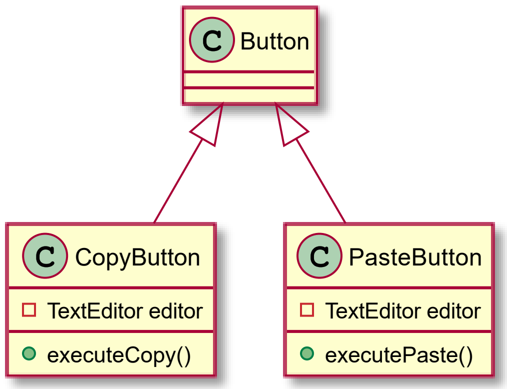
第2章 命令模式

提出问题

> 设计一个文档编辑器以及相应的工具按钮，例如单击一个复制按钮实现编辑器内容的复制。这里针对该请求设计一个专属按钮如下：

```
class CopyButton extends Button{
    private TextEditor editor;
    public void executeCopy(){
        editor.copy();
    }
}
class TextEditor{
    String content;
    public void copy(){
        System.out.println("Copy the content");
    }
}
```

当需要增加功能时，例如增加内容的粘贴功能，则需要再设计一个专属按钮：



以此类推，更多的交互功能则需要设计更多的专属按钮类，可能导致按钮类修改和维护困难。另外，如果存在多种交互形式触发相同的命令呢，例如右键弹出的菜单项也能实现复制功能，则有请求触发的代码需要重复编写：

```
class CopyMenuItem extends MenuItem{
    private TextEditor editor;
    public void executeCopy(){
        editor.copy();
    }
}
```

造成上述例子问题的主要原因在于调用者类和接收者类之间的耦合关系较强，需要思考如何解耦。

模式名称

Command或Action, Transaction

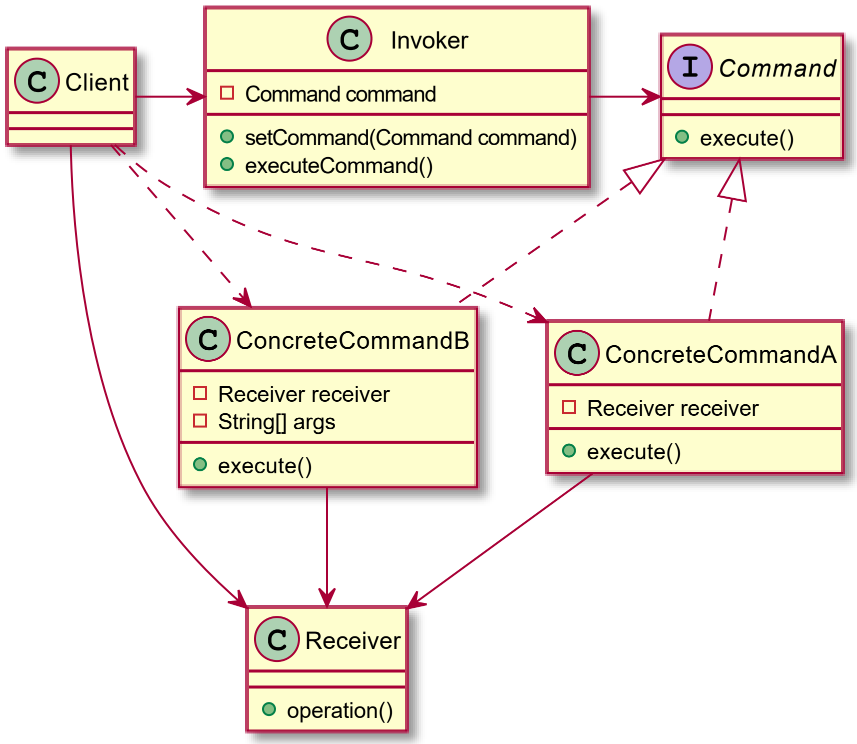
设计意图

命令模式（Command）将请求封装成一个对象，将客户端对命令的调用参数化，实现调用者和接收者之间的动态绑定，将其应用于命令执行的排队、日志记录和撤销等操作。

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

设计结构

类图



参与者

调用者（Invoker）：接收客户端命令，并执行命令
接收者（Receiver）：负责具体实施或执行一个请求
命令（Command）：定义需要执行的所有命令行为
具体命令（ConcreteCommand）：指定一个命令接收者，再执行方法中调用接收者提供的操作。

实现代码

设计命令接口以及具体命令：

```java
// 命令对象
public interface Command {
    public void execute();
}
public class CommandA implements Command {
    Receiver r;

    @Override
    public void execute() {
        System.out.println("2.调用接收者处理请求");
        r.operation();
    }
    public CommandA(Receiver r) {
        super();
        this.r = r;
    }
}
```

设计命令接收者

```java
// 命令接收者
public class Receiver {
    public void operation() {
        System.out.println("3.接收者处理请求");
    }
}
```

设计命令调用者：

```java
public class Invoker {
    private Command command;

    public void executeCommand() {
        System.out.println("1.调用命令对象处理请求");
        command.execute();
    }

    public Command getCommand() {
        return command;
    }

    public void setCommand(Command command) {
        this.command = command;
    }
}
```

设计客户端：

```java
public class Client {
    public static void main(String[] args) {
        Invoker inv = new Invoker();
        Command command = new CommandA(new Receiver());
        inv.setCommand(command);
        inv.executeCommand();
    }
}
```

运行结果：

```
1.调用命令对象处理请求
2.调用接收者处理请求
3.接收者处理请求
```

解决问题

设计一个命令接口，定义命令执行操作：

```java
public interface Command {
    public void execute();
}
```

设计具体的操作接口，这里涉及到复制和粘贴两个操作：

```java
public interface ICopy {
    public void copy();
}
public interface IPaste {
    public void paste();
}
```

设计具体命令类，这里包括复制命令和粘贴命令：

```java
public class CopyCommand implements Command {
    ICopy receiver;
    @Override
    public void execute() {
        receiver.copy();
    }
    public CopyCommand(ICopy receiver) {
        super();
        this.receiver = receiver;
    }
}
public class PasteCommand implements Command {
    IPaste receiver;
    @Override
    public void execute() {
        receiver.paste();
    }
    public PasteCommand(IPaste receiver) {
        super();
        this.receiver = receiver;
    }
}
```

设计调用者类，这里包括按钮和菜单项两种调用者：

客户端类：

```
public class Button {
    public Button(String name) {
        super();
        this.name = name;
    }
    private String name;
    private Command command;
    public void click() {
        command.execute();
    }
}
public class MenuItem {
    public MenuItem(String name) {
        super();
        this.name = name;
    }
    private String name;
    private Command command;
    public void click() {
        command.execute();
    }
}
```

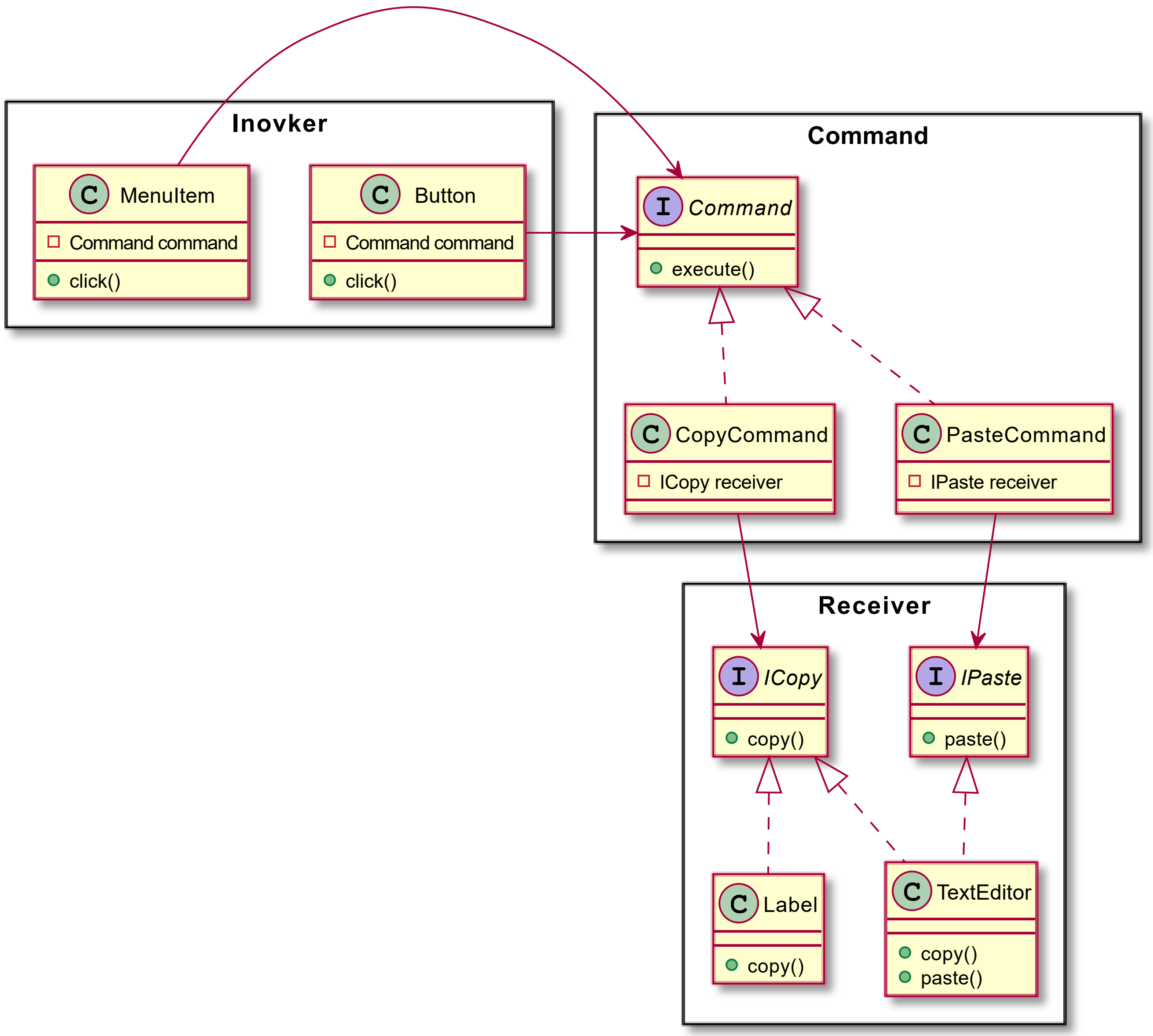
设计接收者类，这里包括编辑器和标签：

```
public class Label implements ICopy{
    @Override
    public void copy() {
        System.out.println("复制标签的内容");
    }
}
public class TextEditor implements ICopy, IPaste{
    @Override
    public void paste() {
        System.out.println("粘贴内容到编辑器");
    }
    @Override
    public void copy() {
        System.out.println("复制编辑器的内容");
    }
}
```

客户端类：

```java
public class Client {
    public static void main(String[] args) {
        Button editorCopyBtn = new Button("编辑器复制按钮");
        Button editorPasteBtn = new Button("编辑器粘贴按钮");
        Button labelCopyBtn = new Button("标签复制按钮");
        MenuItem editorCopyItem = new MenuItem("编辑器复制菜单项");
        MenuItem editorPasteItem = new MenuItem("编辑器粘贴菜单项");
        MenuItem labelCopyItem = new MenuItem("标签复制菜单项");

        Label label = new Label();
        TextEditor editor = new TextEditor();

        Command labelCopyCommand = new CopyCommand(label);
        Command editorCopyCommand = new CopyCommand(editor);
        Command editorPasteCommand = new PasteCommand(editor);

        editorCopyBtn.setCommand(editorCopyCommand);
        editorPasteBtn.setCommand(editorPasteCommand);
        labelCopyBtn.setCommand(labelCopyCommand);

        editorCopyItem.setCommand(editorCopyCommand);
        editorPasteItem.setCommand(editorPasteCommand);
        labelCopyItem.setCommand(labelCopyCommand);

        editorCopyBtn.click();
        editorPasteBtn.click();
        labelCopyBtn.click();

        editorCopyItem.click();
        editorPasteItem.click();
        labelCopyItem.click();
    }
}
```

类图如下：

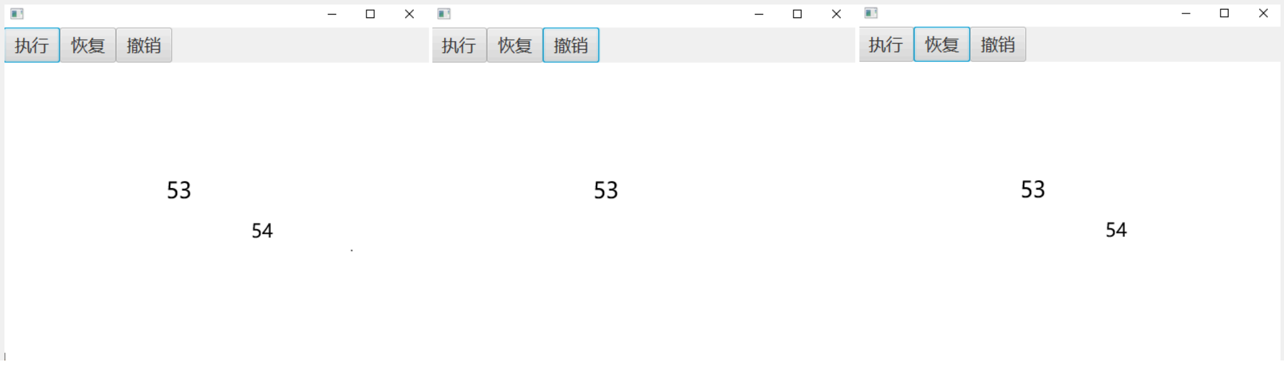


效果与适用性

扩展案例

命令的撤销与恢复

> 设计一个在画板上绘制随机数字符号的程序，包括执行、撤销和恢复三个功能，执行功能生成一个指定大小和位置的数字文本，撤销功能是将之前绘制的符号撤销一个，恢复功能是将撤销的最后一个符号恢复。例如第一步单击执行按钮两次生成53、54两个数字，第二

> 步单击撤销按钮，将最后绘制的数字54删除；第三步再单击恢复按钮，将删除的数字54恢复：
>
> 
>

这里的三个按钮并不是对应三个命令，而是一个命令的三种操作。先设计支持以上三种操作的命令接口：

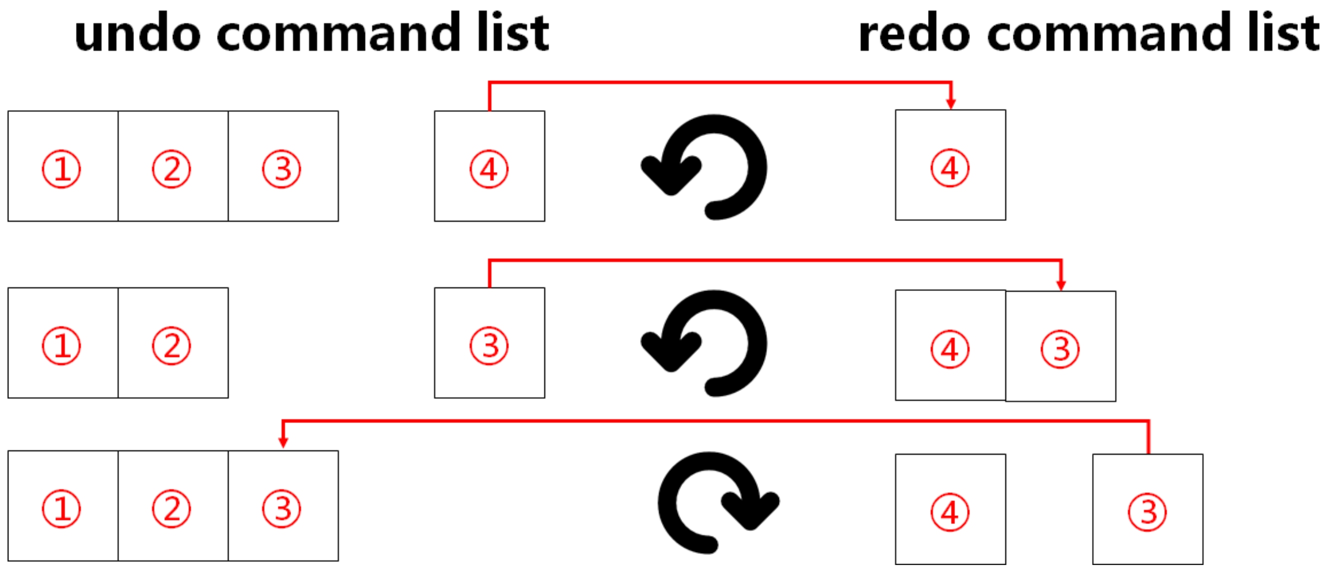
```
public interface Command {
    // 这里恢复操作实际就是再执行一遍，因此只需要定义两种操作
    public void execute();
    public void undo();
}
```

具体命令需要命令接收者和执行命令需要的参数（或数据），这里接收者为一个渲染器（`Render`），参数为一个文本对象（`Text`）。渲染器中维护着一个文本列表，每次渲染都会将所有文本重新绘制一遍。文本对象中包含文本内容、大小和位置信息。

```java
public class DrawTextCommand implements Command{
    Text text;
    Render render;
    @Override
    public void execute() {
        render.add(text);
    }
    @Override
    public void undo() {
        render.remove(text);
    }
}
public class Text {
    private String text;
    private Point2D location;
    private int size;
}
public class Render {
    private Canvas canvas;
    private List<Text> list;
    public Render(Canvas canvas) {
        this.canvas = canvas;
        list = new ArrayList<Text>();
    }
    public void add(Text t) {
        list.add(t);
    }
    public void remove(Text t) {
        list.remove(t);
    }
    public void rendering() {
        GraphicsContext gs = canvas.getGraphicsContext2D();
        gs.setFill(Color.WHITE);
        gs.fillRect(0, 0, canvas.getWidth(), canvas.getHeight());
        gs.setFill(Color.BLACK);
        for(Text t:list) {
            gs.setFont(new Font(t.getSize()));
            gs.fillText(t.getText(), t.getLocation().getX(), t.getLocation().getY());
        }
    }
}
```

调用者的设计需要考虑执行、撤销、恢复三个功能以及功能之间的配合。这里需要维护已执行命令和已经撤销命令，这里采用堆栈式管理。如下图所示，当执行命令时，命令入已执行栈（`stack`），当撤销命令时，命令调用`undo()`操作，并入已撤销栈（`undoStack`），当恢复命

令时，命令再次执行，并入已执行栈。

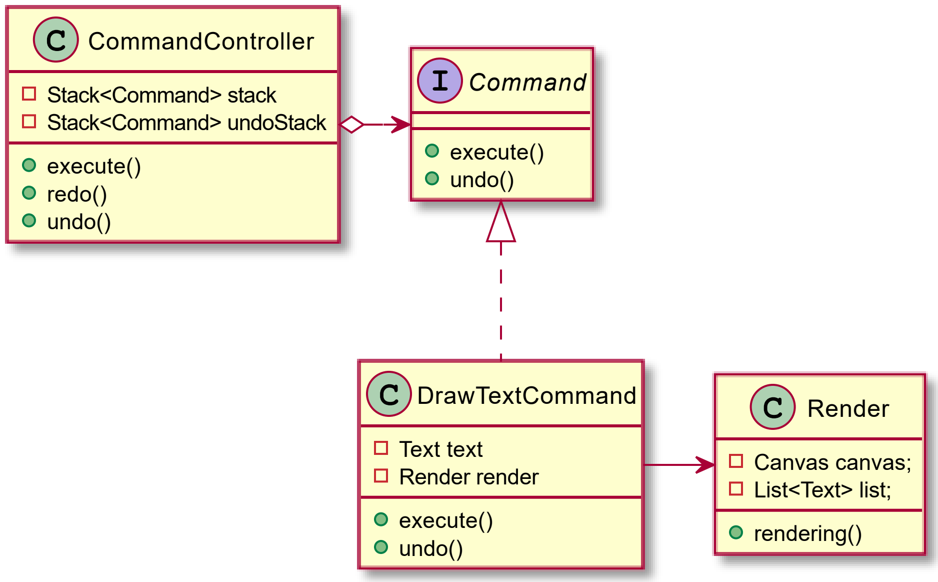


```java
public class CommandController {
    private Stack<Command> stack;
    private Stack<Command> undoStack;
    public CommandController() {
        stack = new Stack<Command>();
        undoStack = new Stack<Command>();
    }
    // 第一次执行命令
    public void execute(Command command) {
        command.execute();
        stack.push(command);
        undoStack.clear();
    }
    // 恢复并再次执行命令
    public void redo() {
        if(!undoStack.empty()) {
            Command c = undoStack.pop();
            c.execute();
            stack.push(c);
        }
    }
    // 撤销命令
    public void undo() {
        if(!stack.empty()) {
            Command c = stack.pop();
            c.undo();
            undoStack.push(c);
        }
    }
}
```

设计界面程序，单击执行按钮时创建新命令，调用者执行命令，单击撤销命令时调用者执行撤销操作，单击恢复命令时调用者执行恢复操作：

```java
public class Client extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        Canvas canvas = new Canvas(500, 400);
        Render render = new Render(canvas);
        CommandController controller = new CommandController();

        // 布局略

        exeBtn.setOnAction(e -> {
            Text t = new Text();
            t.setSize(10 + (int)(20 * Math.random()));
            t.setText(String.valueOf((int)(Math.random() * 100)));
            Point2D p = new Point2D(Math.random()*canvas.getWidth(),
                                    Math.random()*canvas.getHeight());
            t.setLocation(p);
            Command c = new DrawTextCommand(t, render);
            controller.execute(c);
            render.rendering();
        });

        redoBtn.setOnAction(e -> {
            controller.redo();
            render.rendering();
        });

        undoBtn.setOnAction(e -> {
            controller.undo();
            render.rendering();
        });

        Scene scene = new Scene(layout);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

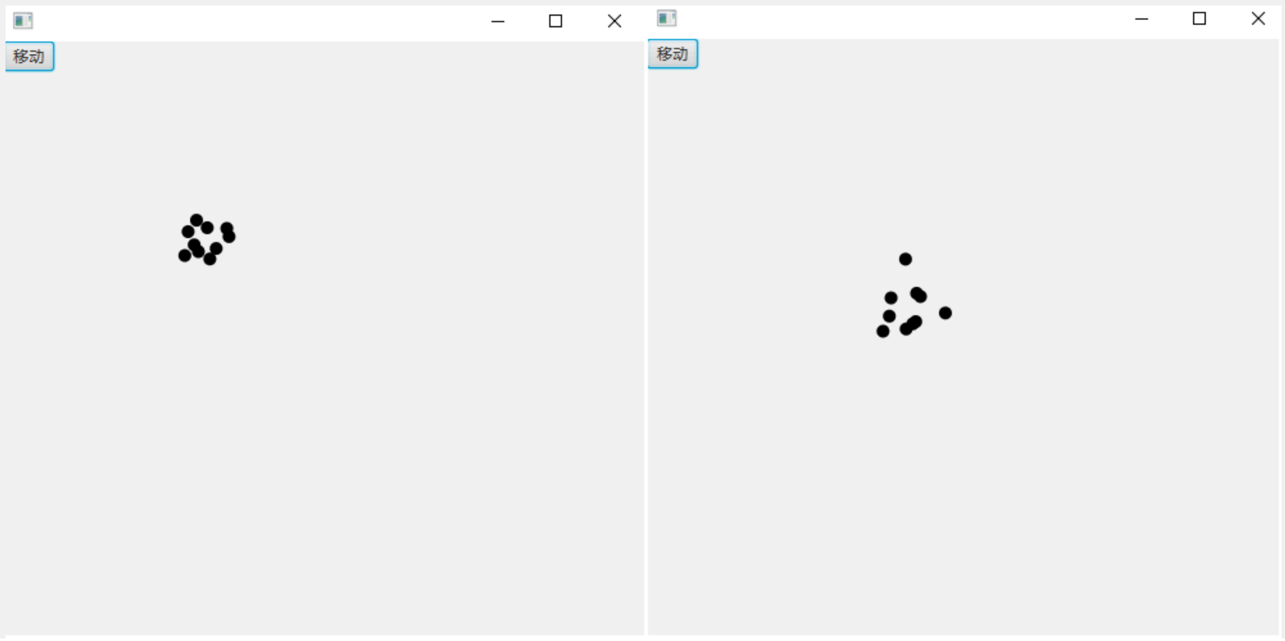
类图：



多目标控制（宏命令）

宏（Macro）是一种批量处理的称谓，宏命令由多个命令组合。

> 设计一个程序通过按钮随机移动多个圆形。



设计命令接口：

```
public interface Command {
    public void execute();
}
```

具体的命令处理目标的移动，接收者为圆形对象。

```
public class MoveCommand implements Command{
    public MoveCommand(MoveCircle target) {
        super();
        this.target = target;
    }
    private MoveCircle target;
    @Override
    public void execute() {
        target.move();
    }
}
```

接收者需要实现移动功能：

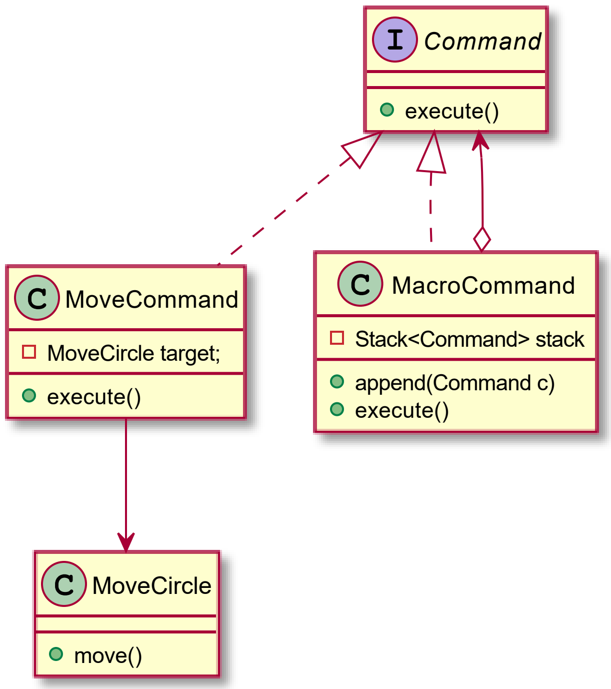
```
public class MoveCircle extends Circle{
    public void move() {
        double offx = 10 * Math.random();
        double offy = 10 * Math.random();
        this.setCenterX(this.getCenterX() + offx);
        this.setCenterY(this.getCenterY() + offy);
    }
}
```

宏命令实现单个命令的管理和批量执行：

```
public class MacroCommand implements Command{
    private Stack<Command> stack;
    public MacroCommand(){
        stack = new Stack<Command>();
    }
    @Override
    public void execute() {
        Iterator<Command> it = stack.iterator();
        while(it.hasNext()) {
            it.next().execute();
        }
    }
    public void append(Command comand) {
        stack.push(comand);
    }
}
```

客户端:

```
public class Client extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {

        MacroCommand macroCom = new MacroCommand();

        Pane pane = new Pane();
        Button btn = new Button("移动");
        for(int i = 0; i < 10; i++) {
            MoveCircle mc = new MoveCircle();
            mc.setCenterX(100);
            mc.setCenterY(100);
            mc.setRadius(5);
            macroCom.append(new MoveCommand(mc));
            pane.getChildren().add(mc);
        }
        pane.getChildren().add(btn);
        btn.setOnAction(e -> {
            macroCom.execute();
        });

        Scene scene = new Scene(pane, 500, 500);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

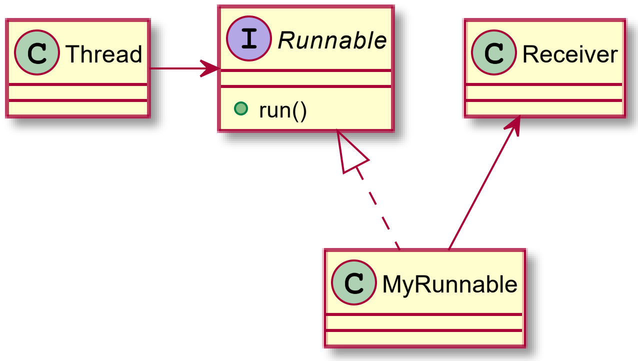


JDK中的命令模式

> JDK中的可运行（`Runnable`）接口类似于命令接口，而线程（`Tread`）对象充当调用者角色负责执行具体可运行对象的`run()`方法。例如：

```java
// 具体可运行对象
public class MyRunnable implements Runnable {
    public MyRunnable(Receiver receiver) {
        super();
        this.receiver = receiver;
    }
    // 接收者引用
    private Receiver receiver;

    @Override
    public void run() {
        receiver.hello();
    }

}
// 接收者
public class Receiver {
    public void hello() {
        System.out.println("你好");
    }
}
// 客户端
public class Client {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable(new Receiver()), "MyThread");
        thread.start();
    }
}
```

类图：


C Thread
I Runnable
run()
C Receiver
C MyRunnable