INFO-Y-080: Software Architectures
Assignment 1: Design Patterns

Dehouck Samuel and Delhaye Quentin

6th March 2014

Factory Method

It allows classes to ask `ShapeEditFactory` to instanciate classes implementing `EditPart` by simply specifying the type of object they want.

`EditPartFactory:` Interface declaring the createEditPart method, which returns an EditPart.
`ShapeEditPartFactory:` Implements EditPartFactory and overrides the EditPart creation method. It will create one object or an other based on its Object argument.
`EditPart:` Interface of the concrete products the factory creates.
`DiagramEditPart, ShapeEditPart, ConnectionEditPart:` Concrete products implementint EditPart.

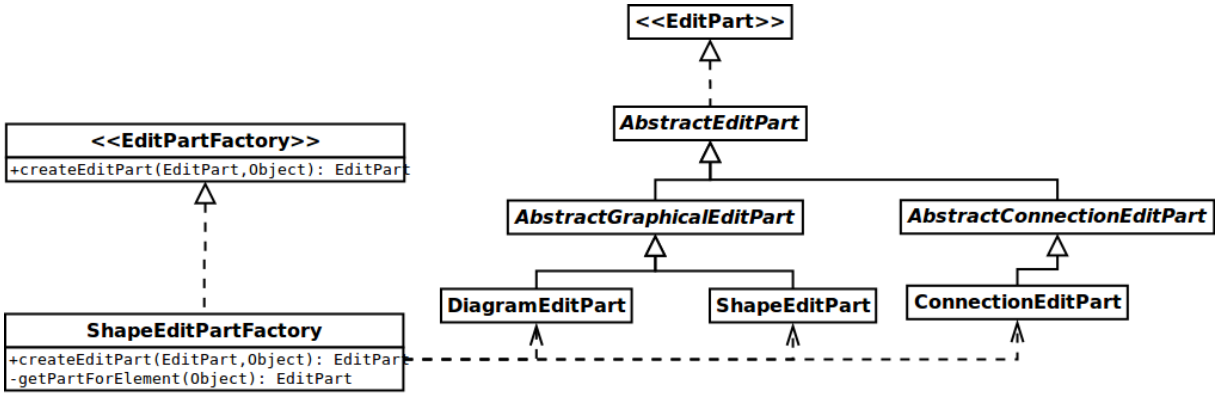

Figure 1: Factory method class diagram

Command

`ShapeEditPart` creates a bunch of commands for some objects. When someone wants to acts on those, he simply requests the command and executes it through his command stack.

`CommandStack:` Acts as the invoker and asks Command to execute the request.
`Command:` General class delcaring a method execute.
`ShapeEditPart:` Client of the pattern. It will create the concrete command.
`ConnectionCreateCommand:` Executes the action specified in its implementation (i.e. creating a connection between two shapes).
`Connection, Shape:` The receivers of the command. The concrete command creates a new Connection between two existing Shapes.

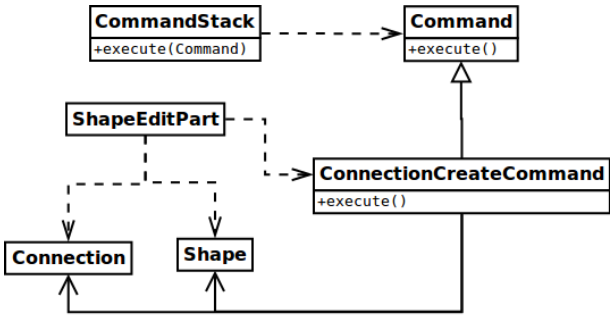

Figure 2: Command class diagram

State

When we ask to load the image of a shape through `ShapeTreeEditPart`, the latter does not know for whom he's about to load an image. It posses an instance of that object and just gives it to the state manager who will do the right thing depending on the object type.

`ShapeTreeEditPart:` Class offering the possibility to load the image of the shape. It holds an instance of the current shape which is accessed through the `getModel()` method.
`Shape:` Class encapsulating the behaviour of all its subclasses regarding the loading if their image.
`RectangularShape, EllipticalShape:` Specific class, each loading a different image.

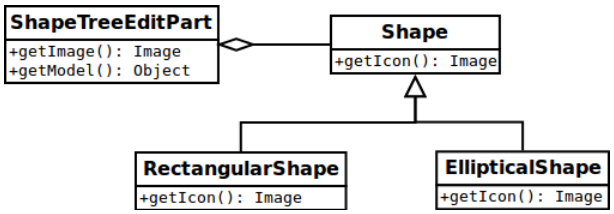

Figure 3: State class diagram

Composite

Organizes the elements in a tree so that the client can handle the whole tree as well as subelements the same way.

`EditPart:` Interface of all the elements in the tree.
`AbstractEditPart:` Nodes of the tree. It contains a list of its children as well as its parent.
`AbstractGraphicalPart:` Leaf of the tree (as are all the classes inheriting this abstract class).
`AbstractEditPartViewer:` Client playing with the `EditPart` components.

Singleton

`ShapesPlugin` holds – among other things – the preferences of the plugin. This does not need to be instanciated more than once and can be shared between all the classes needing it. Therefore, only one instance needs to be kept, hence the singleton.

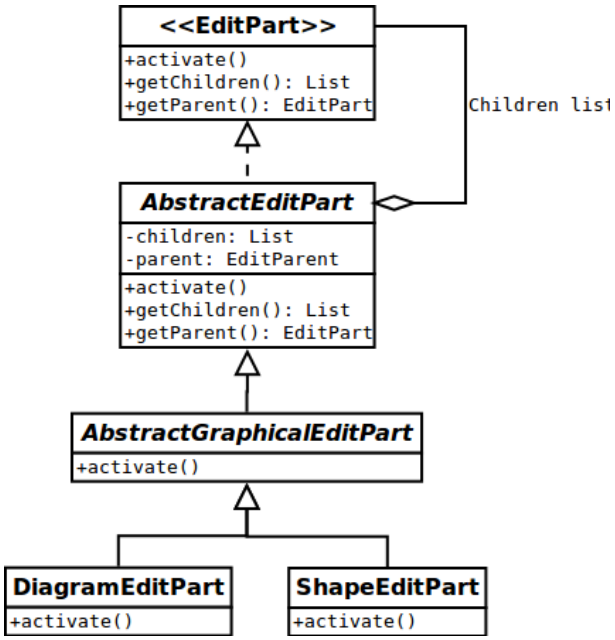

Figure 4: Composite class diagram

`ShapesPlugin:` Creates an instance of ShapePlugins and offers a method to access it.

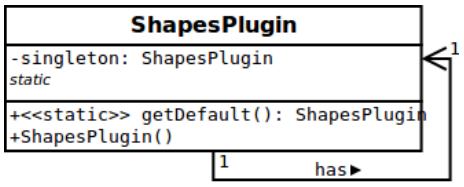

Figure 5: Singleton class diagram

MVC

The MVC pattern used by GEF differs from the usual by the fact that the model does not directly notify the view of its changes. Instead, it notifies de controller by the means of listeners. The controller will then get the model and update its corresponding view.

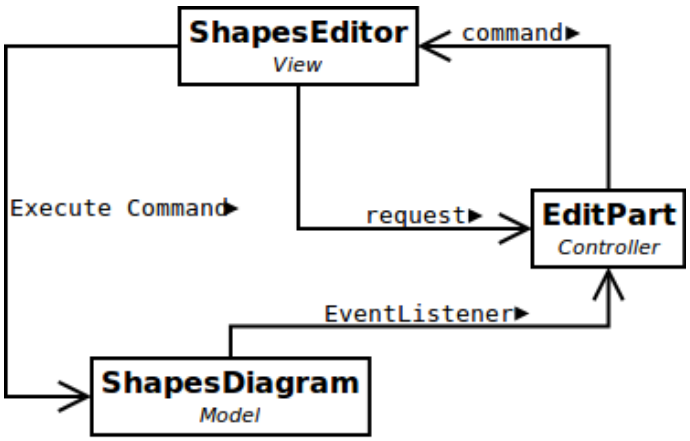

Figure 6: MVC class diagram

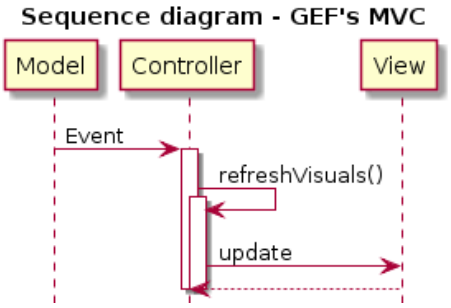

Figure 7: Singleton class diagram