The Composite Pattern
Brahma Dathan

Most drawing programs on the market allow users to select multiple shapes in the drawing and apply common operations such as delete, move, change color, etc., so the operations are carried out on all selected shapes.

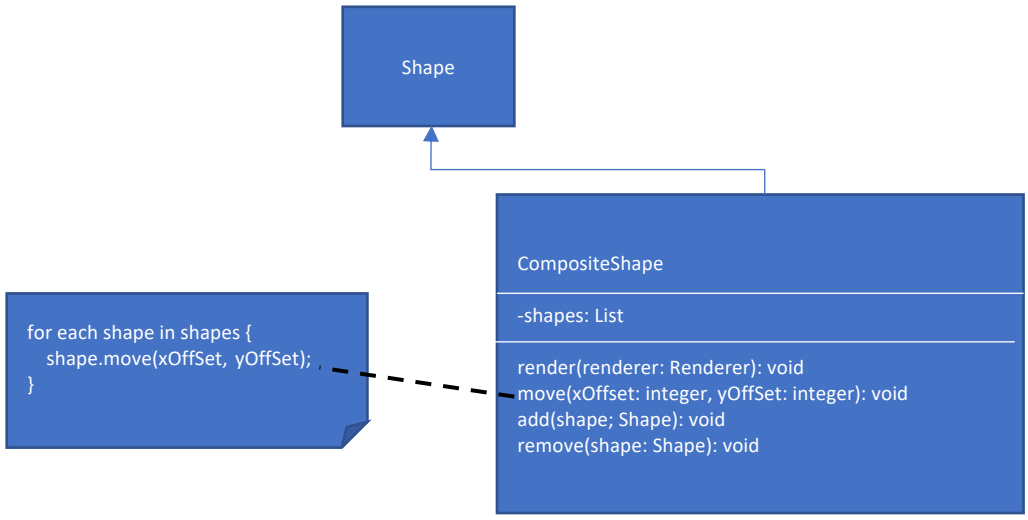
Suppose we want to do something similar in our drawing program. We would like to do this in a manner, so the view can still use the existing code given below.

```
public void draw(Renderer renderer) {
   for (PropertyChangeListener listener : propertyChangeSupport.getPropertyChangeListeners()) {
      Iterator<Shape> items = Model.instance().getShapes();
      while (items.hasNext()) {
         Shape item = items.next();
         item.render(renderer);
      }
   }
}
```

We may also have other operations such as move() defined for Shape objects, which we would like to apply to all selected Shape objects, when we execute code like the following.

Shape shape = …
shape.move(xOffset, yOffSet);

To facilitate such seamless operations on all selected Shapes without modifying the view code, we use the design pattern named **Composite**. In this context, the design would be as in the following structure.



To support the operation, when selection is indicated (perhaps with the push of a button or a mouse click with the Control key pressed), the controller creates a CompositeShape object and adds it to the model. As shapes are selected, the selected shape is removed from the Model and added to the CompositeShape object, using the add() method.

The pattern has the following advantages:

1) It leaves the view (the client of the Shape class) oblivious of the complexity of multiple selected Shape objects. The view code is unchanged.
2) Any combination of the Shape objects can be grouped into a single Shape.
3) If needed, any group created in (2) can be ungrouped using the remove() method of the CompositeShape.