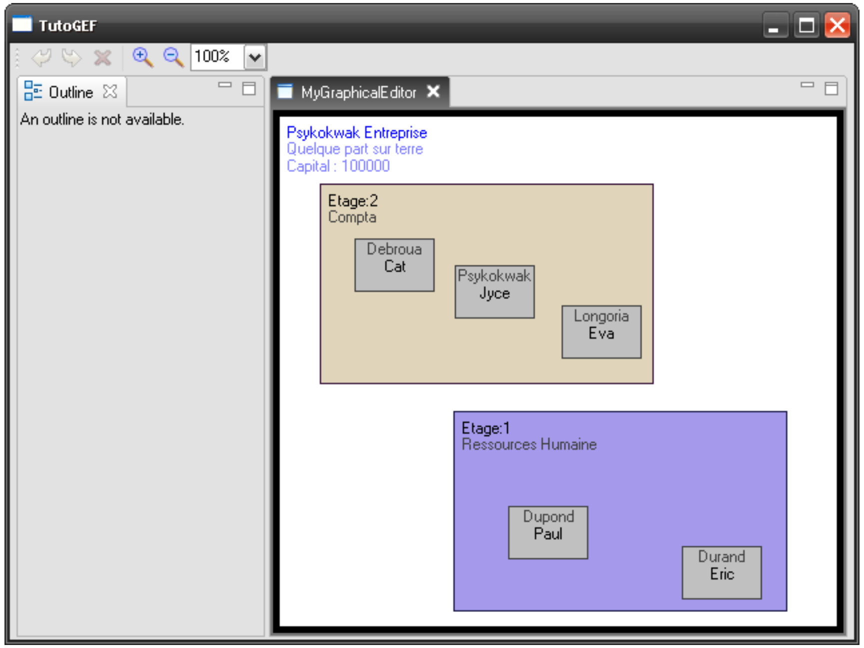
It is in fact a special Eclipse view called "outline".

The first thing to do is to add this view to the RCP plug-in.

The **Perspective** class defines views in our plug-in. We add the outline view to it.

```
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);
	}
}
```

If the application is launched at this level, the new view will appear on the left hand side of the graph.



The outline view has its own EditParts. It enables to have a different display than for the graph. But here, it is not the case.

We're going to create all necessary EditParts for this view, and also the associated factory in a brand new package: **tutogef.part.tree**.

Let's start with a base abstract class that will be extended by the EditParts.

```
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();
  }
}
```

Then the EditParts. We notice that they have exactly the same structure than those of the graph.

```
public class EntrepriseTreeEditPart extends AppAbstractTreeEditPart {


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

	@Override
	public void propertyChange(PropertyChangeEvent evt) {
		if(evt.getPropertyName().equals(Node.PROPERTY_ADD)) refreshChildren();
		if(evt.getPropertyName().equals(Node.PROPERTY_REMOVE)) refreshChildren();
	}
}
```

```
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().getImage(ISharedImages.IMG_
OBJ_ELEMENT));
	}

	@Override
```

```
        public void propertyChange(PropertyChangeEvent evt) {
                if(evt.getPropertyName().equals(Node.PROPERTY_ADD)) refreshChildren();
                if(evt.getPropertyName().equals(Node.PROPERTY_REMOVE)) refreshChildren();
        }
}
```

```
public class EmployeTreeEditPart extends AppAbstractTreeEditPart{
        protected List<Node> getModelChildren() {
                return ((Employe)getModel()).getChildrenArray();
        }

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

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

setWidgetImage(PlatformUI.getWorkbench().getSharedImages().getImage(ISharedImages.IMG_
DEF_VIEW));
        }

        @Override
        public void propertyChange(PropertyChangeEvent evt) {
                if(evt.getPropertyName().equals(Node.PROPERTY_ADD)) refreshChildren();
                if(evt.getPropertyName().equals(Node.PROPERTY_REMOVE)) refreshChildren();
        }
}
```

Then the factory:

```
public class AppTreeEditPartFactory implements EditPartFactory {

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

            if (model instanceof Entreprise)
              part = new EntrepriseTreeEditPart();
            else if (model instanceof Service)
                      part = new ServiceTreeEditPart();
            else if (model instanceof Employe)
                              part = new EmployeTreeEditPart();

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

            return part;
         }
}
```


Now, we're going to create a nested class inside **MyGraphicalEditor** that will take care of the outline view. We'll perform some changes on the go, to make the model and the key handler become accessible.

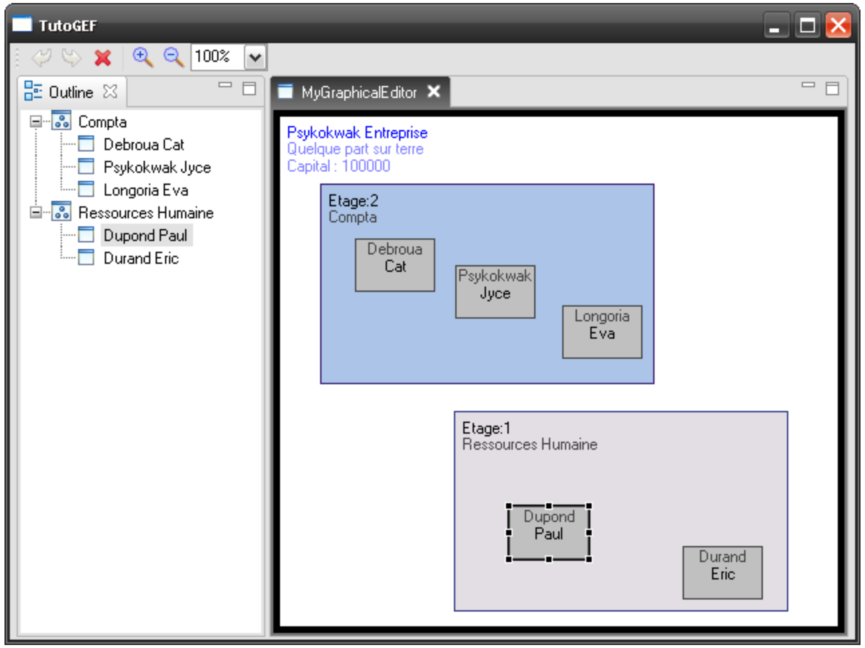
```
public class MyGraphicalEditor extends GraphicalEditor {

        (...)
        private Entreprise model;
        private KeyHandler keyHandler;

        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);

                        // On hook les actions de l'editeur sur la toolbar
                        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();

                        // On associe les raccourcis clavier de l'editeur a l'outline
                        getViewer().setKeyHandler(keyHandler);
                }

                public Control getControl() {
                        return sash;
                }

                public void dispose() {
                        getSelectionSynchronizer().removeViewer(getViewer());
                        super.dispose();
                }
        }
        (...)
        protected void initializeGraphicalViewer() {
                GraphicalViewer viewer = getGraphicalViewer();
                model = CreateEntreprise();
```

```
        viewer.setContents(model);
    }
    (...)
protected void configureGraphicalViewer() {

        (...)

        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);
    }
}
```

Inside “plugin.xml”, the dependency to *org.eclipse.ui.views* needs to be added. Last but not least, the nested class needs to be called when needed. The *getAdapter()* method is the one to check:

```
public class MyGraphicalEditor extends GraphicalEditor {

    (...)

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

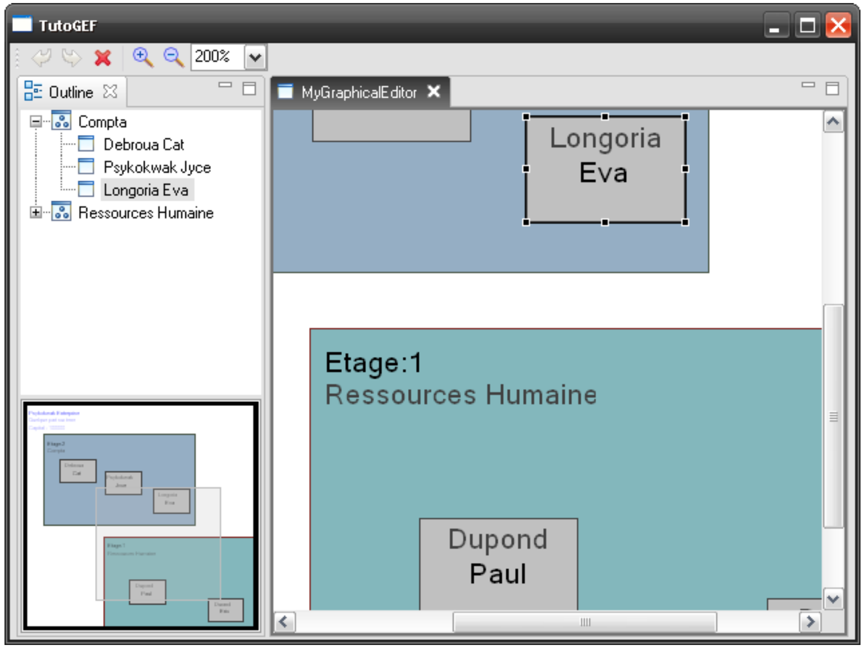

Feel better with a tree, isn’t it?



Part 7: Miniature view

In this tutorial part, we'll see how to add a miniature view of the graph in the outline view. It is very practical when using zoom functions. The mechanism relies upon the (nested) class in the outline view, in the **MyGraphicalEditor** class.

```
protected class OutlinePage extends ContentOutlinePage{

	private ScrollableThumbnail thumbnail;
	private DisposeListener disposeListener;

	(...)


	public void createControl(Composite parent) {
		(...)
		// Creation de la miniature.
		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(disposeListener);
	}

	(...)

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

Part 8: Context menu

In this tutorial part, we're going to add a context menu binded to right mouse click. The first thing to do is add the menu class.

This class inherits from **ContextMenuProvider**, and is build roughly the same way than for the toolbar.

We add "delete" and "undo/redo" actions to the menu:

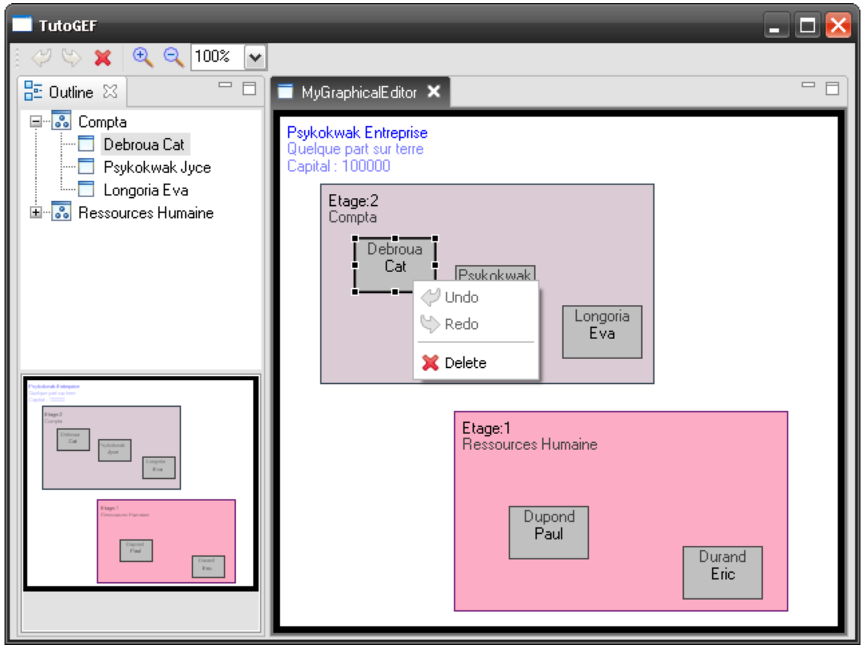
```
public class AppContextMenuProvider extends ContextMenuProvider{

	private ActionRegistry actionRegistry;

	public AppContextMenuProvider(EditPartViewer viewer, ActionRegistry registry) {
		super(viewer);
		setActionRegistry(registry);
	}

	@Override
	public void buildContextMenu(IMenuManager menu) {
		IAction action;

		GEFActionConstants.addStandardActionGroups(menu);

		action = getActionRegistry().getAction(ActionFactory.UNDO.getId());
		menu.appendToGroup(GEFActionConstants.GROUP_UNDO, action);

		action = getActionRegistry().getAction(ActionFactory.REDO.getId());
		menu.appendToGroup(GEFActionConstants.GROUP_UNDO, action);


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

	private ActionRegistry getActionRegistry() {
		return actionRegistry;
	}

	private void setActionRegistry(ActionRegistry registry) {
		actionRegistry = registry;
	}
}
```


Then we apply the menu where we want it to be.

We will put it in the tree and the graph.

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

The long-awaited result:



Part 9: Creating a custom action

We have seen before that we were able to add some EditPolicy to an EditPart, but these were only using already existing actions, handled by GEF (e.g. Component removal).

We will now create our own action and add it in an EditPolicy, as well as in the context menu of the application. Say that we want to be allowed to rename services but not the employees neither companies.

For this, we will create the following elements:

- A Wizard to ask the user for a new name
- A Command that will carry out the rename, and thus will be integrated in the undo/redo system
- An Action that will launch the wizard and create a request
- An EditPolicy to generate the command from the rename request, and that will eventually be completed for other edit operations.

Wizard creation

It is not really GEF, but wizards are very valuable tools in Eclipse, and very customizable, that's why we're going to create one here for the simple task consisting in asking the user a new name. The Wizard will only contain a Label, and a Text typing field to let the user enter a name, while still displaying the old one. The Wizard is in fact a container of IWizardPage, each of these representing a step in the wizard process. Our wizard will only contain one page, that we implement by creating a class derived from WizardPage.

```
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) {
			Composite composite = new Composite(parent, SWT.NONE);

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

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

			RowLayout l = new RowLayout();
			composite.setLayout(l);
```



```
			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() {
		RenamePage page = (RenamePage)getPage("MyRenamePage");
		if (page.nameText.getText().isEmpty()) {
			page.setErrorMessage("Le champ nom est vide!");
			return false;
		}
		newName = page.nameText.getText();
		return true;
	}

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

The **performFinish()** method is called whenever the user pushes the **Finish** button. If it returns **true**, the wizard finishes with success, otherwise it remains blocked waiting for the user correcting its error(s).

Command creation

We now enter again in the meanders of GEF. We will create a Command that will execute the rename action itself. As for the implementation of *DeleteCommand*, we implement the *execute()* and *undo()* methods of the command. We backup the old and new name in the command fields so that we can restore them during undo/redo process.

```
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);
        }
}
```

Action creation

Now that we're able to perform the rename operation and ask the user a new name, an action needs to be created to launch the wizard and create the command. The action we're going to create is called RenameAction. During its execution, it will create a GEF *Request*, that has the "rename" type (a new type created for this occasion), and we use its "extended data" (a map between Object and Object) to save the new name in the request (we choose "newName" as a key, and its value is the new name string). The "rename" type will be recognized by our EditPolicy, what we will see later. Finally, we call *getCommand()* on the first selected EditPart, to ask GEF to handle the request thanks to its EditPolicies and to return the generated command (that can be made of several chained commands, but that will be here a simple rename command).

```
public class RenameAction extends SelectionAction {

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

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

                // On spécifie l'identifiant utilise pour associer cette action a l'action globale de
renommage
                // intégrée a Eclipse
                setId(ActionFactory.RENAME.getId());

                // Ajout d'une icone pour l'action. N'oubliez pas d'ajouter une icone dans le dossier
"icones"
                // du plugin :)
                ImageDescriptor icon = AbstractUIPlugin.imageDescriptorFromPlugin("TutoGEF",
"icons/rename-icon.png");
                if (icon != null)
                        setImageDescriptor(icon);
                setEnabled(false);
        }

        @Override
        protected boolean calculateEnabled() {
          // On laisse les EditPolicy decider si la commande est disponible ou non
                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, 180);

                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();
        }
}
```

Don't forget that once the action is created, we need to register it in the **MyGraphicalEditor** ActionRegistry…

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

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

…and specify its shortcut in the context menu of the editor (by retrieving the global action using its Eclipse identifier in AppContextMenuProvider)…

```
    public void buildContextMenu(IMenuManager menu) {
        IAction action;

        // ...

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

…then in the application global menu (in MyGraphicalEditorActionBarContributor)

```
    public void contributeToMenu(IMenuManager menuManager) {
    // TODO
    }
```

EditPolicy creation

We have the command that carries out the said rename process, and the action that is able to create a request to retrieve this command from the EditPolicies. What is remaining is adding an EditPolicy that understands the new request and creates the rename command. We're going to create a new EditPolicy, called **AppRenamePolicy**, that will be derived directly from AbstractEditPolicy because we will define ourselves the **getCommand()** method.

```
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("newName"));
        return command;
    }
}
```

This class takes back the principle of **org.eclipse.gef.editpolicies.ComponentEditPolicy** to create a rename command: it verifies the request type, then it builds the corresponding command (using the extended data to read the new name).

Associate the new EditPolicy with the EditParts

We have now to specify which of our EditParts will have the privilege to be rename-enabled. Let's install the new EditPolicy in ServicePart and ServiceTreePart (one line to add in each corresponding files to offer service renaming in the graph and in the tree).


```
protected void createEditPolicies() {
    // ...
    installEditPolicy(EditPolicy.NODE_ROLE, new AppRenamePolicy());
}
```

Last, property activation to update views

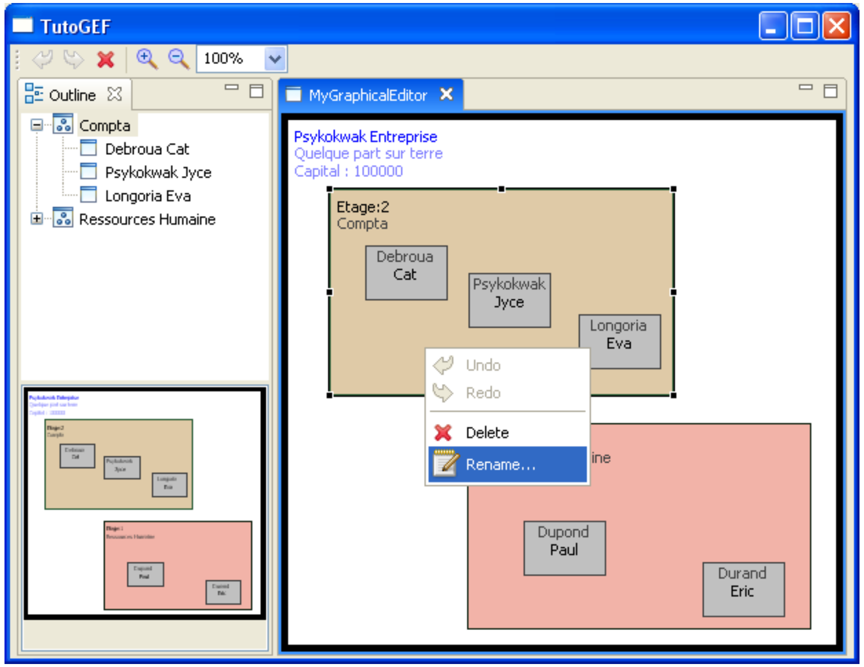
The rename is now functional, but views have no way to know that they need to update themselves. For this, we will create a new property in Node that we'll trigger whenever the new name chages, and the **propertyChange()** methods in concerned EditParts will listen to this new property to update themselves automatically. In coding words, that gives:

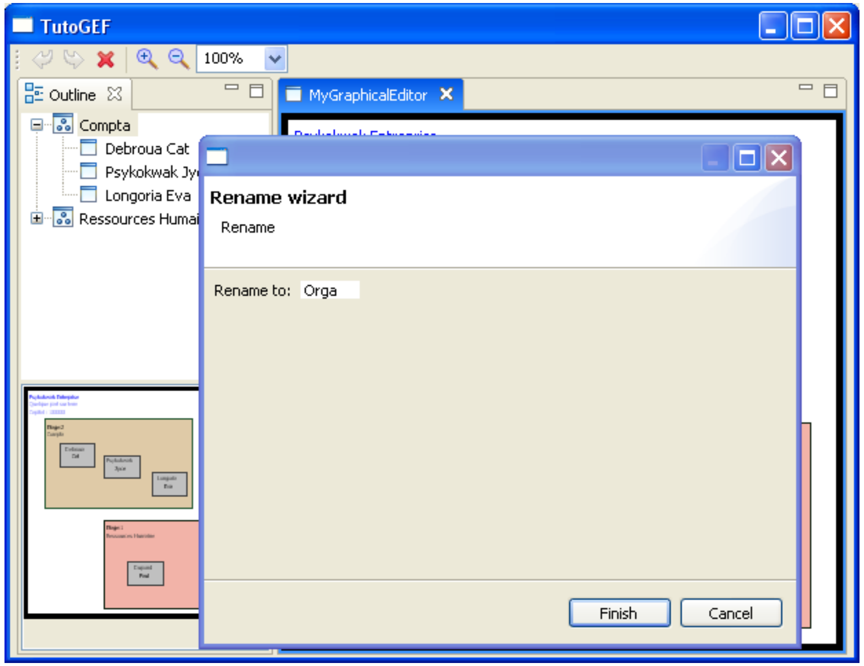
- In Node:

```
 // Declarations...
  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);
    }
```

- In ServicePart and ServiceTreePart:

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





Part 10: Property Sheet

In this tutorial, we will try to implement a property window, to display and modify model properties directly, and make the previous tutorial obsolete.

Some code parts are a bit fastidious to create but actually quite simple and logical.

You can refer to this page (http://www.eclipse.org/articles/Article-Properties-View/properties-view.html) for more complete explanations about the properties page.

Adding the color of the services in the model

First, before entering in the subject body, let's add the possibility to modify the color of a service after it has been chosen at random.

In the previous tutorial's version, color was choosen at random in services' Figure, meaning that we had no control on it. We're going to store the service color in a model field, and we will create accessors to read and modify it, as well as trigger a color property change when modifying to inform views that they need an update.

In ServiceFigure, modify the constructor not to decide of the background color anymore, and fix the border color to black:

```
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(ColorConstants.black);

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

Then in Service, add the color property, still initialized randomly at start-up:

```
private Color color;

// ...

private Color createRandomColor() {
      /** Just for Fun :) **/
      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;

        // mise-à-jour des vues
        getListeners().firePropertyChange(PROPERTY_COLOR, oldColor, color);
}
```

Define a property source

The Eclipse platform uses a standard property sheet which reacts automatically to selection changes, and updates the display of selected objects' properties. Our selection is already represented directly by model objects, thus we will define them as property sources.

A property source can be defined in two manners:

- Either by implementing directly the **IPropertySource** interface in the model object

- Or by implementing the **IAdaptable** interface in the model object, and by returning an object that implements IPropertySource in the *getAdapter()* method, in the case where an IPropertySource type is passed as argument to the method. This technique has the advantage of a better separation between the model and the property source, and it's the one we will be implementing in Node, what rises from this is that all objects inheriting from Node will be able to have properties (thus, Entreprise, Service and Employe).

Node.java:

```
public class Node implements IAdaptable
{
        // mise en cache de la IPropertySource pour ne la créer qu'au premier appel de getAdapter()
        private IPropertySource propertySource = null;

        // ...

        @Override
        public Object getAdapter(Class adapter) {
                if (adapter == IPropertySource.class) {
                        if (propertySource == null)
                                propertySource = new NodePropertySource(this);
                        return propertySource;
                }
                return null;
        }
}
```


This code doesn't compile yet, as it is supposed to be, since the **NodePropertySource** class is missing, and we'll create it now, by implementing IPropertySource. We'll cut its description in a few steps to detail each method. But before that, we're going to create the unique property identifiers, which are static fields, defined in the model, and of any type (here, String).

Definition of static fields in Service:

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

Those of Entreprise:

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

Then Employe:

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

By the way, let's update property listeners in the different EditParts, so that views can update themselves after any editable property change.

In EntreprisePart: we add the handling of PROPERTY_RENAME and PROPERTY_CAPITAL.

```
@Override
public void propertyChange(PropertyChangeEvent evt) {
    if (evt.getPropertyName().equals(Node.PROPERTY_ADD)) refreshChildren();
    if (evt.getPropertyName().equals(Node.PROPERTY_REMOVE)) refreshChildren();
    if (evt.getPropertyName().equals(Node.PROPERTY_RENAME)) refreshVisuals();
    if (evt.getPropertyName().equals(Entreprise.PROPERTY_CAPITAL)) refreshVisuals();
}
```

The modification is similar in all graph and tree EditParts. We won't enumerate them all here but they are of course included in the tutorial archive.

Now, let us return to our NodePropertySource class.

First, we store the model object to be able to refer to it later.

```
public class NodePropertySource implements IPropertySource {

        private Node node;

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

```java
/**
 * Returns the property value when this property source is used as a value. We can
 * return <tt>null</tt> here
 */
@Override
public Object getEditableValue() {
	return null;
}
```

In this tutorial, we've chosen to implement all model properties in one single class, hence the different tests in order to know if each property is available to one or the other model elements, but it is indeed possible to create a distinct class for each model object and reimplement these methods, or implement IPropertySource directly in all of the model objects which have properties.

The following method returns IPropertyDescriptors, each of these representing a property of the model. If the property is read-only, a simple PropertyDescriptor is enough; the role of derived classes is to return to the property sheet a CellEditor fitted to edit the cell. As we don't want to be able to edit the name and forename of employees, we return a PropertyDescriptor.

```java
@Override
public IPropertyDescriptor[] getPropertyDescriptors() {
	ArrayList<IPropertyDescriptor> properties = new ArrayList<IPropertyDescriptor>();
	if (node instanceof Employe)
		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 Entreprise) {
		properties.add(new TextPropertyDescriptor(Entreprise.PROPERTY_CAPITAL, "Capital"));
	}
	else if (node instanceof Employe) {
		properties.add(new PropertyDescriptor(Employe.PROPERTY_FIRSTNAME, "Prenom"));
	}
	return properties.toArray(new IPropertyDescriptor[0]);
}
```


Retrieval of property values depending on its identifier.

```java
@Override
public Object getPropertyValue(Object id) {
	if (id.equals(Node.PROPERTY_RENAME))
		return node.getName();
	if (id.equals(Service.PROPERTY_COLOR))
		// ColorCellEditor, renvoyé comme éditeur de cellule par ColorPropertyDescriptor,
		// utilise la classe RGB de SWT pour lire et écrire une couleur.
		return ((Service)node).getColor().getRGB();
	if (id.equals(Service.PROPERTY_FLOOR))
		return Integer.toString(((Service)node).getEtage());
	if (id.equals(Entreprise.PROPERTY_CAPITAL))
		return Integer.toString(((Entreprise)node).getCapital());
	if (id.equals(Employe.PROPERTY_FIRSTNAME))
		return (((Employe)node).getPrenom());
	return null;
}
```

```java
// Returns if the property with the given id has been changed since its initial default value.
// We do not handle default properties, so we return <tt>false</tt>.
@Override
public boolean isPropertySet(Object id) {
	return false;
}
/**
 * Reset a property to its default value. Since we do not handle default properties, we do
 * nothing.
 */
@Override
public void resetPropertyValue(Object id) {
}
```

Recording of a property value after the user edited it:

```java
@Override
public void setPropertyValue(Object id, Object value) {
	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);
			((Service)node).setEtage(floor);
		}
		catch (NumberFormatException e) { }
	}
	else if (id.equals(Entreprise.PROPERTY_CAPITAL)) {
		try {
			Integer capital = Integer.parseInt((String)value);
			((Entreprise)node).setCapital(capital);
		}
		catch (NumberFormatException e) { }
	}
}
```

Integration of the property sheet

Voila, our property source is ready to give the property of the model objects. But we still don't have integrated the property sheet in the Eclipse perspective.

We will in fact proceed in two steps: first, we define a placeholder for this property sheet, then we ask for its display when the user double-clicks a GEF object.

Let's start by specifying the window placement: it takes place in the default perspective definition (perspective.java):

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

		IFolderLayout tabs = layout.createFolder(
				ID_TABS_FOLDER, IPageLayout.LEFT, 0.3f, editorArea);
		tabs.addView(IPageLayout.ID_OUTLINE);
		tabs.addPlaceholder(IPageLayout.ID_PROP_SHEET);
	}
```

In the previous version of the tutorial, the outline was attached directly to the main window, now we have an IFolderLayout (tabs) on which we attach the two windows, but the outline alone will be displayed at startup.

We'll now associate the double-click on an EditPart to the opening or focus-in of the property sheet, what takes us back for some time to GEF to finish this tutorial part.

Double-clicking on a GraphicalEditPart generates in fact a *Request* of type REQ_OPEN. This request is not transmitted to EditPolicies, and has to be handled in the *performRequest()* method of the EditPart.

In AppAbstractEditPart (to handle the double-click on graphical elements):

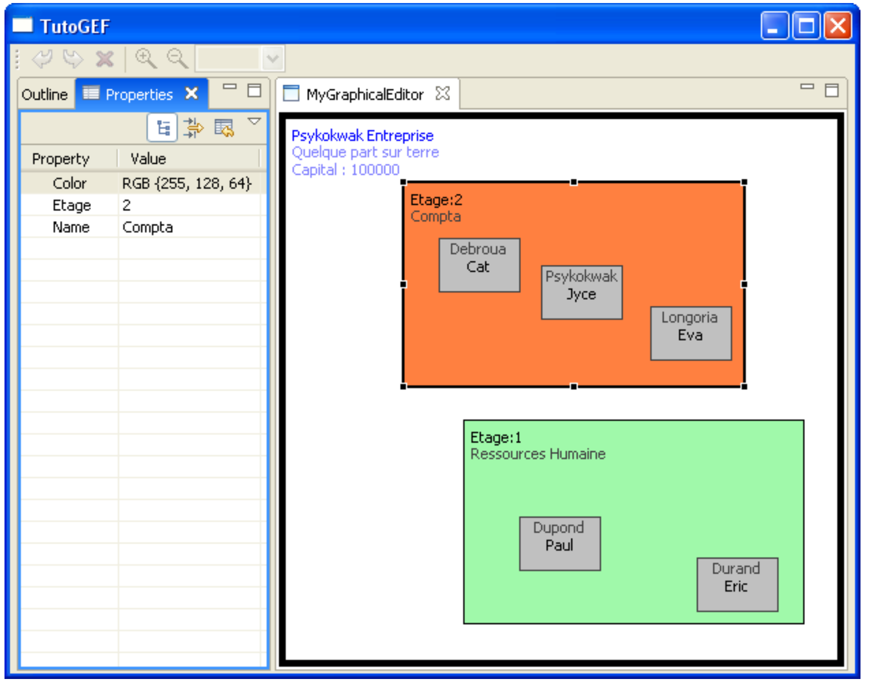
```
	@Override
	public void performRequest(Request req) {
		if (req.getType().equals(RequestConstants.REQ_OPEN)) {
			try {
				IWorkbenchPage page =
PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage();
				page.showView(IPageLayout.ID_PROP_SHEET);
			}
			catch (PartInitException e) {
				e.printStackTrace();
			}
		}
	}
```

That is enough for EditParts inheriting from AbstractGraphicalEditPart, because it returns a DragTracker to handle selection in the *getDragTracker()* method. As this method returns nothing in AbstractTreeEditPart (for the tree view), we need further to implement it to handle double-click in the tree.

In AppAbstractTreeEditPart:

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

	@Override
	public void performRequest(Request req) {
		if (req.getType().equals(RequestConstants.REQ_OPEN)) {
			try {
				IWorkbenchPage page =
PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage();
				page.showView(IPageLayout.ID_PROP_SHEET);
			}
			catch (PartInitException e) {
				e.printStackTrace();
			}
		}
	}
```

That's it. The model is now visualizable and editable with a property sheet.



Part 11: Adding new graphical elements

In the next parts of this tutorial, we will concentrate on all that have to deal with **graphical** edition and element addition. In this part, we will create a palette, add to it some tools to basically manipulate the graph and insert services and employees into it.

Palette insertion

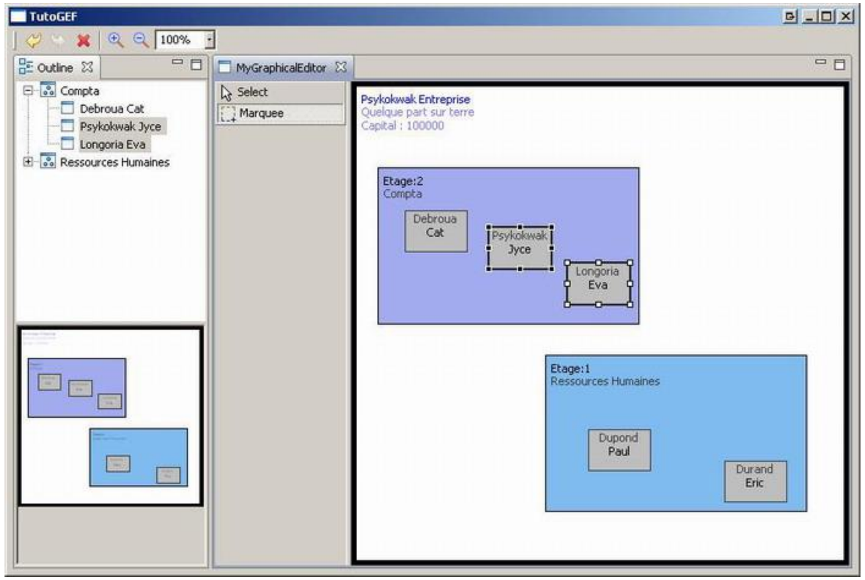
The first thing to do is, obviously, to add the palette to the editor. For that, we're going to modify the class which the editor (*MyGraphicalEditor*) inherits from: we transit from *GraphicalEditor* to *GraphicalEditorWithPalette* (even *GraphicalEditorWithFlyoutPalette* for those who prefer a more "interactive" palette).

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette {
  // ...
  }
```

This new inherited class force us to implement the *getPaletteRoot()* method. This one must return an object of type *PaletteRoot*, containing the palette tree. It is in general composed of several groups (*PaletteGroup*), which we can separate graphically (*PaletteSeparator*), each of these groups containing entries of different kinds. To begin with, we're going to add the most basic palette tools which are the selection tool and the marquee tool.

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette {
        // ...
        @Override
        protected PaletteRoot getPaletteRoot() {
                // Racine de la palette
                PaletteRoot root = new PaletteRoot();

                // Creation d'un groupe (pour organiser un peu la palette)
                PaletteGroup manipGroup = new PaletteGroup("Manipulation d'objets");
                root.add(manipGroup);

                // Ajout de l'outil de selection et de l'outil de selection groupe
                SelectionToolEntry selectionToolEntry = new SelectionToolEntry();
                manipGroup.add(selectionToolEntry);
                manipGroup.add(new MarqueeToolEntry());

                // Definition l'entree dans la palette qui sera utilise par defaut :
                //      1.lors de la premiere ouverture de la palette
                //      2.lorsqu'un element de la palette rend la main
                root.setDefaultEntry(selectionToolEntry);
                return root;
        }
  }
```


If you launch the application, you will obtain a result looking like this:



The marquee tool is used to select multiple elements belonging to a user-defined region. It is notably useful to mass move, delete entities or interact with an entity group (here, services or employees) without having to hold the *CTRL* key.

Adding a service

Now that we have the palette, it could be interesting to fill it somewhat, and especially to add the possibility to **graphically** create a service. In this purpose, we will first create a class derived from *CreationFactory*. Factories are classes used to create new object instances depending on the context. The benefit of using this class tree is, as we're going to see later, calling its methods is transparent and will be handled by GEF internally. So let's create the factory, that we will name *NodeCreationFactory* (a generic name since we will modify it so that it will handle employee creation later), inheriting from *CreationFactory*.

```
public class NodeCreationFactory implements CreationFactory
{
        private Class<?> template;

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

        @Override
        public Object getNewObject() {
                if (template == null)
                        return null;
                if (template == Service.class)
                {
```

```
            Service srv = new Service();
            srv.setEtage(42);
            srv.setName("Factorouf");
            return srv;
        }
        return null;
    }

    @Override
    public Object getObjectType() {
        return template;
    }
}
```

Obviously, you're free to improve the new object generation; especially concerning the name and the specification of the floors (we're more concerned about the concept rather than the finality here). Now that we have a class able to generate an object almost from scratch, we're going to create the command that will use this *CreationFactory*. This command has to memorize useful information (in this case, the newly-created service and the company it belongs to and it will be added to), notably for undo/redo integration (in the command stack). This command (that will be called *ServiceCreateCommand*) has to be derived from *Command*.

```
public class ServiceCreateCommand  extends Command
{
    private Entreprise 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 setEntreprise(Object e) {
        if (e instanceof Entreprise)
            this.en = (Entreprise)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);
    }
}
```

Once again, feel free to sophisticate all this at will, for example, to accept the creation of a new service only if the company has more than $10,000 capital (which is of course the case of the *Psykokwak* Company). Fair enough, now it won't compile without the following code snippet, to be put in *Node* class, and which is a method that checks a node's presence in its parent.

```
public class Node implements IAdaptable
{
    // ...
    public boolean contains(Node child) {
        return children.contains(child);
    }
}
```

As you know now, in GEF, without any *EditPolicy*, the *Command* is nothing. What we have to concentrate on now, is integrate the command birth in our favourite *EditPolicy*, in our case *AppEditLayoutPolicy*.

```
public class AppEditLayoutPolicy extends XYLayoutEditPolicy {
    // ...
    protected Command getCreateCommand(CreateRequest request) {
        if (request.getType() == REQ_CREATE && getHost() instanceof EntreprisePart)
{
            ServiceCreateCommand cmd = new ServiceCreateCommand();
            cmd.setEntreprise(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;
    }
}
```

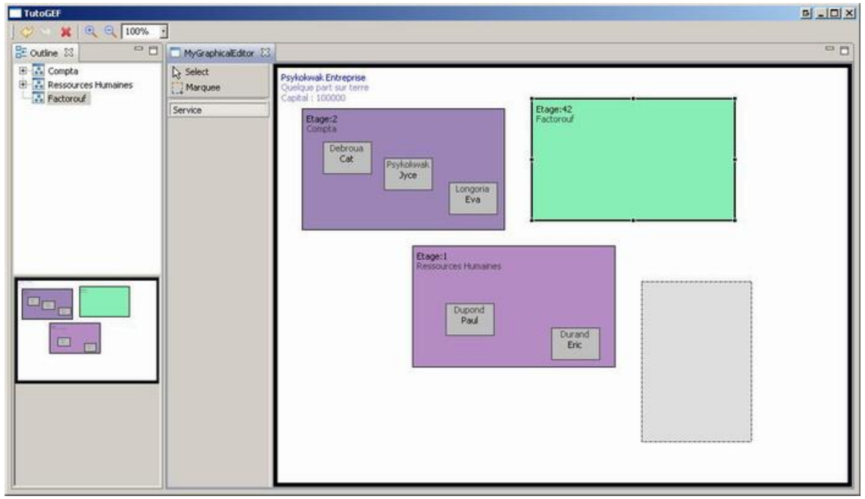
Here, if the request type is a creation request (well, this test is at first sight not really necessary since we already are in *getCreateCommand()*) and if the *EditPart* is an *EntreprisePart*, then we create the command, filling it with useful information. By the way, the most observant will have noticed that the two size constants need to be added in the *ServiceFigure* class (relative to services and their representation, so it needs to be put here).

```
public class ServiceFigure extends Figure {
	public static final int SERVICE_FIGURE_DEFWIDTH = 250;
	public static final int SERVICE_FIGURE_DEFHEIGHT = 150;
	// ...
}
```

Next, we have to check that the *EditPolicy* we just changed is used by the *EditParts* in which we would like to be able to put a new element. In our case, *EntreprisePart* is concerned, indeed, we would not allow the creation of a service that would cover another service (at least, whose top-left corner would be in another service space, to be accurate). In our case, it's already done, all that is missing is adding the entry in the palette.

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette
{
	// ...
	protected PaletteRoot getPaletteRoot()
	{
		// ...
		PaletteSeparator sep2 = new PaletteSeparator();
		root.add(sep2);

		PaletteGroup instGroup = new PaletteGroup("Creation d'elemnts");
		root.add(instGroup);

		instGroup.add(new CreationToolEntry("Service", "Creation d'un service type",
				new NodeCreationFactory(Service.class),
				null, null));
		// ...
		root.setDefaultEntry(selectionToolEntry);
		return root;
	}
}
```


We thus define a new entry in the palette (that we place yet in a new palette group), and we give to it, in this order: a name, a description, an instance of the factory to generate the objects, then the *ImageDescriptor*s referring to the icons respectively for small and large views. Now, let's test all this!



Adding an employee in a service

It's possible to create a new service in the company; it would still be interesting to add employees in it. We will reproduce globally the same procedure than before.

First, we will modify the *CreationFactory* so that it's able to produce *Employe* objects. We proceed as follows:

```
public class NodeCreationFactory implements CreationFactory
{
	// ...
	public Object getNewObject() {
		// ...
		else if (template == Employe.class) {
			Employe emp = new Employe();
			emp.setPrenom("Halle");
			emp.setName("Berry");
			return emp;
		}
		return null;
	}
}
```

Now let's define a new command, called *EmployeCreateCommand*, that corresponds to adding an employee in a precise service.

```
public class EmployeCreateCommand extends Command
{
	private Service srv;
	private Employe emp;

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

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

	public void setEmploye(Object e) {
		if (e instanceof Employe)
			this.emp = (Employe)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);
	}
}
```

In the concept, not much change, we memorize useful information to be able to undo/redo the command, and we add the employee in the specified service. This new command has to be built, after the context is correct.

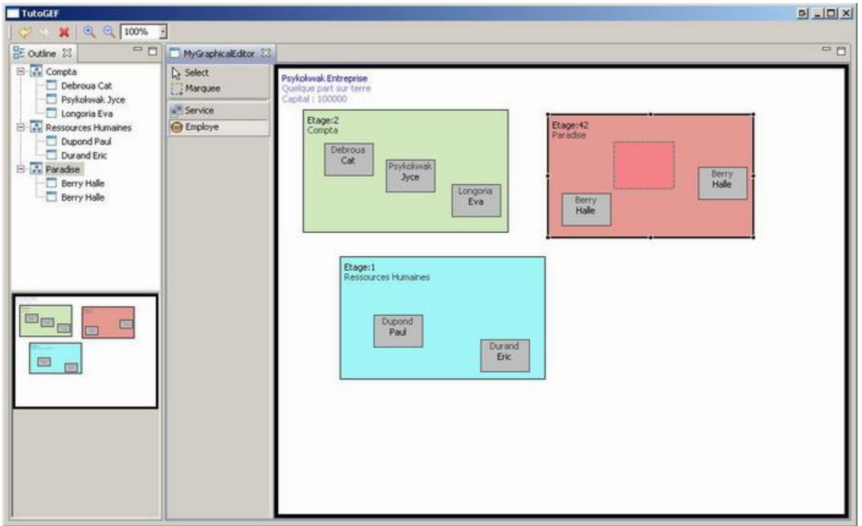
```
public class AppEditLayoutPolicy extends XYLayoutEditPolicy
{
	// ...
	@Override
	protected Command getCreateCommand(CreateRequest request) {
		if (request.getType() == REQ_CREATE && getHost() instanceof EntreprisePart)
{
			// ...
		}
		else if (request.getType() == REQ_CREATE && getHost() instanceof
ServicePart) {
			EmployeCreateCommand cmd = new EmployeCreateCommand();

			cmd.setService(getHost().getModel());
			cmd.setEmploye(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) ?
EmployeFigure.EMPLOYE_FIGURE_DEFWIDTH :  constraint.width;
			constraint.height = (constraint.height <= 0) ?
EmployeFigure.EMPLOYE_FIGURE_DEFHEIGHT : constraint.height;
			cmd.setLayout(constraint);
			return cmd;
		}
		return null;
	}
}
```

Once again, we add in *EmployeFigure* the default values and to homogenise the whole, we also use them during graph creation. All is now done, only a palette entry is missing, that will use all this beautiful code. Let's go!

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette {
	// ...
	protected PaletteRoot getPaletteRoot() {
		// ...
		PaletteGroup instGroup = new PaletteGroup("Creation d'elemnts");
		root.add(instGroup);

		instGroup.add(new CreationToolEntry("Service", "Creation d'un service type",
				new NodeCreationFactory(Service.class),
				AbstractUIPlugin.imageDescriptorFromPlugin(Activator.PLUGIN_ID,
"icons/services-low.png"),
				AbstractUIPlugin.imageDescriptorFromPlugin(Activator.PLUGIN_ID,
"icons/services-high.png")));

		instGroup.add(new CreationToolEntry("Employe", "Creation d'un employe
model",
				new NodeCreationFactory(Employe.class),
				AbstractUIPlugin.imageDescriptorFromPlugin(Activator.PLUGIN_ID,
"icons/employe-low.png"),
				AbstractUIPlugin.imageDescriptorFromPlugin(Activator.PLUGIN_ID,
"icons/employe-high.png")));
	}
}
```

By the way, let's add some little icons to our two entries: one for our service creation entry, the other for the employee's one. We retrieve an *ImageDescriptor* with the plug-in identifier and the relative path to the icons inside it. If you prefer the interactive palette (*FlyoutPalette*), you will have access to some nice configurations, and especially the one that will use the big icons version.

All this is very nice, we now have to test the whole system and take a little break before the next tutorial part. Enjoy.



Part 12 : Drag and drop

We have the possibility to create graphically services and employees, by clicking and drawing these entities, but it's not that user-friendly and practical. We will now integrate a must-have functionality, available in every so-called modern interface: the well named **drag-and-drop** (alias *dnd*).

Almost all the process will take place in our editor (*MyGraphicalEditor*). What we want to be done is, from the palette, selection (*drag)* of an element that will be put (*drop*) in the editor. The first (well, the only) thing to do is to add a specific *listener* of type *TemplateTransfertDragSourceListener* on our palette, and a listener of type *TemplateTransferDropTargetListener* on our editor. Names are self-explanatory: they are transfer templates (component moving from a source element *x* to a destination element *y*, thus the so-called transfer), one being the source (*drag* movement), the other being the destination (*drop* movement).

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette
{
	// ...
	protected void initializeGraphicalViewer() {
		GraphicalViewer viewer = getGraphicalViewer();
		model = CreateEntreprise();
		viewer.setContents(model);
		viewer.addDropTargetListener(new MyTemplateTransferDropTargetListener(viewer));
	}

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

You'll probably have noticed that we don't use directly a *TemplateTransferDropTargetListener*, but our own derived version. Indeed, the "drop" notion is fully dependant to the model, so it can't in its generic version fullfill our specific needs, and the class functions will have to be overloaded. Let's type!

```
public class MyTemplateTransferDropTargetListener extends
TemplateTransferDropTargetListener
{
	public MyTemplateTransferDropTargetListener(EditPartViewer viewer) {
		super(viewer);
	}

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

The only interesting part in this code snippet is the factory, from which we return an instance when asked for. In the scope of this tutorial, we basically use the object *class* that we want to use. However, as we will see just now, this object is used as a template. So it is advised to use a template version of an object (a single shared instance) to make all we have seen until now. We don't have done this here because our objects are very simple and don't need specific information from a model object (the template) to be built. You should think about adapting your listeners, and above all, your *CreationFactory* consequently.

Now, there is only one thing still to do: adapt palette entries to this new type of interaction. There is a kind of *CreationToolEntry* that supports, not only the creation as we've done initially, but also *drag-and-drop*: *CombinedTemplateCreationEntry*.

```
public class MyGraphicalEditor extends GraphicalEditorWithPalette
{
	// ...
	@Override
	protected PaletteRoot getPaletteRoot()
	{
		// ...
		instGroup.add(new CombinedTemplateCreationEntry("Service", "Creation d'un service 
type",
				Service.class, new NodeCreationFactory(Service.class),
				AbstractUIPlugin.imageDescriptorFromPlugin(Activator.PLUGIN_ID, 
"icons/services-low.png"),
				AbstractUIPlugin.imageDescriptorFromPlugin(Activator.PLUGIN_ID, 
"icons/services-high.png")));

		instGroup.add(new CombinedTemplateCreationEntry("Employe", "Creation d'un 
employe model",
				Employe.class, new NodeCreationFactory(Employe.class),
				AbstractUIPlugin.imageDescriptorFromPlugin(Activator.PLUGIN_ID, 
"icons/employe-low.png"),
				AbstractUIPlugin.imageDescriptorFromPlugin(Activator.PLUGIN_ID, 
"icons/employe-high.png")));
		// ...
	}
}
```

The difference between both prototypes is simple: the template object that we've talked about has to be inserted just before the factory. In our case and for aforementioned reasons, we will insert object *classes*. We can test now!

Part 13 : Cut and paste

After having seen how to integrate *drag-and-drop* functionality, it is time for us to focus on another must-have functionality in every respectable interface nowadays: the **cut/paste** function. It's the third and last method, after by-palette insertion and *drag-and-drop*, able to provide graphical creation of new objects.

As opposed to the majority of GEF functionalities, cut and paste is done, as the name suggests, in two steps. In the first step, we will create a command that represents the *copy*, as well as an action that will build this so-called command when the context is suited to it. In the second step, we will repeat that for the *paste*. We will finally have to integrate these actions into the application.

Let's go, and we start with this command. It surely inherits from *Command* and will memorize in an *ArrayList* elements to copy, if these are usable in this context. In our case, we decided to be able to copy, not only employees but also services.

```
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 Employe)
			return true;
		return false;
	}
}
```

As the code above suggests, we apply our selection (our *ArrayList*) to the internal mechanism of GEF passing through the static *Clipboard.getDefault().setContents(list)* method. Later, it will be shown how to retrieve this information.

Now that we have our command, we have to define an action that will be integrated in the application. This will build the command, if all goes well (if there is something to copy, etc.). We tried to simplify the action to its simplest form, and thus abstract the complexity (is there really one?) in the command.

```
public class CopyNodeAction extends SelectionAction
{
	public CopyNodeAction(IWorkbenchPart part) {
		super(part);
		// force calculateEnabled() to be called in every context
		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
_DISABLED));
		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() {
		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();
		}
	}

}
```

In a first step, it's worth noting in the *Init()* method that we will load icons available in *Eclipse*. These will be visible in the context menu as well as in the toolbar (and, if we had defined one, in the application global menu).

In a second step, note that in the *run()* method, we call directly the *execute()* method of the command, and not the *execute()* method into which we would pass the command as an argument. This avoids integrating the *copy* action in the GEF command stack; indeed, we don't consider as good practice to be able to undo a copy operation (but it will not be the case for the paste command).

Copy makes the guy happy, but being able to paste what we just copied, is the Holy Grail to reach. We will proceed quite in the same manner. We create a command and an associated action that will create the command if required and sensible.

```
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 Employe) {
                                Employe emp = (Employe) node;
                                Employe clone = (Employe) 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 Employe)
                return true;
        return false;
    }
}
```


We retrieve the *Clipboard* contents that we defined in the copy command. Next, a *HashMap* is used to store, as keys, the source object and as values, the clone generated from the key. We then use the value to retrieve the parent (which is the same as the key since our *clone()* methods keep the parent relationships, see below). Created objects need to be stored so that the cut and paste can be integrated in the GEF undo/redo facilities. Let's now walk to the corresponding action:

```
public class PasteNodeAction extends SelectionAction
{
        public PasteNodeAction(IWorkbenchPart part) {
                super(part);
                // force calculateEnabled() to be called in every context
                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_PAST
E_DISABLED));
                setEnabled(false);
        }

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

        @Override
        protected boolean calculateEnabled() {
                Command command = createPasteCommand();
        return command != null && command.canExecute();
        }

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

Nothing is really exciting here. We create an action that builds the command. Let's see some details though. If you're aware, you would have remarked the call to the *clone()* method. This one is generic and needs to be personalized to be able to return a new object (a new instance), fully similar to the *calling* instance. Two clones have, if all goes well, the same parent, which certifies that the paste command will correctly run.

```java
public class Employe extends Node {
	@Override
	public Object clone() throws CloneNotSupportedException {
		Employe emp = new Employe();
		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;
	}
}
```

In the case of a *Service*, we want the *clone()* method to return a new *Service* having precisely the same employees than its model; we thus also had to clone the object children. There is a little tweak on the cloned employees layout, that enables to keep precisely the same layout inside our *Services*.

```java
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 Employe) {
				Employe child = (Employe)node;
				Node clone = (Node)child.clone();
				srv.addChild(clone);
				clone.setLayout(child.getLayout());
			}
		}
		return srv;
	}
}
```


Now that our actions (and their respective commands) are implemented, we have to integrate them into the application. First, we add the actions in the toolbar in *MyGraphicalEditorActionBarContributor*.

```java
public class MyGraphicalEditorActionBarContributor extends ActionBarContributor
{
	protected void buildActions() {
		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()));
		// ...
	}
}
```

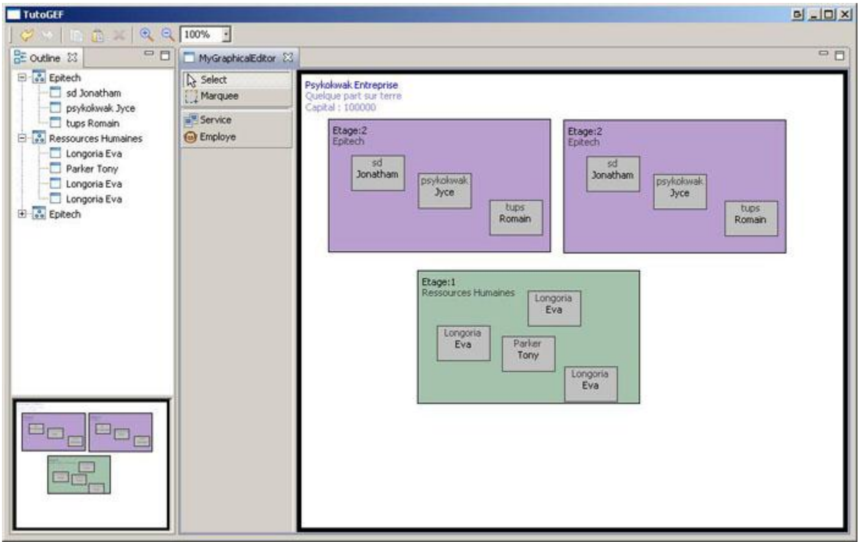
Then, the actions have to be added with the editor controls. This is done, of course, in *MyGraphicalEditor* like this:

```java
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() {
		super.createActions();

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

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

Don't run away, it's not finished yet! One last little thing is to be done, unsure that keyboard shortcuts will be active, and above all, work. For this, we register the actions in the *ApplicationActionBarAdvisor*.

```
public class ApplicationActionBarAdvisor extends ActionBarAdvisor
{
   public ApplicationActionBarAdvisor(IActionBarConfigurer configurer) {
      super(configurer);
   }

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

   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) {
   }
}
```

So, this part is finished, let's test all that great things.



Conclusion

With this tutorial, you should have looked at the most useful and used functionalities of GEF and also some useful parts of Eclipse (RCP, property sheet, actions).

References

GEF website: http://download.eclipse.org/tools/gef

GEF in the Eclipse wiki: http://wiki.eclipse.org/index.php/Graphical_Editing_Framework

Another GEF tutorial (Japanese): http://www13.plala.or.jp/observe/GEF