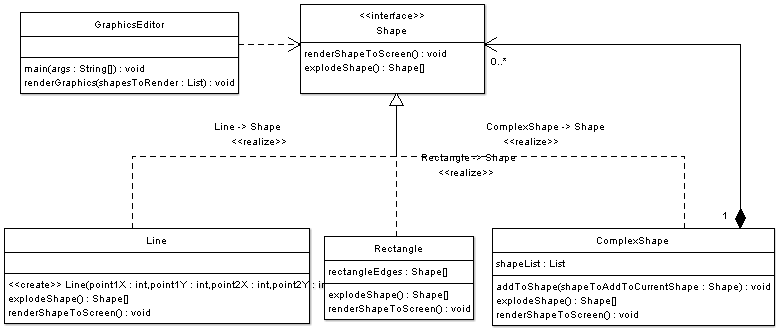
Composite Pattern Example

In graphics editors a shape can be basic or complex. An example of a simple shape is a line, where a complex shape is a rectangle which is made of four line objects. Since shapes have many operations in common such as rendering the shape to screen, and since shapes follow a part-whole hierarchy, composite pattern can be used to enable the program to deal with all shapes uniformly.

The code below illustrates the Shape Interface which defines the component interface:

|  |
| --- |
| package composite;  /\*\*  \*  \* Shape is the Component interface  \*  \*/  public interface Shape {    /\*\*  \* Draw shape on screen  \*  \* Method that must be implemented by Basic as well as  \* complex shapes  \*/  public void renderShapeToScreen();    /\*\*  \* Making a complex shape explode results in getting a list of the  \* shapes forming this shape  \*  \* For example if a rectangle explodes it results in 4 line objects  \*  \* Making a simple shape explode results in returning the shape itself  \*/  public Shape[] explodeShape();    } |

The code below illustrates a Simple shape object which is a line, note that this object represents a leaf.

|  |
| --- |
| package composite;  /\*\*  \*  \* Line is a basic shape that does not support adding shapes  \*/  public class Line implements Shape {    /\*\*  \* Create a line between point1 and point2  \* @param point1X  \* @param point1Y  \* @param point2X  \* @param point2Y  \*/  public Line(int point1X, int point1Y, int point2X, int point2Y) {    }    @Override  public Shape[] explodeShape() {  // making a simple shape explode would return only the shape itself, there are no parts of this shape    Shape[] shapeParts = {this};    return shapeParts;    }  /\*\*  \* this method must be implemented in this simple shape  \*/  public void renderShapeToScreen() {  // logic to render this shape to screen  }  }  } |

The code below illustrates a complex object, which is a rectangle object. This object represents a Composite.

|  |
| --- |
| package composite;  /\*\*  \* Rectangle is a composite  \*  \*Complex Shape  \*/  public class Rectangle implements Shape{  // List of shapes forming the rectangle  // rectangle is centered around origin  Shape[] rectangleEdges = {new Line(-1,-1,1,-1),new Line(-1,1,1,1),new Line(-1,-1,-1,1),new Line(1,-1,1,1)};      @Override  public Shape[] explodeShape() {    return rectangleEdges;    }  /\*\*  \* this method is implemented directly in basic shapes  \* in complex shapes this method is implemented using delegation  \*/  public void renderShapeToScreen() {  for(Shape s : rectangleEdges){    // delegate to child objects  s.renderShapeToScreen();    }    }  } |

The code below illustrates a more complex shape object which also represents a composite

|  |
| --- |
| package composite;  import java.util.ArrayList;  import java.util.List;  /\*\*  \* Composite object supporting creation of more complex shapes  \* Complex Shape  \*/  public class ComplexShape implements Shape {    /\*\*  \* List of shapes  \*/  List shapeList = new ArrayList();  /\*\*  \*  \*/  public void addToShape(Shape shapeToAddToCurrentShape) {  shapeList.add(shapeToAddToCurrentShape);    }    public Shape[] explodeShape() {  return (Shape[]) shapeList.toArray();  }  /\*\*  \* this method is implemented directly in basic shapes  \* in complex shapes this method is handled with delegation  \*/  public void renderShapeToScreen() {  for(Shape s: shapeList){    // use delegation to handle this method  s.renderShapeToScreen();    }  }  } |

The code below illustrates the Graphics Editor Driver class which represents the client. Note how the client of the composite pattern deals with all Shape objects uniformly despite the fact that some of the shapes are leafs while others are branches.

|  |
| --- |
| package composite;  import java.util.ArrayList;  import java.util.List;  /\*\*  \* Driver Class  \*  \*/  public class GraphicsEditor {  public static void main(String[] args) {    List allShapesInSoftware = new ArrayList();    // create a line shape  Shape lineShape = new Line(0,0,1,1);    // add it to the shapes  allShapesInSoftware.add(lineShape);    // create a rectangle shape  Shape rectangelShape = new Rectangle();    // add it to shapes    allShapesInSoftware.add(rectangelShape);    // create a complex shape  // note that we have dealt with the complex shape  // not with shape interface because we want  // to do a specific operation  // that does not apply to all shapes  ComplexShape complexShape = new ComplexShape();    // complex shape is made of the rectangle and the line    complexShape.addToShape(rectangelShape);    complexShape.addToShape(lineShape);    // add to shapes    allShapesInSoftware.add(complexShape);      // create a more complex shape which is made of the  // previously created complex shape  // as well as a line    ComplexShape veryComplexShape = new ComplexShape();    veryComplexShape.addToShape(complexShape);    veryComplexShape.addToShape(lineShape);      allShapesInSoftware.add(veryComplexShape);    renderGraphics(allShapesInSoftware);        // you can explode any object  // despite the fact that the shape might be  // simple or complex    Shape[] arrayOfShapes = allShapesInSoftware.get(0).explodeShape();        }    private static void renderGraphics(List shapesToRender){  // note that despite the fact there are  // simple and complex shapes  // this method deals with them uniformly  // and using the Shape interface    for(Shape s : shapesToRender){  s.renderShapeToScreen();  }    }  } |