M6 (a) - Composition

Jin L.C. Guo

Objective

- Design Principle:
Divide and Conquer

- Programming mechanism:
Aggregation and Delegation

- Design Techniques:
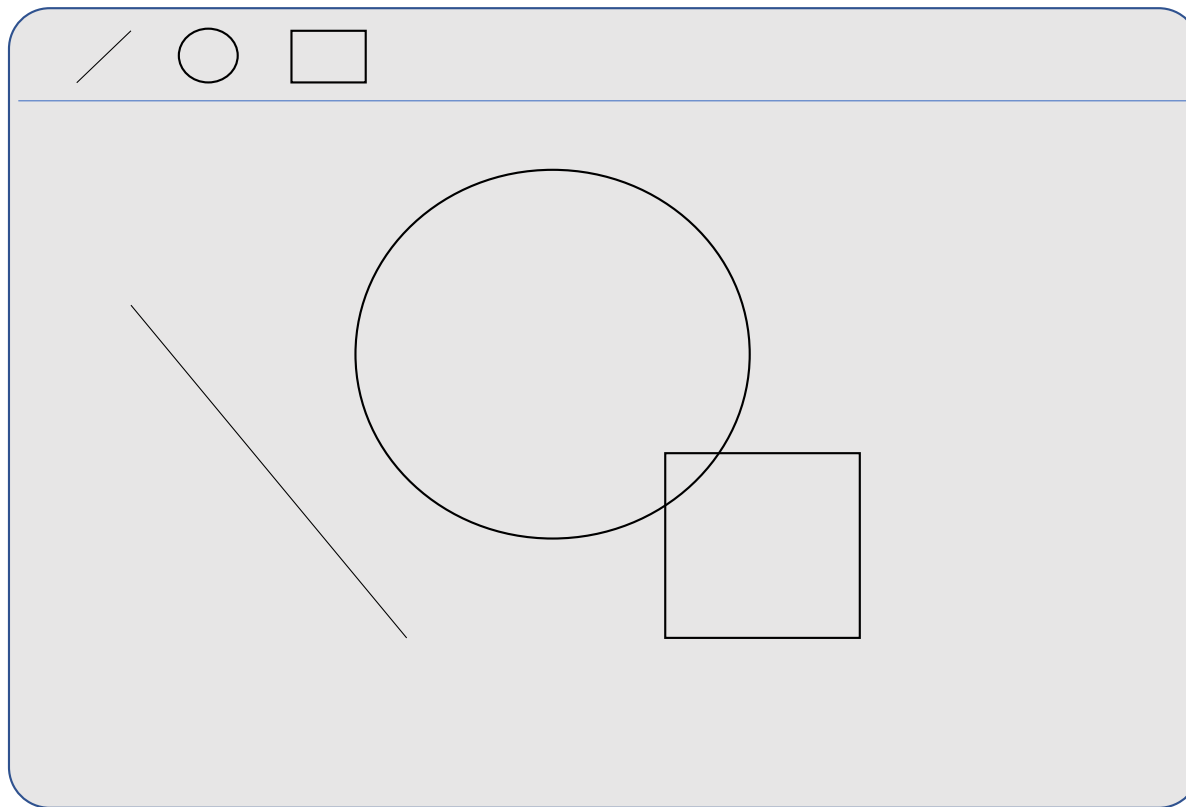Sequence Diagram

- Patterns and Anti-patterns:
Composite Pattern, Decorator Pattern, God class

Design Problem – Drawing Editor
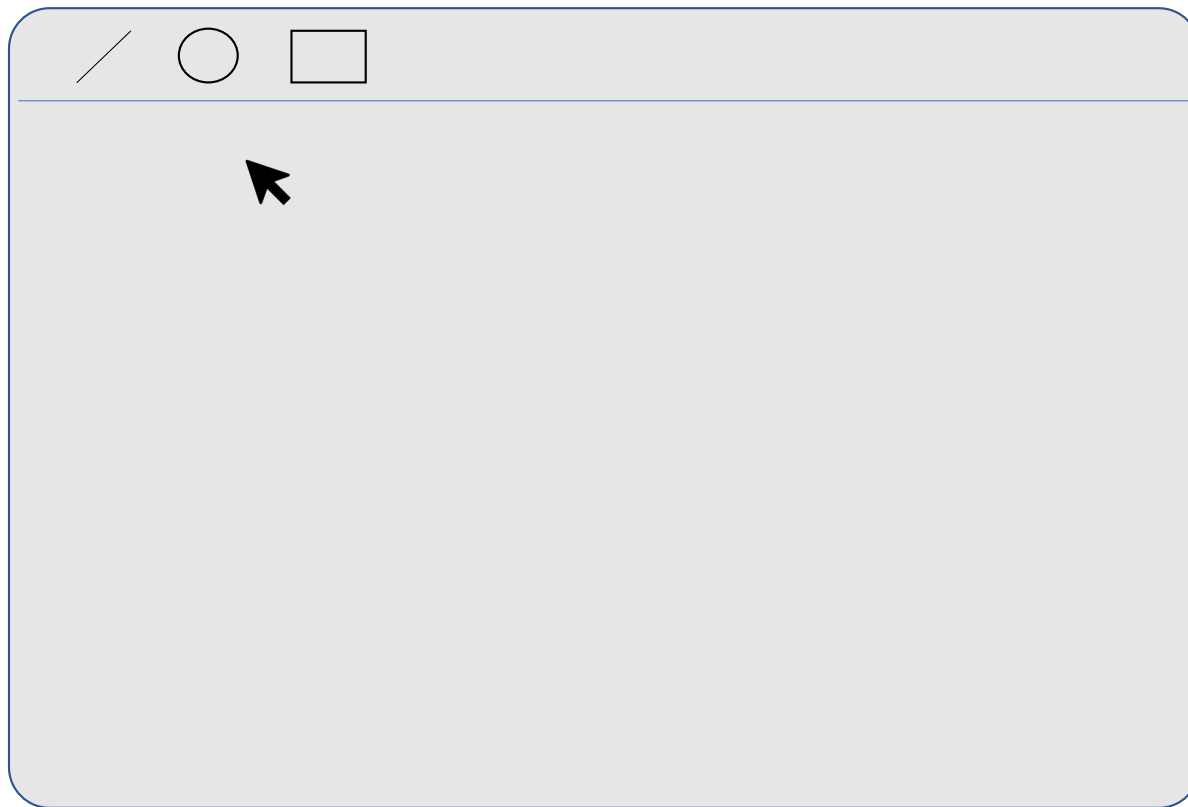
Design Problem

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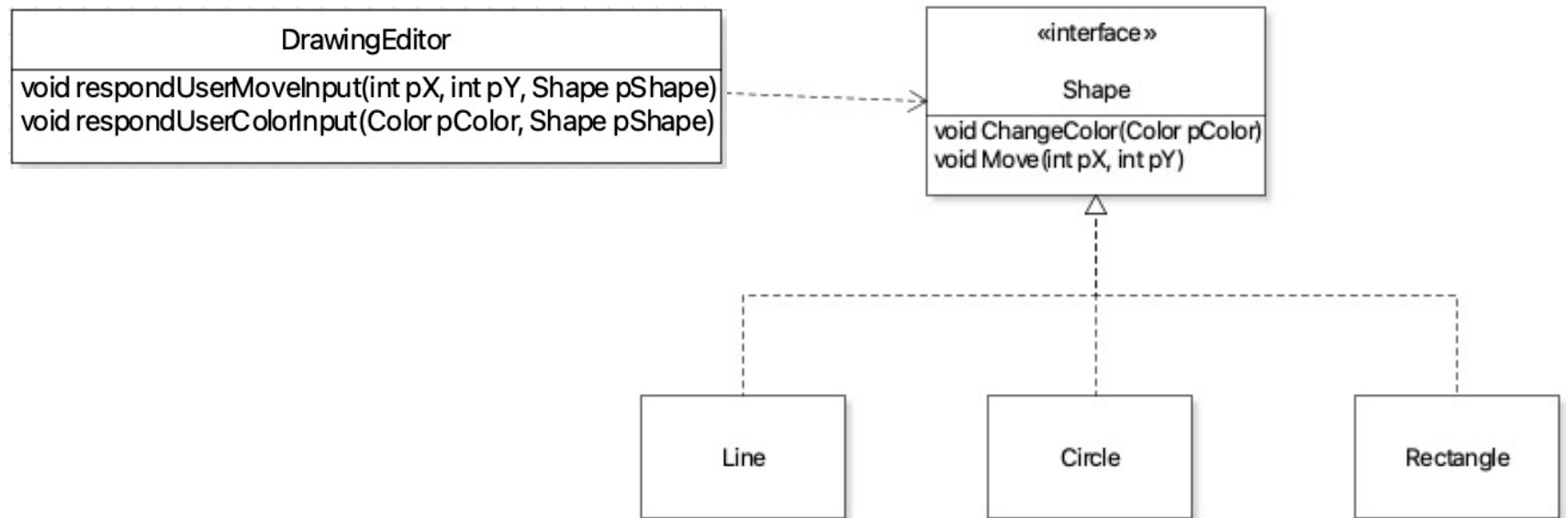

Line

- int x_start
- int y_start
- int x_end
- int y_end;
- Color currentColor

Line(int, int, int, int)

Circle

Rectangle

```
void respondUserMoveInput(int x, int y, Shape pShape) {
    pShape.move(x, y);
}
```
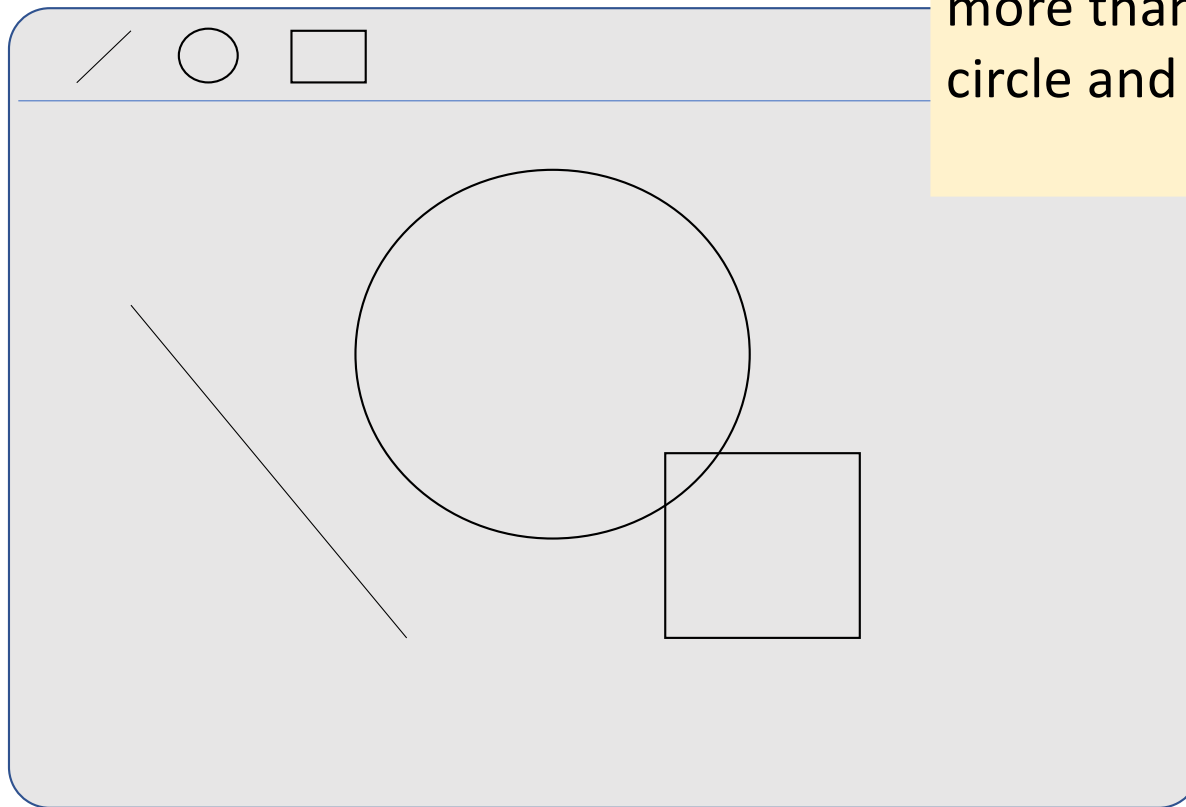
DrawingEditor
void respondUserMoveInput(int pX, int pY, Shape pShape)
void respondUserColorInput(Color pColor, Shape pShape)

«interface»
Shape
void ChangeColor(Color pColor)
void Move(int pX, int pY)

Line

Circle

Rectangle

Solution 1:

DrawingEditor

void respondUserMoveInput(int pX, int pY, Shape pShape)
void respondUserColorInput(Color pColor, Shape pShape)
void respondUserMoveInput(int pX, int pY, List<Shape> pShape)
void respondUserColorInput(Color pColor, List<Shape> pShape)

«interface»
Shape

void ChangeColor(Color pColor)
void Move(int pX, int pY)

Line

Circle

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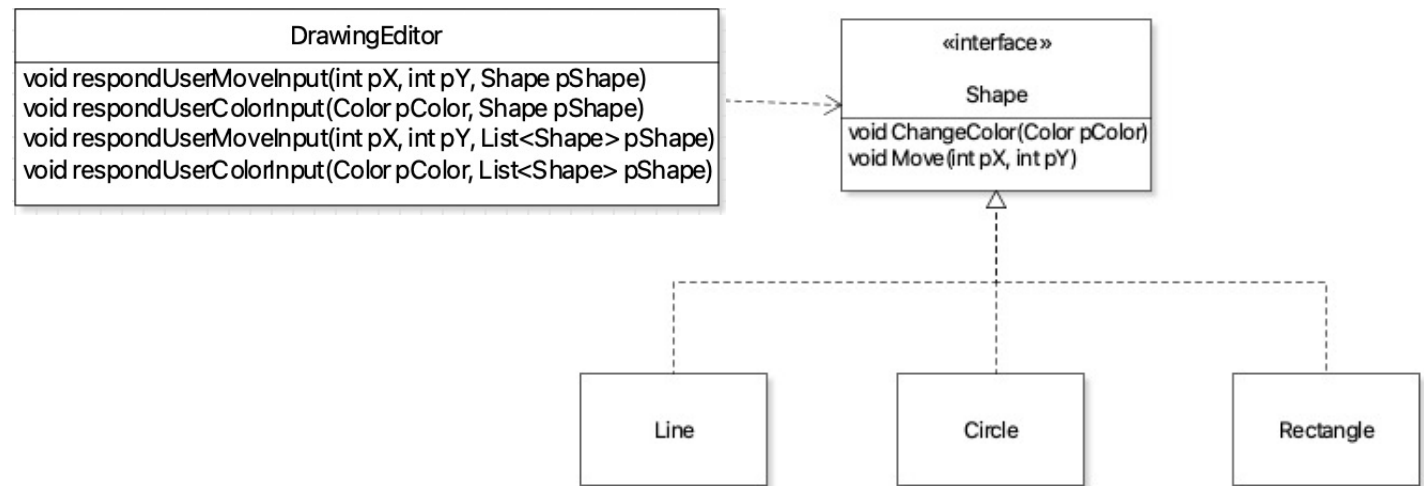
```
void respondUserMoveInput(int x, int y, List<Shape> pShapes) {
    for(Shape aShape: pShapes) {
        aShape.move(x, y);
    }
}
```

causes some complications

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

even though the action is similar
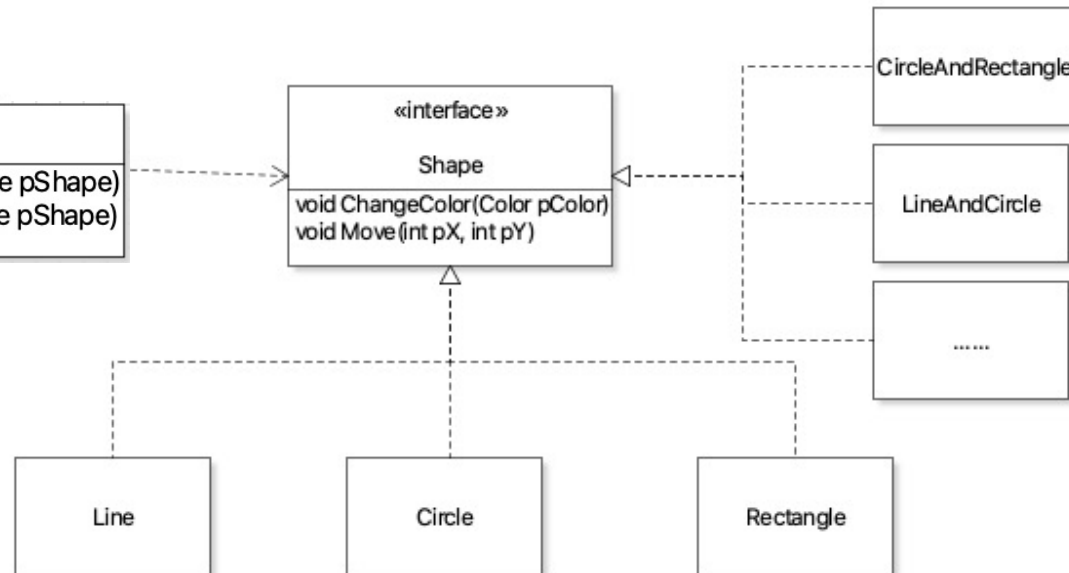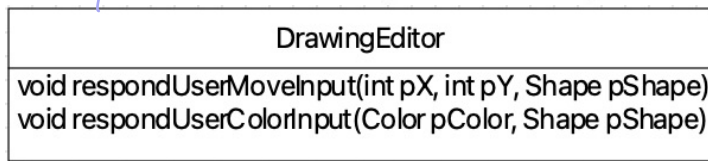
additional selection step

Solution 2:

Large number of possible structures.

Each option requires a class definition, even rare cases.

Impossible to accommodate all situations.

lots of things to write (handwritten)

doesn't need to change (handwritten)

new objects that represent grouped objects (handwritten)

1 client side is more stable (handwritten)

DrawingEditor

void respondUserMoveInput(int pX, int pY, Shape pShape)
void respondUserColorInput(Color pColor, Shape pShape)

«interface»

Shape

void ChangeColor(Color pColor)
void Move(int pX, int pY)
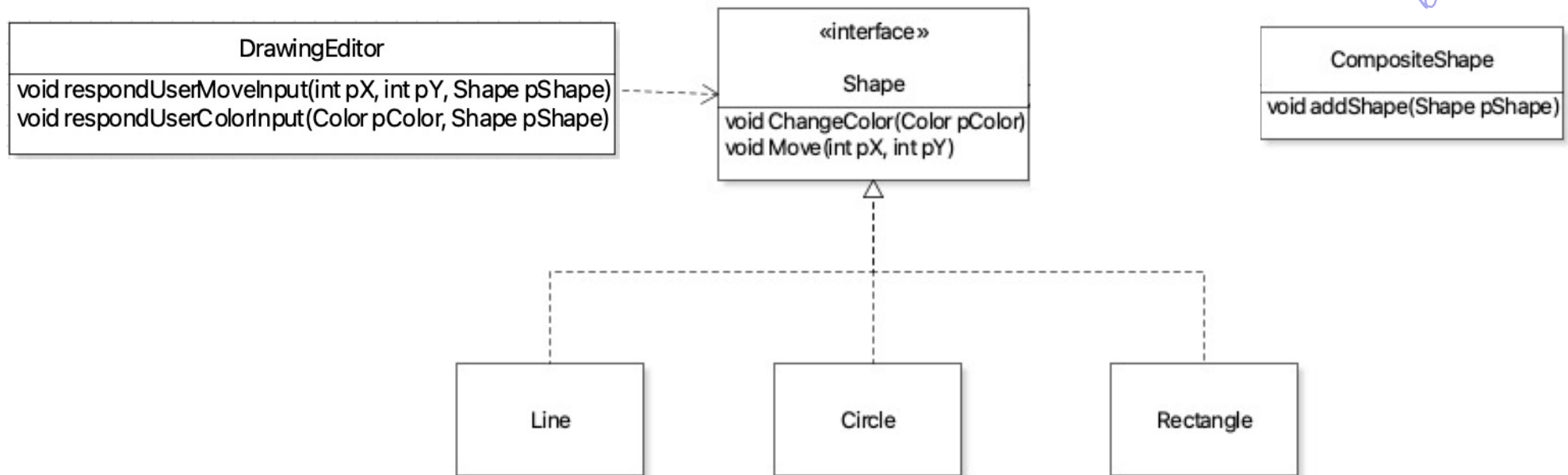
CircleAndRectangle
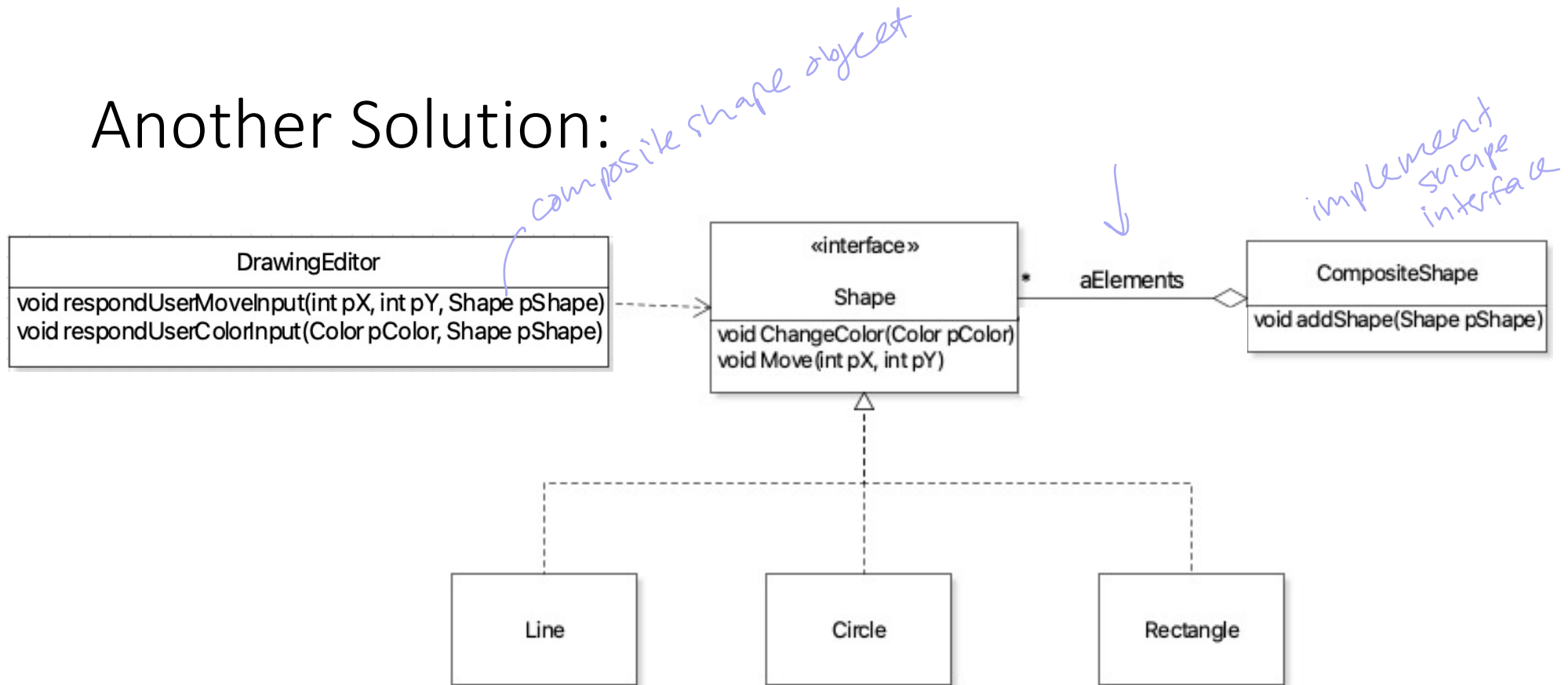
LineAndCircle

......

Line

Circle

Rectangle

Another Solution

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

Another Solution:

DrawingEditor

void respondUserMoveInput(int pX, int pY, Shape pShape)
void respondUserColorInput(Color pColor, Shape pShape)

«interface»

Shape

void ChangeColor(Color pColor)
void Move(int pX, int pY)

CompositeShape

void addShape(Shape pShape)

Line

Circle

Rectangle

Another Solution:

composite shape object (handwritten annotation)

implement shape interface (handwritten annotation)

DrawingEditor

void respondUserMoveInput(int pX, int pY, Shape pShape)
void respondUserColorInput(Color pColor, Shape pShape)

«interface»

Shape

void ChangeColor(Color pColor)
void Move(int pX, int pY)

* aElements

CompositeShape

void addShape(Shape pShape)

Line

Circle

Rectangle

Another Solution:

DrawingEditor

void respondUserMoveInput(int pX, int pY, Shape pShape)
void respondUserColorInput(Color pColor, Shape pShape)

«interface»

Shape

void ChangeColor(Color pColor)
void Move(int pX, int pY)

* aElements

CompositeShape

void addShape(Shape pShape)

Line

Circle

Rectangle

:DrawingEditor

aShape1 =

aShape2 =

:CompositeShape

aElements =

:Circle

:Rectangle

:Line

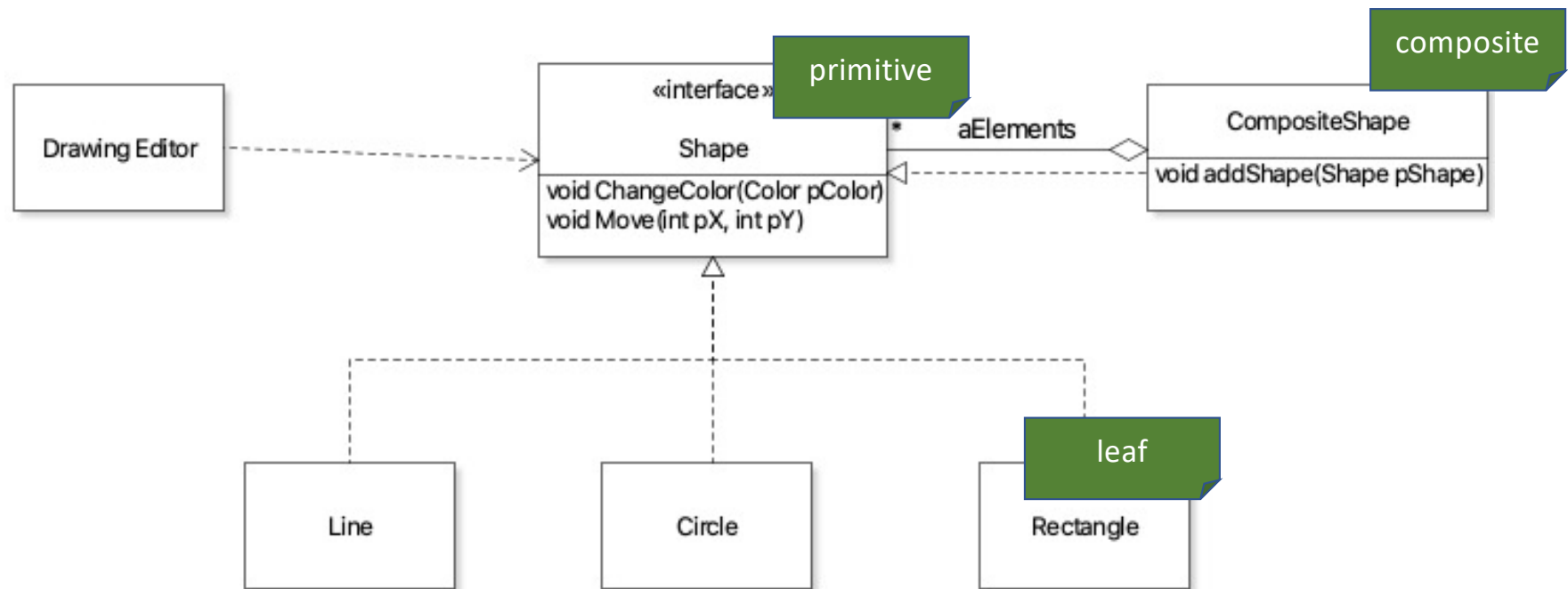atomic Shapes

atomic Shape

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

    concrete *Defines behaviour for primitives* atomic shapes
  - Composite

    concrete *Defines behaviour for primitives to have children*

Composite Pattern

Implement Composite Pattern

```java
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) {/* … */}


}
```

of atomic shapes

enables client to
add a shape
to it

Activity2: how to add Primitive instances to Composite?

```java
public class CompositeShape implements Shape
{

    … …


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

}
```

What are other options?
What are their tradeoffs?

put add method in interface
↳ but leafs do not need add method (could provide empty body)

Other considerations

- Is the order of accessing the children important? *– iterator?*

- Should the references to the parent be maintained from the children? *Should children point to the parent*

- Should a child be allowed to be added to more than one component?

Object Collaboration

```java
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.
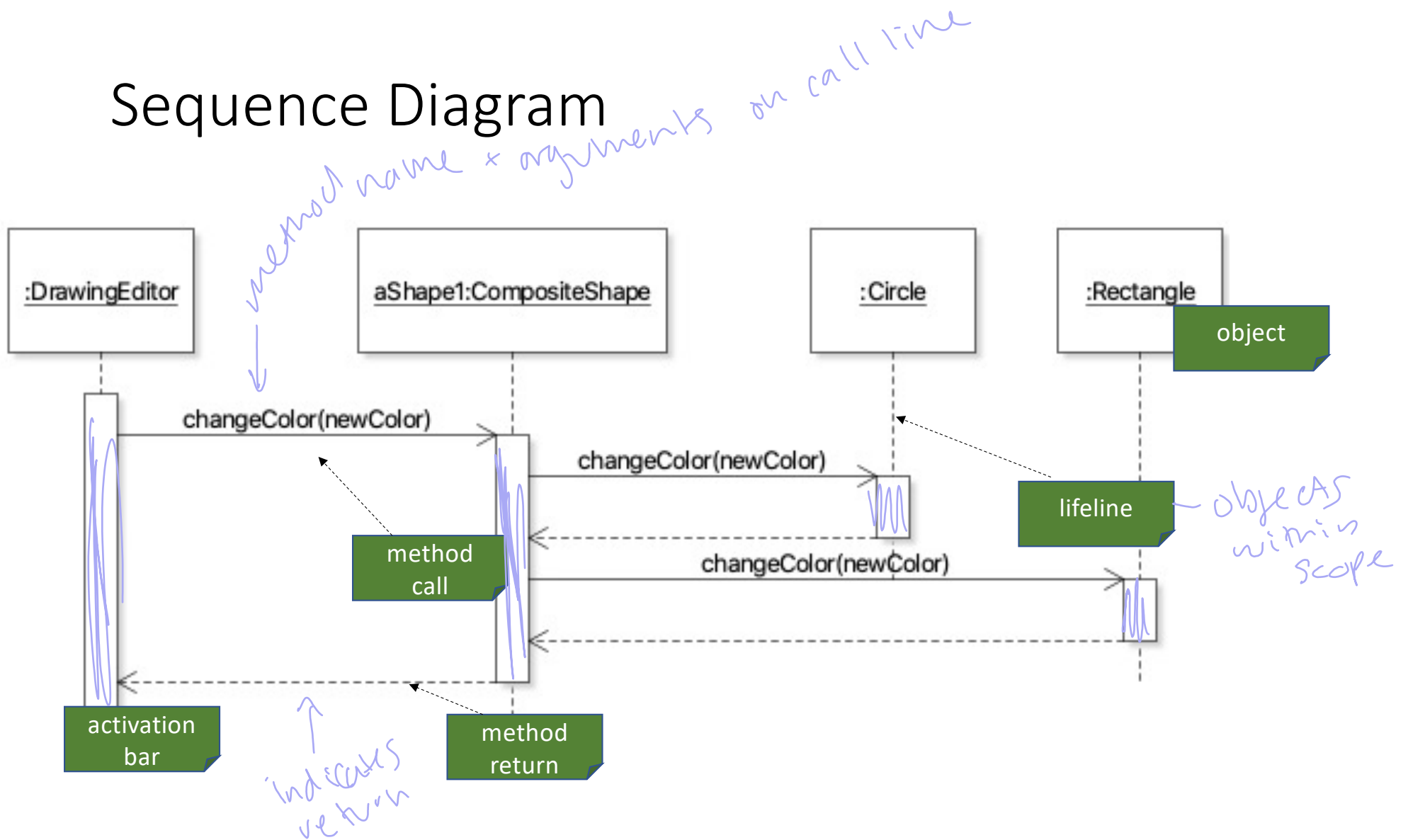
Modeling object call sequences?
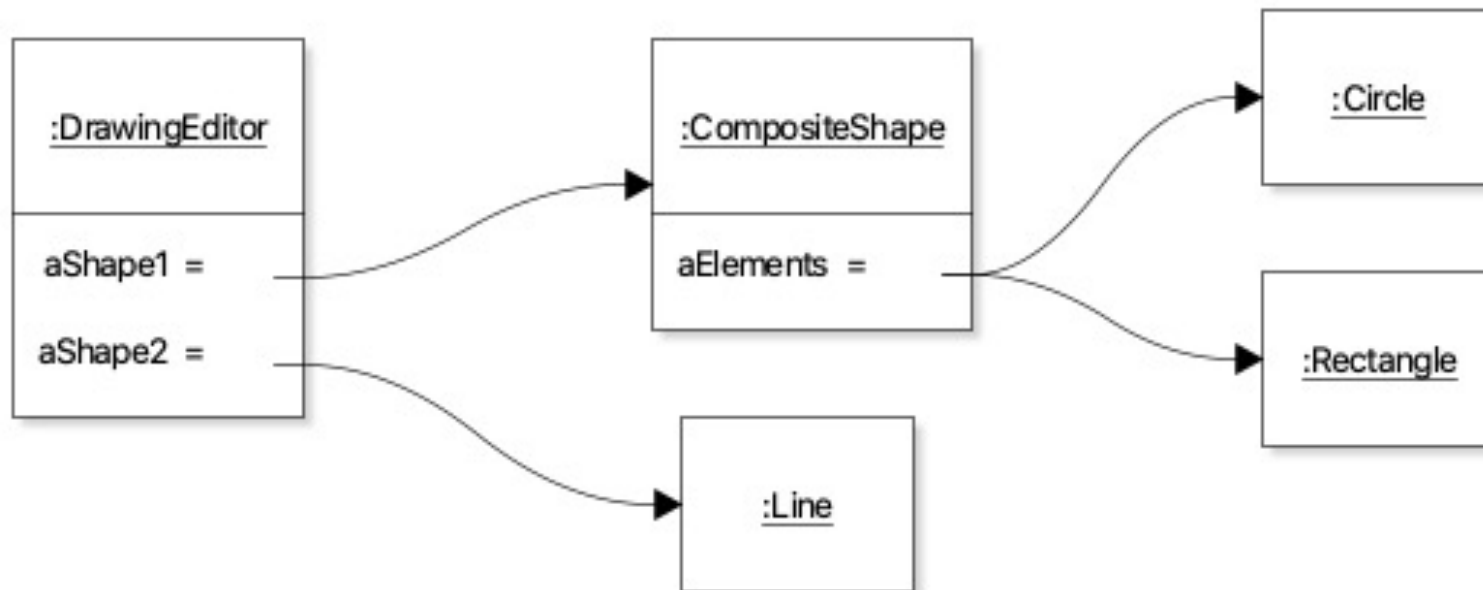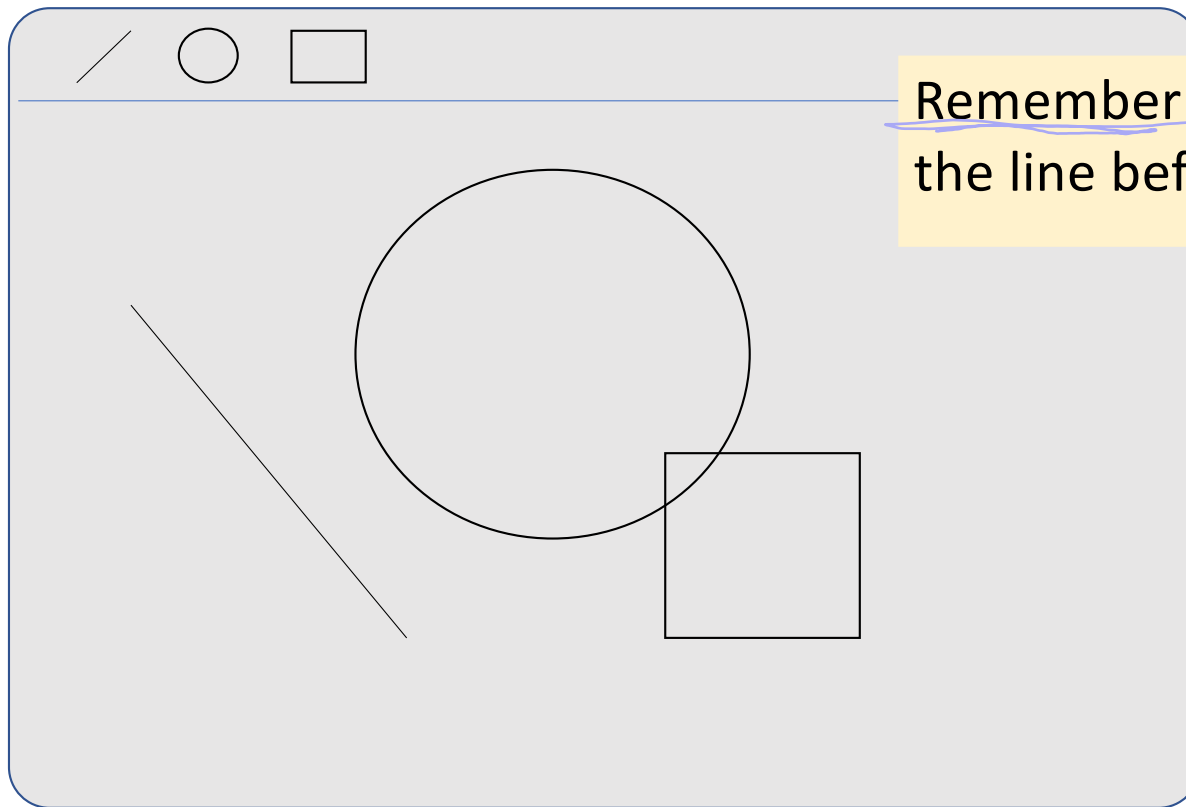
Objective

- 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

method name + arguments on call line

:DrawingEditor	aShape1:CompositeShape	:Circle	:Rectangle

object

changeColor(newColor)

changeColor(newColor)

changeColor(newColor)

method call

lifeline

~ objects within scope

activation bar

method return

indicates return

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

Remember the previous color of the line before changing.

```java
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;

}
```

ChangeColor()

Skin Versus Gut

Performing essential things

functionalities that don't change fundamentals

Separate the essential and additional state and behavior

```java
public class MemoryLine implements Shape
{

    private Line aLine;          ← composite of gut



    private Color aPreviousColor;
    @Override
    public void changeColor(Color pColor)
    {
        aPreviousColor = aLine.getColor();   ← additional stuff
        aLine.chageColor(pColor);
    }                 ∟ delegated to gut object

}
```

1. Delegate the original request
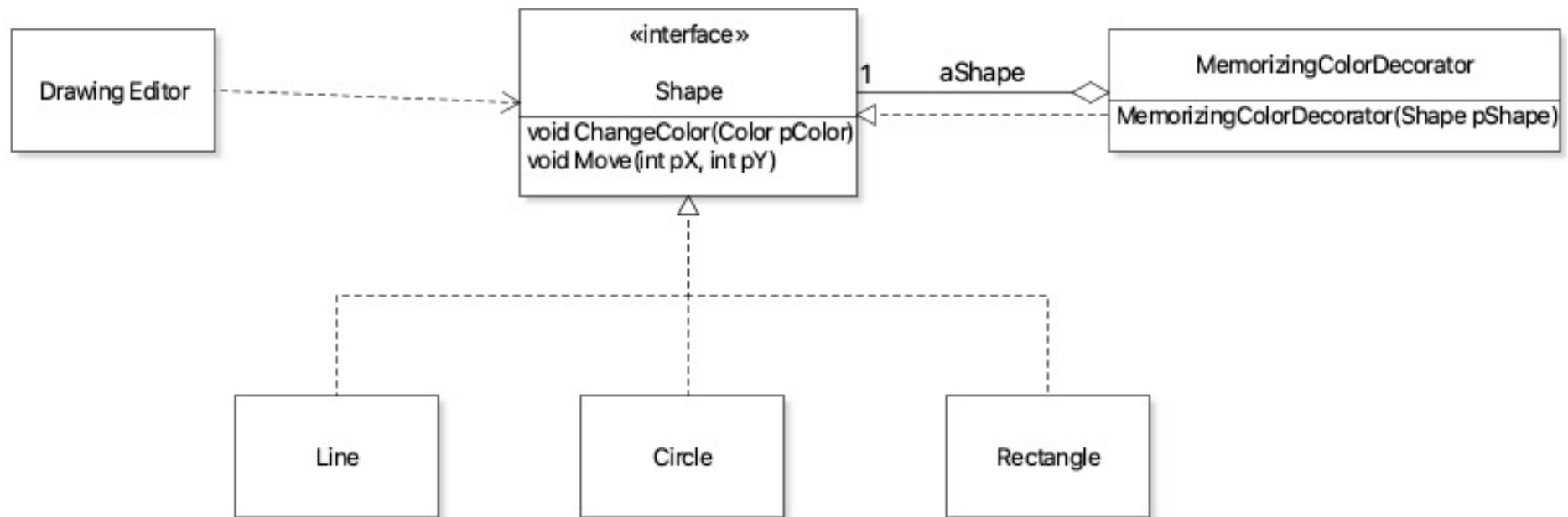
2. Implement additional feature

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)

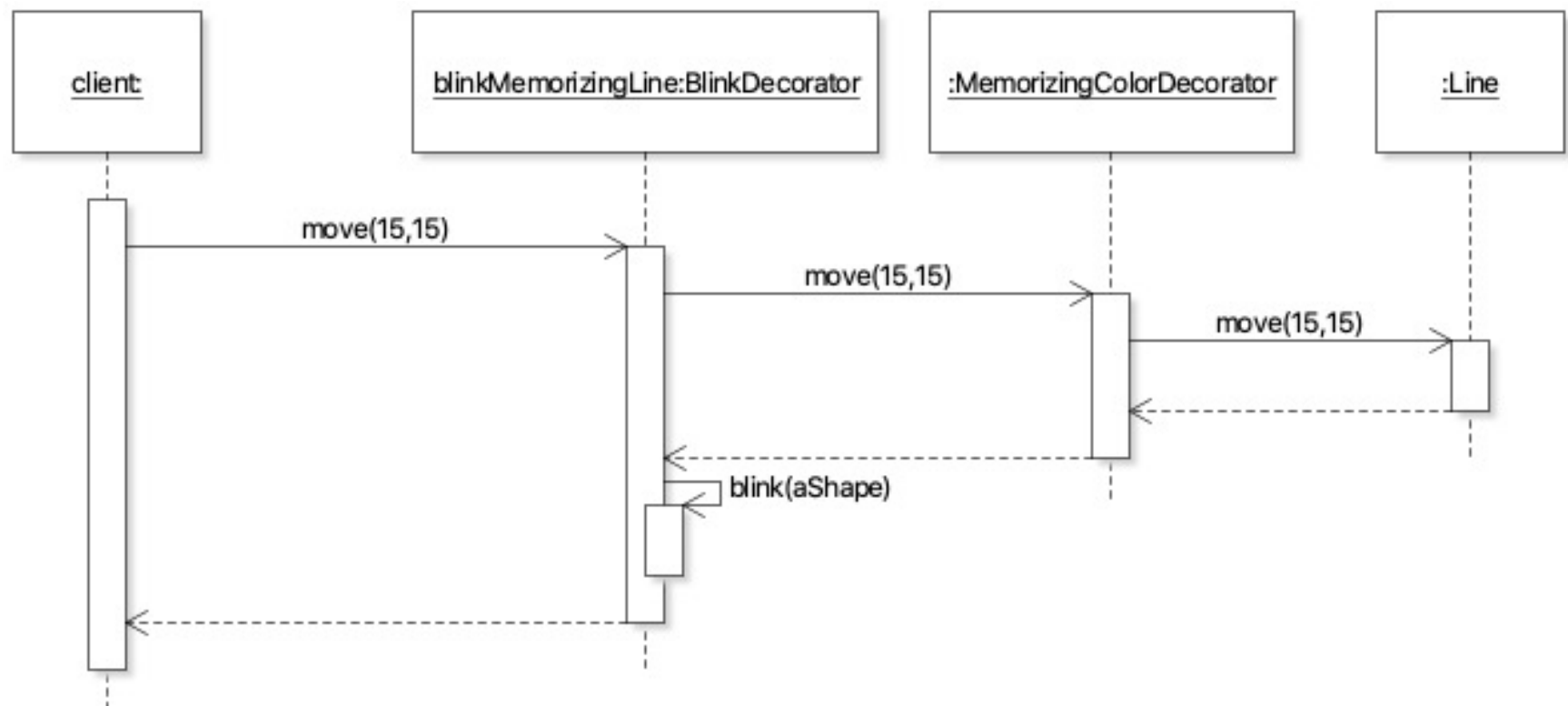| Line |
| --- |

Circle

Rectangle

```java
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
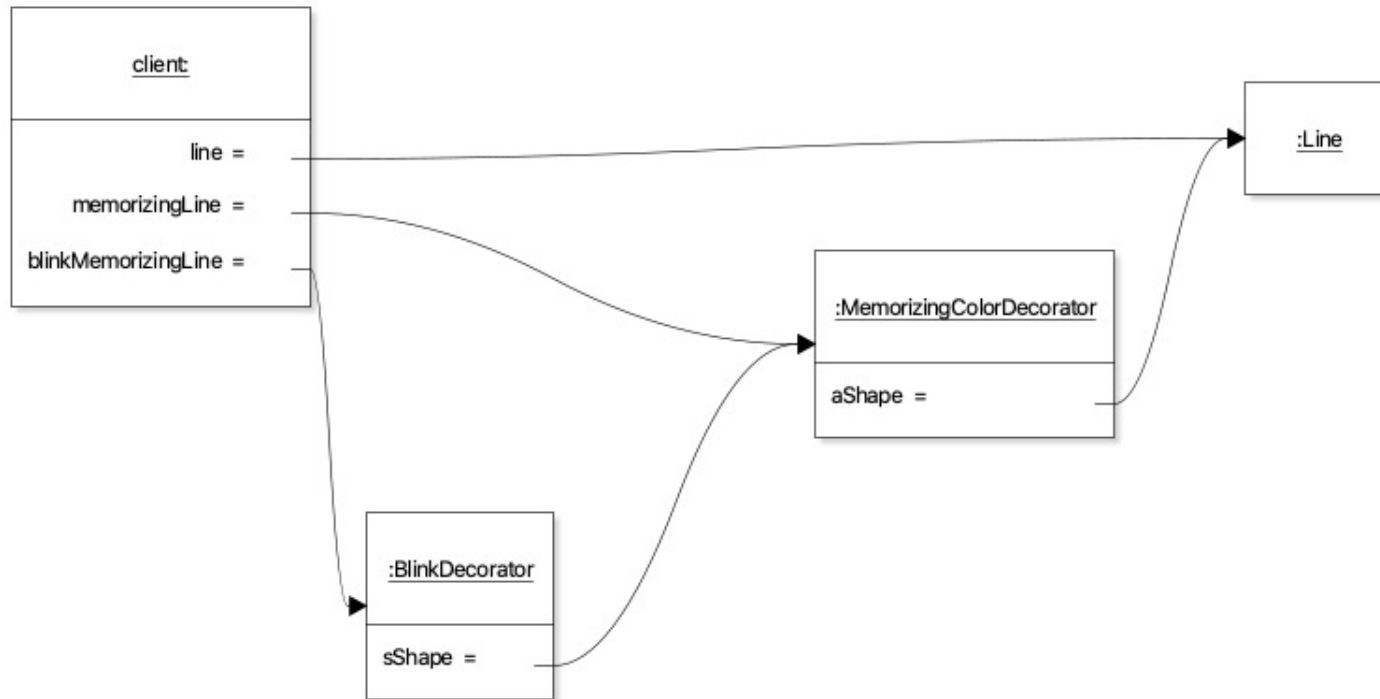
sShape =

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?

all implement shape
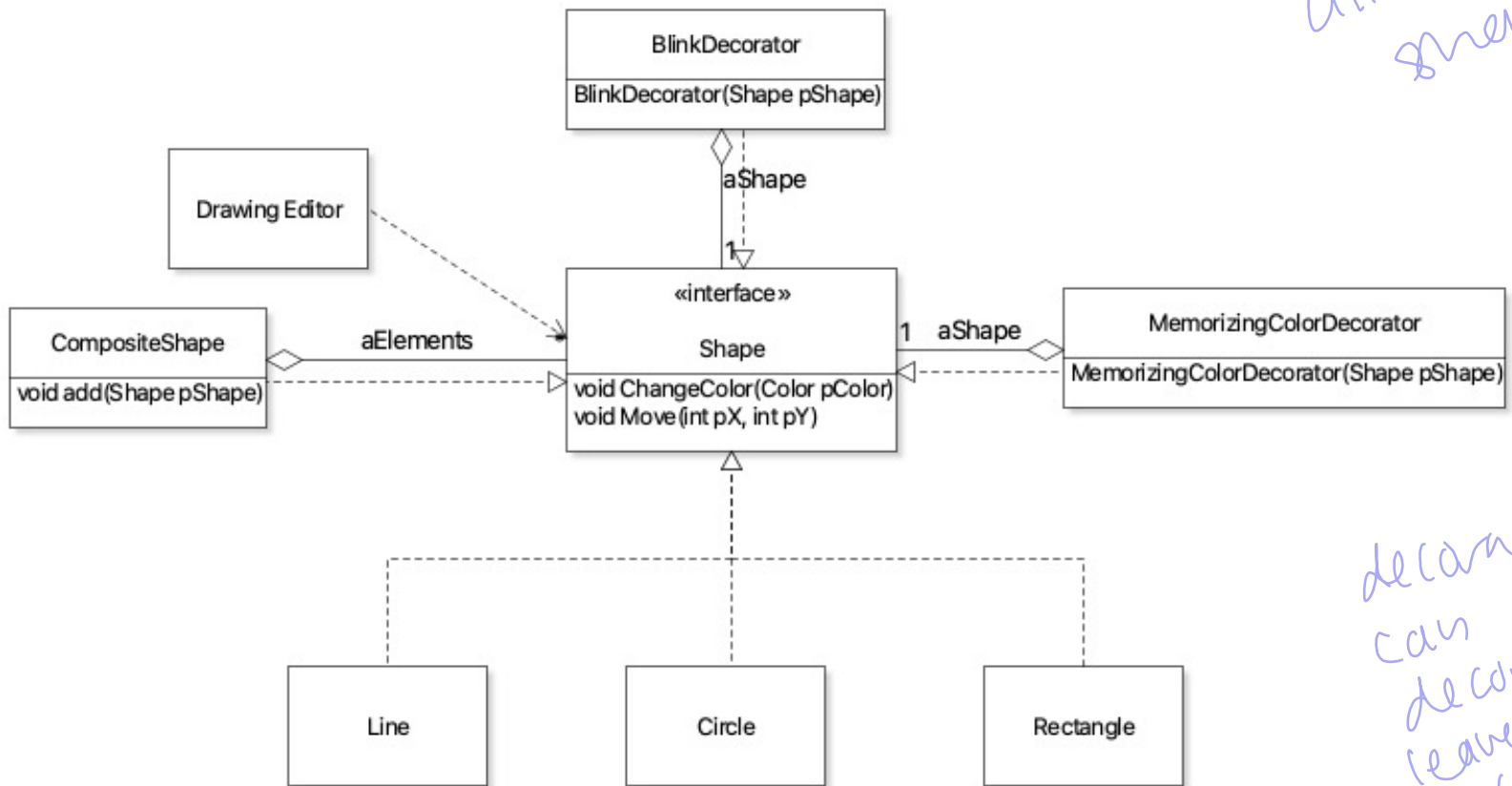
decorators can decorate leaves or/and composites

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:
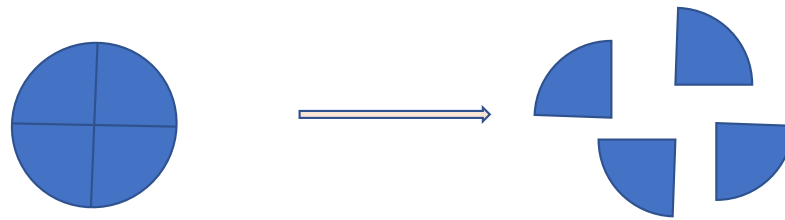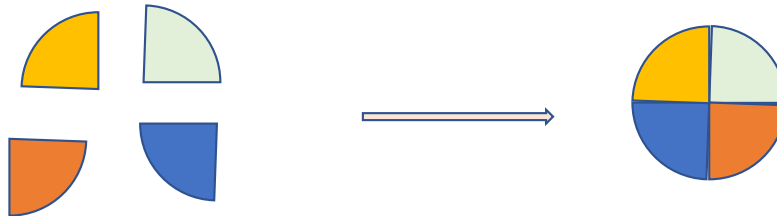Composite Pattern, Decorator Pattern, God class

See M6_b

Manage Complexity -- Divide and conquer
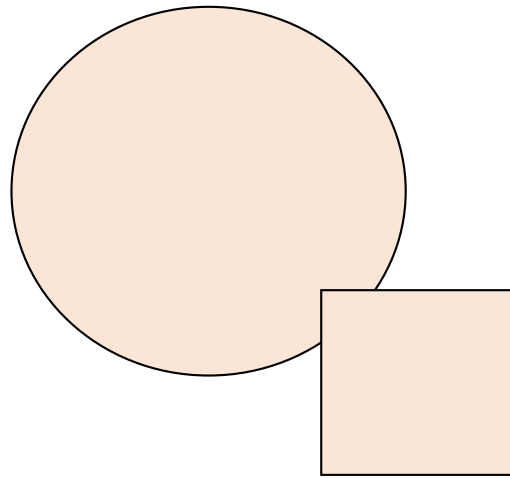
- Modularization
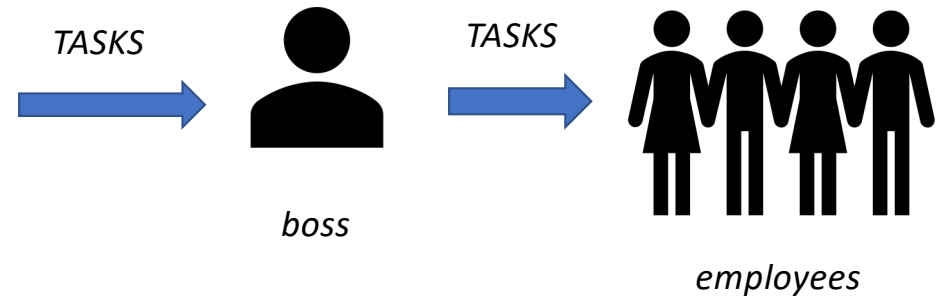
  - Decomposable

  - Composable

Purpose 1

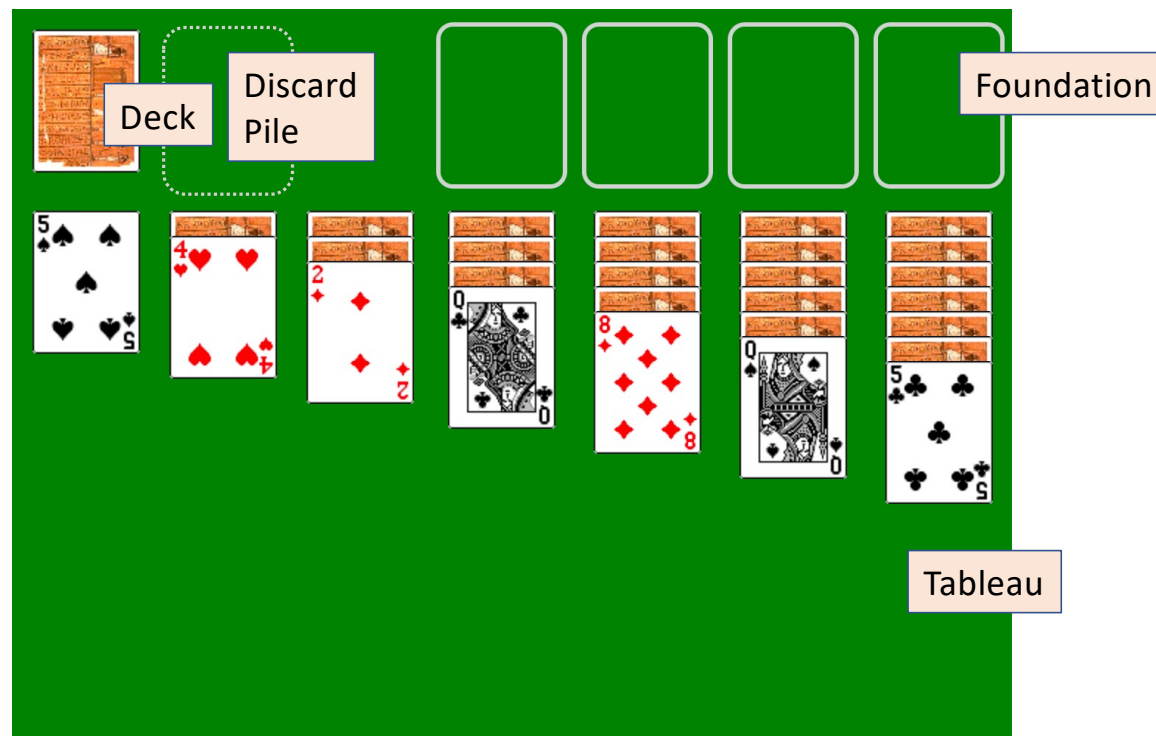- Aggregation: Representation of collections

Purpose 2

- Delegation: Redirect duties

GameModel in Solitaire

13 piles of cards?

God Class

Deck

Discard Pile

Foundation

Tableau

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

Class Diagram