**Description of the design:**

I have used the **Composite pattern** to implement my solution to this problem. The Shape class implements an interface such as renderShapeToScreen()[to render shape to screen] and explodeShape() that returns the basic components (lines) that make up the shape. The line class and all subsequent classes define and override these methods and implement their own individual algorithms to draw the shape on the screen. The composite shapes i.e. Triangle, Arrow and Rectangle use the leaf’s i.e. Line’s renderShapeToScreen() method to render their own individual shapes to screen. The composite class (Rectangle, Triangle, Arrow) also have an add method to add individual Shapes. For instance the Arrow class takes the address of a Line object and a Triangle object to render itself to the screen. Thus, for the Arrow class’s add method only two arguments are needed. For other class’s add methods the arguments are equal to the number of lines that make up the object. The method explodeShape() returns the basic components i.e. lines to the caller and the caller can then use this to display the individual components. Thus, in this manner the Composite classes use the common interface that they inherit from to render their shapes on the screen using the leaf’s method indirectly via the interface.

**Application of the design pattern to this problem and justification:**

The Composite pattern is used as this pattern composes objects to represent whole-part hierarchies. Composite pattern lets clients treat individual objects and compositions of objects uniformly. Thus, I have defined the class Line as my leaf node and the Triangle, Rectangle and Arrow classes represent objects composed of this single leaf i.e. class Line. A common interface in the form of Shape lets the composite objects (Rectangle, Triangle, Arrow) use the functions of Line which thus allows the composite objects to be treated in the same way as the primitive object which in this case is the Line. Thus, when I call renderShapeToScreen() of a particular composite class object, it invokes the corresponding function of the Line class and this renders a single line on the screen. I can thus render the entire shape by calling renderShapetoScreen of the Line class and applying an algorithm. Hence, in this way it becomes a question of decomposing the problem in terms of parts that make up the whole and hence the composite pattern is the design pattern of my choice.

**Class Diagram:**

Class Line:

void renderShapeToScreen()

Shape \*\* explodeShape()

Class Triangle:

void renderShapeToScreen()

Shape \*\* explodeShape()

void addLines(Shape \*, Shape \*, Shape \*);

Class Rectangle:

void renderShapeToScreen()

Shape \*\* explodeShape()

void addLines(Shape \*, Shape \*, Shape \*, Shape \*);

Class Arrow:

void renderShapeToScreen()

Shape \*\* explodeShape()

void addLines(Shape \*, \*Shape);

<interface>

Class Shape:

virtual void renderShapeToScreen()

virtual Shape \*\* explodeShape()