



M6 (c) - Composition

Jin L.C. Guo

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

Recap of Module 6 so far

- Design Principle:
Divide and Conquer
- Programming mechanism:
Aggregation and Delegation, Polymorphic Object Cloning
- Design Techniques:
Sequence Diagram
- Patterns and Anti-patterns:
Composite Pattern, Decorator Pattern, Prototype Pattern, God class

Objective

- Design Principle:
Law of Demeter
- Patterns and Anti-patterns:
Command Pattern

Design Problem

Support shortcut for certain behavior, for example, move the selected shape 1pixel to the left.

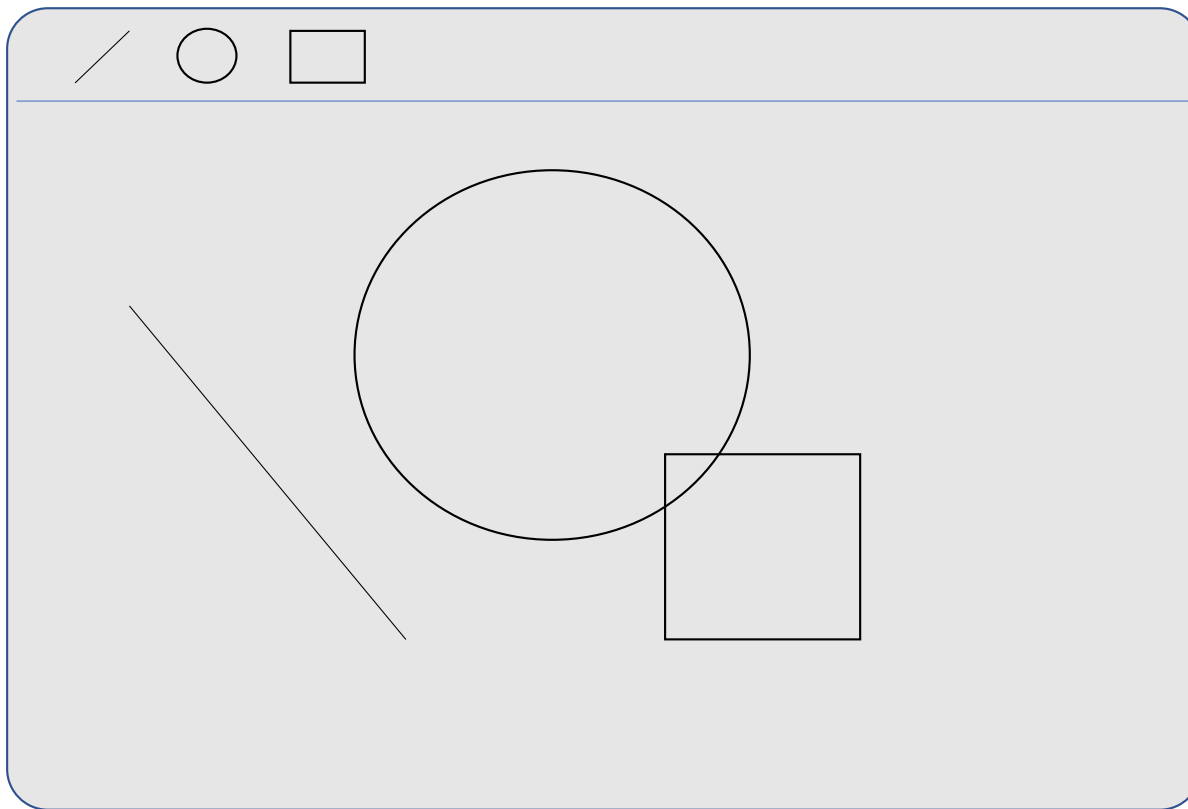
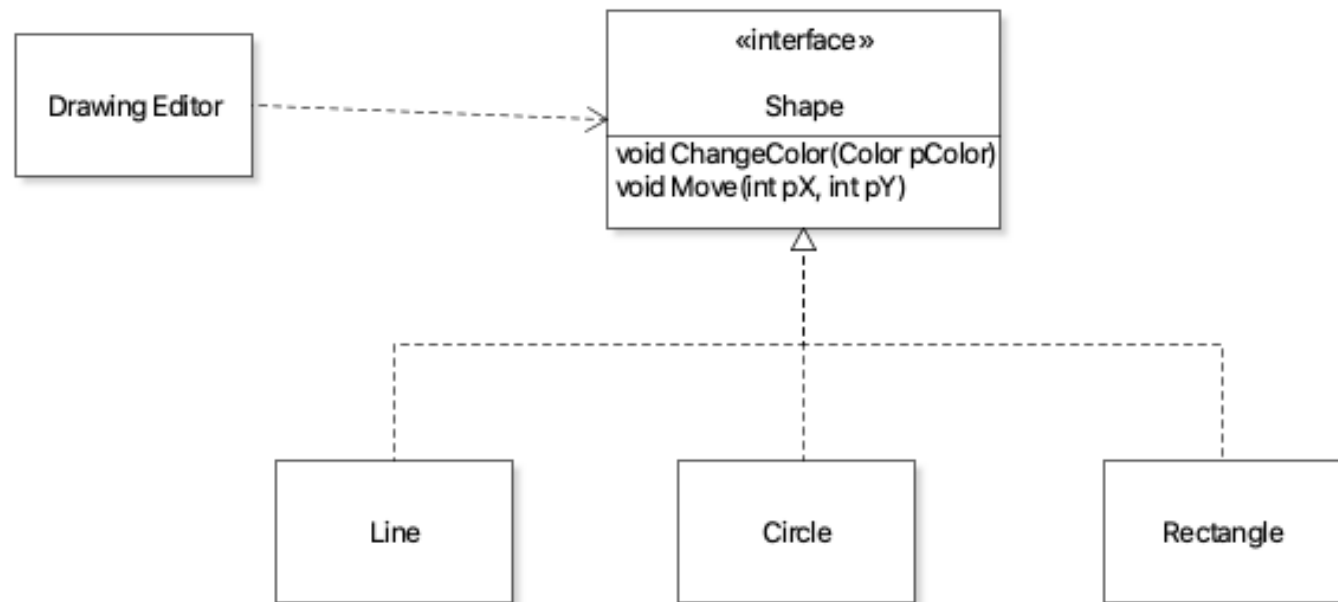


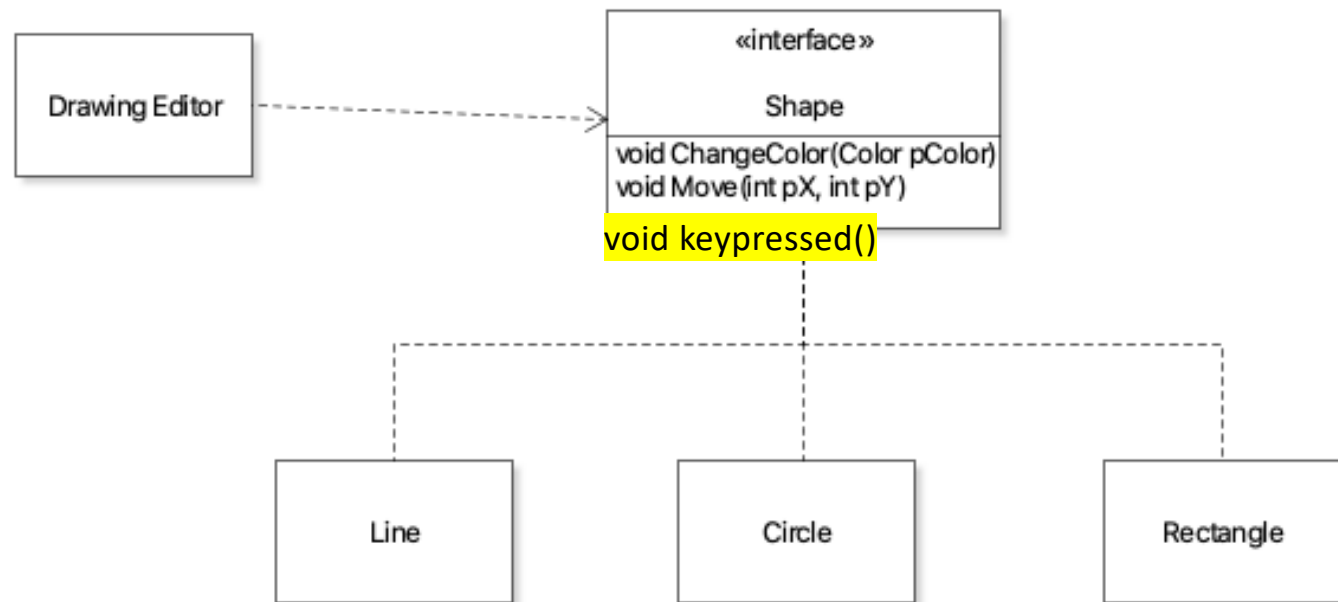


Image source: https://c1.staticflickr.com/4/3111/3145776919_74f05d63fd_b.jpg

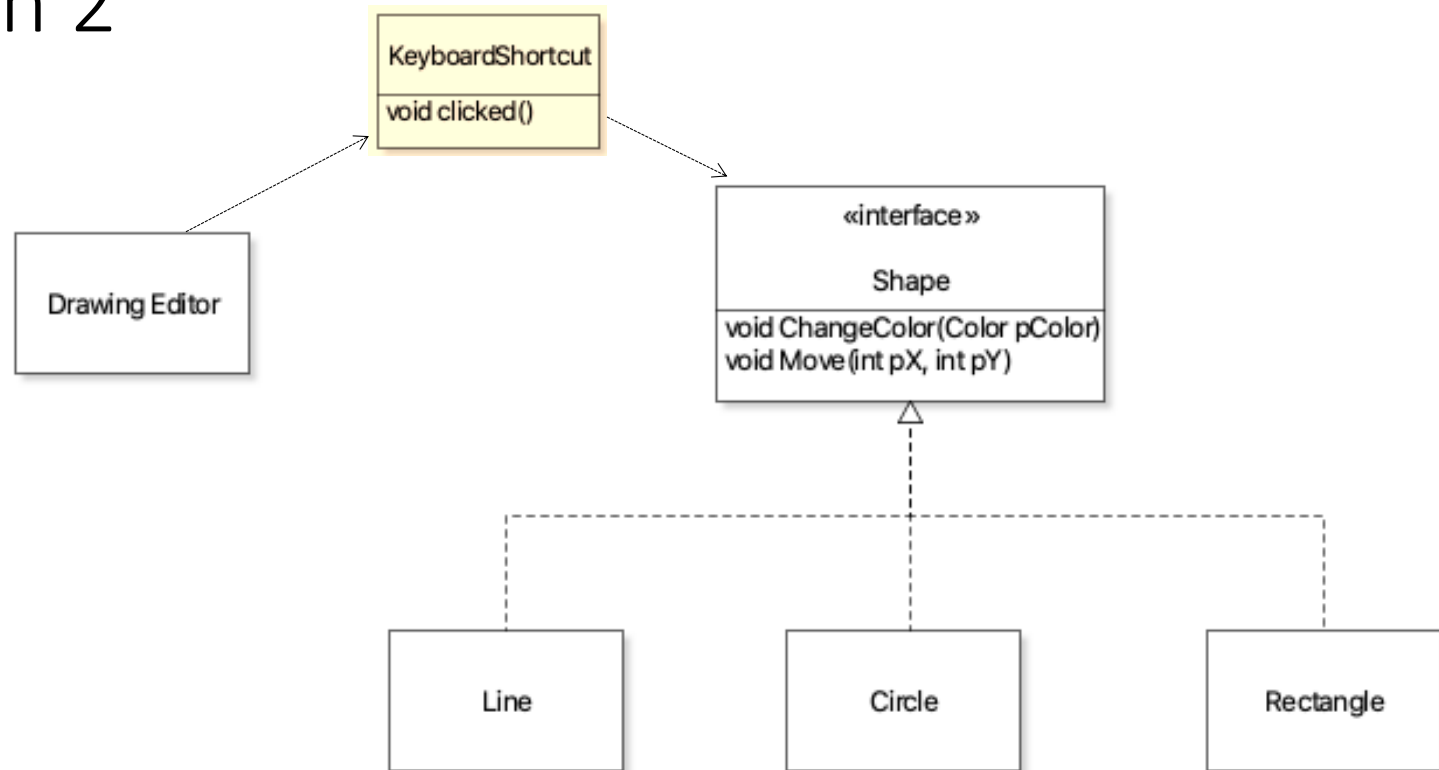
Ideas?



Solution 1

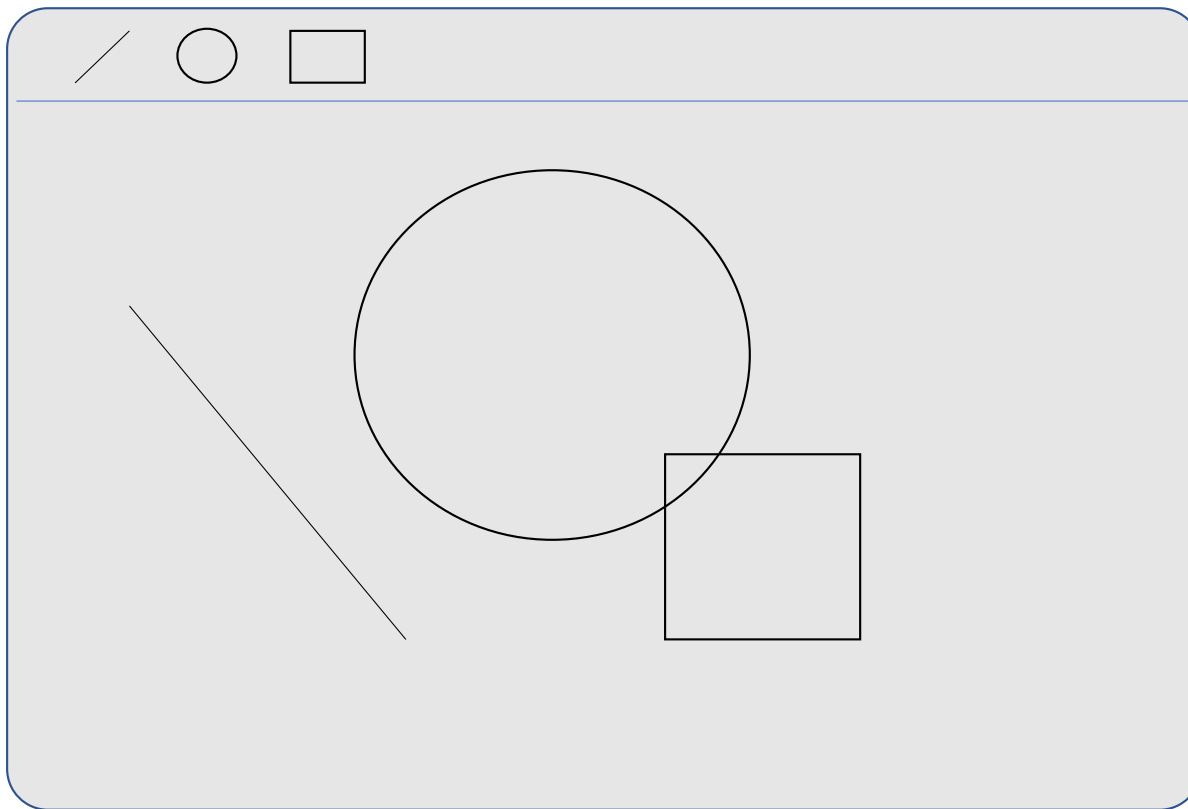


Solution 2

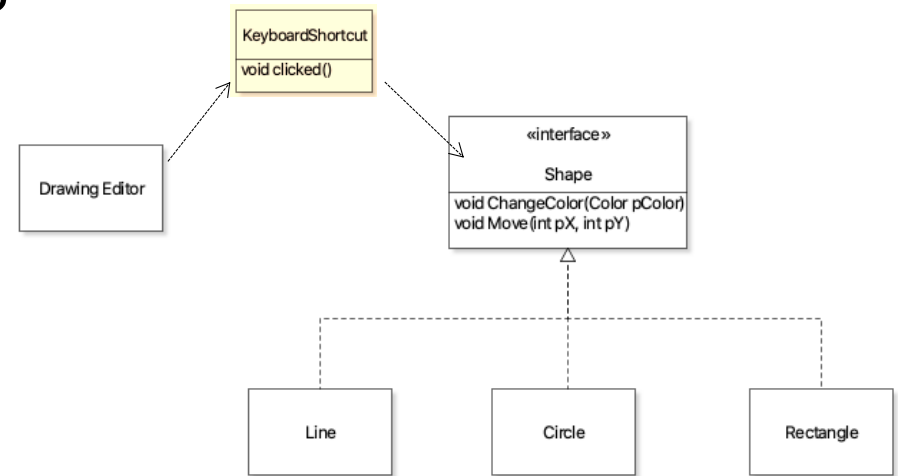
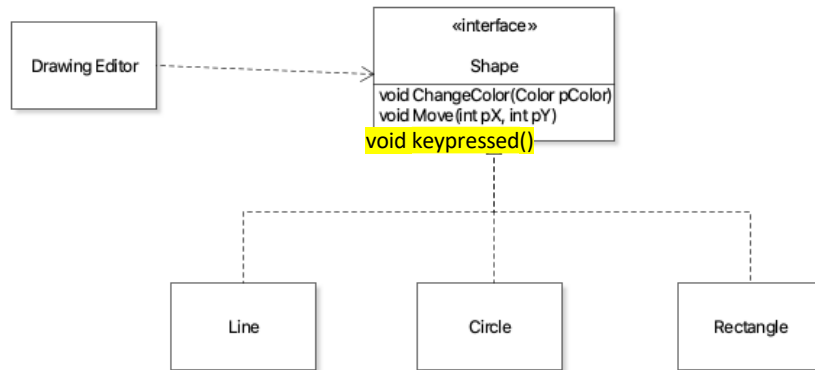


Design Problem

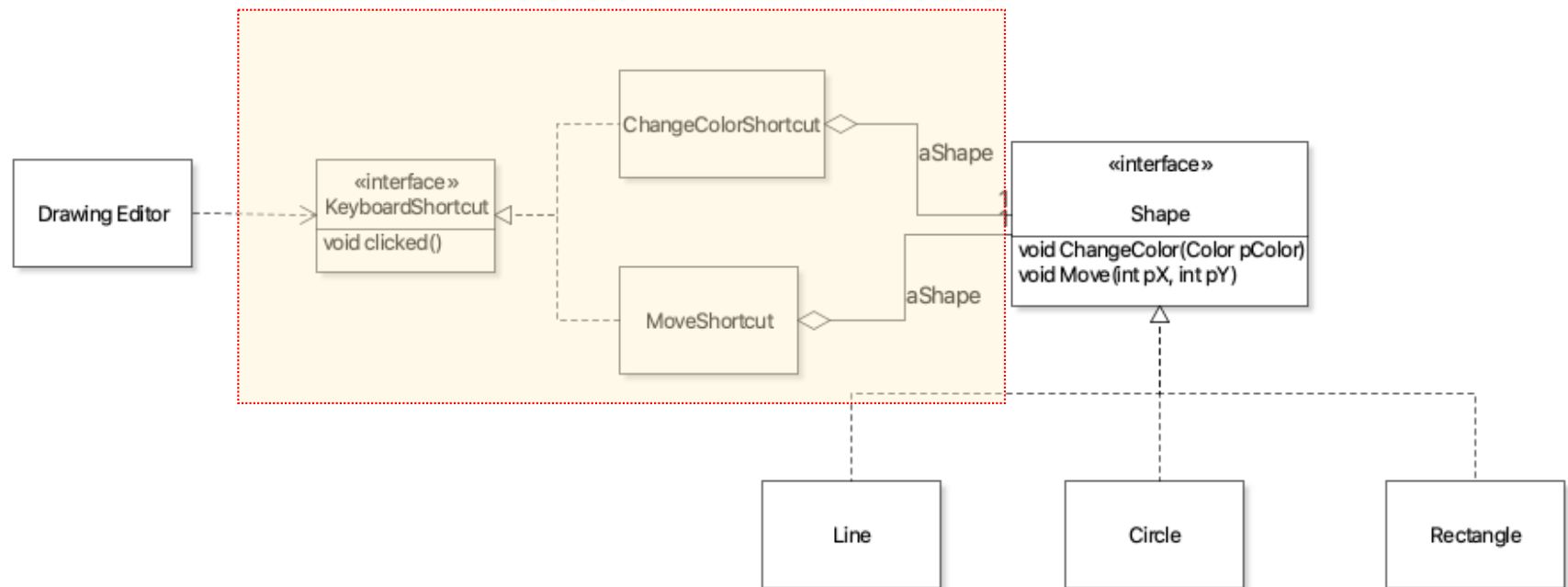
Support different shortcut for different behaviors, and reconfigurable.



Compare previous designs



Polymorphic shortcut behavior



```
public class MoveShortcut implements KeyboardShortcut {  
    private Shape aShape;  
    private int aX;  
    private int aY;  
  
    MoveShortcut(Shape pShape, int pX, int pY) {  
        aShape = pShape;  
        aX = pX;  
        aY = pY;  
    }  
  
    @Override  
    public void clicked() {  
        aShape.move(aX, aY);  
    }  
}
```

```
public class ChangeColorShortcut implements KeyboardShortcut {  
    private Shape aShape;  
    private Color aColor;  
  
    ChangeColorShortcut(Shape pShape, Color pColor) {  
        aShape = pShape;  
        aColor = pColor;  
    }  
  
    @Override  
    public void clicked() {  
        aShape.changeColor(aColor);  
    }  
}
```

```
public class DrawingEditor {  
    KeyboardShortcut aShortcut;  
  
    void setKeyboardShortcut(KeyboardShortcut pShortcut){  
        aShortcut = pShortcut;  
    }  
  
    void respondToShortcut(){  
        aShortcut.clicked();  
    }  
}
```

```
Shape lineObj = new Line(5, 5, 10, 10);  
KeyboardShortcut ks = new MoveShortcut(lineObj, 1,0);  
editor.setKeyboardShortcut(ks);
```

```
public class DrawingEditor {  
    KeyboardShortcut aShortcut;  
  
    void setKeyboardShortcut(KeyboardShortcut pShortcut){  
        aShortcut = pShortcut;  
    }  
  
    void respondToShortcut(){  
        aShortcut.clicked();  
    }  
}
```

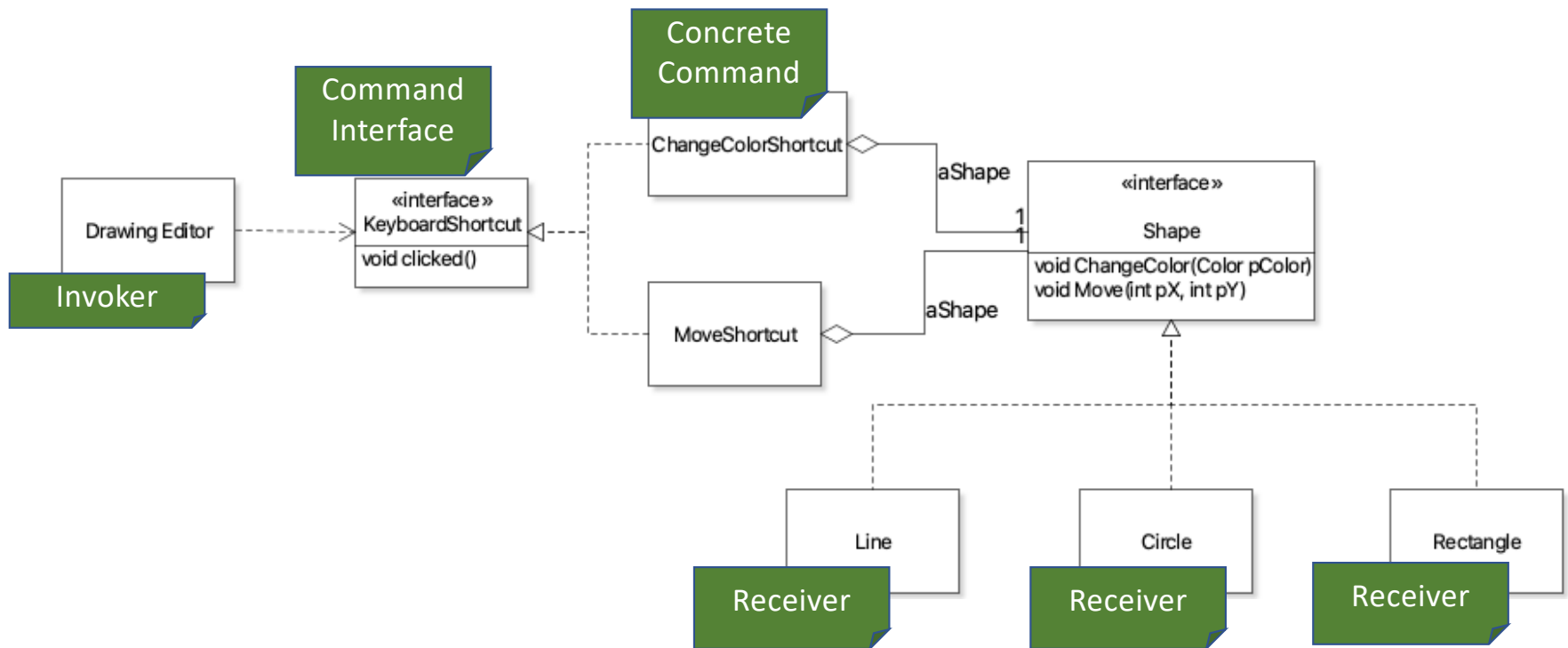
```
Shape lineObj = new Line(5, 5, 10, 10);  
KeyboardShortcut ks = new MoveShortcut(lineObj, 1,0);  
editor.setKeyboardShortcut(ks);
```

Client code:

```
editor.respondToShortcut();
```

Command

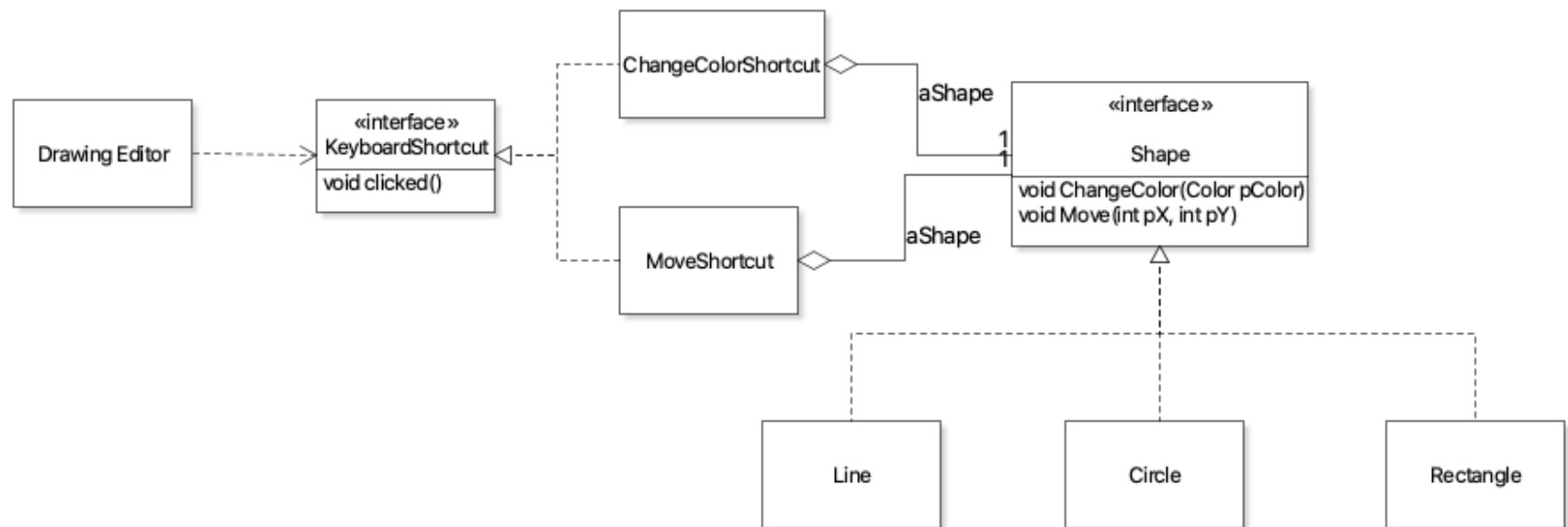
- Intent:
 - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Participants:
 - Command
 - declares an interface for executing an operation.*
 - ConcreteCommand
 - implements execute by invoking the corresponding operation(s) on Receiver.*
 - Receiver
 - knows how to perform the actual operation*
 - Invoker
 - execute the operation through function calls declared in Command Interface.*



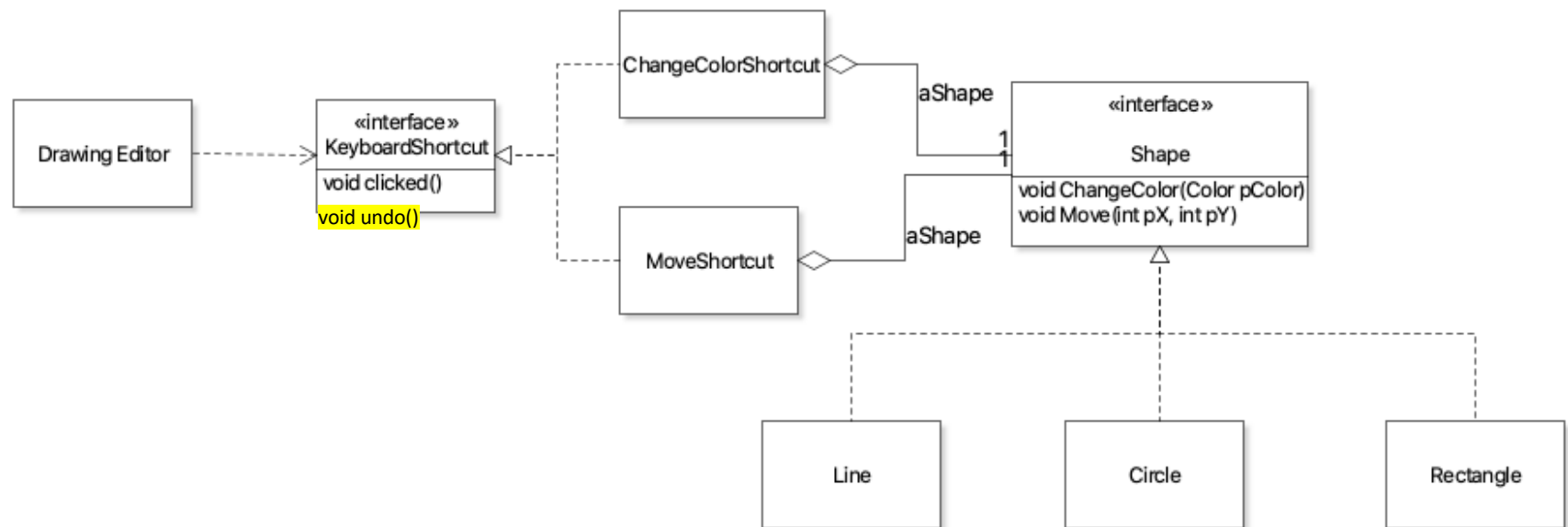
Command Pattern

- Intent:
 - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Participants:
 - Command
 - declares an interface for executing an operation.*
 - ConcreteCommand
 - implements execute by invoking the corresponding operation(s) on Receiver.*
 - Receiver
 - knows how to perform the actual operation*
 - Invoker
 - execute the operation through function calls declared in Command Interface.*

Activity1: How to support undo function



How to support undo function?



```

public class MoveShortcut implements KeyboardShortcut {
    private Shape aShape;
    private int aX;
    private int aY;
    private Shape aPreviousShape;

    MoveShortcut(Shape pShape, int pX, int pY) {
        aShape = pShape;
        aX = pX;
        aY = pY;
    }

    @Override
    public void clicked() {
        aPreviousShape = aShape.clone();
        aShape.move(aX, aY);
    }

    @Override
    public void undo() {
        aShape = aPreviousShape;
    }
}

```

Why won't this solution work?

```

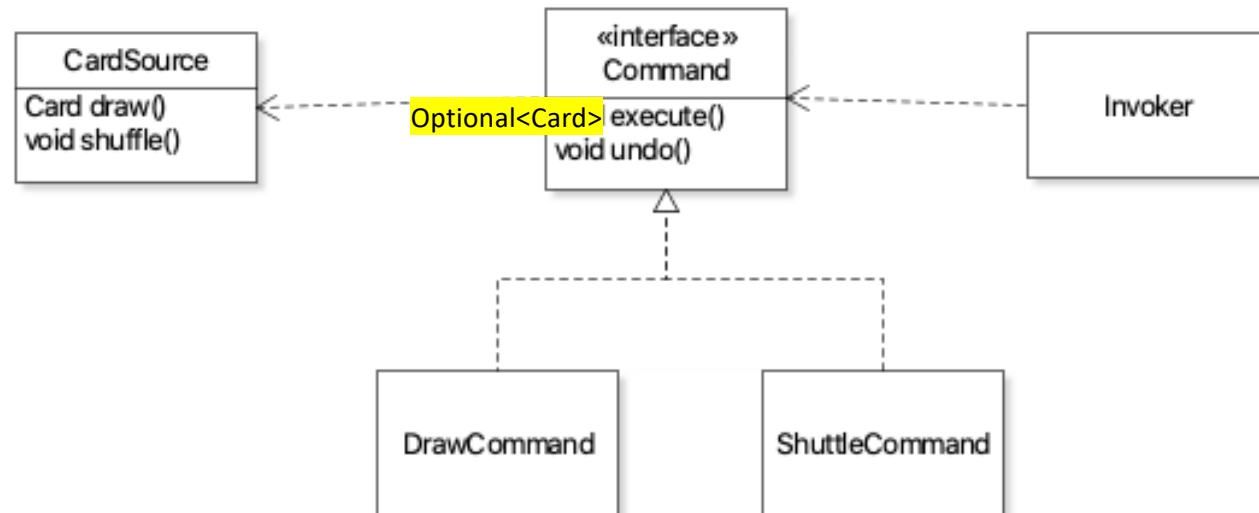
public class Line implements Shape {
    private int x_start;
    private int y_start;
    private int x_end;
    private int y_end;

    public KeyboardShortcut getMoveShortcut(int pX, int pY) {
        return new KeyboardShortcut() {
            int pre_x_start;
            int pre_y_start;
            int pre_x_end;
            int pre_y_end;

            @Override
            public void clicked() {
                pre_x_start = x_start;
                pre_y_start = y_start;
                pre_x_end = x_end;
                pre_y_end = y_end;
                move(pX,pY);
            }
            @Override
            public void undo() {
                x_start = pre_x_start;
                y_start = pre_y_start;
                x_end = pre_x_end;
                y_end = pre_y_end;
            }
        };
    }
}

```

What if some functions has return value?



```
public interface CardSourceCommand
{
    /**
     *
     * @return the production of the execution if it's a card,
     * empty if the execute doesn't produce output.
     */
    Optional<Card> execute();
    /**
     * Undo the immediate previous execution.
     */
    void undo();
}
```



```
public class DrawCommand implements CardSourceCommand
{
    private CardSource aCardSource;
    private Optional<Card> aCard;
    DrawCommand(CardSource pCardSource)
    {
        aCardSource = pCardSource;
    }

    @Override
    public Optional<Card> execute()
    {
        if(aCardSource.size()>0)
        {
            Card card = aCardSource.draw();
            aCard = Optional.of(card);
            return Optional.of(card);
        }
        else
        {
            return Optional.empty();
        }
    }
}
```

Consideration

- Access of command target and its state

Pass target as argument or use inner class

- Data flow

Return value of execution

- Command execution correctness

Respect precondition

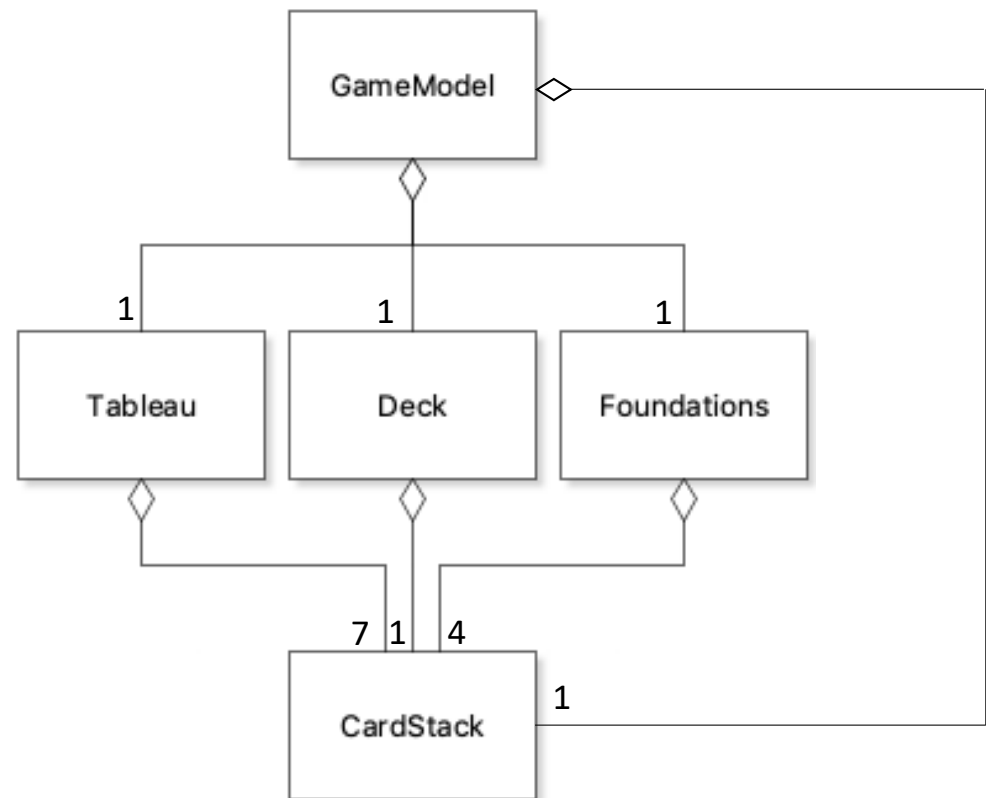
- Storing state

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Integreter
			Template Method ✓
Object	Abstract Factory	Adapter (class)	Chain of Responsibility
	Builder	Bridge	Command ✓
	Prototype ✓	Composite ✓	Iterator ✓
	Singleton ✓	Decorator ✓	Mediator
		Flyweight ✓	Memento
		Façade	Observer ✓
		Proxy	State
			Strategy ✓
			Visitor ✓

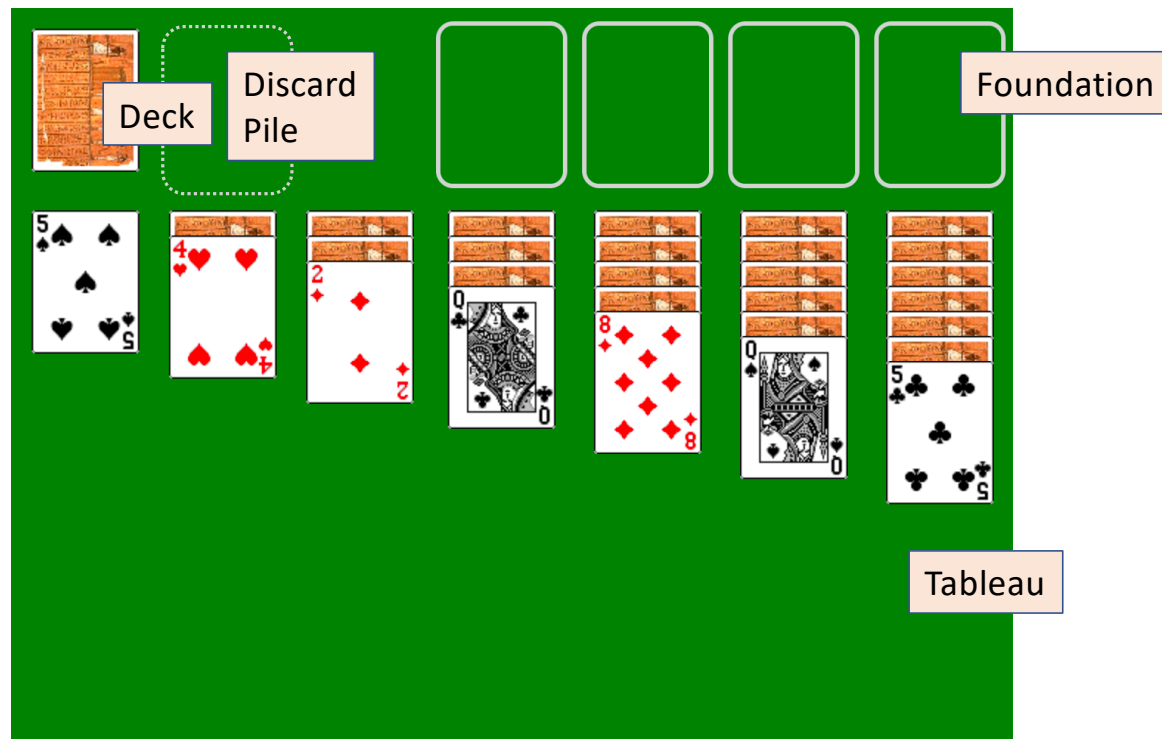
Objective

- Design Principle:
Law of Demeter
- Patterns and Anti-patterns:
Command Pattern

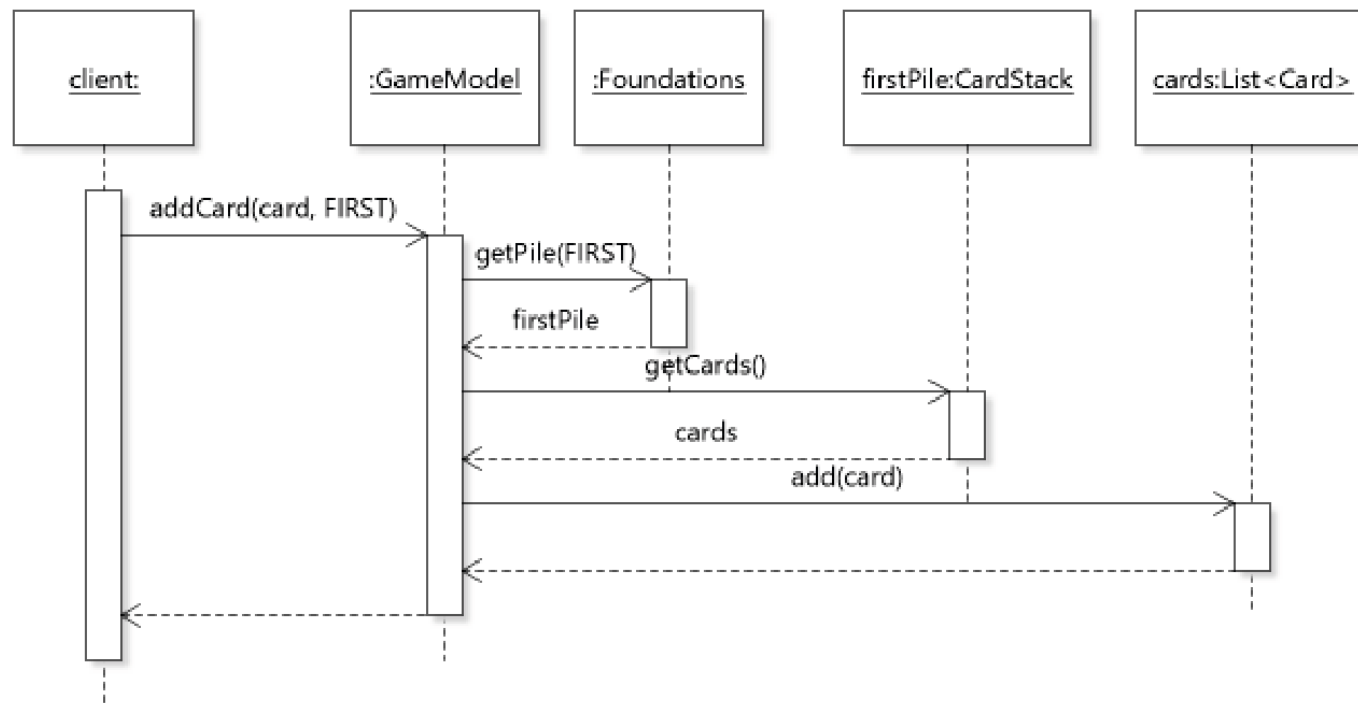
Delegation Chains



Scenario of adding a card to the first Foundation pile.



One option



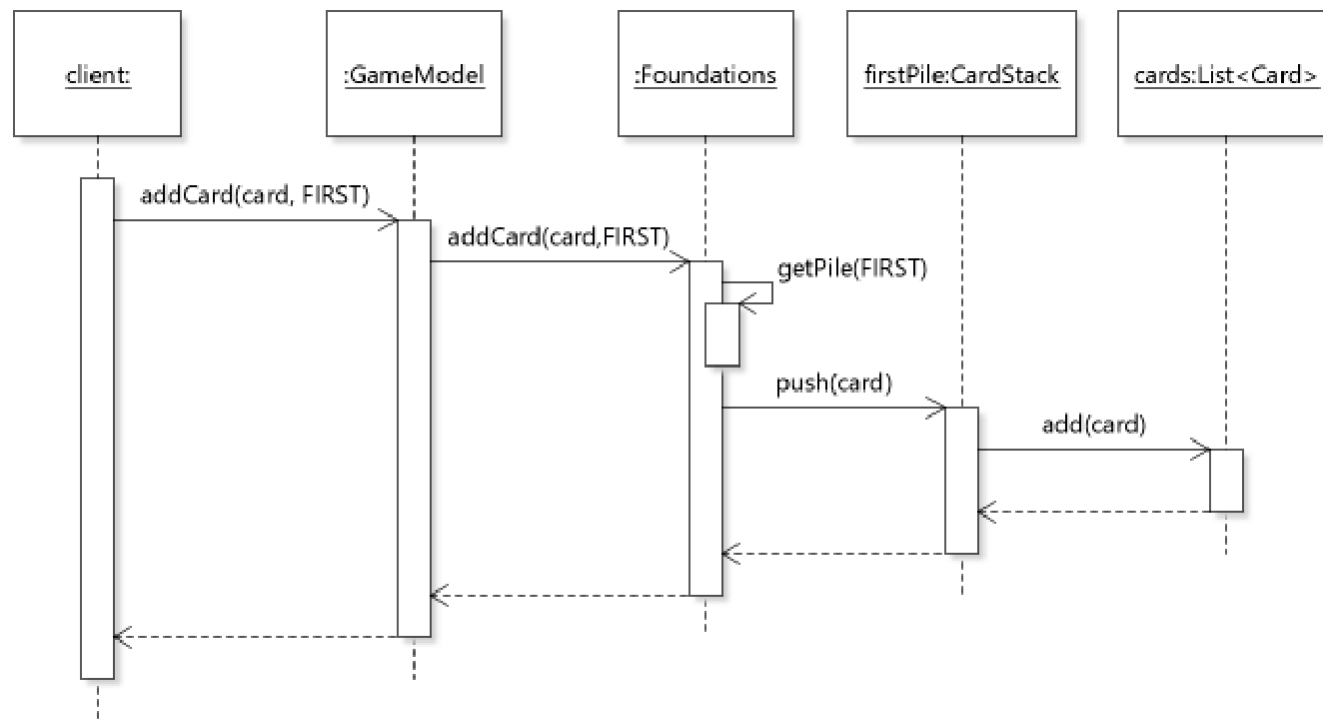
Law of Demeter

“Only talk to your friends”

The code of a method should only access:

- The instance variables of its implicit parameter;
- The arguments passed to the method;
- Any new object created within the method;
- (If need be) globally available objects.

Following the Law of Demeter



Activity 2:

- Determine if the method calls are allowed according to the Law of Demeter:

```
public class Colada {  
    private Blender aBlender;  
    private Vector aIngredients;  
  
    public Colada()  
    {  
        aBlender = new Blender();  
        aIngredients = new Vector();  
    }  
    public void addIngredientsToBlender()  
    {  
        aBlender.addIngredients(aIngredients.elements());  
    }  
    public void printReceipt(Inventory pInventory)  
    {  
        PriceCalculator priceCalculator = pInventory.getPriceCalculator();  
        Price price = priceCalculator.compute(aIngredients.elements());  
        System.out.print(price);  
    }  
}
```

Law of Demeter

“Only talk to your friends”

The code of a method should only access:

- The instance variables of its implicit parameter;
- The arguments passed to the method;
- Any new object created within the method;
- (If need be) globally available objects.

```
public class Colada {  
    private Blender aBlender;  
    private Vector aIngredients;  
  
    public Colada()  
    {  
        aBlender = new Blender();  
        aIngredients = new Vector();  
    }  
    public void addIngredientsToBlender()  
    {  
        aBlender.addIngredients(aIngredients.elements());  
    }  
    public void printReceipt(Inventory pInventory)  
    {  
        PriceCalculator priceCalculator = pInventory.getPriceCalculator();  
        Price price = priceCalculator.compute(aIngredients.elements());  
        System.out.print(price);  
    }  
}
```

Acknowledgement

- Some examples are from the following resources:
 - *COMP 303 Lecture note* by Martin Robillard.
 - *The Pragmatic Programmer* by Andrew Hunt and David Thomas, 2000.
 - *Effective Java* by Joshua Bloch, 3rd ed., 2018.