M6 (a) - Composition

Jin L.C. Guo

Image source: https://cdn.pixabay.com/photo/2017/11/05/21/21/container-2921882_960_720.jpg

Objective

- Design Principle:

Divide and Conquer

- Programming mechanism:

Aggregation and Delegation

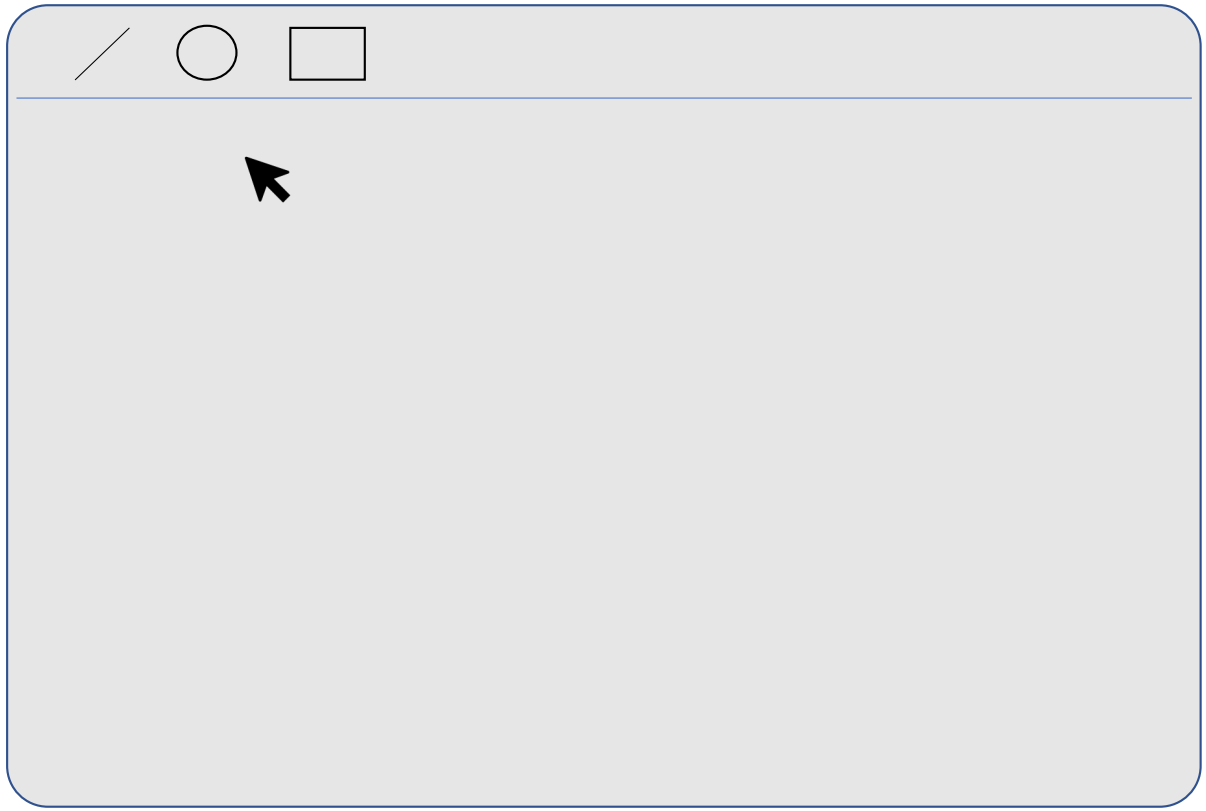
- Design Techniques:

Sequence Diagram

- Patterns and Anti-patterns:

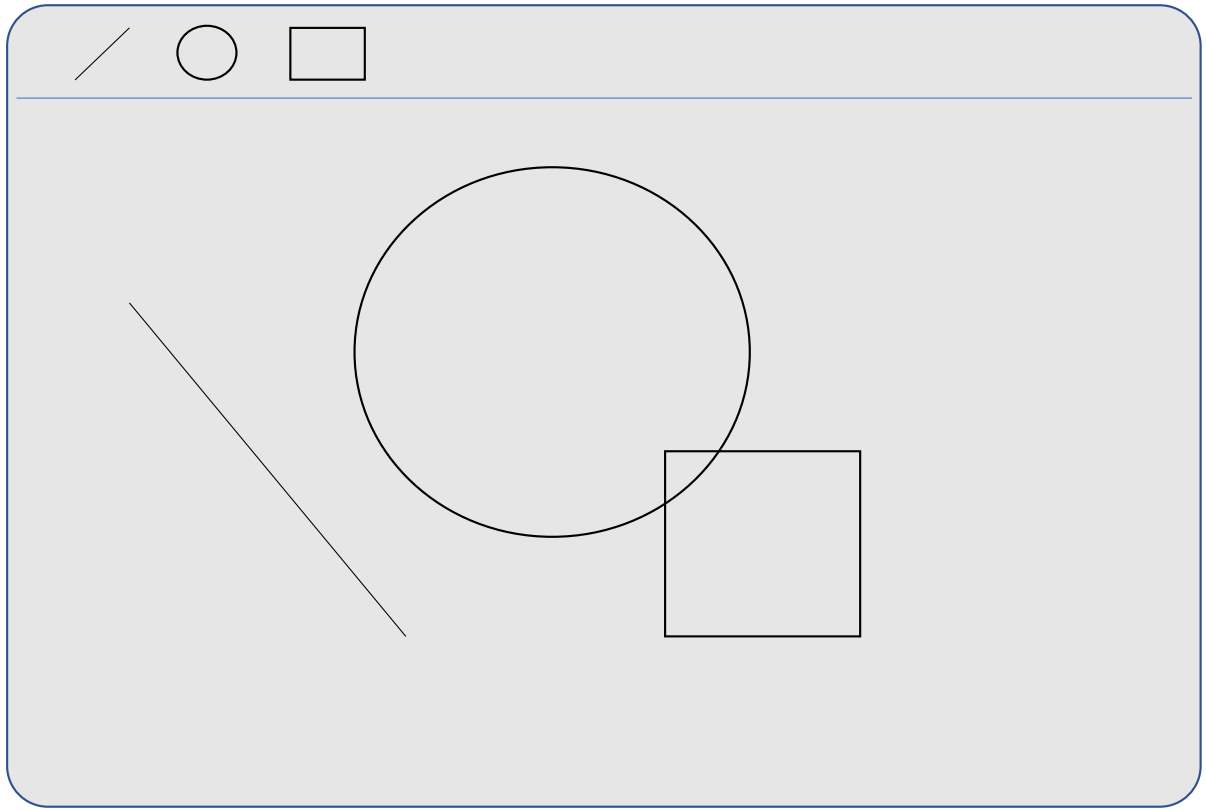
Composite Pattern, Decorator Pattern, God class 

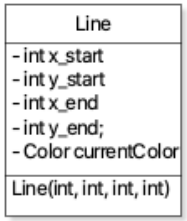
Design Problem – Drawing Editor

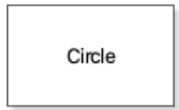


Design Problem

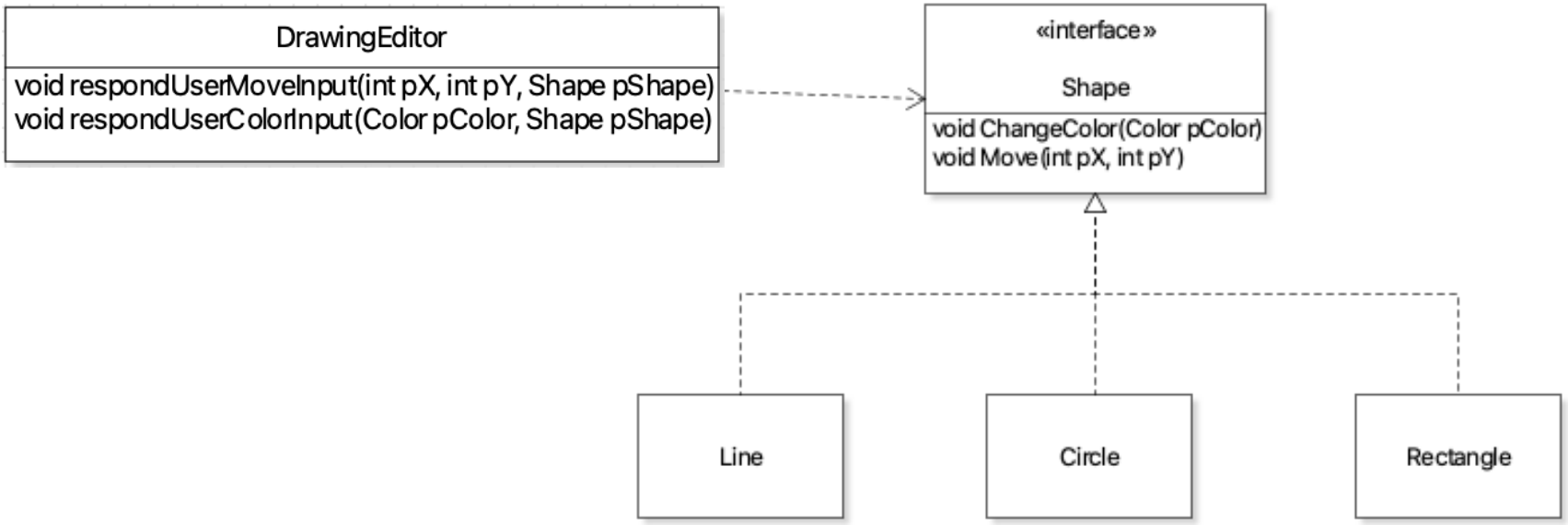
The users need to be able to change the color of the lines, circles and rectangles, as well as their position.



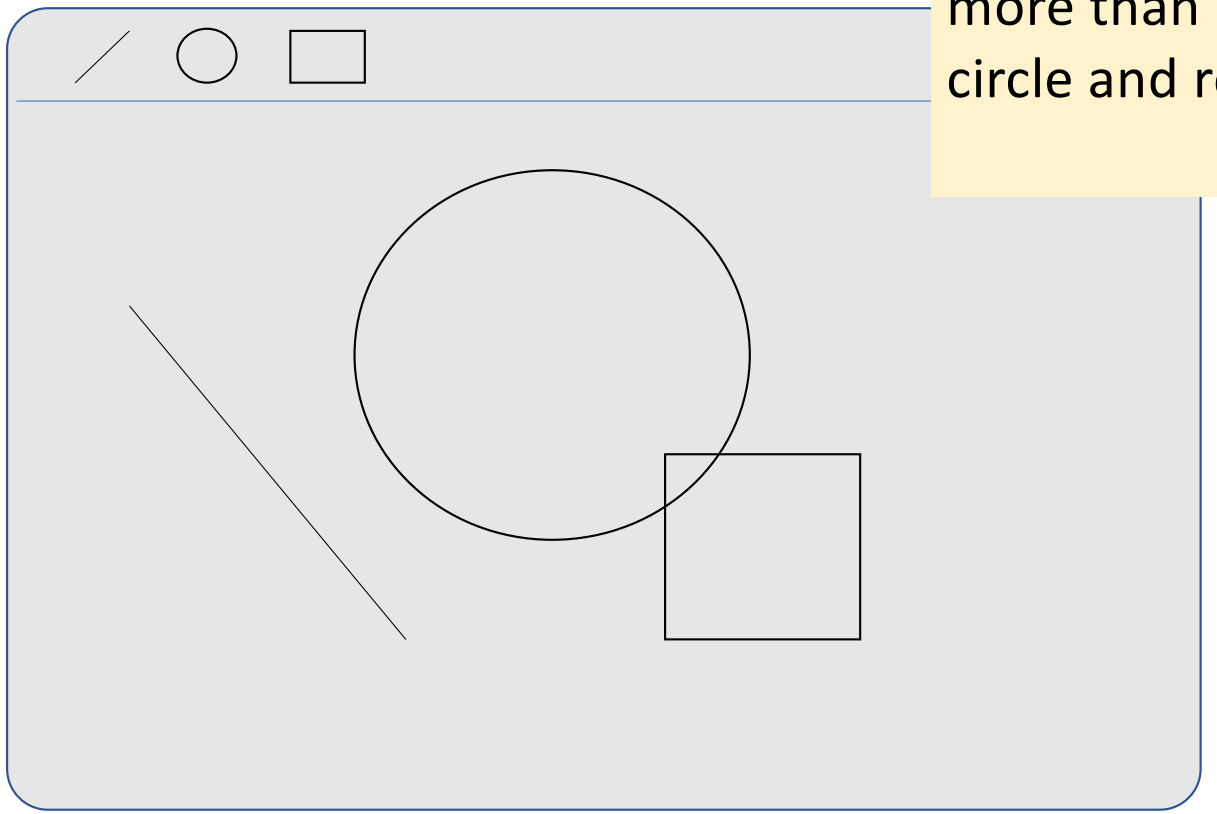




```
void respondUserMoveInput(int x, int y, Shape pShape) {
    pShape.move(x, y);
}
```

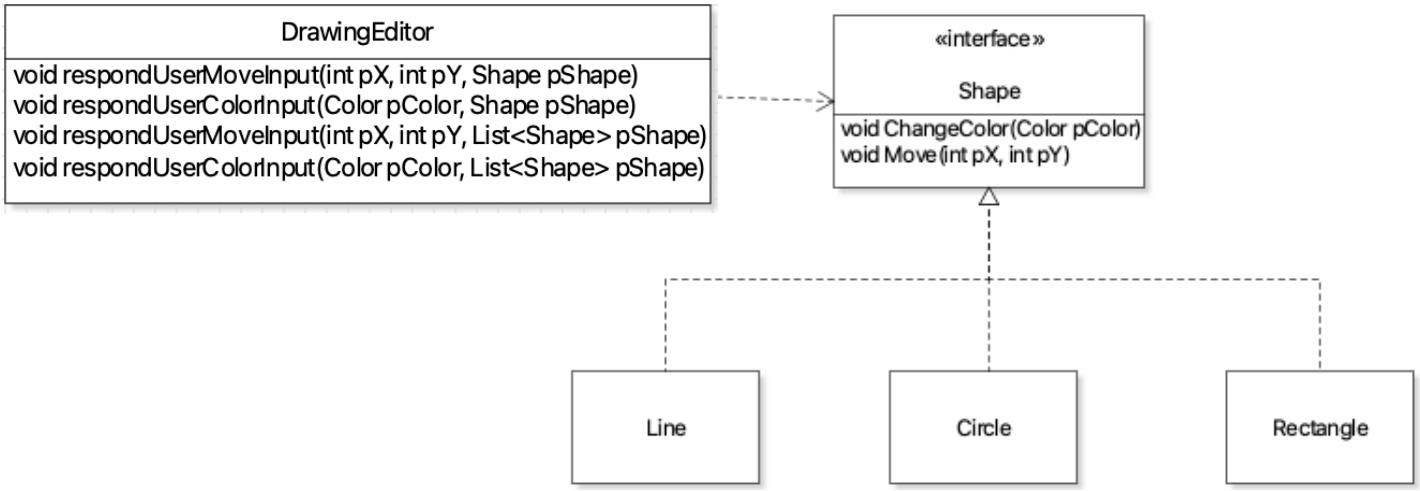


Activity 1: Design Problem



Add the function of grouping shapes so that the users can move or change the color for more than one element, e.g., circle and rectangle.

Solution 1:




```
void respondUserMoveInput(int x, int y, List<Shape> pShapes) {
    for(Shape aShape: pShapes) {
        aShape.move(x, y);
    }
}
```

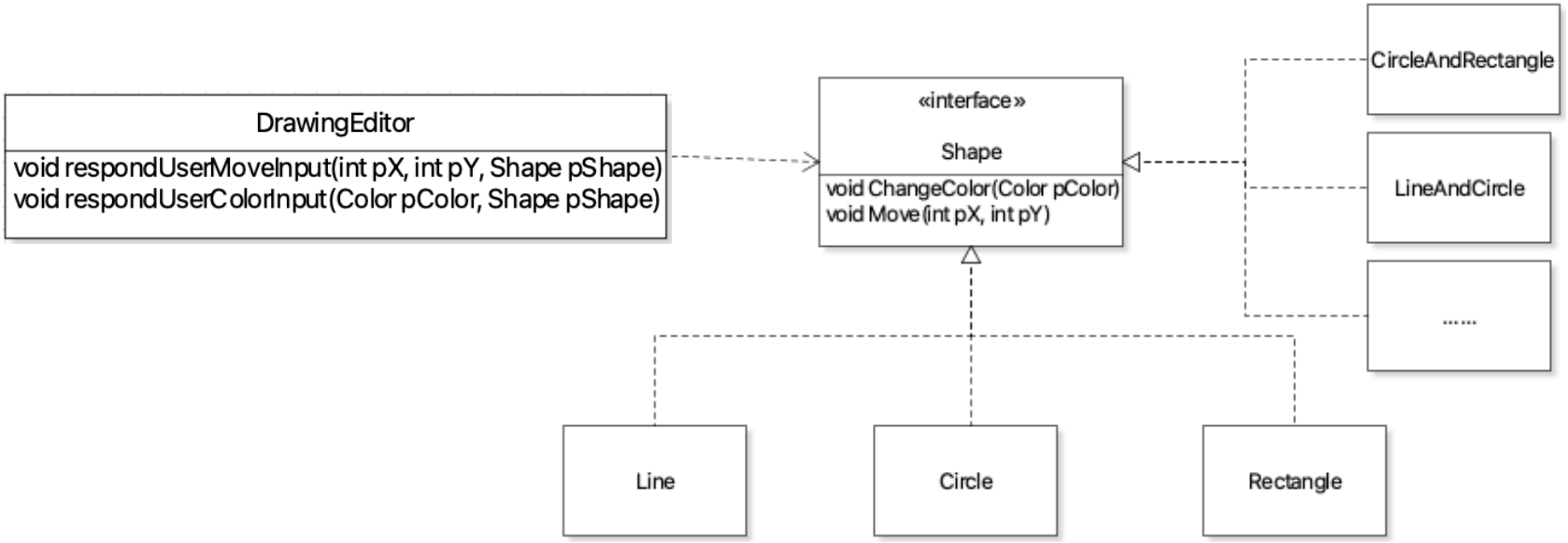
Client code has to treat a single shape and a group of shapes differently.

Solution 2:

Large number of possible structures.

Each option requires a class definition, even rare cases.

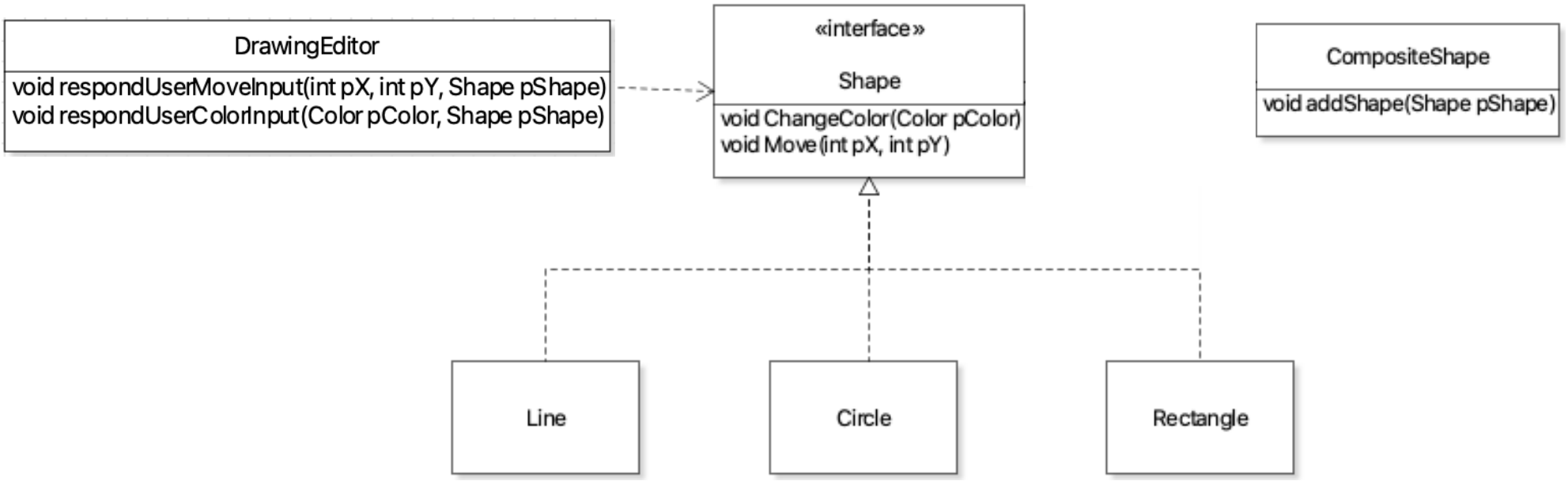
Impossible to accommodate all situations.



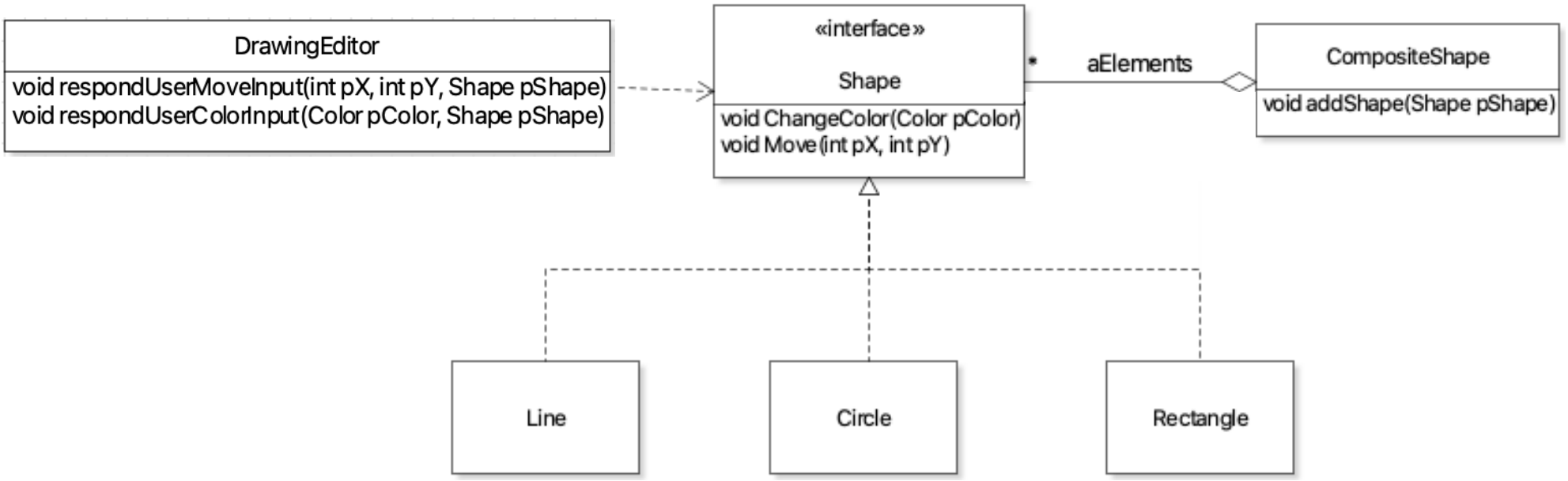
Another Solution

Image Source: https://upload.wikimedia.org/wikipedia/commons/6/61/Lego_blocks.jpg

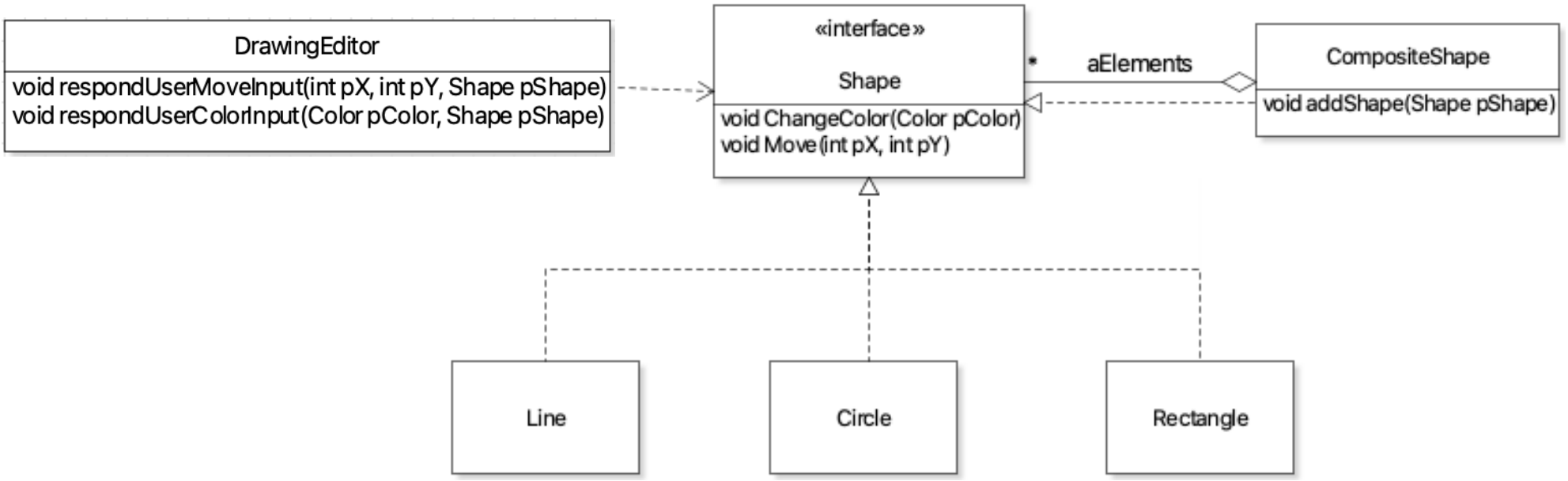
Another Solution:

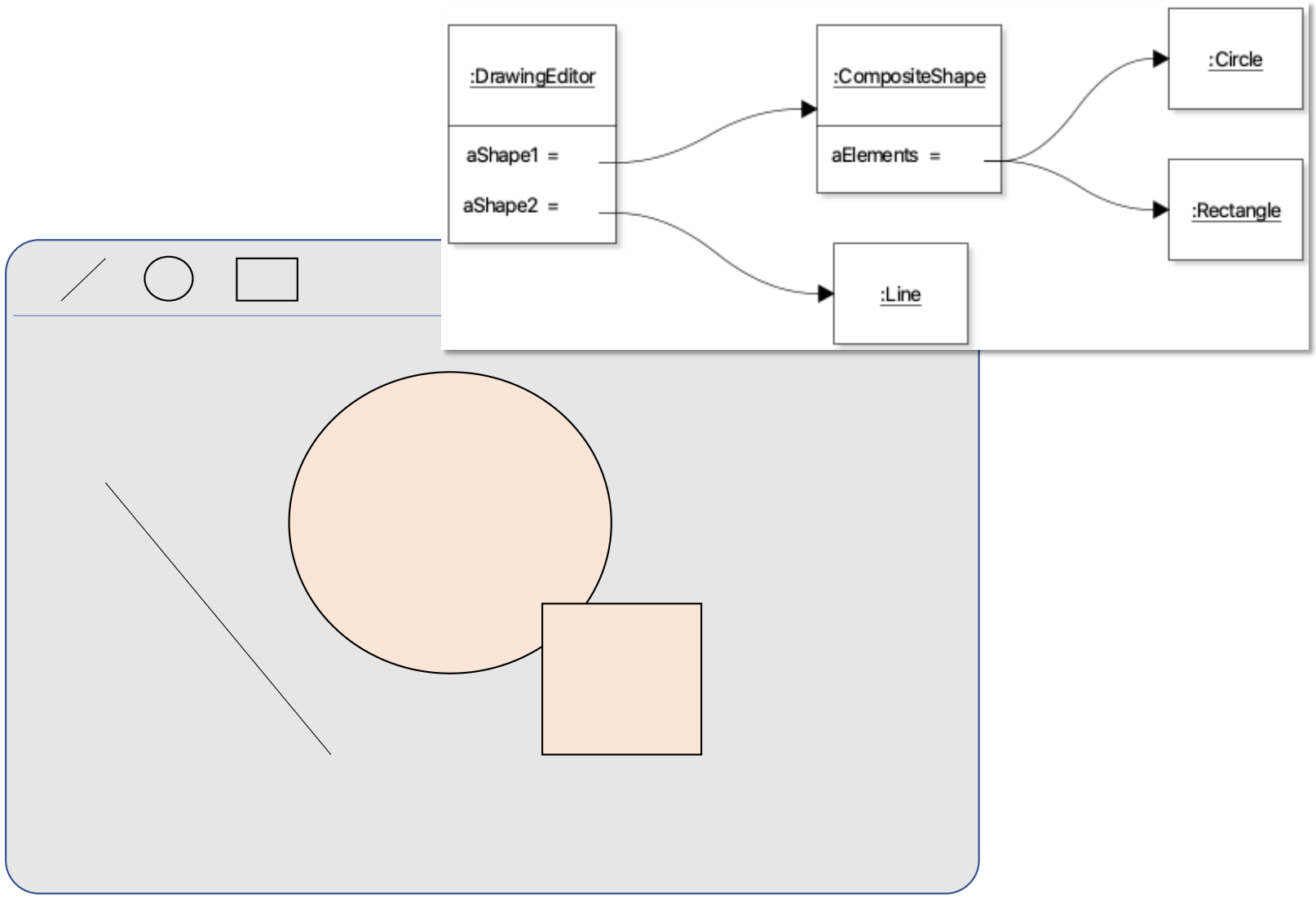


Another Solution:



Another Solution:




:DrawingEditor
aShape1 =
aShape2 =
:CompositeShape
aElements =
:Circle
:Rectangle
:Line

Objective

- Design Principle:

Divide and Conquer

- Programming mechanism:

Aggregation and Delegation

- Design Techniques:

Sequence Diagram

- Patterns and Anti-patterns:

Composite Pattern, Decorator Pattern, God class 

Composite Pattern

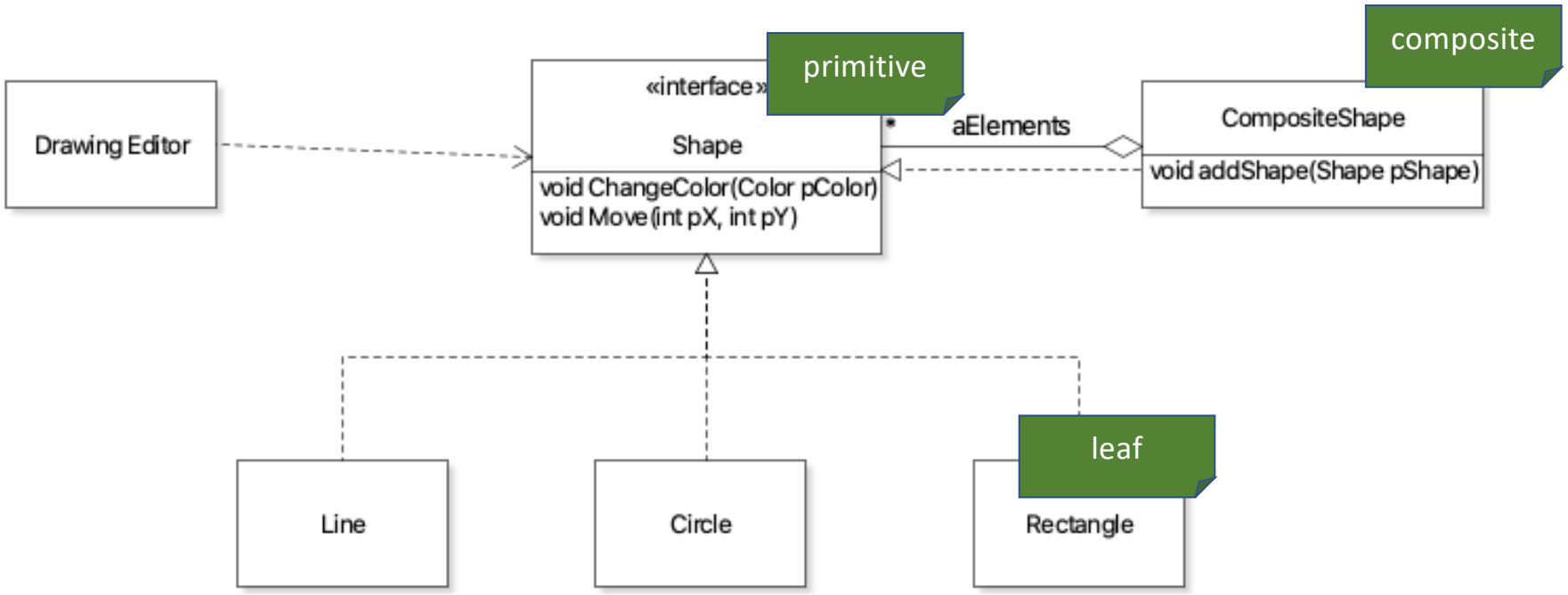
- Intent
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly
- Participants:
  - Primitive (Component)

    Declares the interface for objects in the composition.
  - Leaf

    Defines behaviour for primitives
  - Composite

    Defines behaviour for primitives to have children

Composite Pattern



Implement Composite Pattern

```
public class CompositeShape implements Shape
{
    private List<Shape> aElements = new ArrayList<>();


    @Override
    public void changeColor(Color pColor){/* … */}


    @Override
    public void move(int pX, int pY) {/* … */}


}
```

Activity2: how to add Primitive instances to Composite?

```
public class CompositeShape implements Shape
{
    … …


    public void add(Shape pShape)
    {
        aElements.add(pShape);
    }


}
```

What are other options?
What are their tradeoffs?

Other considerations

- Is the order of accessing the children important?
- Should the references to the parent be maintained from the children?
- Should a child be allowed to be added to more than one component?

Object Collaboration

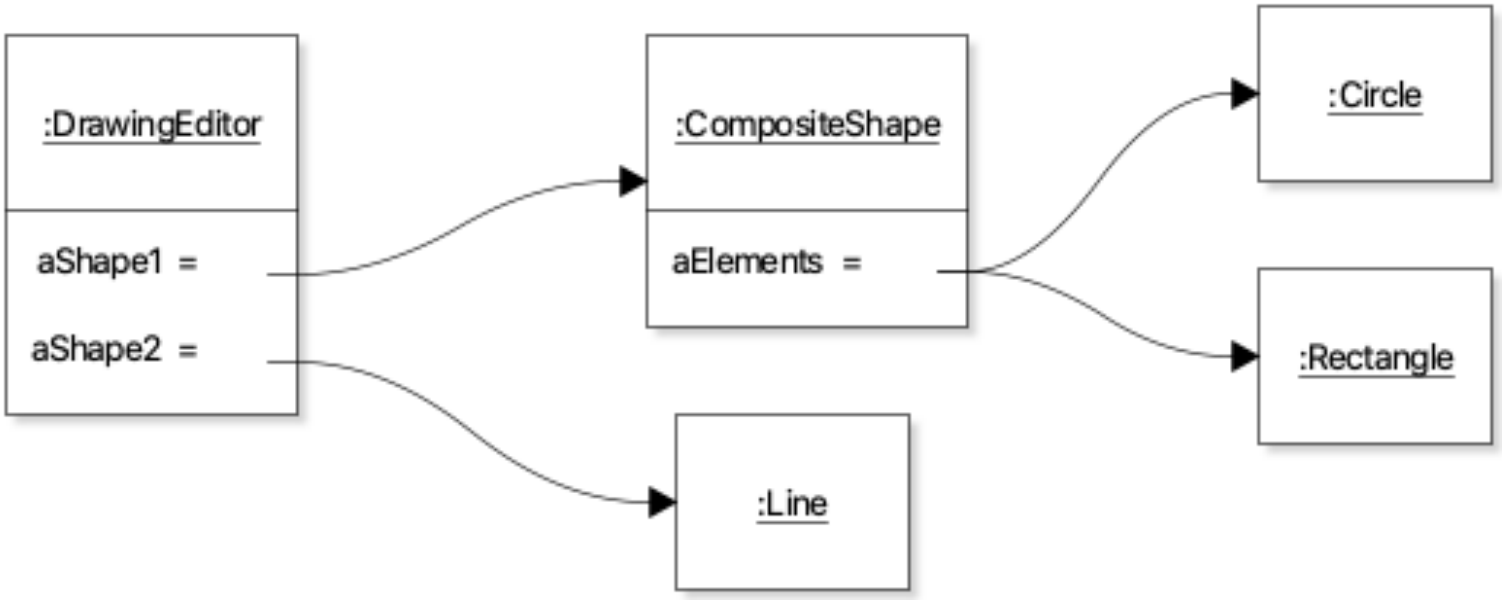
```
public class CompositeShape implements Shape
{
    private List<Shape> aElements = new ArrayList<>();


    @Override
    public void changeColor(Color pColor){
         for(Shape shape : aElements)
         {
             shape.changeColor(pColor);
         }
    }

}
```

The use of composition implies that we are designing how objects collaborate with each other, through methods calls.

Modeling object call sequences?



Objective

- Design Principle:

Divide and Conquer

- Programming mechanism:

Aggregation and Delegation

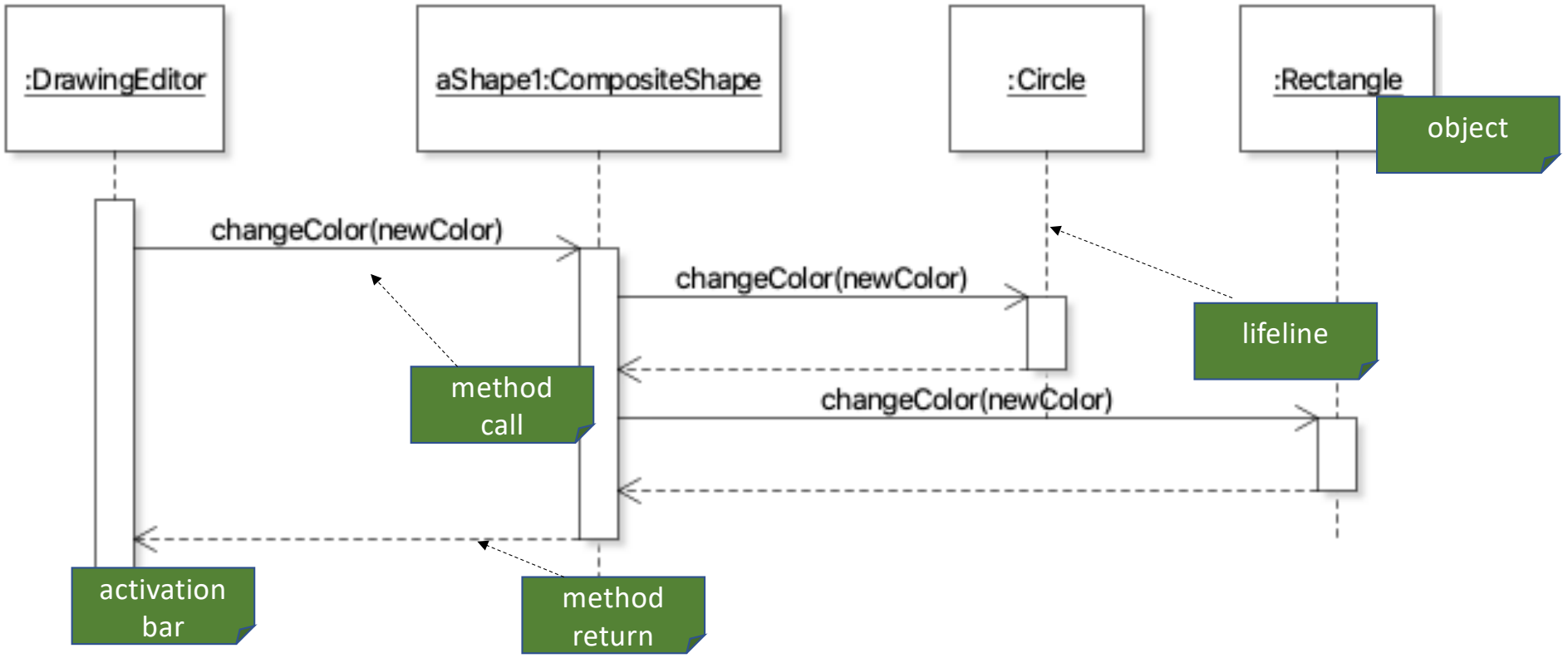
- Design Techniques:

Sequence Diagram

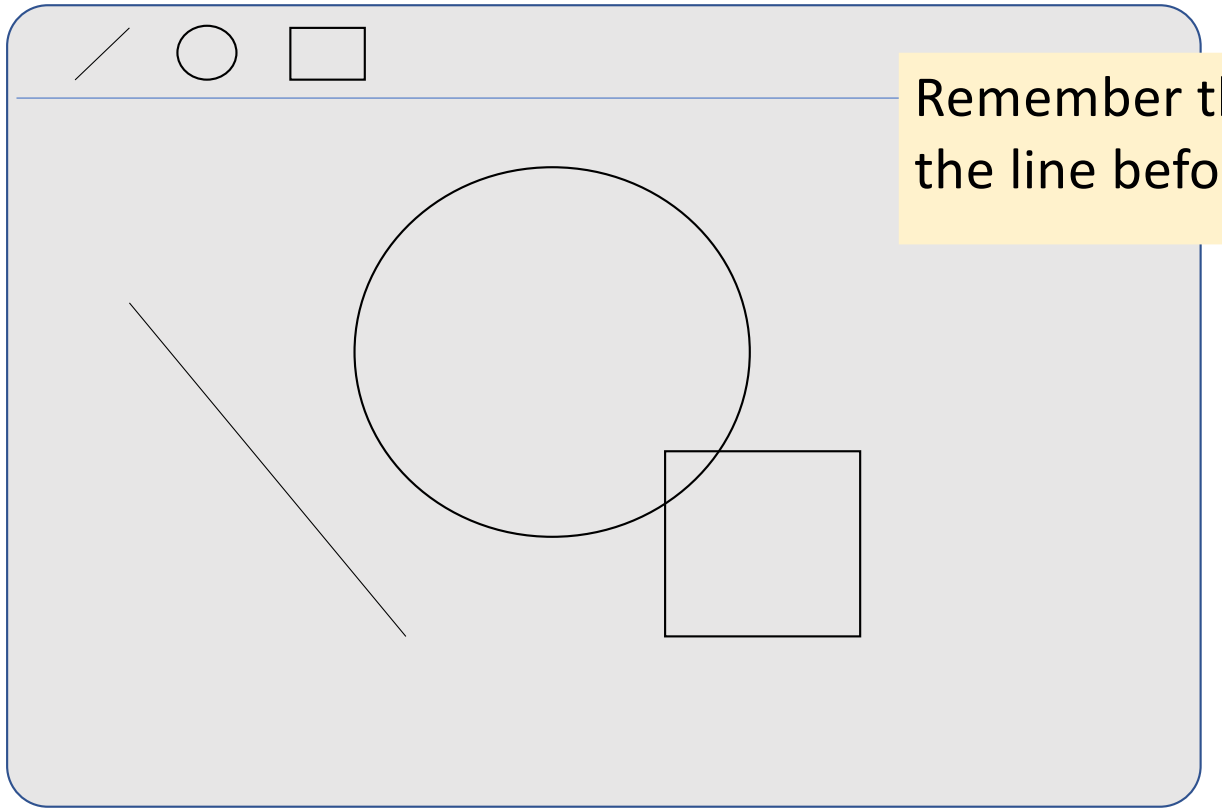
- Patterns and Anti-patterns:

Composite Pattern, Decorator Pattern, God class 

Sequence Diagram



Activity 3: Attach additional responsibility dynamically to `changeColor` of `Line`?



Remember the previous color of the line before changing.

```
public class MemoryLine implements Shape
{
   private int x_start;
   private int y_start;
   private int x_end;
   private int y_end;
   private Color aColor;
   private Color aPreviousColor;


   @Override
   public void changeColor(Color pColor)
   {
      aPreviousColor = aColor;
      aColor = pColor;
   }
}
```

Specialized Class, hard to extend

Cannot turn responsibility on and off at runtime

Other design options?

Separate the essential and additional state and behavior

```
public class MemoryLine implements Shape
{
   private int x_start;
   private int y_start;
   private int x_end;
   private int y_end;
   private Color aColor;
   private Color aPreviousColor;




}
```

Skin Versus Gut

Separate the essential and additional state and behavior

```
public class MemoryLine implements Shape
{


   private Line aLine;


   private Color aPreviousColor;
   @Override
   public void changeColor(Color pColor)
   {
      aPreviousColor = aLine.getColor();
      aLine.chageColor(pColor);
   }

}
```

1. Delegate the original request
2. Implement additional feature

Objective

- Design Principle:

Divide and Conquer

- Programming mechanism:

Aggregation and Delegation

- Design Techniques:

Sequence Diagram

- Patterns and Anti-patterns:

Composite Pattern, Decorator Pattern, God class

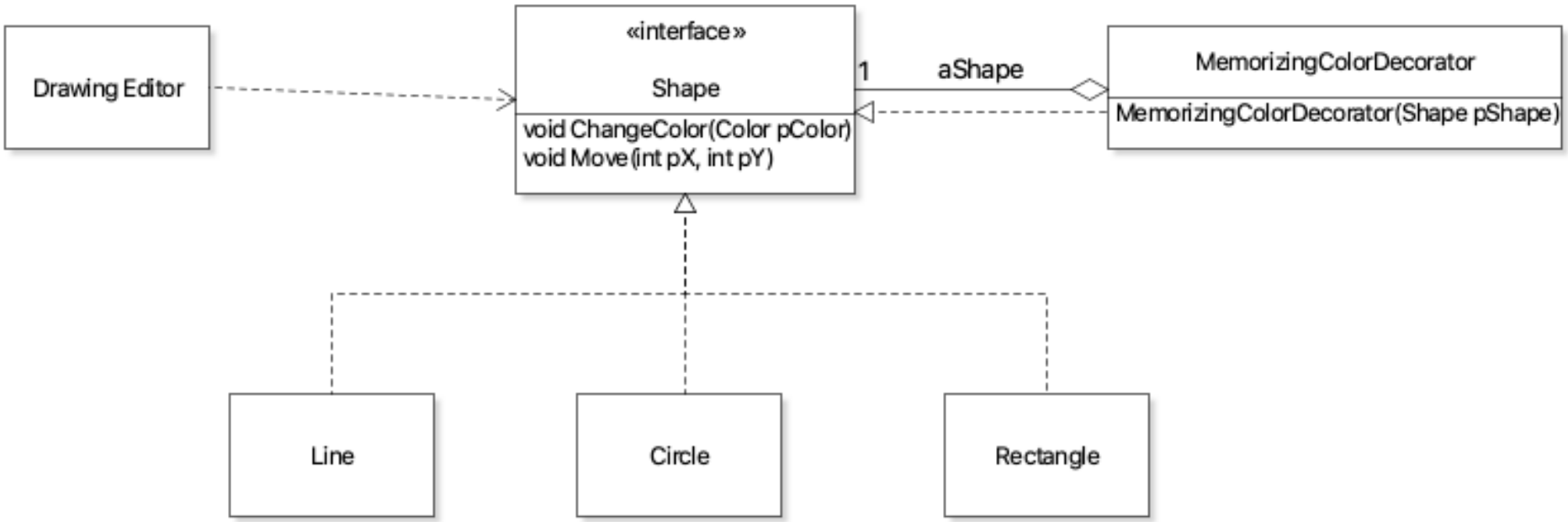
Decorator Pattern

- Intent:
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Participants:
  - Primitive

    Declares the interface for objects that can have responsibilities added to them dynamically
  - Leaf

    Defines the class to which additional responsibilities can be attached.
  - Decorator

    Maintains a reference to the primitive and defines the interface that confirms the primitive's interface

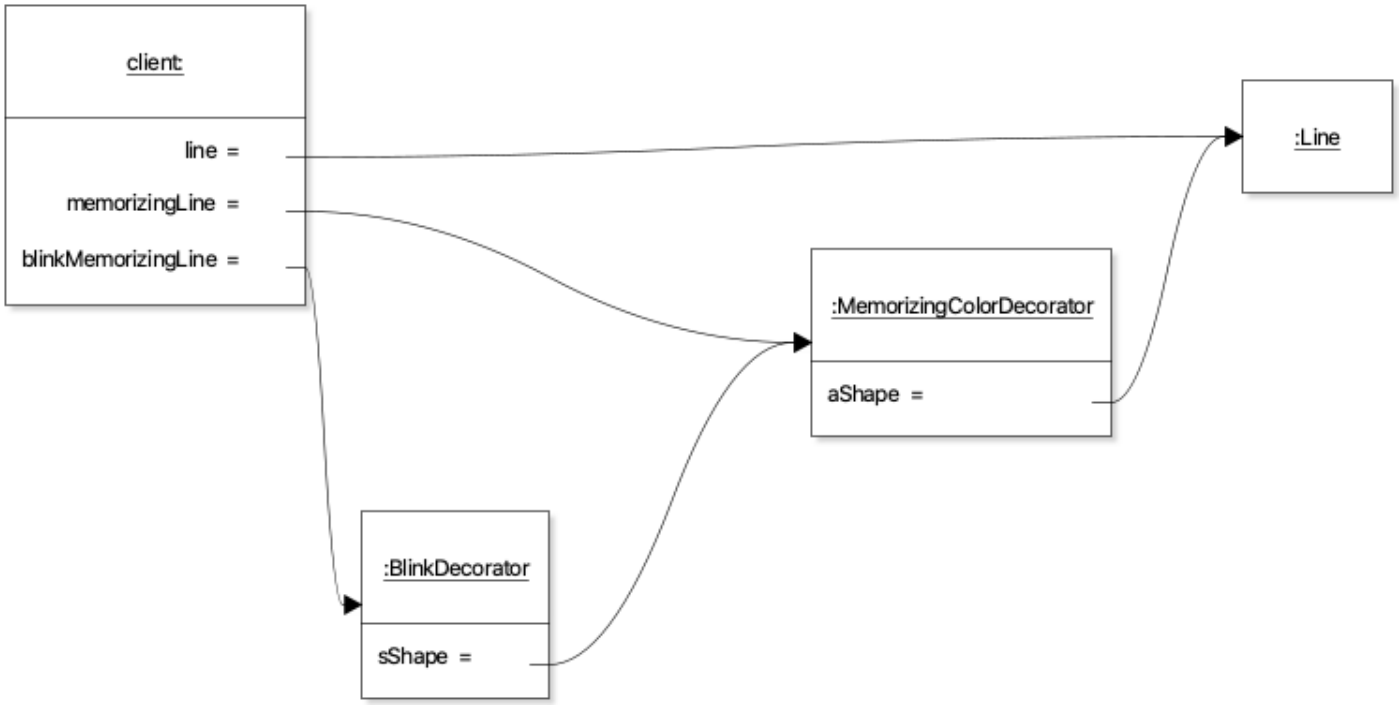

Drawing Editor
«interface»
Shape
void ChangeColor(Color pColor)
void Move(int pX, int pY)
1
aShape
MemorizingColorDecorator
MemorizingColorDecorator(Shape pShape)
Line
Circle
Rectangle

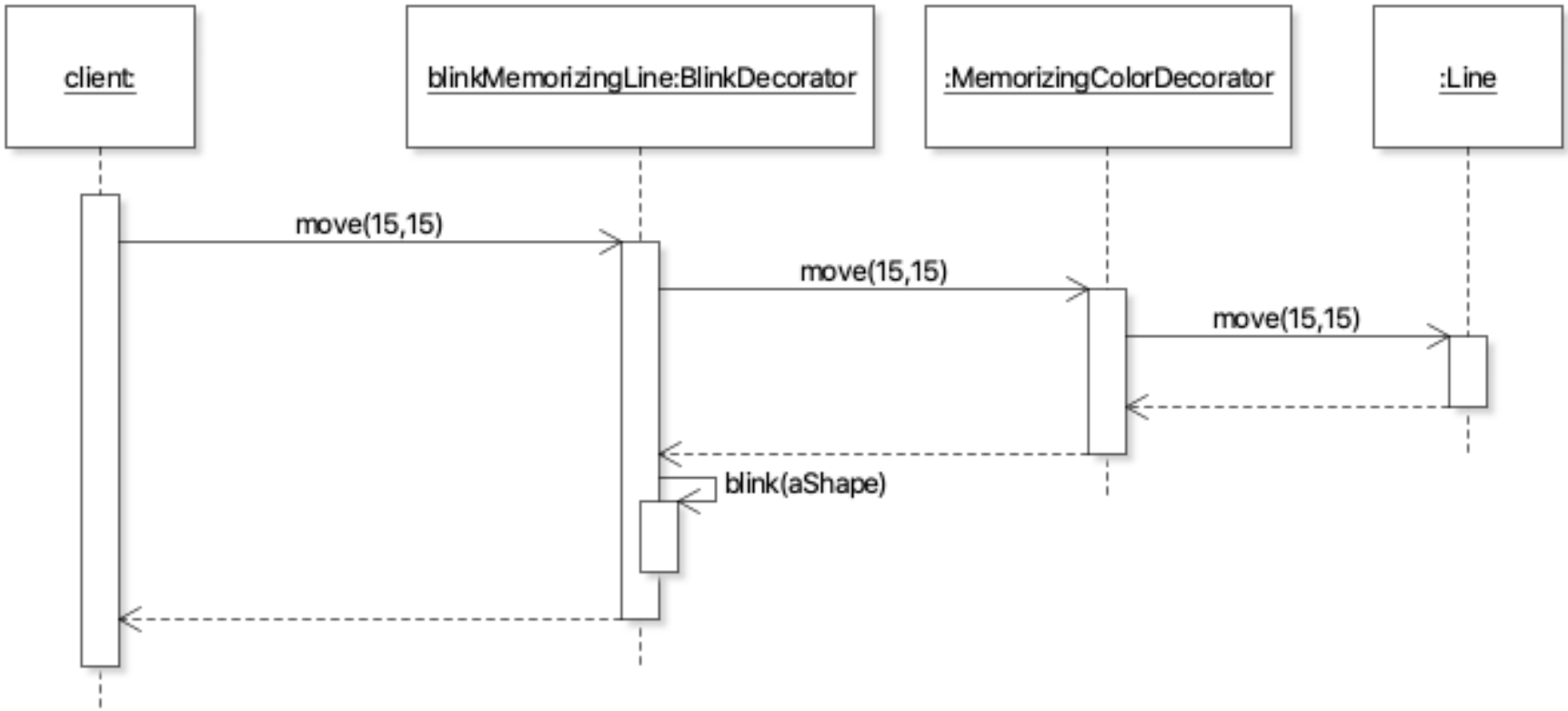
```
public class BlinkDecorator implements Shape
{
   ……
   @Override
   public void move(int pX, int pY)
   {
      aShape.move(pX, pY);
      this.bink(aShape);
   }
}
```

Activity 4

- Draw the object and sequence diagram when executing the last line of the following client code.

```
Shape line = new Line(3,3,10,10);//start x, start y, end x, end y
Shape memorizingLine = new MemorizingColorDecorator(line);
Shape blinkMemorizingLine = new BlinkDecorator(memorizingLine);
blinkMemorizingLine.move(15, 15);
```


client:
line =
memorizingLine =
blinkMemorizingLine =
:Line
:MemorizingColorDecorator
aShape =
:BlinkDecorator
sShape =


client:
blinkMemorizingLine:BlinkDecorator
:MemorizingColorDecorator
:Line
move(15,15)
move(15,15)
move(15,15)
blink(aShape)

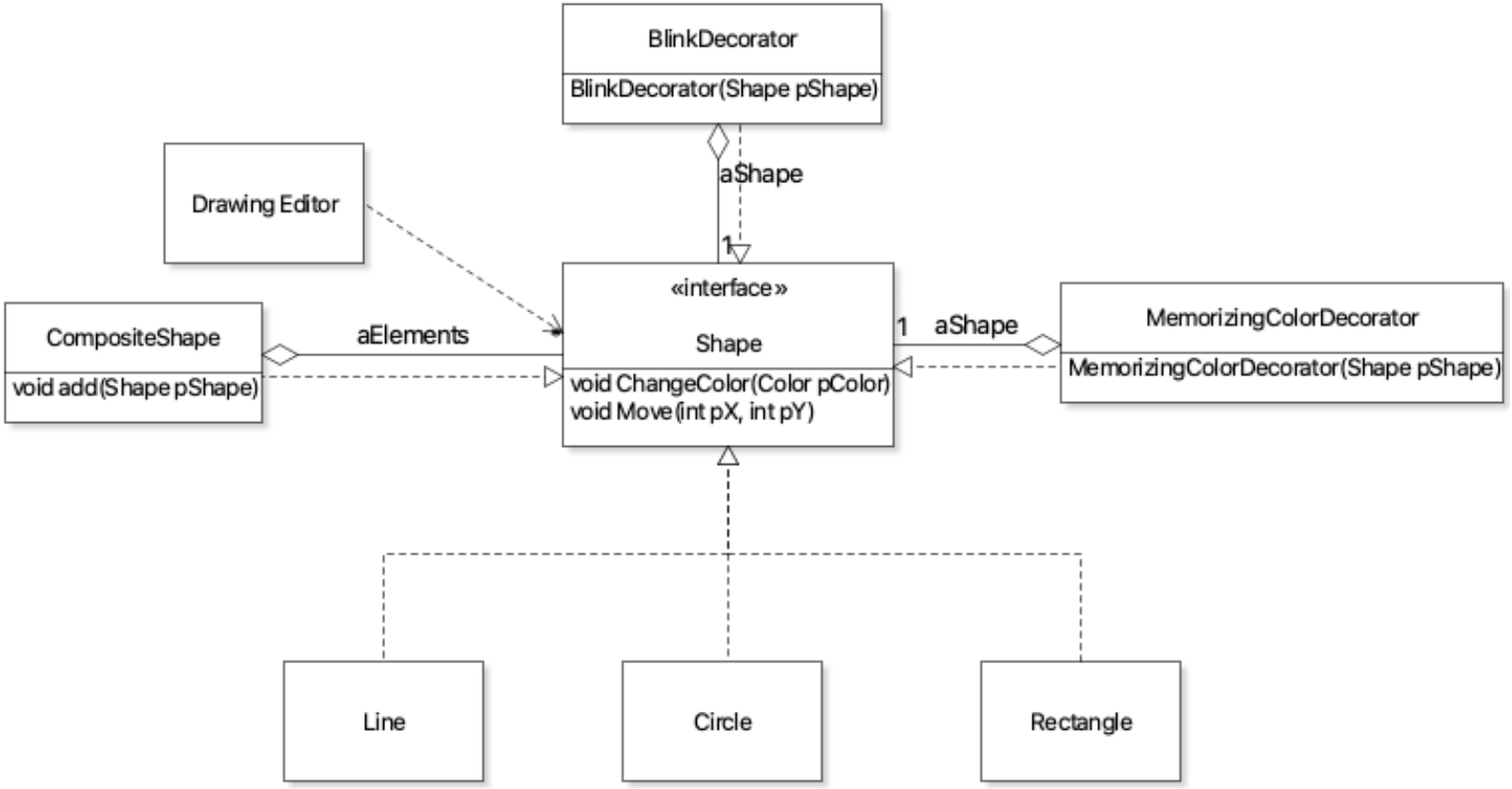
Identity of decorator and decorated object

```
Shape line = new Line(3,3,10,10);//start x, start y, end x, end y
Shape memorizingLine = new MemorizingColorDecorator(line);
Shape blinkMemorizingLine = new BlinkDecorator(memorizingLine);
blinkMemorizingLine.move(15, 15);
```

```
System.out.println(line == blinkMemorizingLine); // true or false?
```

Decorated object identity lost

Combining Decorator and Composite?



Combining Decorator and Composite

- Allow decorating behaviors of the composite, e.g., blinking the group of shapes.

Objective

- Design Principle:

Divide and Conquer

- Programming mechanism:

Aggregation and Delegation

- Design Techniques:

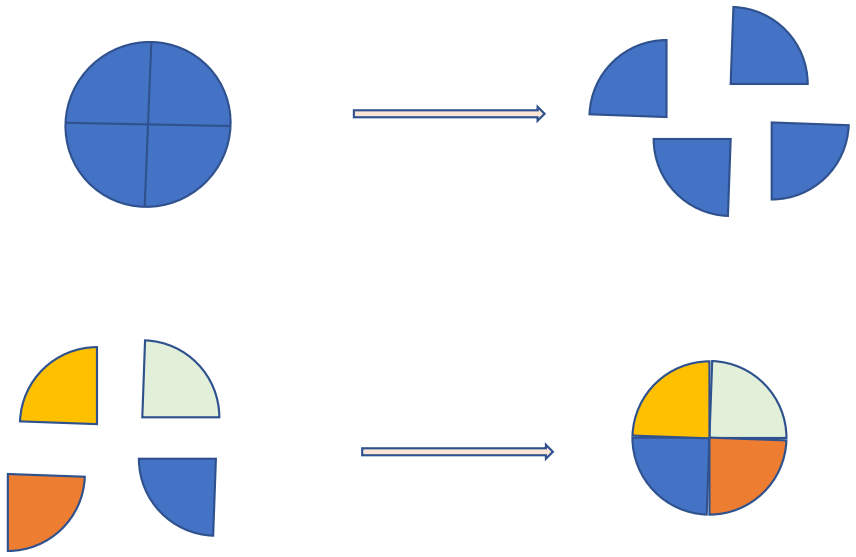
Sequence Diagram

- Patterns and Anti-patterns:

Composite Pattern, Decorator Pattern, God class

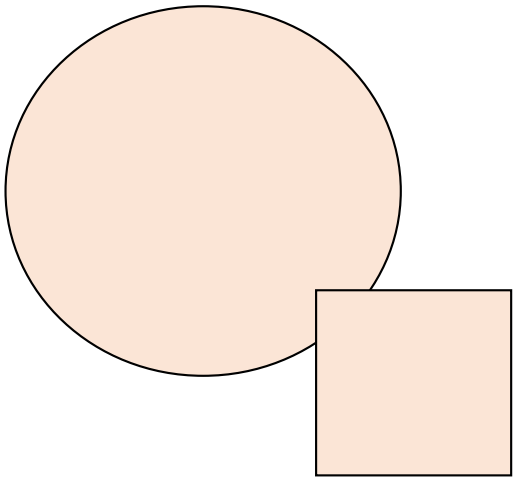
Manage Complexity -- Divide and conquer

- Modularization
  - Decomposable
  - Composable



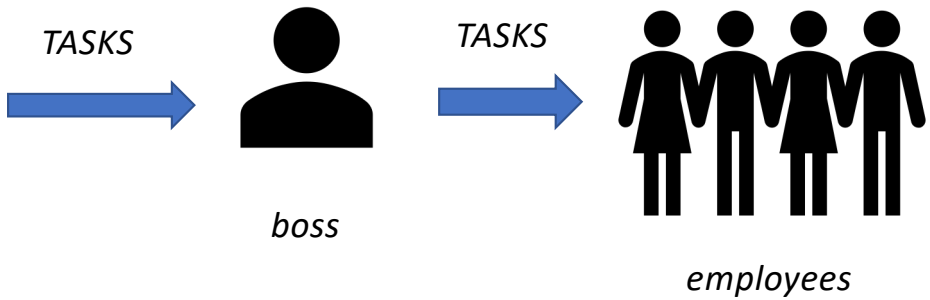
Purpose 1

- Aggregation: Representation of collections



Purpose 2

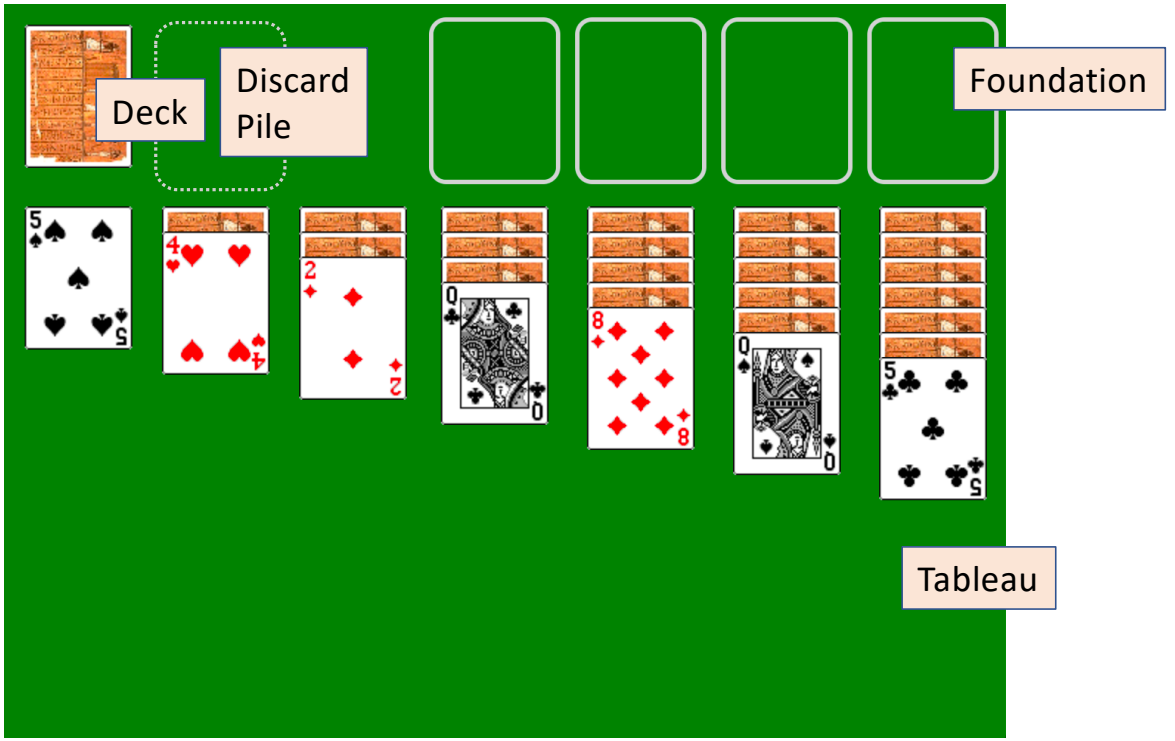
- Delegation: Redirect duties



GameModel in Solitaire

13 piles of cards?

God Class



The elements are both the component, and also entities providing services.

Class Diagram