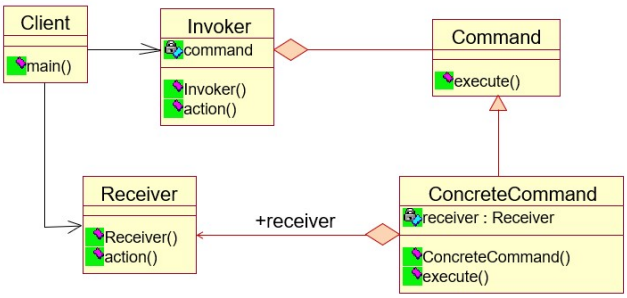
1 命令模式 (Command)

命令（Command）模式属于对象的行为模式。命令模式又称为行动（Action）模式或事务（Transaction）模式。命令模式是对命令的封装。命令模式把发出命令的责任和执行命令的责任分割开，委派给不同的对象。每一个命令都是一个操作：请求的一方发出请求要求执行一个操作；接收的一方收到请求，并执行操作。命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行，何时被执行，以及是怎么被执行的。

1.1 命令模式的结构

命令模式的类图: 下面以一个示意性的系统为例，说明命令模式的结构。



命令模式涉及到 5 个角色:

- 客户端（Client）角色：创建了一个具体命令（ConcreteCommand）对象并确定其接收者。
- 命令角色（Command）：声明了一个给所有具体命令类的抽象接口。
- 具体命令(ConcreteCommand)角色:定义一个接收者和行为之间的弱偶合;实现 execute() 方法，负责调用接收者的相应操作。execute() 方法通常叫做执行方法。
- 请求者（Invoker）角色：负责发出一个请求。
- 接收者 (Receiver) 角色: 任何一个类都可以成为接收者; 实施和执行请求的方法叫做行动方法。

```
1 public class Invoker {
2   private Command command;
3   public Invoker(Command command) { this.command=command; }
4   public void action() { /* 行动方法 */
5     command.execute();
6   }
7 }
```

Receiver 类是命令的接收者，在命令的控制下执行行动方法。这里仅给出一个 action() 方法的示意性实现。代码如下：

```
1 public class Receiver {
```

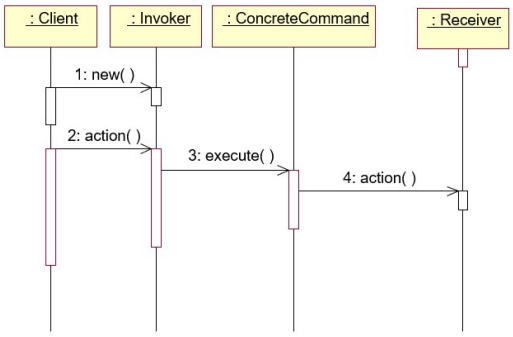
```
  public Receiver() { }
  public void action() {
    System.out.println("Action has been taken.");
  }
}
```

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

```
public class ConcreteCommand implements Command {
  private Receiver receiver;
  public ConcreteCommand(Receiver receiver) {
    this.receiver=receiver;
  }
  public void execute() { receiver.action(); }
}
```

```
public class Client {
  public static void main(String[] args) {
    Receiver receiver=new Receiver();
    Command command=new ConcreteCommand(receiver);
    Invoker invoker=new Invoker(command);
    invoker.action();
  }
}
```

命令模式的活动序列:



1.2 命令模式的优劣

- 优点:
  1. 命令模式把请求一个操作的对象与知道怎么执行一个操作的对象分割开。
  2. 命令类与其他任何别的类一样，可以修改和扩展。

3. 可以把命令对象聚合在一起，合成为合成命令。比如上面的例子里所讨论的宏命令便是合成命令的例子。合成命令是合成模式的应用。
4. 由于加进新的具体命令类不影响其他的类，因此增加新的具体命令类很容易。

- 缺点：使用命令模式会导致某些系统有过多的具体命令类。某些系统可能需要几十个，几百个甚至几千个具体命令类，这会使命令模式在这样的系统里变得不实际。

1.3 例:AudioPlayer 系统

系统的描述: User 有一个录音机，该录音机有 play、rewind 和 stop 功能，录音机的键盘是请求者角色（Invoker），User 是客户端类，录音机是接收者角色。Command 类是抽象命令角色，PlayCommand、StopCommand 和 RewindCommand 是具体命令类。User 不需要知道 play、rewind 和 stop 功能是怎么执行的，这些命令执行的细节由键盘 Keypad 具体实施。User 只需要在键盘上按下相应的键就行了。

系统的角色:

- 客户端（Client）角色：User
- 请求者（Invoker）角色：Keypad
- 抽象命令（Command）角色：Command
- 具体命令角色：PlayCommand、RewindCommand、StopCommand
- 接收者角色: 录音机 AudioPlayer

请求者 Invoker 角色 Keypad 源代码:

```
1  public class Keypad {
2    private Command playCmd;
3    private Command rewindCmd;
4    private Command stopCmd;
5    public Keypad(Command play, Command stop, Command rewind) {
6      playCmd=play;
7      stopCmd=stop;
8      rewindCmd=rewind;
9    }
10   public void play() { // 具体action方法
11     playCmd.execute();
12   }
13   public void stop() { stopCmd. execute(); }
14   public void rewind() { rewindCmd. execute(); }
15 }
```

抽象命令角色 Command 源代码:

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

具体命令类 PlayCommand 源代码:

```
public class PlayCommand implements Command {
   private AudioPlayer myAudio;
   public PlayCommand(AudioPlayer a) { myAudio=a; }
   public void execute() { myAudio.play(); }
}
```

具体命令类 StopCommand 源代码:

```
public class StopCommand implements Command {
  private AudioPlayer myAudio;
  public StopCommand(AudioPlayer a) { myAudio=a; }
  public void execute() { myAudio.stop(); }
}
```

具体命令类 RewindCommand 源代码:

```
public class RewindCommand implements Command {
   private AudioPlayer myAudio;
   public RewindCommand(AudioPlayer a) { myAudio=a; }
   public void execute() { myAudio.rewind(); }
}
```

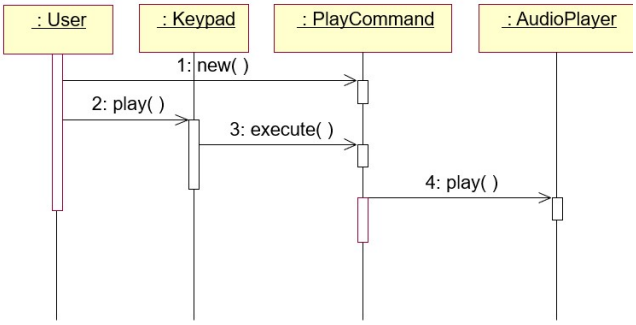
接收者类 AudioPlayer 源代码:

```
public class AudioPlayer {
  public void play() { System.out.println("Playing"); }
  public void stop() { System.out.println("Stopped"); }
  public void rewind() { System.out.println("Rewinding"); }
}
```

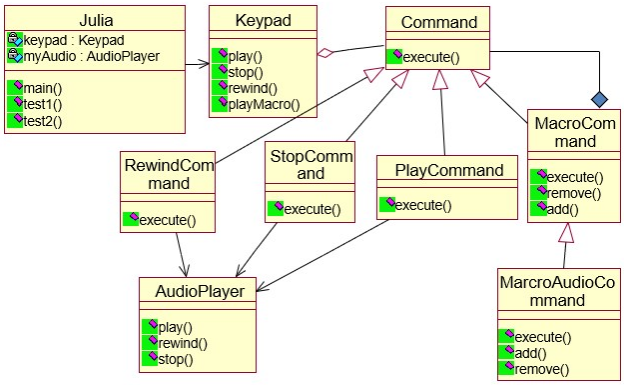
```
public class User {
  private static Keypad keypad;
  private static AudioPlayer myAudio = new AudioPlayer();
  public static void main(String[] args) { test1(); }
  private static void test1() {
    Command play=new PlayCommand(myAudio);
    Command stop=new StopCommand(myAudio);
    Command rewind=new RewindCommand(myAudio);
    keypad=new Keypad(play,stop,rewind);
    kepad.play();
    kepad.stop();
    kepad.rewind();
```

```
    kepad.stop();
    kepad.play();
    kepad.stop();
  }
}
```

系统的时序图如下：



增加宏命令功能：设想 User 的录音机有一个记录功能，可以把每一次执行的命令记录下来，再在任何需要的的时候重新把这些记录下来的命令一次性执行，这就是所谓的宏命令集功能。因此，User 的录音机系统现在有四个键，分别为播音，倒带，停止和宏命令功能。此时系统的设计与前面的设计相比有所增强，主要体现在 User 类现在有了一个新的方法，用以操作宏命令键。带有宏命令集功能的录音机模拟系统的设计图如下：




```
public interface MacroCommand extends Command {
  void execute(); /* 执行方法 */
  /* 宏命令聚集的管理方法，可以删除一个成员 */
  void remove(Command toRemove);
  /* 宏命令聚集的管理方法，可以添加一个成员 */
  void add(Command toAdd);
}
```

具体宏命令 MacroAudioCommand 类负责把个别的命令合成宏，代码如下：

```
import java.util.Vector;
public class MacroAudioCommand implements MacroCommand {
  private Vector commandList=new Vector();
  public void add(Command toAdd) { commandList.addElement(toAdd); }
  public void remove(Command toRemove) {
    commandList.removeElement(toRemove);
  }
  public void execute() {
    Command nextCommand;
    for (int i=0; i<commandList.size(); i++) {
      nextCommand = (Command)commandList.elementAt(i);
      nextCommand.execute();
    }
  }
}
```

Julia 类的方法 test2() 源代码如下：

```
public static void test2() {
  Command play=new PlayCommand(myAudio);
  Command stop=new PlayCommand(myAudio);
  Command rewind=new PlayCommand(myAudio);
  MacroCommand macro=new MacroCommand();
  macro.add(play);
  macro.add(stop);
  macro.add(rewind);
  macro.add(stop);
  macro.add(play);
  macro.add(stop);
  KeyPad keypad=new KeyPad(play,stop,rewind,macro);
  keypad.playMacro();
}
```