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

Question from previous lecture

Can we achieve polymorphic copying through static factory method?

No.

Java’s override mechanism

“If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass _hides_ the one in the superclass.

The distinction between hiding a static method and overriding an instance method has important implications:

- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.”

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.

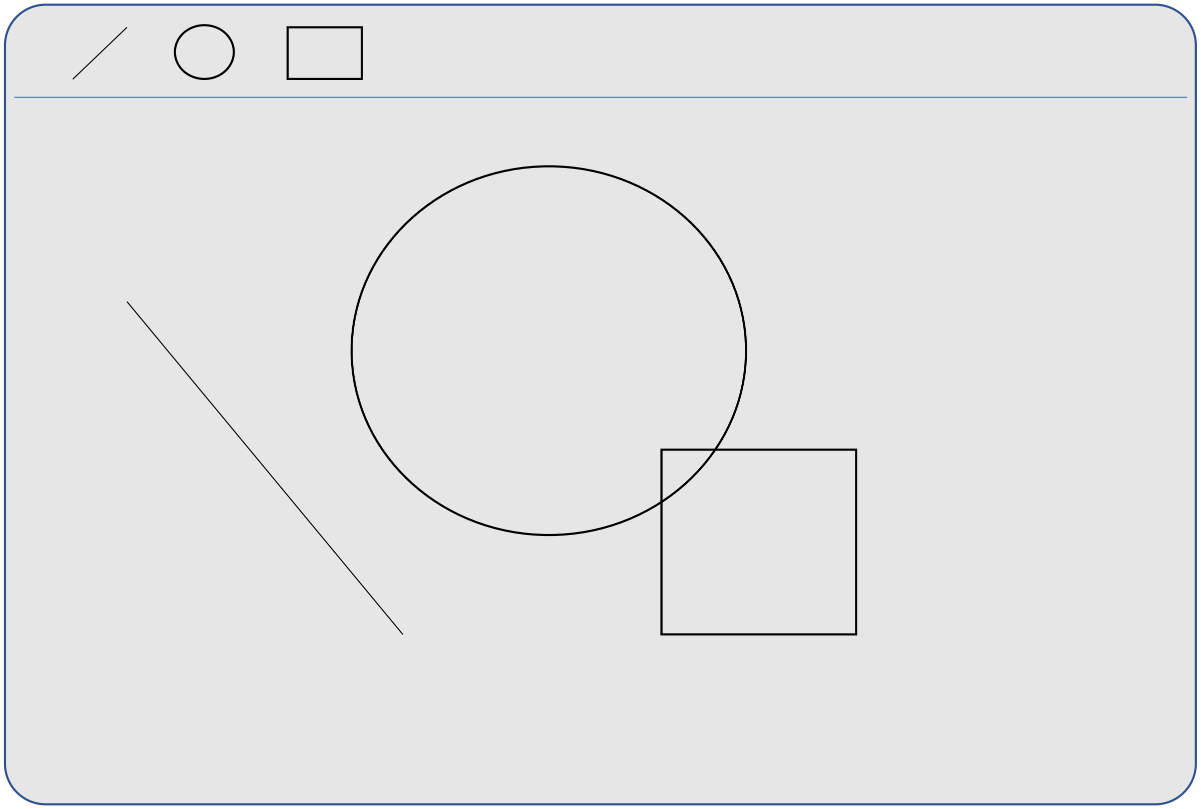
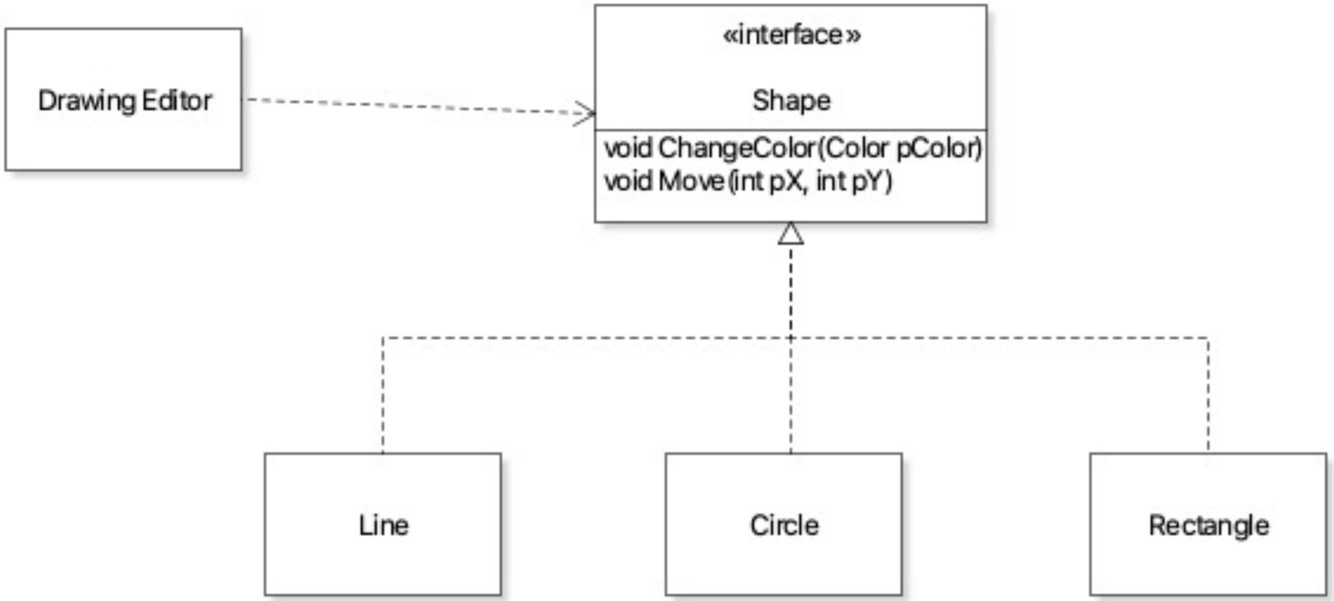
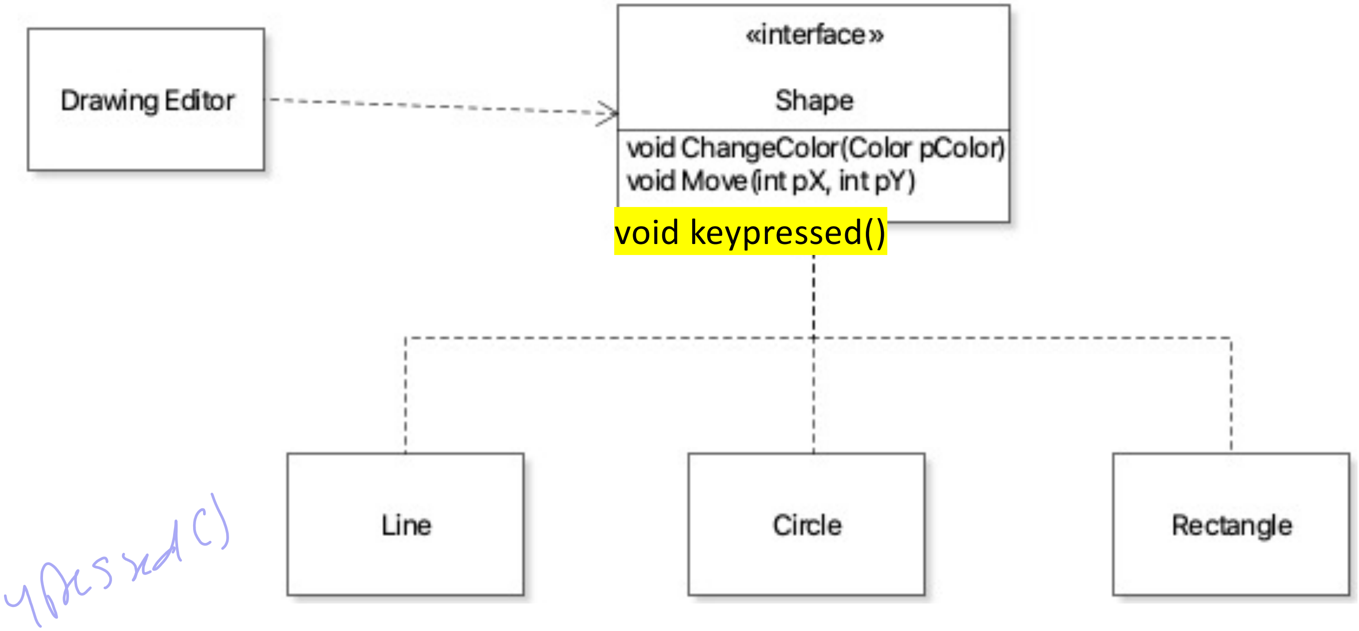


Image source: https://c1.staticflickr.com/4/3111/3145776919_74f05d63fd_b.jpg

Ideas?


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

Solution 1

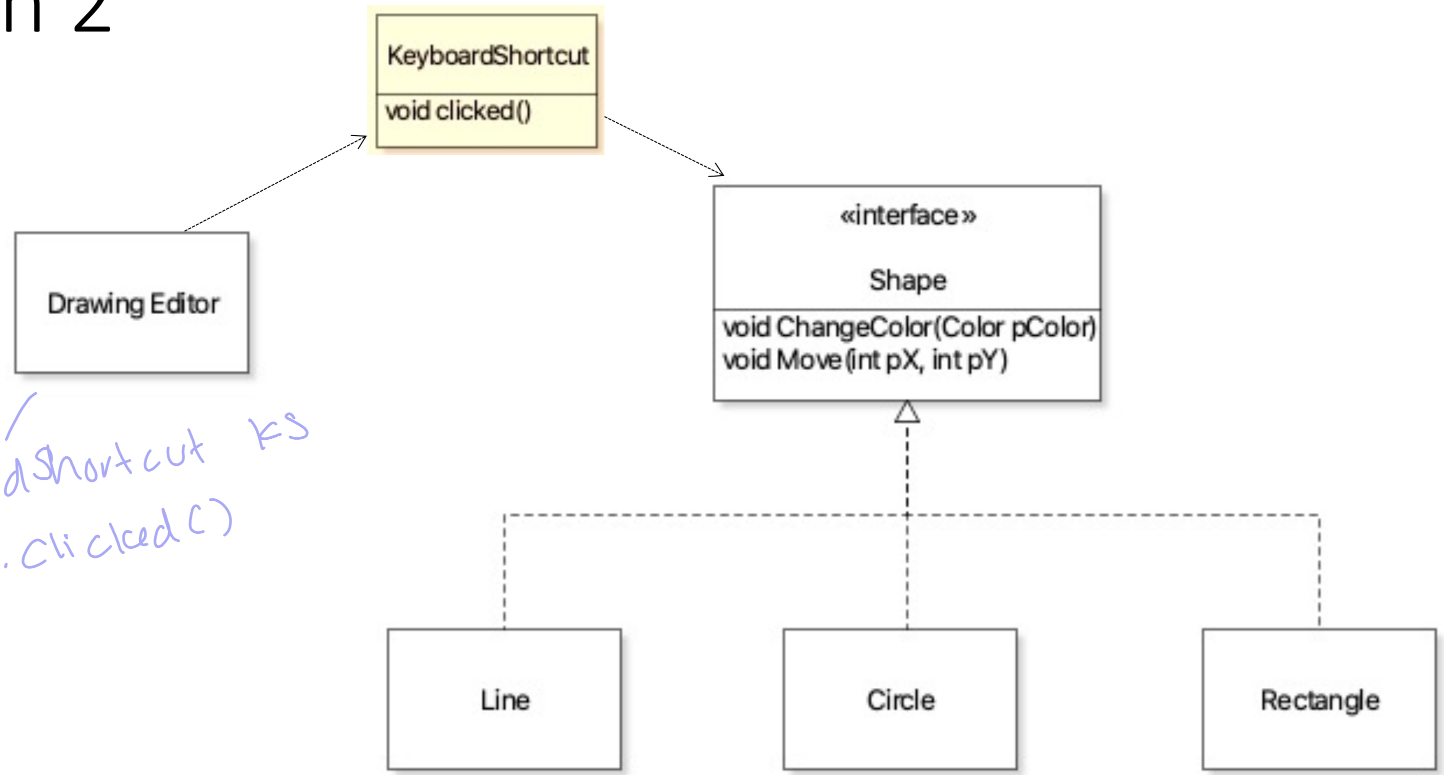


Shape s.keypressed()

create a new interface

encapsulates invocation

Solution 2



KeyboardShortcut

void clicked()

«interface»

Shape

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

Drawing Editor

Line

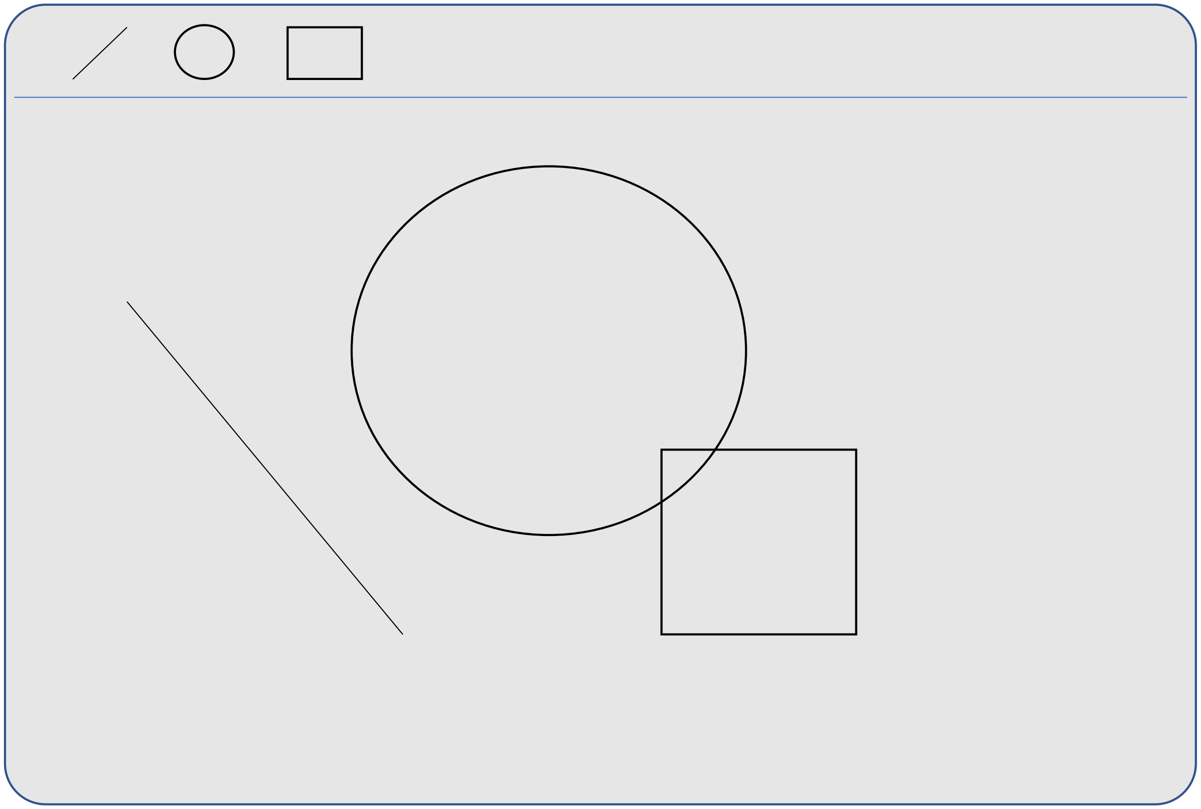
Circle

Rectangle

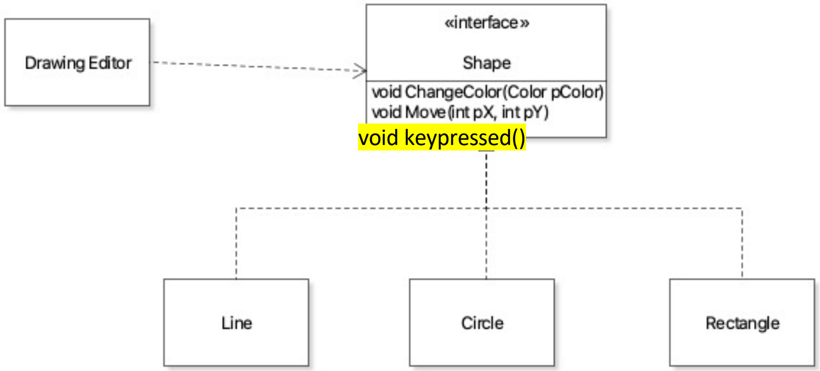
KeyboardShortcut ks
ks.clicked()

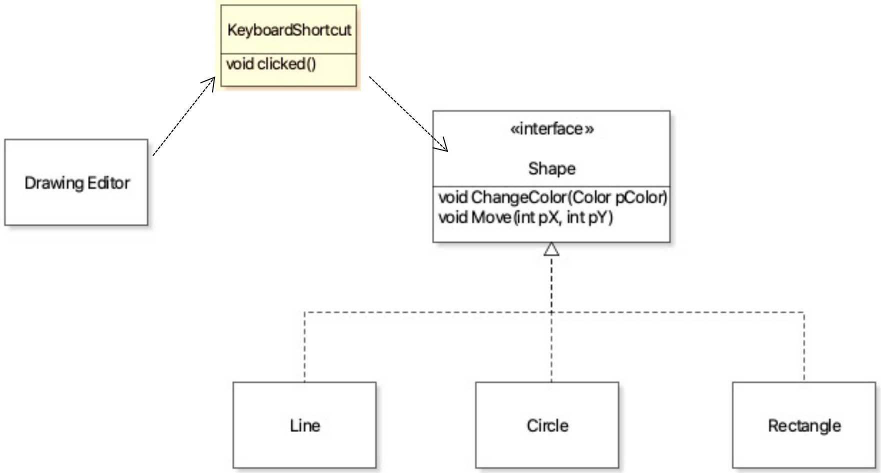
Design Problem

Support different shortcut for different behaviors, and reconfigurable.



Compare previous designs

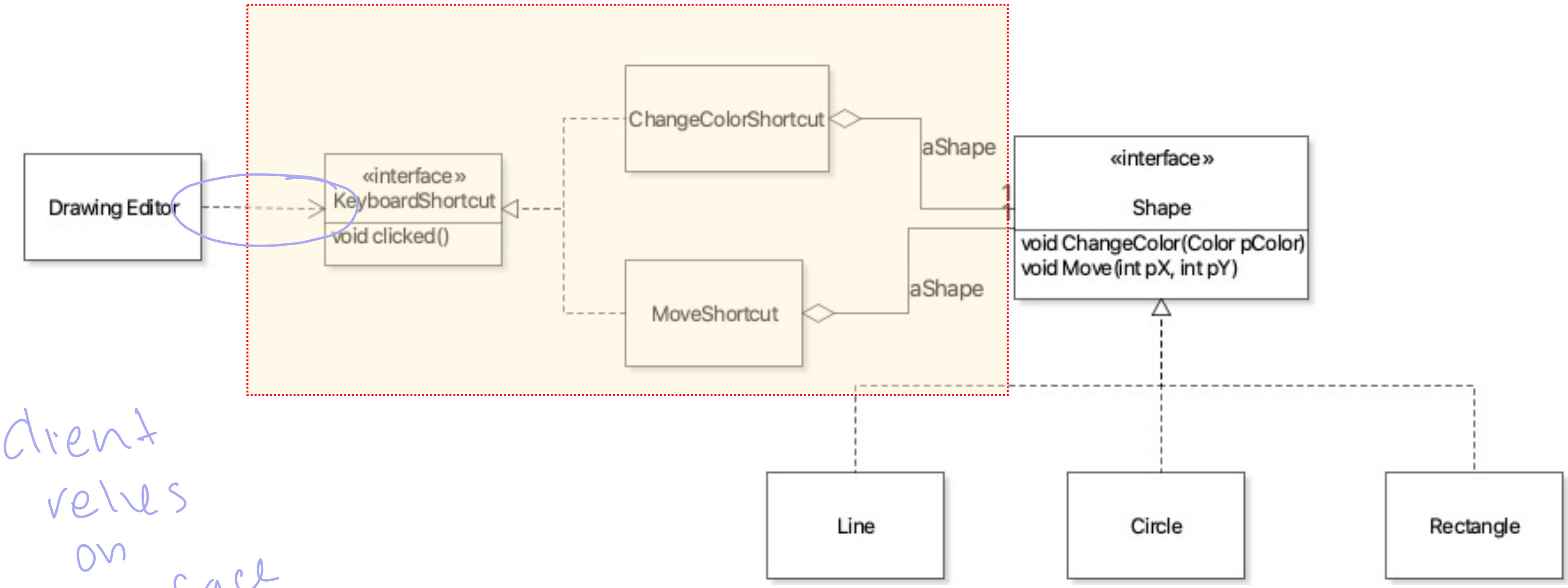




need to add more methods
to accomodate new behaviurs

need to make configurable

Polymorphic shortcut behavior



client
relies
on
interface

```java
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);
    }
}
```

delegate call to object

```java
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);
    }
}
```

delegate

```
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);
```

```java
public class DrawingEditor {
    KeyboardShortcut aShortcut;

        void setKeyboardShortcut(KeyboardShortcut pShortcut){
            aShortcut = pShortcut;
        }

        void respondToShortcut(){
            aShortcut.clicked();
        }

}
```

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

Client code:

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

using polymorphism to encapsulate an invocation of a method call

Command

- Intent:
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

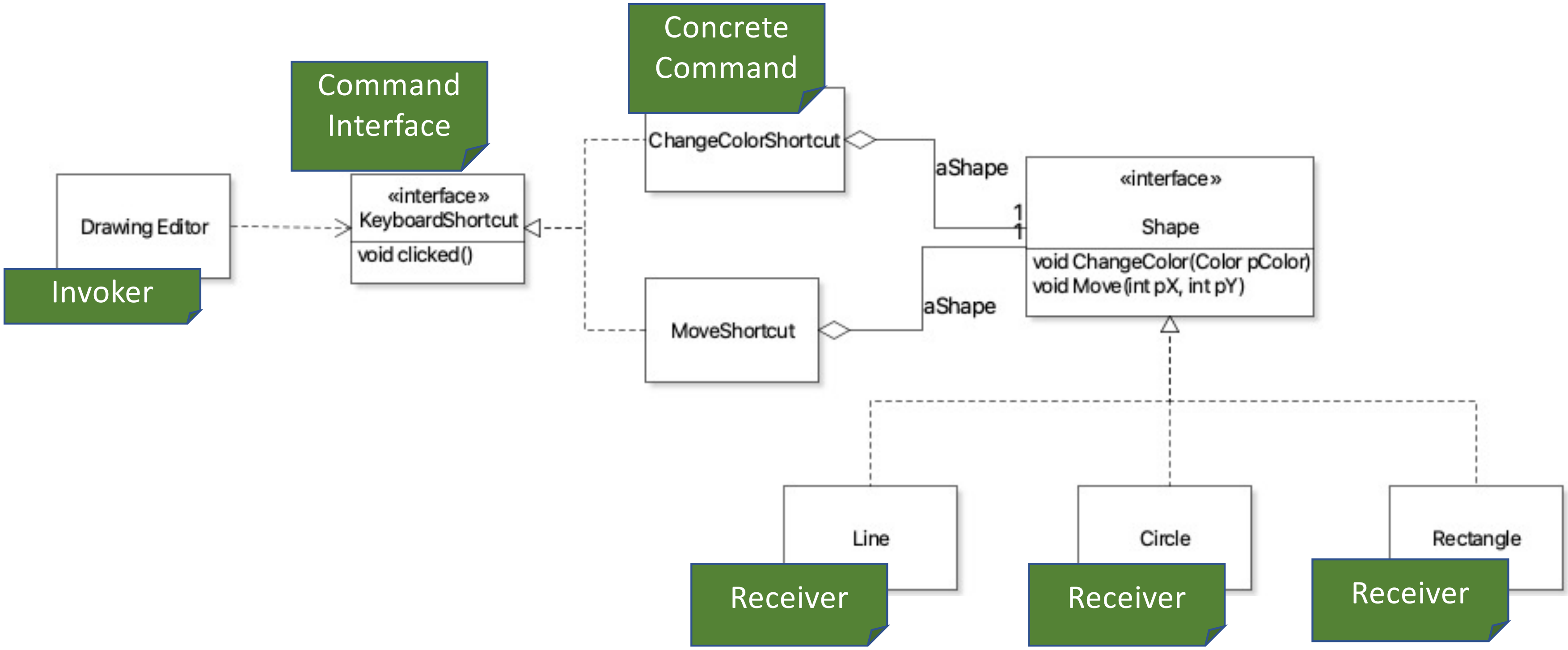
- Participants:
  - Command

    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
Interface
Concrete
Command
Drawing Editor
Invoker
«interface»
KeyboardShortcut
void clicked()
ChangeColorShortcut
MoveShortcut
aShape
aShape
1
1
«interface»
Shape
void ChangeColor(Color pColor)
void Move(int pX, int pY)
Line
Circle
Rectangle
Receiver
Receiver
Receiver

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

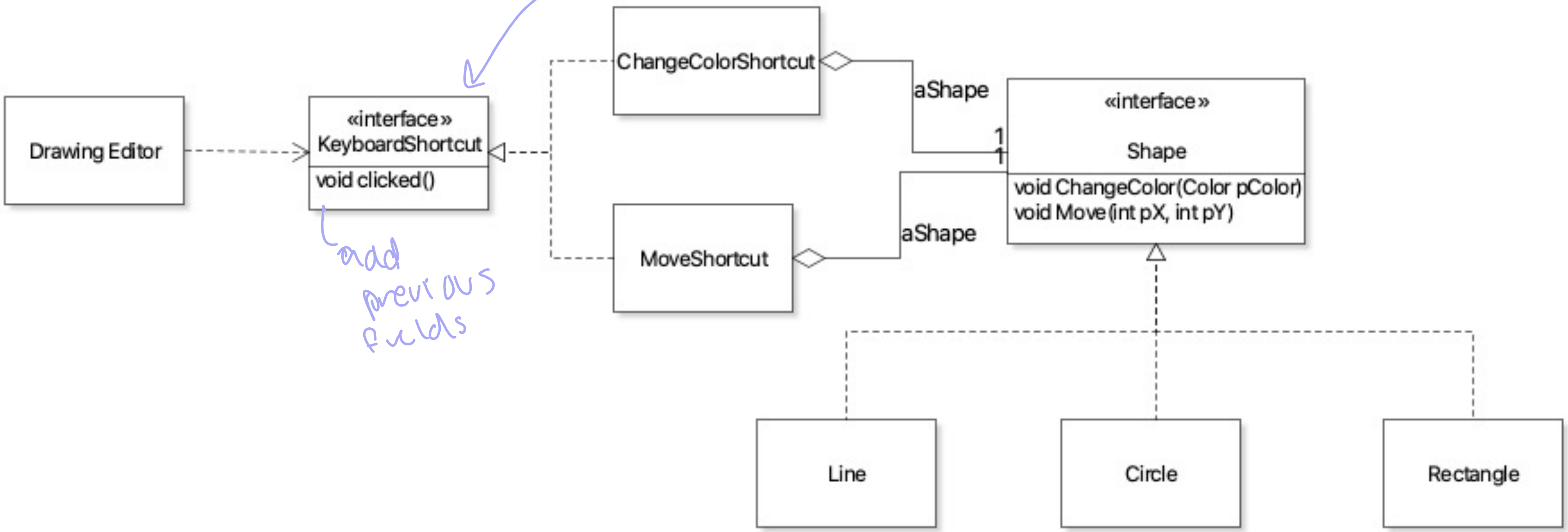
    declares an interface for executing an operation.
  - ConcreteCommand

    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.

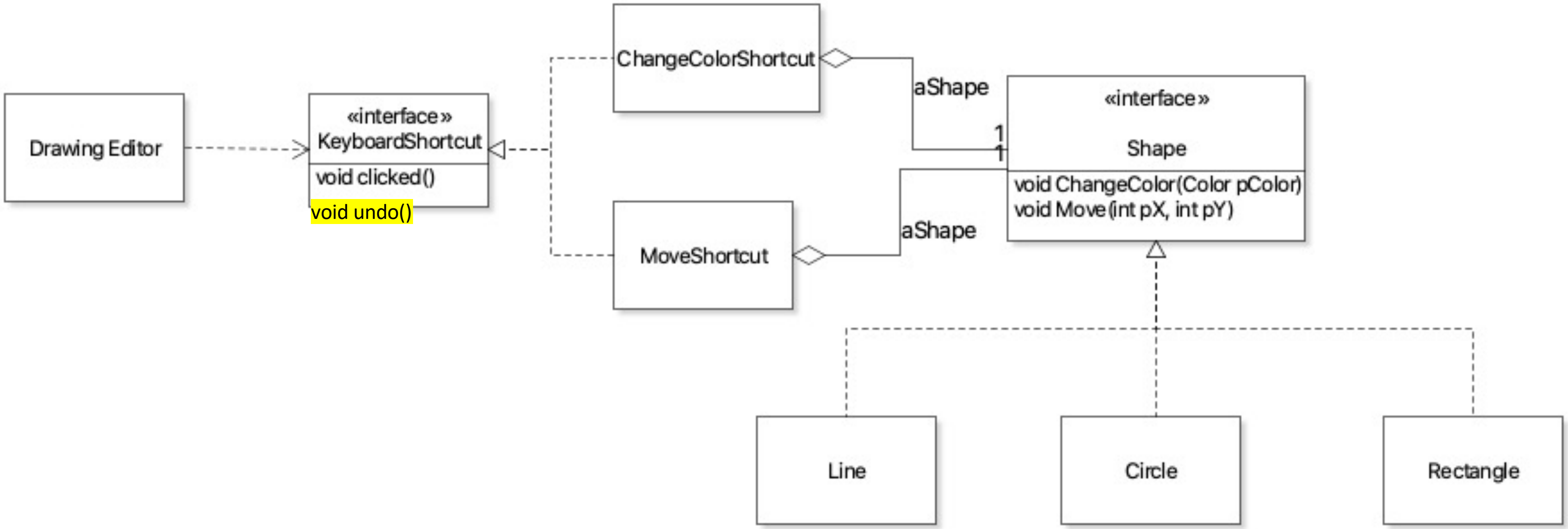
Activity1: How to support undo function



undo – reverts to previous

add previous fields

How to support undo function?



```java
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?

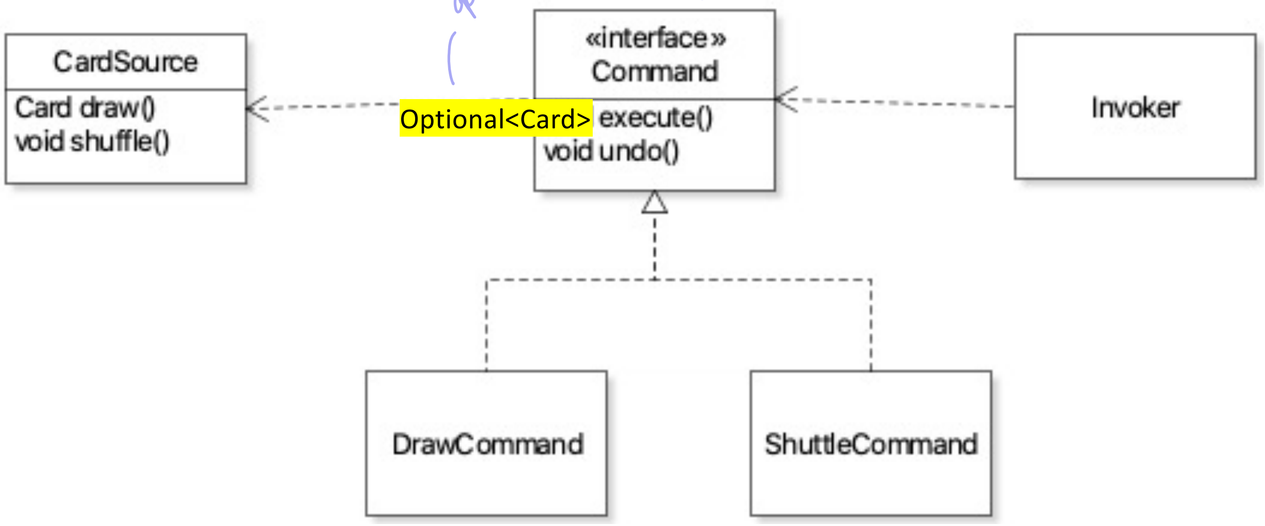
```java
public class Line implements Shape {
    private int x_start;
    private int y_start;
    private int x_end;
    private int y_end;

    public KeyboardShortcut getShortcut(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;
            }
        };
    }
}
```

Saving previous internal fields

What if some functions has return value?



optional return type

↳ draw() needs it

↳ shuffle() doesn't

```java
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();


}
```

```java
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 ? use optional ?

- 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 ✓	Momento
		Façade	Observer ✓
		Proxy	State
			Strategy ✓
			Visitor ✓

Objective

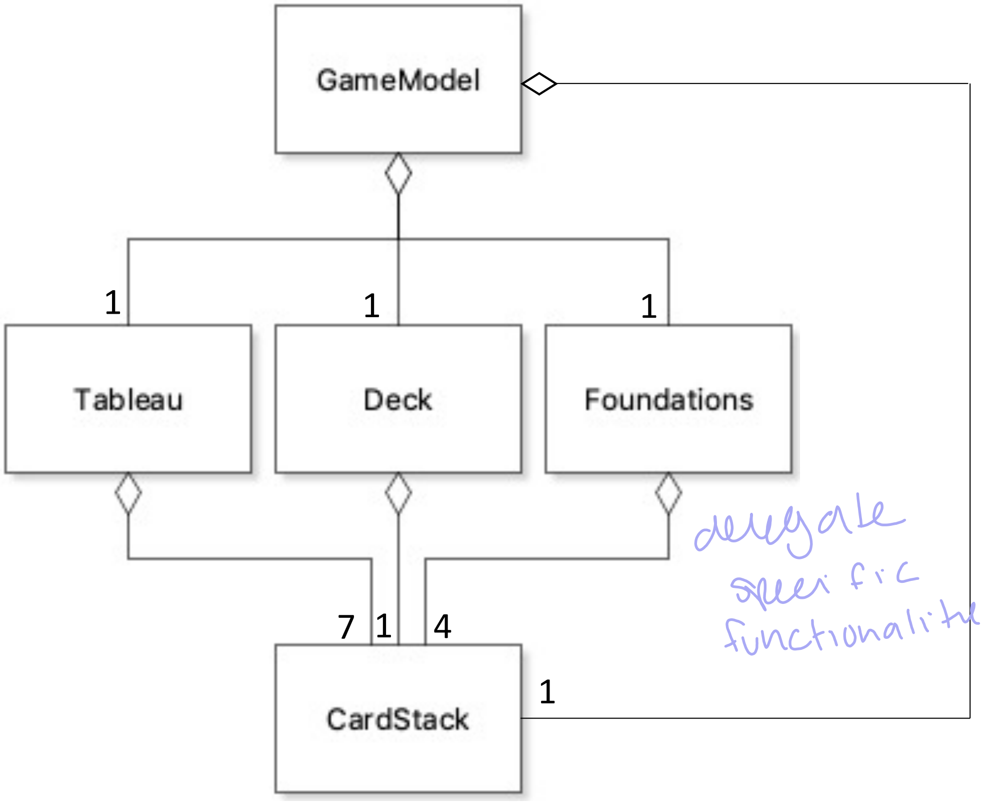
- Design Principle:

Law of Demeter ←

- Patterns and Anti-patterns:

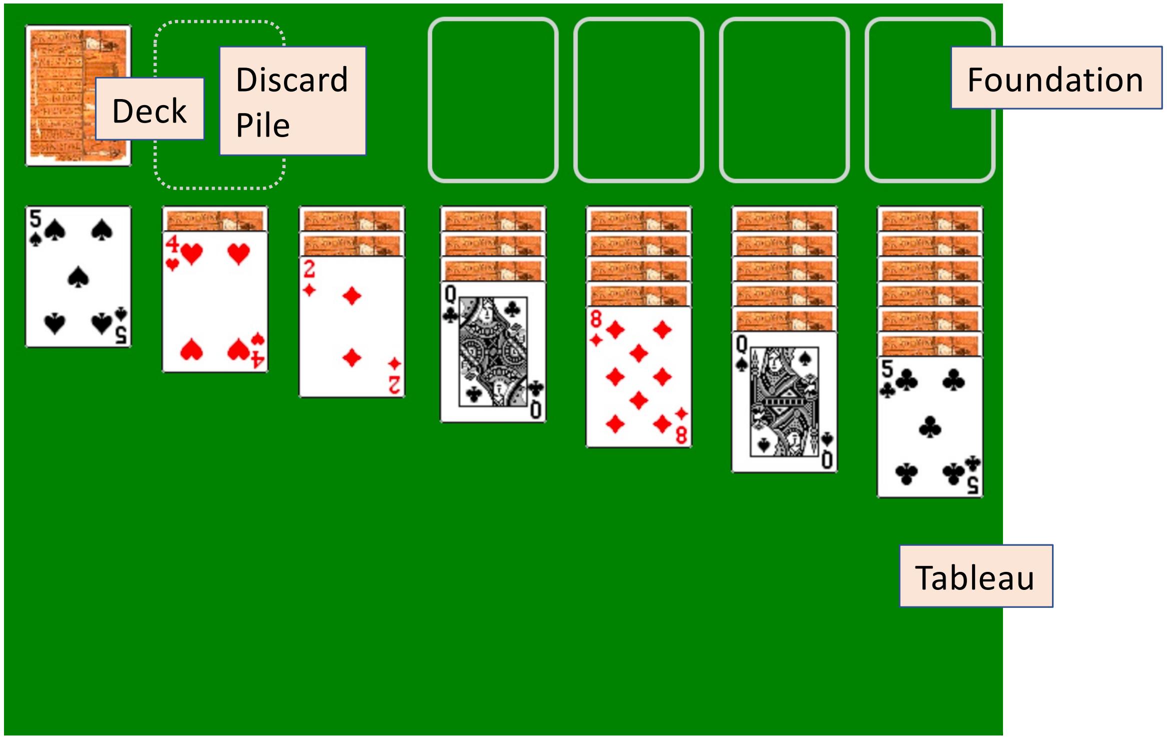
Command Pattern

Delegation Chains



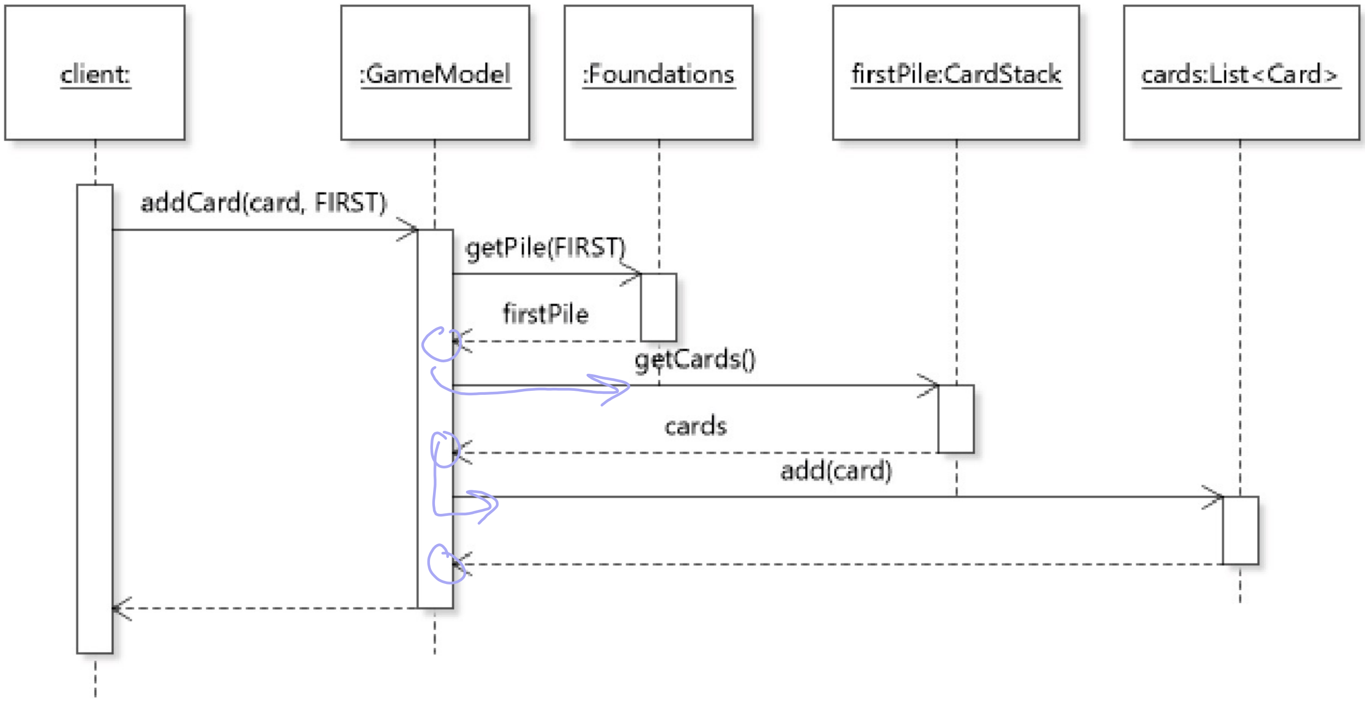
delegate
specific
functionalities

Scenario of adding a card to the first Foundation pile.



Sequence diagram

One option



Law of Demeter

“Only talk to your friends”

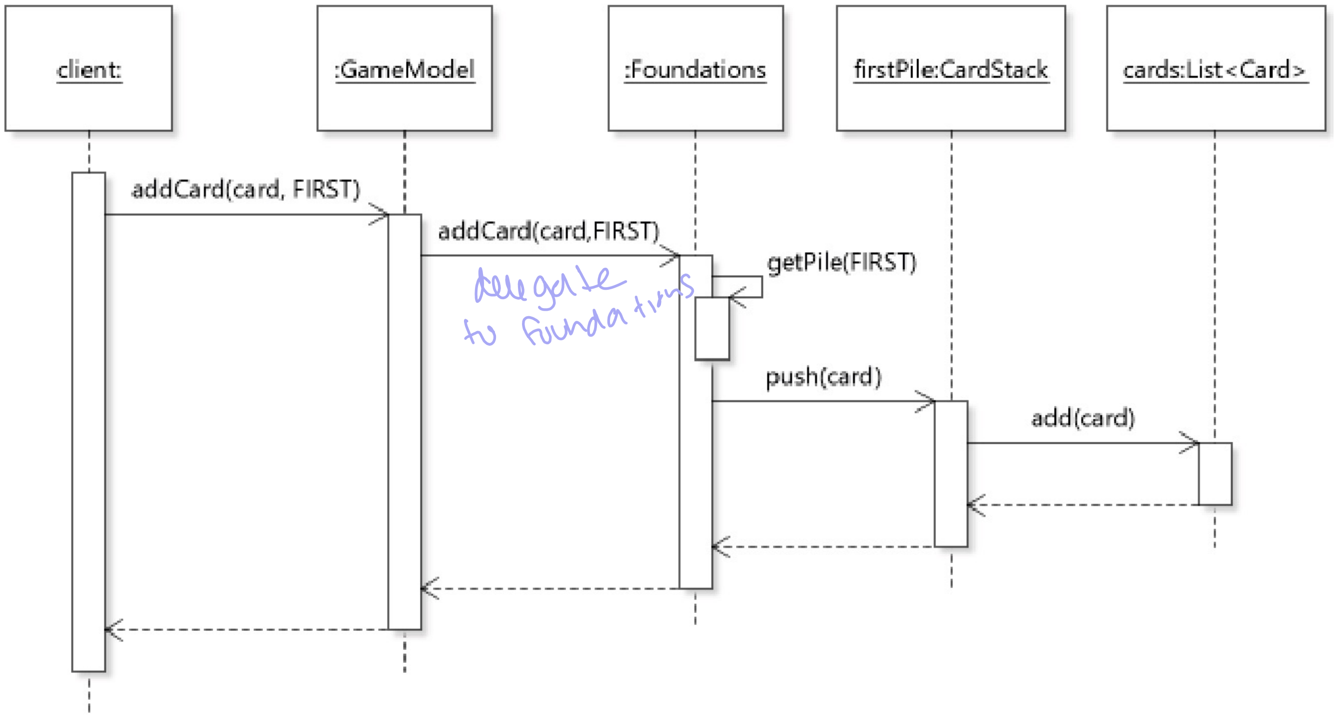
The code of a method should only access:

- 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.

rules out things in previous slide

(return values that don't belong to these categories)

Following the Law of Demeter



delegate to foundations

each object only interacts with a limited number of objects

Activity 2:

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

```java
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);
   }
}
```

instance variables

follows law of demeter ✓

✗

argument ✓

red flag - violates law

instance variables ✓

if compute is changed a bunch of other stuff will need to be changed too

creates dependences

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.

```java
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);
   }
}
```

pInventory.print(aIngredients.elements())

delegates

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.