GefDescription

What this page is about and what you should know before reading it :

This page is intended to give somebody interested in GEF most of the things he has to know to get started. This includes a description of the purpose of the library, a global view of the library with descriptions of the different components and how to use them, and some practical guidelines to build a simple GEF editor. At least that's what I have tried to do. I wrote most of this for myself a few months ago but I think this is better here than on my personal hard drive. Maybe it could be useful to you.

I didn't try to avoid redundancy with currently available documentation. When I got started, I read the Randy Hudson tutorial, the IBM redbook and the docs of the library. So what is written here is probably strongly inspired by all that.

My mother tongue is French, I am not an experienced eclipse and GEF programmer and I really don't feel like some kind of genius. **So what follows probably contains mistakes, and is by far not so reliable as the official docs**.

Feel free to correct all the mistakes you notice and to add your own content to the page. There are a lot of todo's in the page, feel free to complete them if you want to. Please leave comments to point out what you don't understand, what is not clear, what you would like to find here and things like that.

rlemaigr

1) The problem GEF helps you to solve :

GEF stands for Graphical Editing Framework. So this is a library to ease the task of building a graphical editor, in Eclipse.

1.1) a general problem of graphical editing :

You want to build an application with the following requirements :

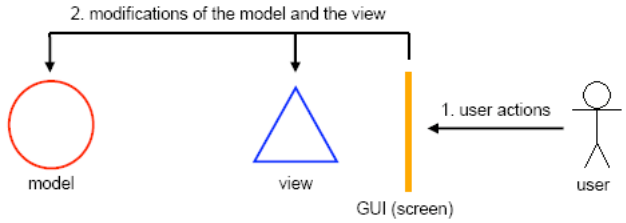
- on one hand there is a **model** composed of objects holding some datas,
- on the other hand a **view** composed of objects defining some paintings on the screen,
- the user must be able to modify the view with the mouse and the keyboard,
- some **link** between the view and the model defines what happens to the model when the view is modified and vice versa.

Such an application providing the user with a way to modify a model graphically is called a graphical editor.

The view, the model and the link between them are illustrated in this picture, which introduces the symbolic conventions I will try to follow for the rest of this page:

In this general problem, the link between the model and the view can be anything, so it is possible that for two identical states of the model, the view showed to the user is different. For example, if the user can move the different figures of the view without any trace being kept of their positions in the model, there will be several possible views for a same state of the model (one for each possible location the user can choose for the figures in the view).

Your graphical editor should work like that :

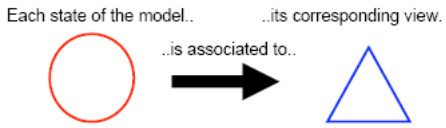


But defined like that, I think that **this problem is too fuzzy** to start a good explanation, and to give clear ideas to people who want to get started with GEF. So I will define a problem which is less general and I will try to show in this page how GEF allows you to solve this less general problem. Anyway, I may be wrong but I think that GEF was thought to solve that less general problem, and even if it wasn't, I believe that it is good practice to follow the limitations introduced by the less general problem to keep things clear and simple.

1.2) a less general problem of graphical editing :

This problem is just the same as above except that this time **the link between the model and the view is more specific** : for each state of the model, you have defined **one** view which should be displayed to the user for that particular state.

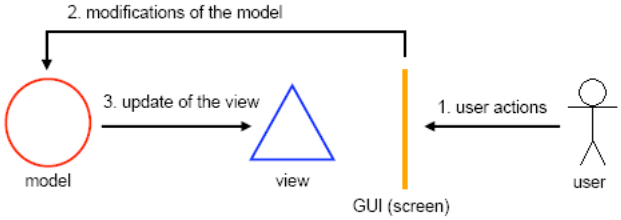
So this time the view shown to the user depends only on the current state of the model and is fully defined by it : the view is a function of the state of the model. This is illustrated on this picture :



This loss of generality leads to a simplification of the architecture of the application : now the user's actions on the graphical interface can be interpreted in terms of model modifications only because we know that the new view will be fully defined by the new state of the model, whatever were the user actions on the GUI. We can make a clean separation between :

- the modifications of the model triggered by the user actions on the graphical interface,
- the updates of the view triggered by the modifications of the model according to its new state.

This is illustrated in these pictures :



Please don't get me wrong : the model is not really responsable of updating the view. This is just a schematic view of what happens when the user acts on the GUI. Some pieces are absent from the picture.

To implement this, you have three problems to solve :

1. You have to implement some mechanism which automatically **builds and displays the view** you have defined for the model when the editor is opened.
2. You have to implement some mechanism which **updates the view when the model is changed**, according to the view you have defined for the new state of the model.
3. You have to implement some mechanism which **captures the user's actions on the graphical interface, and translates them into changes applied to the model**.

These are the problems that GEF helps you to solve and the following explanations will look at them one by one.

2) Model - view - controller architecture :

I know what your are thinking : you are thinking that this section is only made for those masochists of us who absolutely want to know how the things work in the GEF black box and so you can skip it. **Don't think that ! Because :**

- It is true that GEF is based on the model - view - controller architecture, but it only provides the pieces of this puzzle. You will have to assemble it by yourself, and you will have to build and extend a lot of these pieces by yourself. If you don't know how to do that properly, then everything will go wrong for sure.
- I have found on Internet several different explanations about this topic. The idea is always the same but the details differ. The description I give here is the one which applies to GEF.

So please read this before going further, this is not a waste of time.

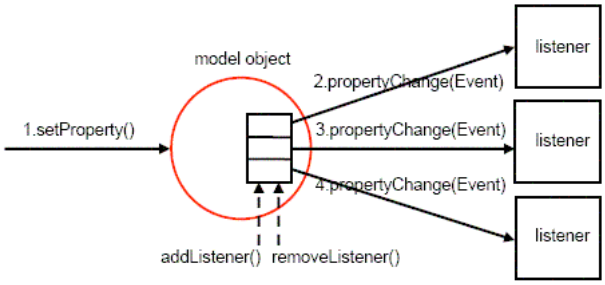
2.1) Description :

The model - view - controller architecture (MVC) is a software architecture which applies to graphical editors. I will explain some of the advantages of this architecture later. All the pieces of GEF are built to take place into this architecture. But the right implementation of this architecture with GEF is up to you.

2.1.1) Model :

I think everybody has an intuitive view of what this term means so I will not try to define it more clearly. However, there are some requirements about the model that must be known :

- **The model must hold all the interesting data you want to be edited by the use**r. All the data which will be made persistent at the closure of the editor should be in your model and only there. **Even data defining the graphical properties of the view** should be stored there (if you don't want to mix the graphical datas with the business datas, see the Randy Hudson tutorial about GEF and other articles on this site).
- **The model must know nothing about the view or about any other part of the editor.** The model must not hold any references to its view. This is very important. The model is just a container of data which get modified during the editing process and signals its changes through a notification mechanism (this is explained in the following point).
- **The model must implement some kind of notification mechanism** : it must fire events when it changes and it must be possible to register listeners to catch these events. This is illustrated here :

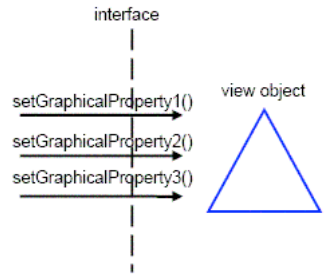


GEF doesn't assume anything about the model you use. This means that you can use almost every model with GEF, and this is good news. But on the other hand, this also means that you will have to take care of the preceding requirements by yourself ! In particular you will have to implement by yourself the notification mechanism, and the listeners for it. But this is easy to do because there are some ready-to-use supports for notification in Eclipse and in the java.beans package. I will give examples of that later.

2.1.2) View :

The view is the set of building blocks which compose the graphical interface. As for the model, the MVC architecture specifies some requirements for the view :

- **The view must not hold any important data which aren't already stored in the model.** This is just a consequence of the requirements for the model.
- **The view must know nothing about the model or about any other part of the editor.** The view must not hold any references to the model. This is very important. The view must be completly dumb and doesn't participate in any way to the logic of the editor. The view can be seen as just a map used by the painting algorithm to paint the graphical interface. Moreover, I think it is good practice to access the properties of the different building blocks of the view through interfaces. This is illustrated here :



In the rest of the page, I will assume that the view is built with Draw2d Figures. GEF provides support for Draw2d Figures and for some kind of tree item figures, but I have never used the second ones so I can't talk about them. I believe (?) that GEF can be used with other kinds of graphical objects than tree figures and Draw2D Figures but it doesn't provide as much support for them as it does for Draw2d figures and for the tree items.

I will give a very little description of Draw2d later but only what is necessary to understand GEF.

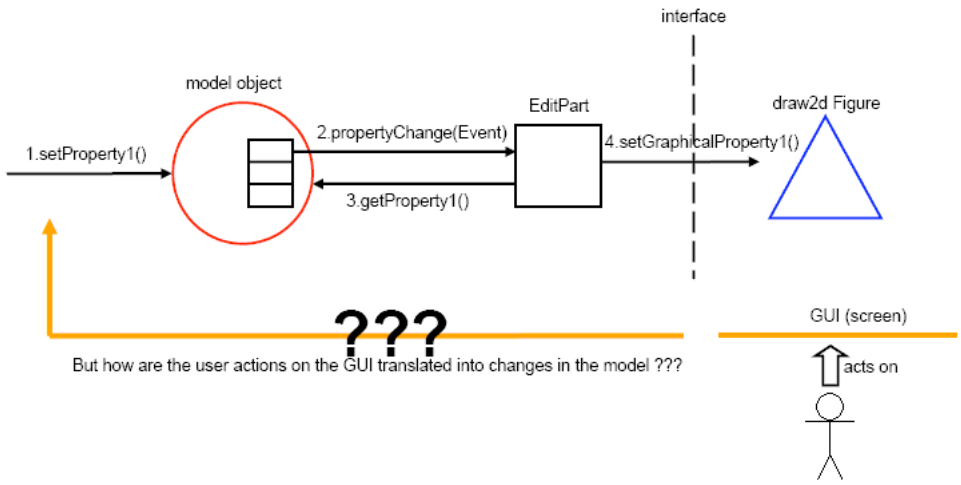
2.1.3) Controllers :

As the model doesn't hold any references to the view and as the view doesn't hold any references to the model, you should be wondering what makes the link between the two. This link is made by the controllers. **In GEF the controllers are subclasses of EditPart**.

There is an EditPart between each model object which has to be graphically represented and its view. The EditPart knows about the two of them (it holds a reference to the model object and to its view) so it is able to gather information about the model object, and to set the graphical properties of the view. The EditPart is registered as a listener of its model object to be informed about its changes, and when these changes occur, it knows how to update the view according to the new state of the model object. The controller is also involved in the editing process of its model object in some way. This will be explained later.

2.1.4) Model - view - controller working together :

Here is an illustration to summarize all this :



So now you should have a good idea about how the changes in the model result in an update of the view, and about which elements are involved in this process. But you still don't know how the actions of the user are translated into changes of the model. This involves too many things that are still to be introduced so it will be explained later.

Remember that this picture doesn't show how GEF works, it shows how you will have to assemble the different pieces provided by GEF.

2.2) Why do you have to use this architecture ?

First of all, GEF is built to be used like this. So if you don't, you will have a lot of problems.

There are many advantages to use this architecture, here are the ones that come to my mind by now :

- The model stays very clean because all the logic of the editor is elsewhere, and it doesn't have any references or dependencies with the other parts of the editor. So it can be reused in another application without any changes.
- The same applies to the view.
- If you have defined interfaces between the controllers and the views, you can easily replace the view by another one without changing anything else.
- Each piece of the editor has a well-defined role : the model stores the data, the view displays the data, and the controllers implement the logic to bind them together. So the code stays very clear because there is a clean separation between the different things, and you don't end up with everything mixed in some kind of a spaghetti plate.
- ...?

3) A short description of Draw2d :

I give here a description of Draw2d, but just the basic knowledge which is necessary to understand GEF. To build a nice GEF editor, with a nice GUI it will not be sufficient, but to follow the rest of the explanations, it should be enough.

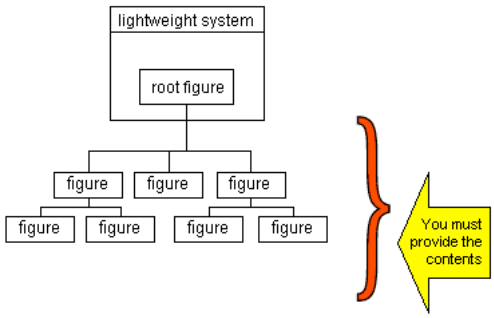
3.1) What it is :

Draw2d is a lightweight system of graphical components which follows the same philosophy as Swing. But unlike swing which has its own MVC architecture, Draw2d is almost purely graphical. There are no draw2d models behind what is shown on the screen, so you won't find in draw2d things like JTables, JList, JTextField, and things like that. Its purpose is only to show things on the screen, not to hold and manipulate any data.

3.2) Figures :

3.2.1) Tree of Figures

Figures are the building blocks of Draw2d. A Draw2d GUI is defined by a **tree of Figures** used by the lightweight system to paint the GUI.



You can add a Figure to its parent by calling parent.add(child) and remove it by calling parent.remove(child).

But what is the meaning of this tree ? What does it change to put some Figure as a child of a parent A or as a child of a parent B ?
This changes a lot of things, and you should understand them with the rest of the explanation and with the pictures.

The Figure class implements the IFigure interface which defines it and which is used in every place where only the generic properties of all the figures are needed. Draw2d provides a lot of useful ready-to-use figures.

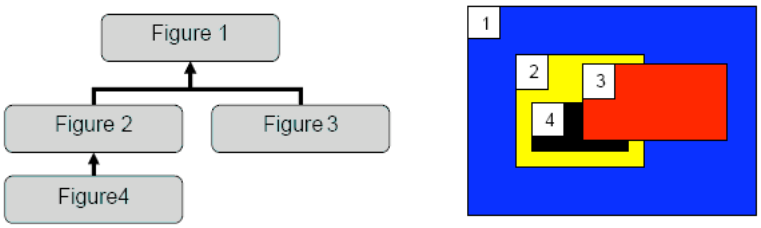
3.2.2) Painting of Figures

Figures have the ability to paint themselves (there is a Figure.paint(Graphics) method).
The Figure.paint(Graphics) method does the following, in the following chronological order :

1. it paints the figure itself by calling the paintFigure(Graphics) method,
2. it paints the children recursively by calling paint(Graphics) on all the children in their appearing order in the list of children,
3. it paints the border of the Figure.

The tree of figures is painted recursively by calling paint(Graphics) on the root figure.

The last two remarks define the painting order of the figures, and thus they define which figure is above another, like this :



This picture shows a tree of figures and its graphical representation if each figure is painted as a full rectangle.

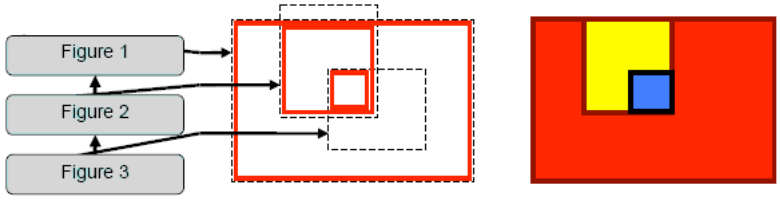
So if A is a child of B in the figures tree, it means that A will be painted above B.

3.2.3) Bounds of Figures :

This is not an easy thing to explain because the usage made of the bounds by draw2d differs from some figures to others.

Each Figure has bounds. For the large majority of the figures, the bounds can be set (for some others, they are calculated based on the content of the figure). **Bounds define a rectangular area outside of which the clipping system of draw2d forbids any painting of the figure or of its children. Each figure is supposed to fill at best the bounds allocated to it with the content it has to display.** The limitation of the space allocated to a figure for painting allows some very important improvements in the painting algorithm.

The clipping windows associated with each figure of a tree are shown here :



The Bounds of the figures are represented as dash lines, each figure is painted as a full rectangle with a black border, the clipping area associated with each figure is represented as a red line.

So if A is a child of B in the figure tree, A will not be allowed to paint itself outside of the bounds of B.
In other words, **if A is a child of B in the figure tree, it means that A is contained by B.**

Moreover, if you set the bounds of a figure in such a way that it moves, then all the children tree will move the same.
In other words, **if A is a child of B in the figure tree, it means that A moves with B.**

3.3) Interesting features :

- **Layout managers :** You can set a layout manager to every figure. Layout managers are objects responsible for adjusting the bounds of the children of their owner figure, given the bounds of their owner figure, the geometrical properties of the children (preferred size, maximum size, minimum size) and possibly some constraints associated with the children. They are also responsible for calculating the preferred size, minimum size and maximum size (?) of their owner figure if these geometrical properties were not explicitly set by the user. Draw2d provides a lot of layout managers to suit your needs. **So if A is a child of B in the figure tree, it means that A is positioned by the layout manager of B.**

- **Hit - testing :** There is an algorithm in draw2d to recursively determine the top most figure which is likely to have put

visible paintings at a given point. This is based on the bounds of the figures and possibly on the IFigure.containsPoint() method if it is overriden.

- **Events :** It is possible to register different kinds of figure listeners, like mouse listeners and mouse move listeners. Draw2d routes the mouse events to the top most figure at the mouse location which is able to understand them (= which has a listener registered for that kind of events, is visible and enabled,...).
  But don't be confused by this : the way GEF reacts on the user actions on the GUI doesn't involve such figure listeners.

- **Predifined types of Figures :** Draw2d provides a lot of figure types. There are scroll panes, geometrical shapes, polylines, labels, figures to show images, buttons, checkboxes, figures to show some texts, ...

- **Predifined types of Borders :** There are a lot of borders too, and they can be composed to build complex borders. This is very similar to Swing.

- **Connections :** Draw2d provides a lot of support for connecting figures and this is not so simple as it seems. **This will be explained later**, when we need it to understand connections in GEF.

- **Cursors and tooltips :** For each figure, you can set the cursor which will be shown when the mouse moves above the top-most visible figure, and the tooltip text which will be shown when the mouse hovers above the top most visible figure.

- **Layers :** Draw2d provides layers which are figures transparent for hit-testing. They are used in GEF to put different things in different stacked levels. There is a special layer for holding connections.

4) Building the view when the editor is opened:

This is the first problem which GEF helps you to solve to build a graphical editor. You have a model, and you want it to be displayed to the user in some way for the first time, when you open the editor. In fact, the problem of creating the view is not really different from the problem of updating the view but I have decided to separate them because it makes the things a lot easier to explain.

The elements involved in the creation of the view are :

- the EditPartFactory, if you use one (I will assume it is the case),
- the EditParts,
- the GraphicalViewer,
- the Draw2d Figures.

Once again, this is not just an explanation of what happens in the GEF black box : if you understand how the view gets built, you will know how to write the code to build the view which suits your needs. Once you have read this section, you will be able to write a code which automatically builds the view associated with your model when the editor is opened. But there will still be no updating of this view if the model changes, and no editing capabilities provided to the user.

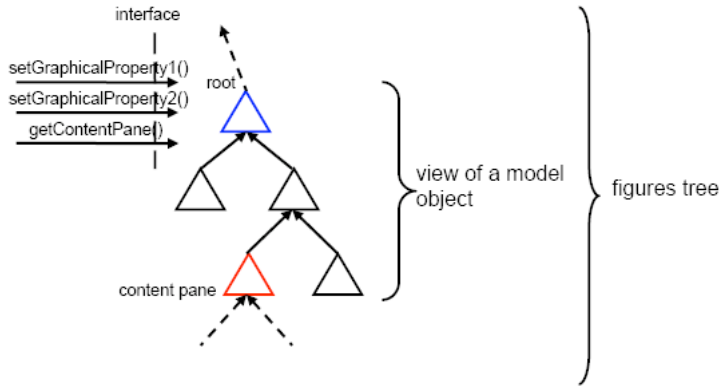
4.1) What you have to do :

For each model object you want to be represented in the view, you will have to write three things :

- the **view** of the model object : a subclass of Figure to represent this model object, and possibly an interface to access to its properties,
- a **subclass of EditPart** to bind the model object to its view,
- some lines of code in the **EditPartFactory** which tells GEF which EditPart to build given a model object.

4.2) The view of a model object :

The view of a model object doesn't have to be a single Figure. It will more likely be a subtree of Figures :



You may wonder about one thing in this picture : the content pane. I didn't talk about that thing till now. Sorry, this will be explained later. For now think about it as a leaf of the figure subtree which serves as a container in the view of the model object and can be filled by GEF with some things if it is needed. You don't necessarily have to explicitly define a content pane. If you don't, GEF will use the root figure of the subtree as the content pane.

Example:
Here is an example of a view which could be used to represent a person with a name and a surname, and the list of the fruits he likes eating (this list of fruits isn't actually a part of the view : it's the content pane).

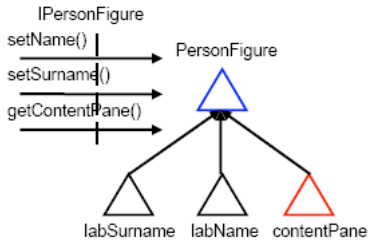
Interface for accessing the properties of the person view

```
public interface IPersonFigure extends IFigure{
   public void setName(String name);
   public void setSurname(String surname);
   public IFigure getContentPane();
}
```

Figure to represent a person

```
public class PersonFigure extends Figure implements IPersonFigure{
```

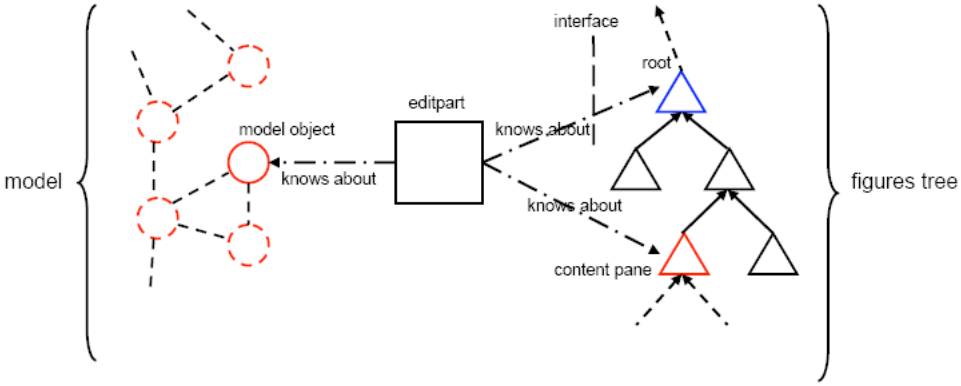
```
        private Label labName;
        private Label labSurname;
        private IFigure contentPane;
        public PersonFigure(){
                setLayoutManager(new ToolbarLayout());
                labName = new Label();
                add(labName);
                labSurname = new Label();
                add(labSurname);
                contentPane = new Figure();
                contentPane.setLayoutManager(new ToolbarLayout());
                contentPane.setBorder(new TitleBorder("fruits list :"));
                add(contentPane);
        }
        public void setName(String name){
                labName.setText(name);
        }
        public void setSurname(String surname){
                labSurname.setText(surname);
        }
        public IFigure getContentPane(){
                return contentPane;
        }
    }
```



Now you should wonder where you have to write the code to instanciate the view associated with a model object, and where you have to write the code to fill the view with the data from the model object to display. You already know that the controller links the view object to the model object and knows about them. So you may already guess the answer : this code belongs in the EditPart between the view object and the model object and I will explain how to do this in the next section.

4.3) The EditPart linking each model object to its view :

This picture shows how the EditPart takes place between the model object and its view :



The EditPart knows about two things in the Figures subtree of the view of the model object :

- the **root** of this subtree,
- a special leaf (I will assume it is a leaf but I don't think it is required) of this subtree called the **content pane**.

The base class for EditParts using Draw2d Figures is AbstractGraphicalEditPart. Each AbstractGraphicalEditPart links a model object to its view. You can set/get the model object associated with an EditPart by calling setModel(Object) and getModel(Object). You can get the (root) Figure associated with an AbstractGraphicalEditPart by calling getFigure(). You cannot set this figure, it is built by GEF when it is needed.

So now you have different types of model objects to represent, and you have written the Figure classes (and possibly interfaces) to represent each of these types of model objects. **For each type of model object to link to a view, you have to write a subclass of AbstractGraphicalEditPart which will link this type of model object to its view.** For each of the AbstractGraphicalEditPart subclasses, you have **some methods to implement / override** (more will come later, here are just the ones necessary to allow GEF to build the view) :

- **protected IFigure createFigure()** : must return a new view associated with the type of model object the EditPart is associated with,
- **public void refreshVisuals()** : must fill the view with data extracted from the model object associated with the EditPart, it will be called just after the creation of the figure,
- **protected IFigure getContentPane()** : must return the content pane of the view (if you don't override this method, the default content pane is the root figure of the view),
- **protected List getModelChildren()** : must return the list of model objects that should be represented as children of the content pane of the view (if you don't override this method, the empty list is returned).
- (for now I assume there are no connections, they will come later)

You can see that the createFigure(), getContentPane() and getModelChildren() methods are protected. So you will not have to call them, they will be called by GEF in the process of building the view when it is needed. The way these methods are used by GEF will become clearer later, for now, just try to understand which code you have to put in them.

4.3.1) createFigure() method :

To continue with the same example as before, suppose you want to represent a Person model object and that you have written a

PersonFigure class and an IPersonFigure interface like before.You now have to write a PersonEditPart which extends AbstractGraphicalEditPart to link the Person model objects with the PersonFigures. For the createFigure() method, you would just have to write:

```
public class PersonEditPart extends AbstractGraphicalEditPart{
...
   protected Figure createFigure(){
           return new PersonFigure();
   }
...
}
```

4.3.2) refreshVisuals() method :

This is very simple too. You just have to get a reference to the model object, another one to the view and then you get the data from the model and apply them to the view :

```
public class PersonEditPart extends AbstractGraphicalEditPart{
...
   public void refreshVisuals(){
           IPersonFigure figure = (IPersonFigure)getFigure();
           Person model = (Person)getModel();
           figure.setName(model.getName());
           figure.setSurname(model.getSurname());
   }
...
}
```

4.3.3) getContentPane() method :

If you have defined a content pane, it should be accessible through your figure interface so you write :

```
public class PersonEditPart extends AbstractGraphicalEditPart{
...
   protected IFigure getContentPane(){
           return ((IPersonFigure)getFigure()).getContentPane();
   }
...
}
```

4.3.4) getModelChildren() method :

You will get a better idea of what you have to put in this method when you see how the view is built by GEF. I give here a little explanation for you to get the general idea.

A lot of models must be represented as a containement hierarchy, with parent views containing their child views (this does not mean that the model itself has to be a tree, I'm talking about the view here). If you consider a UML diagram, then the diagram contains the classes, the classes contain the attributes and the methods, and the methods contain some parameters. Each EditPart has to implement the getModelChildren() method to return to GEF the List of model objects which must be represented as children of the content pane of the EditPart. When GEF creates the EditParts and figures for these model objects, their EditParts will be added as children of the current EditPart and their figures will be added as children of the content pane of the current EditPart.

In the example, the Fruits objects have to be represented as children of the content pane of the PersonEditPart, so we implement the getModelChildren method like that :

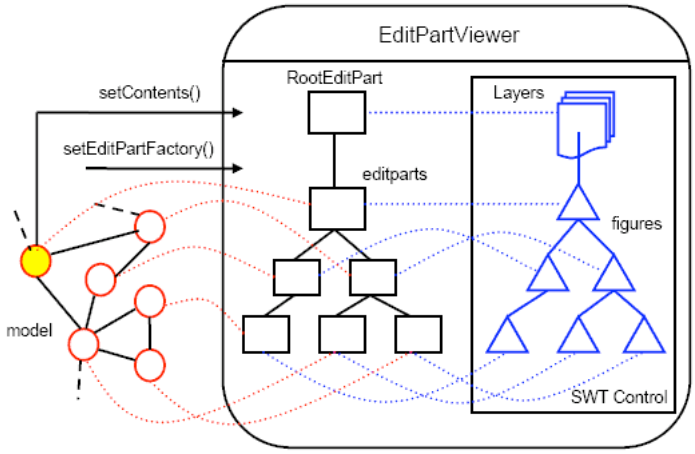
```
public class PersonEditPart extends AbstractGraphicalEditPart{
...
   public List getModelChildren(){
           Person model = (Person)getModel();
           return model.getFuits();//must be non null, if there are no children,
                                   //then return List.EMPTY_LIST (or smthg like that)
   }
...
}
```

4.4) EditPartViewer, RootEditPart :

An EditPartViewer is reponsible for installing one view of your model on a SWT Control. Nothing prevents you from creating multiple views of your model, each of them with its own EditParts and Draw2d Figures, and each of them will be installed on an SWT Control with an EditPartViewer. If your view is composed with Draw2d Figures, you will have to use a subclass of EditPartViewer : GraphicalViewer. You give it the Control, an EditPartFactory, a RootEditPart and a model object called the content and it will display your model to the user. It is also responsible for some other things which concern the editing capabilities but I can't explain them now without having to introduce new GEF elements so I will leave it for later.

The EditParts are organized in a tree structure. You will understand what this structure is when you see how the view of your model is built. The root of this structure is the RootEditPart. The figure associated with the RootEditPart is composed of a set of stacked draw2d Layers where the different elements of the view will take place. So by choosing which RootEditPart you will use, you also choose some properties of the view, like its ability to be scrolled, scalled, and things like that. GEF provides a few implementations of RootEditPart, you can look in the javadoc for a description of each of them and choose the one which suits your needs.

Don't worry too much about this, it is not the important part of the work and you can copy/paste some code to create this part of your editor from some example. What is important (for now) are the model, the EditParts, the Figures and the EditPartFactory.



Some other roles of EditPartViewer are :

- **keeping a Map from the model objects to the EditParts** to allow you to find which EditPart is associated with a given model object. You can get this map by calling EditPartViewer.getEditPartRegistry(). This could be useful for example if you want to put in the current selection the EditParts associated with some given model objects.
- for GraphicalViewer, **keeping a Map from the figures to the EditParts**. For example this allows the GraphicalViewer to know which EditPart is associated to a given figure and to..
- ..**perform hit-testing to know which EditPart satisfying some conditions is at a given location** (this will be explained later),
- providing the workbench ISelectionService with the selected EditParts (not too sure about this).

example :

[TODO : write a code snippet showing the GraphicalEditorPart subclass for the installation of a GraphicalViewer with a FreeformRootEditPart to show a model (if somebody could do that...).]

4.4) EditPartFactory :

In the process of building or refreshing the view installed on a particular EditPartViewer, GEF often needs to create the proper EditPart to represent some model object in this EditPartViewer. When GEF needs it, it asks the EditPartFactory of the EditPartViewer to build an EditPart for the model object to represent in that EditPartViewer. So you have to implement an EditPartFactory yourself for each EditPartViewer you use. Here are the interface provided by GEF for EditPartFactory and an example :

EditPartFactory interface :

```
public interface EditPartFactory{
   EditPart createEditPart(EditPart context, Object model);
}
```

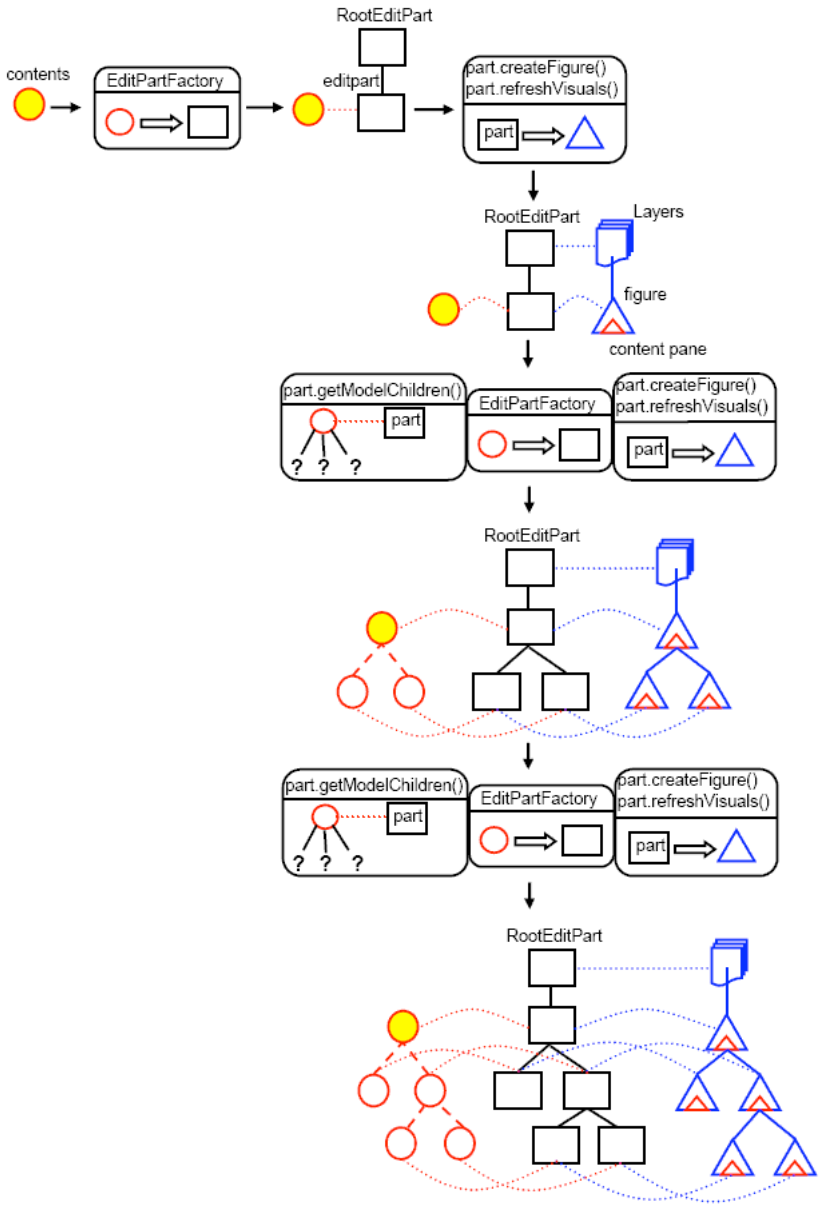
Example :

```
public interface MyEditPartFactory implements EditPartFactory{
   EditPart createEditPart(EditPart context, Object model){
            EditPart part = null;
            if(model instanceof Diagram){
                    part = new DiagramEditPart();
            }
            else if(model instanceof Person){
                    part = new PersonEditPart();
            }
            else if(model instanceof Fruit){
                    part = new FruitEditPart();
            }
            part.setModel(model);//GEF doesn't do this automatically,
                                 //you have to do it yourself here
            return part; //<--(part should never be returned as null)
   }
}
```

4.5) How GEF builds the view :

Now we have all we need to understand the way GEF builds the view. First I will describe what is built and how it depends on the code you have written, and then I will describe how it is built in a more detailled way.

Here is a picture which describes the process approximately :



Here is a chronological description of what happens during the building of the view.
This is a recursive process which takes place in the EditParts themselves and is triggered by the addition of the content EditPart (the one associated with the content object) to the children of the RootEditPart of the viewer. But the only thing that matters and is explained here is what happens, not how exactly it is implemented.

1. A particular model object called the contents is passed to the EditPartViewer.
2. the EditPartFactory is used to build an EditPart for this model object, this EditPart is called the content EditPart.
3. the createFigure() and refreshVisuals() methods are called on the content EditPart to build its Figure (its subtree of Figures...), and refresh its properties, and then this Figure is added the appropriate layer of the RootEditPart.
4. the getModelChildren() method is called on the content EditPart to know which model objects have to be represented as children of the content pane of the content EditPart.
5. For each model object in the list returned by getModelChildren(), the appropriate EditPart is built using the EditPartFactory and added as a child of the content EditPart.
6. the methods createFigure() and refreshVisuals() are called on each of these children EditPart to build their Figures and refresh their properties, and then these figures are added as children of the content pane of the content EditPart.
7. the method getModelChildren() is called on each child EditPart and the process continues untill the tree of EditParts and Figures is completely built (the process will stop when all the getModelChildren() methods of the leaves EditParts return an empty list)

An alternative to EditPartFactory :

Each EditPart has the responsability to build its own children EditParts given the list of the model objects which should be represented as children of the content pane returned by getModelChildren(). This is the job of the **EditPart.createChild(Object model) method**. By default, this method gets the EditPartFactory of the viewer the EditPart belongs to and delegates the job to it. But if you don't like this default behaviour, you can override the createChild method to build the child EditPart given a model object (the context EditPart is no longer needed as parameter because the context EditPart is this).

4.6) Defining a view for your model :

This section summarizes the different ways you hook into the process of building the view, and their effects.

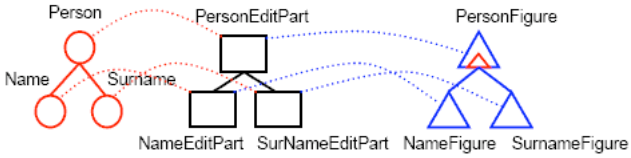
The EditPartFactory defines which EditPart class will be associated with each type of model object, and each EditPart class defines which type of view will be associated with its model through its methods createFigure() and refreshVisuals(). **So by defining the EditPartFactory and the createFigure() and refreshVisuals() methods of the EditParts and the Figures, you define the view associated with each model object and the link between the properties of each model object and the graphical properties of its view.**

The getModelChildren() methods of the EditParts defines the structure of the view, i.e., which model objects will be represented "inside" (in the content pane of) the representation of another model object. **So by defining the getModelChildren() methods of the EditParts, you define the tree structure of the view.**

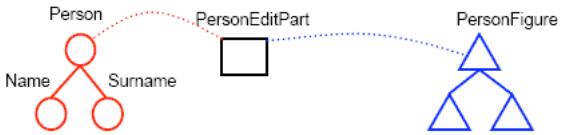
4.7) EditPart or not EditPart ?

But there is one question left : which model objects must be represented with EditParts and their associated Figures, and which model objects must simply be represented as a property of the view associated with another model object, without an associated EditPart ?

For example, if you want to represent a person, with a name and a surname, you could think at least about two different things :



OR



And you could also think about using only one big EditPart to link the whole model to the whole view (but it would be a very very bad idea).

So what is the right thing to do ? How can you choose the right "granularity" of the EditParts ?

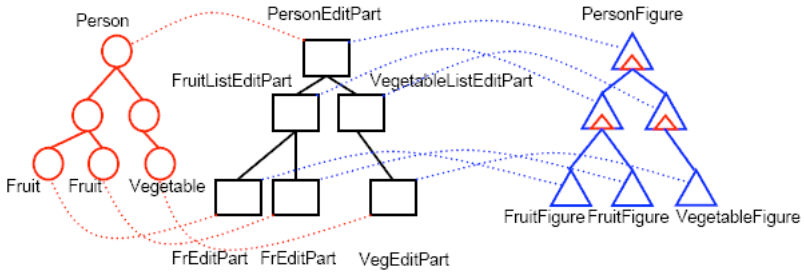
The choice is up to you but there are some things to keep in mind when you make your choice about this :

- GEF doesn't understand the details of the figures which compose the GUI, it only understands the EditParts associated to these figures, so **if the user must be able to manipulate and interact individually with a model object represented in the GUI, this model object must be associated to an EditPart.** You can see that in the GraphicalViewer class : this class provides a method findEditPartAt() (and not findFigureAt()) which is the one used by GEF for editing purposes.
- The selection provided by GEF to the workbench is composed of EditParts, and the content of the properties view depends on the current selection in the workbench. So if you want the properties of a model object to be editable in the properties view, you have to build an EditPart for it in order to make it selectable. In other words, **if you want the representation of a model object to be selectable in the workbench, this model object must be associated to an EditPart**.
- The less EditParts you choose to use, the more complicated they will be, and their figures too.
- ...?

4.8) What if multiple content panes are needed for an EditPart ?

Suppose you want to represent in the view of a Person the list of the fruits, but also the list of the vegetables he likes eating and you want to separate the list of the fruits and the list of the vegatables. This is a problem because GEF only allows one content pane by EditPart. So what can you do ?

If you want the fruits grouped in a content pane and the vegetables grouped in another content pane, you don't have the choice : you have to provide two EditParts, one with a content pane containing the fruits and with a getModelChildren method returning the list of the fruits, the other one for the vegetables, and these two EditParts have to be children of the PersonEditPart in order to subdivide its content pane artificially. Like that :



These two new EditParts must listen to the Person model object to receive events when a fruit or a vegetable is added to the Person model and call refreshChildren on themselves when that happens (I suppose here the lists are too dumb to notify their changes themselves).

But what will be the model of these EditParts and how will you build them ? This was not shown on the picture.

If you set the model of these two EditParts as the Person object, there will be three EditParts with the same model and that will cause problems because the EditPartViewer maintains a Map [model objects -> EditParts] and that map allows only one EditPart by model object. Only the last created EditPart with the Person object as its model will be in the Map and that could be annoying later. If the fruits and vegetables lists are directly accessible, then you could choose to set them as the models of the new EditParts. But what if they are not ? I give here the solution I have found in this case, I don't know if it is the good solution but it seems to work :

PersonEditPart :

```
public class PersonEditPart extends AbstractGraphicalEditPart{
...
   private Object dummyFruitsListModel = new Object();
   private Object dummyVegetablesListModel = new Object();

   public List getModelChildren(){
           List list = new ArrayList();
           list.add(dummyFruitsListModel);
           list.add(dummyVegetablesListModel);
           return list;
   }
   public EditPart createChild(Object model){
           EditPart part;
           if(model == dummyFruitsListModel)
                   part = new FruitsListEditPart();
           else
                   part = new VegetablesListEditPart();
           part.setModel(model);
           return part;
   }
...
```

```
}
```

[**note**: this code snippet includes some elements which will be explained in the next section (updating the view), you may want to read the next section before reading this]
FruitsListEditPart :

```
public class FruitsListEditPart extends AbstractGraphicalEditPart implements propertyChangeListener{
...
   public void activate(){
           if(!isActive()) {
                   Person p = (Person)getParent().getModel();
                   //the Person model object is the model of the parent,
                   //not of this

                   p.addPropertyChangeListener(this);
           }
           super.activate();//do not forget this like I always do or it will not work...
   }
   public void deactivate(){
           Person p = (Person)getParent().getModel();
           p.removePropertyChangeListener(this);
           super.deactivate();
   }
   public List getModelChildren(){
           Person p = (Person)getModel();
           return p.getFruits();//maybe this is not the original list of fruits but a copy
                                //of it to prevent unwanted modifications,
                                //so this list couldn't have been the model of the EditPart
   }
   //I suppose the EditPartFactory of the GraphicalViewer knows what to do with Fruit objects
   //so there is no need to override the createChild() method
   public void propertyChange(PropertyChangeEvent ev){
           if(ev.getPropertyName().equals(Person.PROPERTY_FRUITS)
                   refreshChildren();
   }
...
}
```

5) Updating the view when the model is changed :

5.1) Notification mechanism :

If you remember the MVC pattern I explained before, **the model must notify its changes to its listeners**. Here is an example for the class Person, like before :

Example :

```
import java.beans.*;
import java.util.*;
public class Person{
   private String name;
   private List fruits;
   private PropertyChangeSupport listeners;
   public final static String PROPERTY_NAME = "name";
   public final static String PROPERTY_FRUITS = "fruits";
   public Person(){
           name = "";
           fruits = new ArrayList();
           listeners = new PropertyChangeSupport(this);
   }
   public void setName(String newName){
           String oldName = name;
           name = newName;
           listeners.firePropertyChange(PROPERTY_NAME, oldName, newName);
   }
   public void addFruit(Fruit fruit){
           fruits.add(fruit);
           listeners.firePropertyChange(PROPERTY_FRUITS, null, fruit);
   }
   public void removeFruit(Fruit fruit){...}
   public void addPropertyChangeListener(PropertyChangeListener listener){
           listeners.addPropertyChangeListener(listener);
   }
   public void removePropertyChangeListener(PropertyChangeListener listener){...}
   public String getName(){...}
   public List getFruits(){...}//the returned list should not be modified because
                           //no events would be fired in that case
}
```

5.2) Listening to the model :

In the MVC pattern, the controllers must listen to the changes of the model. In GEF, **EditParts are the controllers so they must listen to their model to update the view according to the new state of the model.**

When an EditPart is added to the EditParts tree and when its figure is added to the figure tree, the method EditPart.activate() is called. When an EditPart is removed from the EditParts tree, the method deactivate() is called. So these are the good places to hook / unhook the EditPart to / from its model. Here is an example for the class PersonEditPart :

Example :

```
import java.beans.*;
import org.eclipse.gef.*;
import org.eclipse.gef.editparts.*;
public class PersonEditPart extends AbstractGraphicalEditPart implements PropertyChangeListener{
   ...
   public void activate(){
           if(!isActive())
```

```
                ((Person)getModel()).addPropertyChangeListener(this);
            super.activate();
        }
        public void deactivate(){
            ((Person)getModel()).removePropertyChangeListener(this);
            super.deactivate();
        }
        public void propertyChange(PropertyChangeEvent ev){
        ...
        }
        ...
    }
```

Don't forget to unhook an EditPart in its deactivate method ! If you don't, some repainting/revalidating code could be called on its figure when events are fired by the model but as its figure is no longer part of the figure tree, this could cause weird errors which would make you scratch your head for days.

5.3) Reacting to the events fired by the model to update the view :

When the model evolves, you always want that :

- the properties of the figures associated with the model objects **stay the ones defined by** the refreshVisuals() methods of the EditParts which link each model object to its view,
- the structure of the view **stays the one defined by** the getModelChildren() methods of the EditParts indicating to GEF which model objects should be represented as children of the content pane of an EditPart.

GEF provides **two methods to help you to achieve this goal** :

- the **refreshVisuals()** method to refresh the view associated with an EditPart according to the new state of its model,
- the **refreshChildren()** method to update the children EditParts according to the getModelChildren() method of the EditPart (changes in the model which need this kind of refreshing are called structural changes).

The refreshChildren() method compares the model objects of the children EditParts of an EditPart to the List of objects returned by its getModelChildren() method. Then it creates new EditParts for the model objects which are represented in the getModelChildren() List but not already represented as model of children EditParts of the EditPart, and it drops the children EditParts which have a model which is not in the getModelChildren() List. So after calling the refreshChildren() method, the children EditParts of the EditPart are the ones defined by its getModelChildren() method, as desired.

You have to call yourself these two methods on an EditPart in response to events fired by the model object of this EditPart.

Here is an example of this for the PersonEditPart class :

Example :

```
import java.beans.*;
import org.eclipse.gef.*;
import org.eclipse.gef.editparts.*;
public class PersonEditPart extends AbstractGraphicalEditPart implements PropertyChangeListener{
    ...
    public void propertyChange(PropertyChangeEvent ev){
            if (ev.getPropertyName().equals(Person.PROPERTY_NAME))
                    refreshVisuals();
            else if (ev.getPropertyName().equals(Person.PROPERTY_FRUITS))
                    refreshChildren();
    }
    public void refreshVisuals(){
            IPersonFigure figure = (IPersonFigure)getFigure();
            Person model = (Person)getModel();
            figure.setName(model.getName());
            figure.setSurname(model.getSurname));
            super.refreshVisuals();
    }
    ...
}
```

Goto page 2

Comments:

From vanto [217.230.8.253] - 29.06.04 08:23

Great work! Thanks!

From Chhil [66.176.91.101] - 07.07.04 17:18

Very impressive...thanx for making it available. I hope this ends up on the eclipse articles page.

From Natan [83.130.225.185] - 26.08.04 19:16

great job!!! and your greatness is twofold))): first, u covered a very popular subject - a ground up GEF tutorial. And secondly, u did it in a very simple and understandable way. thanks a lot))

From Natan [83.130.225.185] - 26.08.04 19:17

great job!!! and your greatness is twofold))): first, u covered a very popular subject - a ground up GEF tutorial. And secondly, u did it in a very simple and understandable way. thanks a lot))

From absolut [83.130.225.185] - 26.08.04 19:17

great job!!! and your greatness is twofold))): first, u covered a very popular subject - a ground up GEF tutorial. And secondly, u did it in a very simple and understandable way. thanks a lot))

From john Lee [63.187.208.187] - 31.08.04 23:52

really appreciate your work. Thank you very much.

From Randy Hudson [129.33.49.251] - 02.09.04 09:39

Very nice! How about we make this thing official? If you want to contribute this to Eclipse.org, we can look into the copyright issues and your donating it. The diagrams you made are great. Some pieces look familiar :-)

From rlemaigr - 02.09.04 15:21

You've got mail ! The mail address I used is the one mentioned here: http://www-106.ibm.com/developerworks/opensource/library/os-gef/#author1 (sorry for the pieces that I...a-hem..."borrowed" from the official documentation but as I was trying to promote your work with them, I believed that there shouldn't be too much complaints about this :D)

From rlemaigr - 04.09.04 20:29

Please log in and describe the changes you made when you make some. "typo" seems to have change something lately but I don't know what. I'm modifying the page offline on my HD, so if somebody makes a change and doesn't describe it, there is a risk it will be deleted when I upload the new version of the page. Thanks.

From frank [217.9.27.150] - 12.09.04 12:02

Please make sure the program code (mostly the comments) in the grey boxes is not to wide. Else the text doesn't wrap and you have to scroll vertically to read.

From rlemaigr [80.201.136.35] - 12.09.04 13:10

You're probably right. I was wondering why the horizontal scrollbar was there...I will correct it, thank you for your advice.

From dhiraj [196.12.47.2] - 05.10.04 11:06

Good piece of work. Neatly explained ...was quite helpful to me

thanx

From Frank [217.9.29.85] - 08.10.04 12:26

I suggest renaming the "next"-link to somehting like "...go to in part 2". It`s easily missed, i know I missed it the first few times.

From sbl - 30.11.04 23:58

I think this page is a very good introduction into GEF :-).
With GEF it's possible to build plugins for eclipse. So it would
also be useful to show how to setup a plugin project in eclipse and get it run.

Yours
sbl

From RudyBOne - 16.02.05 08:32

Excellent article. Need a way to print the article however. Printing the web page directly causes the right edge of the article to be cut off. I had to copy the article body and paste it to a word document to get it to print.

From n16er - 08.06.05 12:55

I'm brand new to GEF and therefrore not too keen on fixing what appears to be a mistake in this fine article but I'll point out the issue all the same hoping one of you more experienced fellows will concur. In section 4.8 in the listing for **`FruitListEditPart`** in the method getModelChildren() it seems to me that

```
Person p = (Person)getModel();
```

is incorrect and should be

```
Person p = (Person)getParent().getModel();
```

What say the rest of you?

From kaliask - 24.11.05 17:46

I can't see all the diagrams on this page. Can you please fix the page.

From mateu - 07.02.06 01:01

I'm reading your document and I wanted to do a 'correction'. When you say that EditPart provides workbrench ISelectionService that enables to get selected EditParts you say that you are not sure. It is true.

Somethin like that will do it (returning a list of the selected editParts:

```
if (!(viewer.getSelection() instanceof IStructuredSelection))
            return Collections.EMPTY_LIST;
        return ((IStructuredSelection)viewer.getSelection()).toList();
```

From diwant - 15.03.06 18:25

Somebody has deleted a LOT from this tutorial! Please restore it (I think Version 34 still had the stuff) as this material was very important and useful.

Great tool. Shame someone thinks the comments page is an opportunity for SEO

Last Modified 14.06.06 15:44