Design Patterns

宋 杰

Song Jie

东北大学 软件学院

Software College, Northeastern University

18. Command Pattern

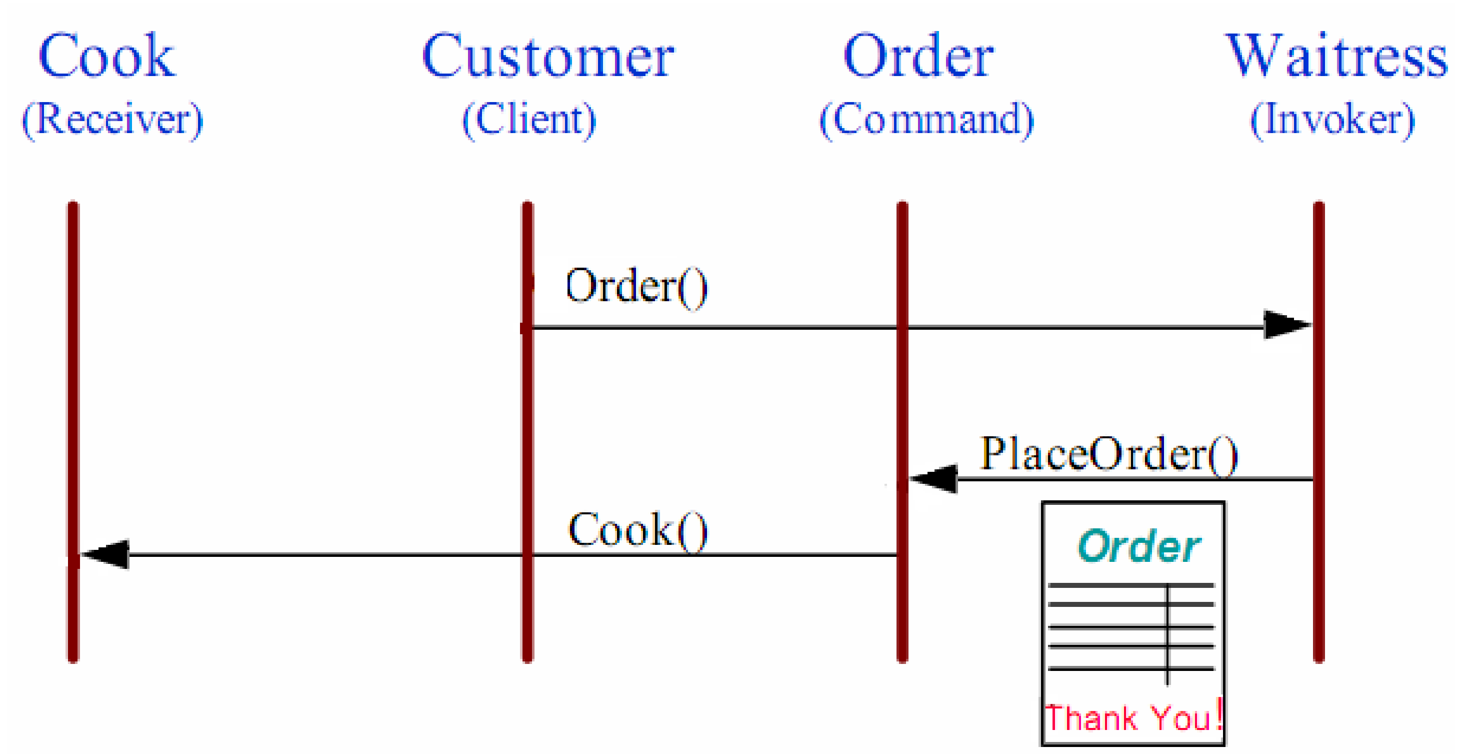
Intent

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable (redoable) operations.
- Action, Transaction
- 命令模式把一个请求封装到一个对象中。命令模式允许系统使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

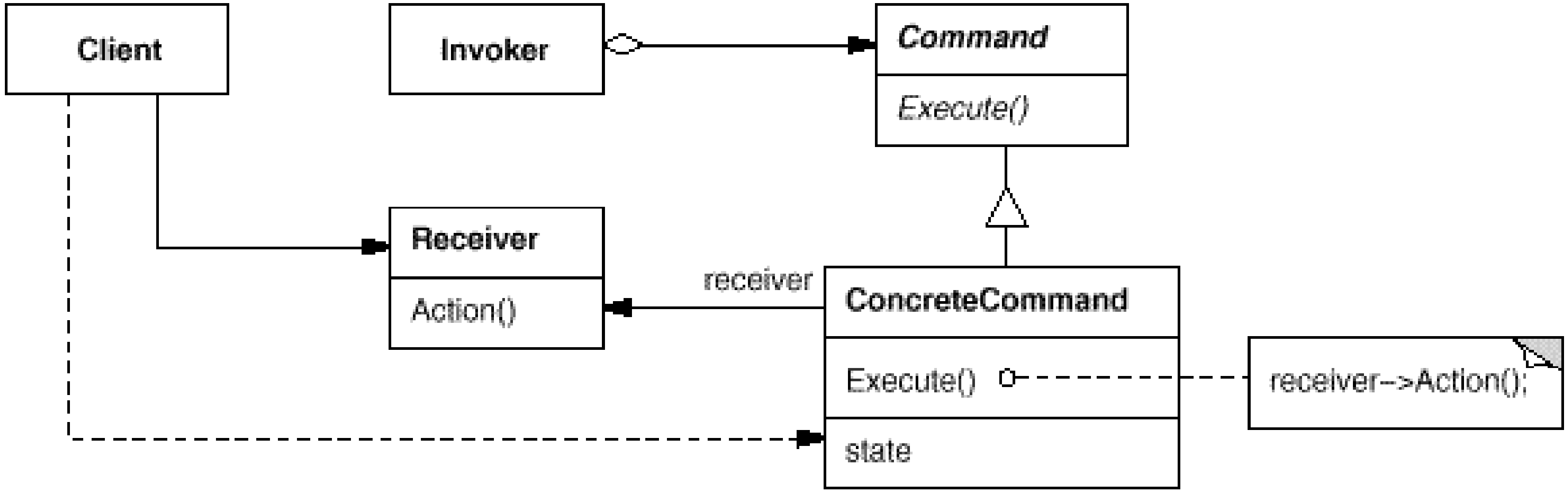
Intent

- Command pattern separate the responsibility of sending command and executing command, delegates command to different objects;
- Each command is an operation;
- Invoker send a command as an request of the operation;
- Receiver take a command and execute the operation;
- Invoker is separate from Receiver, and when, where, how the command is executed.

Example


Cook
(Receiver)
Customer
(Client)
Order
(Command)
Waitress
(Invoker)
Order()
PlaceOrder()
Cook()
Order
Thank You!

Structure

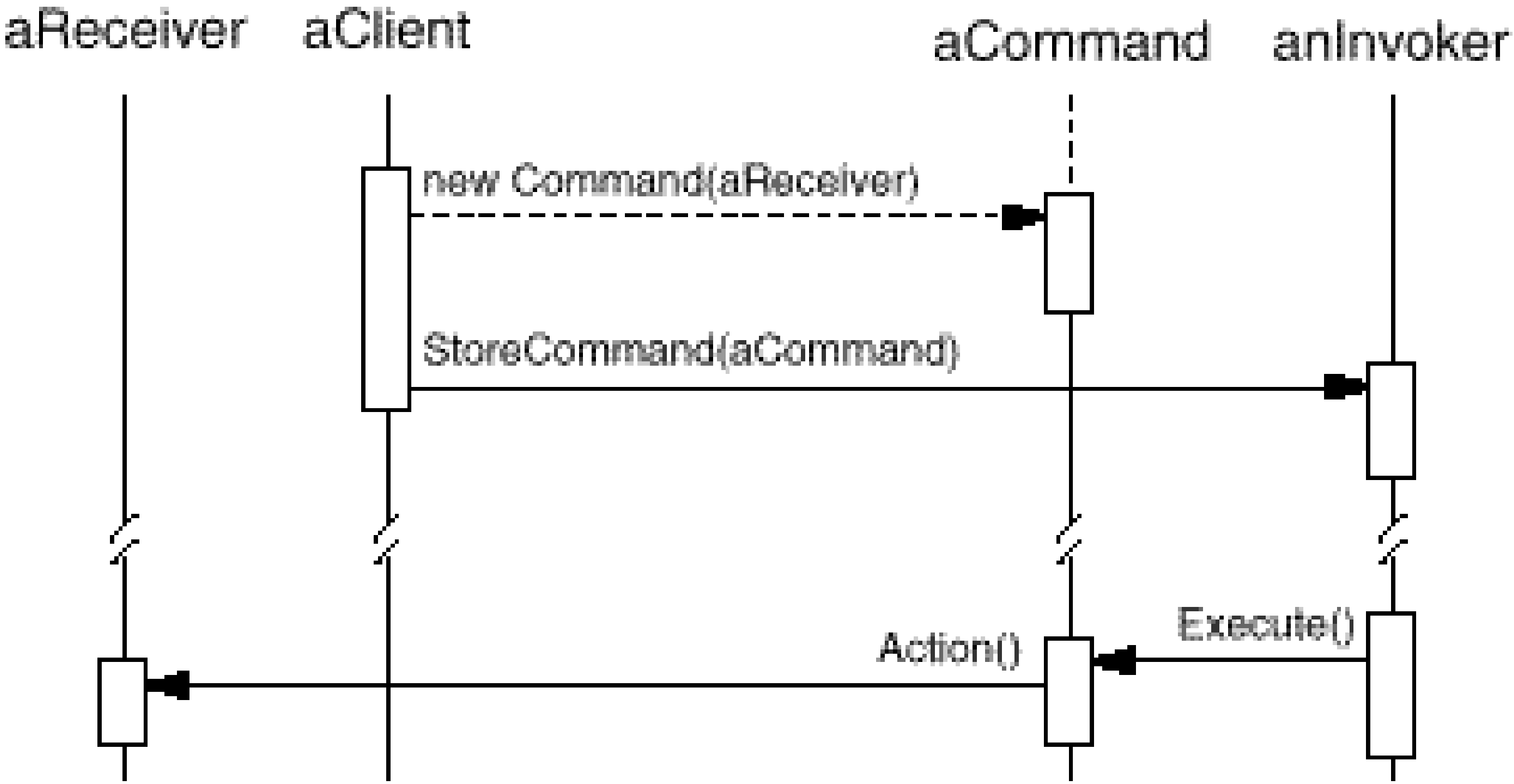

Client
Invoker
Command
Execute()
Receiver
Action()
receiver
ConcreteCommand
Execute()
state
receiver-->Action();

Participants

- Command: Declares an interface for executing an operation.
- ConcreteCommand: Defines a binding between a Receiver object and an action. Implements *Execute()* by invoking the corresponding operation(s) on Receiver.
- Client: Creates a ConcreteCommand object and sets its receiver.
- Invoker: Asks the command to carry out the request.
- Receiver: Knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

Collaborations

- The client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object.
- The invoker issues a request by calling *Execute* on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking *Execute*.
- The ConcreteCommand object invokes operations on its receiver to carry out the request.


aReceiver
aClient
aCommand
anInvoker
new Command(aReceiver)
StoreCommand(aCommand)
Execute()
Action()

Consequences

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- Commands are first-class objects. They can be manipulated and extended like any other object.
- You can assemble commands into a composite command. An example is the MacroCommand class. In general, composite commands are an instance of the Composite pattern.

Consequences

- Allow the receiver veto the command (request);
- It's easy to add new Commands, because you don't have to change existing classes.
- It is easy to implement a command queue;
- It is easy to implement Undo and Redo;
- It is easy to implement Logging mechanisms;
- Command pattern will introduce too many command classes and objects.

Applicability

- Parameterize objects (clients) by an action to perform.
  - You can express such parameterization with a callback function, that is, a function that's registered somewhere to be called at a later point.
  - Commands are an object-oriented replacement for callbacks.
- Specify, queue, and execute requests at different times.
  - A Command object can have a lifetime independent of the original request.

Applicability

- Support undo.
  - The Command's *Execute* operation can store state for reversing its effects in the command itself.
  - The Command interface must have an added *Unexecute* operation that reverses the effects of a previous call to Execute.
  - Executed commands are stored in a history list.
  - Unlimited-level undo and redo is achieved by traversing this list backwards and forwards calling *Unexecute* and *Execute*, respectively.

Applicability

- Support logging changes
    - Adding the Command interface with *Load* and *Store* operations, you can keep a persistent log of changes.
    - They can be reapplied in case of a system crash.
    - Recovering from a crash involves *Load* logged commands from disk and re-executing them with the *Execute* operation.
- Support transactions.
    - Structure a system around high-level operations built on primitives operations.
    - A transaction encapsulates a set of changes to data.
    - Commands have a common interface, letting you invoke all transactions the same way.

Implementation 1:
How intelligent should a command be?

- A command can have a wide range of abilities.
- At one extreme, it merely defines a binding between a receiver and the actions that carry out the request.
  - Sometime commands have enough knowledge to find their receiver dynamically.
- At the other extreme, it implements everything itself without delegating to a receiver at all.
  - It is useful when you want to define commands that are independent of existing classes, when no suitable receiver exists, or when a command knows its receiver implicitly.

Implementation 2: Supporting undo and redo

- Commands can support undo and redo capabilities if they provide a way to reverse their execution (*Unexecute* or *Undo* operation).
- A ConcreteCommand class might need to store additional state to do so.
  - The Receiver object
  - The arguments to the operation performed on the receiver
  - Any original values in the receiver that can change as a result of handling the request.
  - The receiver must provide operations that let the command return the receiver to its prior state.

Implementation 2: Supporting undo and redo

- To support one level of undo, an application needs to store only the command that was executed last.
- For multiple-level undo and redo, the application needs a history list of commands that have been executed,
  - The maximum length of the list determines the number of undo/redo levels.
  - Traversing backward through the list and reverse-executing commands cancels their effect;
  - Traversing forward and executing commands re-executes them.

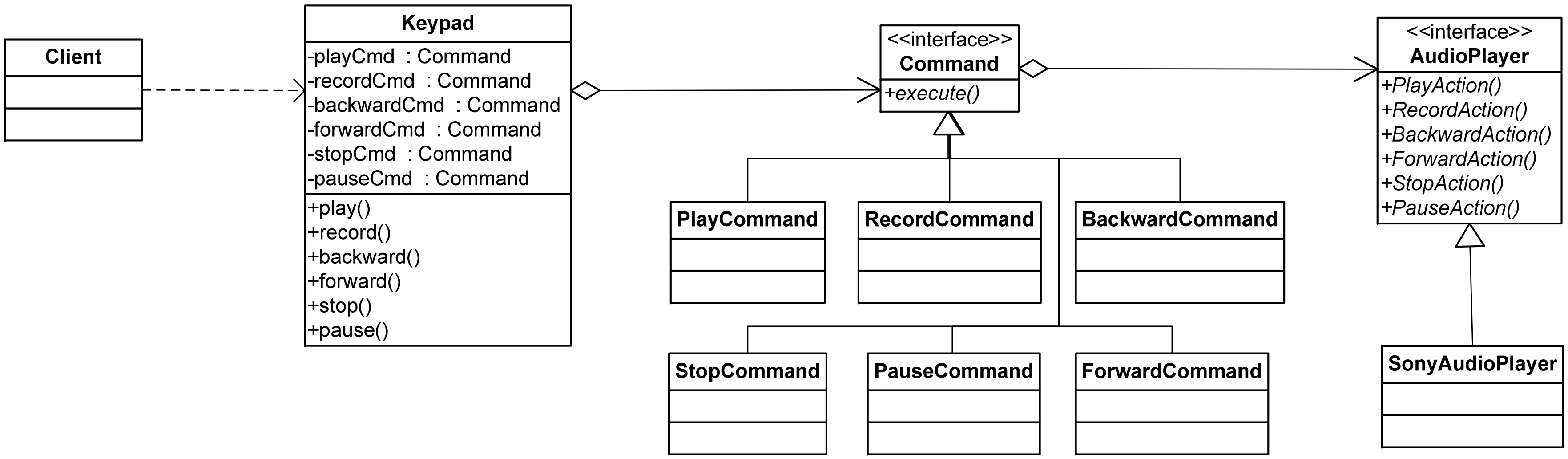
Implementation
Avoiding error accumulation in the undo process.

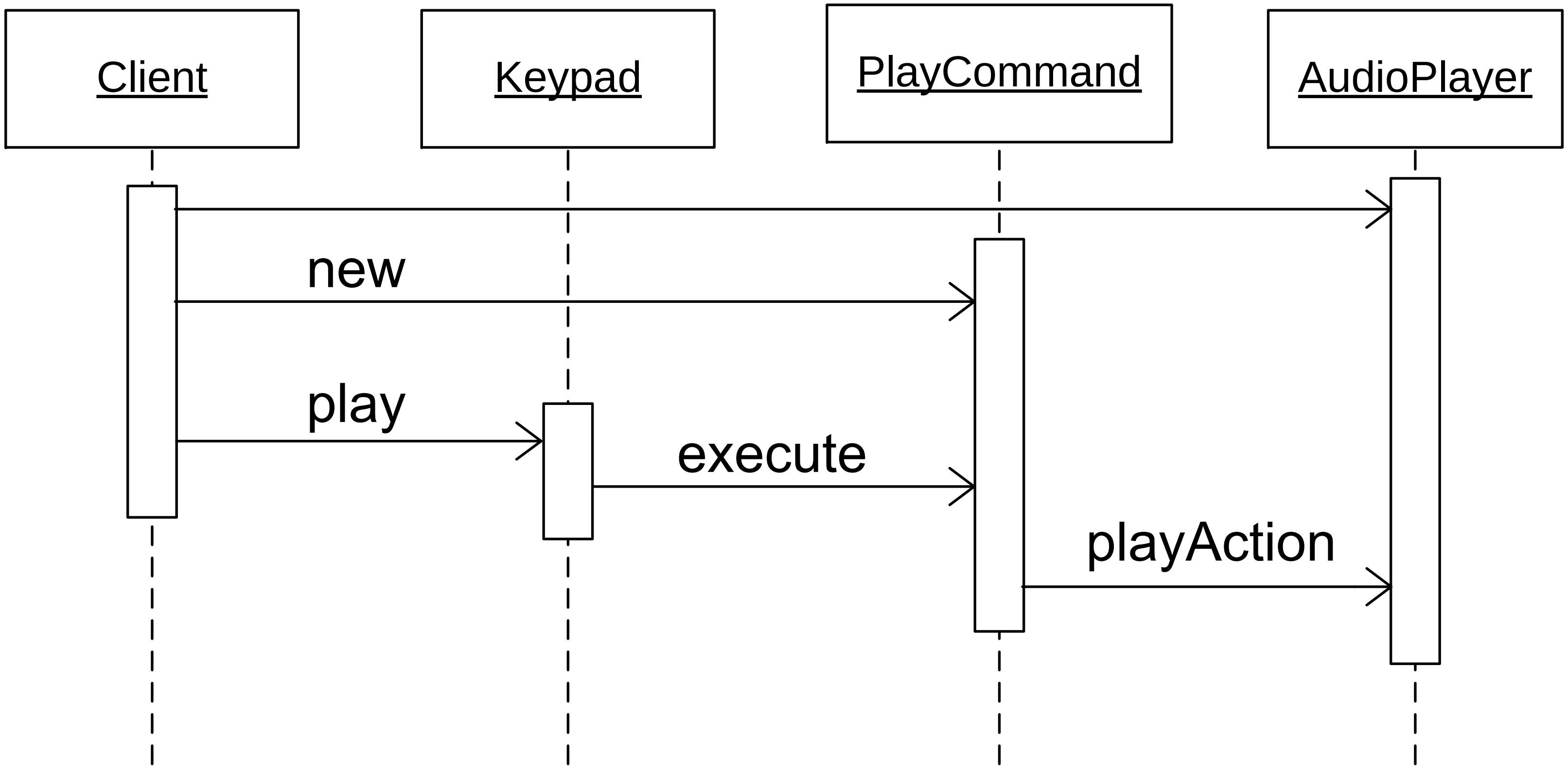
- Errors can accumulate as commands are executed, unexecuted, and re-executed repeatedly, so that an application's state eventually diverges from original values.
- It may be necessary to store more information in the command to ensure that objects are restored to their original state.
- Memento pattern

Example: AudioPlayer system

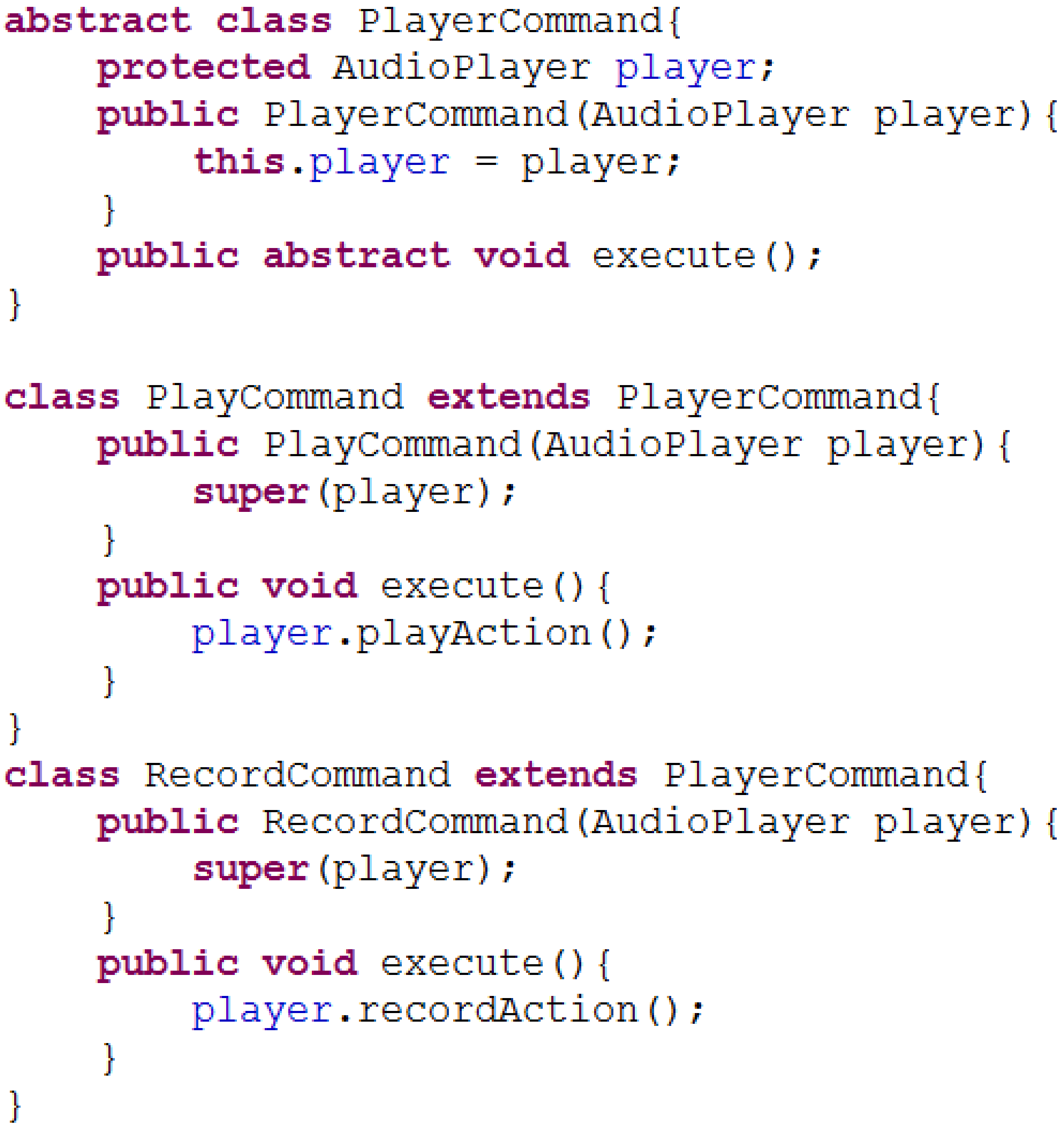
- Play
- Record
- Backward
- Forward
- Stop
- Pause

- Client: Person
- Invoker: Keypad
- Command: Functionalities
- Receiver: AudioPlayer


Client
Keypad
-playCmd : Command
-recordCmd : Command
-backwardCmd : Command
-forwardCmd : Command
-stopCmd : Command
-pauseCmd : Command
+play()
+record()
+backward()
+forward()
+stop()
+pause()
<<interface>>
Command
+execute()
PlayCommand
RecordCommand
BackwardCommand
StopCommand
PauseCommand
ForwardCommand
<<interface>>
AudioPlayer
+PlayAction()
+RecordAction()
+BackwardAction()
+ForwardAction()
+StopAction()
+PauseAction()
SonyAudioPlayer

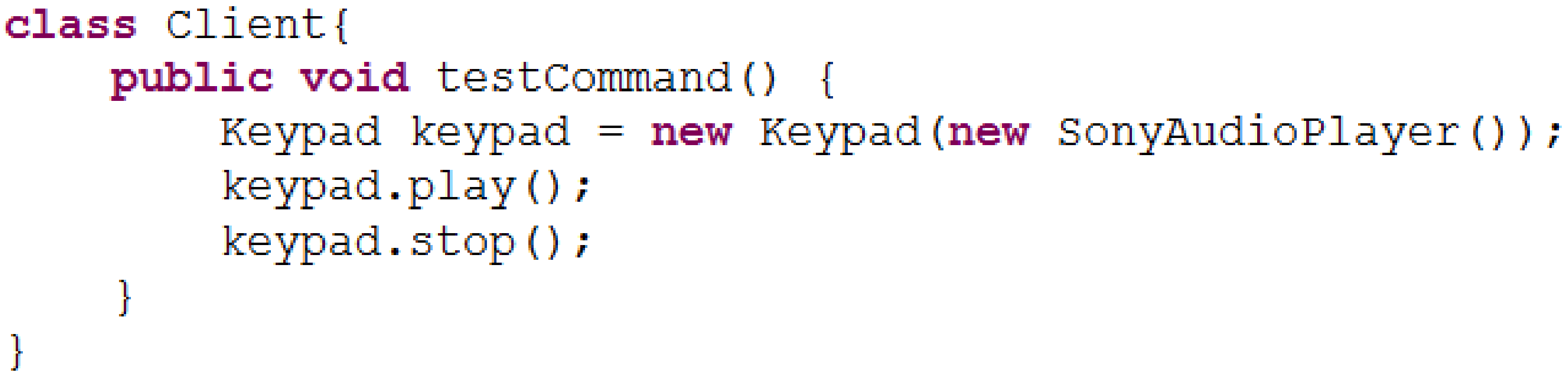

Client
Keypad
PlayCommand
AudioPlayer
new
play
execute
playAction

```java
interface AudioPlayer{
    public void playAction();
    public void recordAction();
    public void backwardAction();
    public void forwardAction();
    public void stopAction();
    public void pauseAction();
}
class SonyAudioPlayer implements AudioPlayer{
    public void backwardAction() {
        // TODO backward
    }
    public void forwardAction() {
        // TODO forward
    }
    public void pauseAction() {
        // TODO pause
    }
    public void playAction() {
        // TODO play
    }
    public void recordAction() {
        // TODO record
    }
    public void stopAction() {
        // TODO stop
    }
}
```



```
abstract class PlayerCommand{
    protected AudioPlayer player;
    public PlayerCommand(AudioPlayer player){
        this.player = player;
    }
    public abstract void execute();
}

class PlayCommand extends PlayerCommand{
    public PlayCommand(AudioPlayer player){
        super(player);
    }
    public void execute(){
        player.playAction();
    }
}
class RecordCommand extends PlayerCommand{
    public RecordCommand(AudioPlayer player){
        super(player);
    }
    public void execute(){
        player.recordAction();
    }
}
```

```
class Keypad{
    private PlayerCommand playCmd;
    private PlayerCommand recordCmd;
    private PlayerCommand forwardCmd;
    private PlayerCommand backwardCmd;
    private PlayerCommand stopCmd;
    private PlayerCommand pauseCmd;

    private AudioPlayer player;

    public Keypad(AudioPlayer player){
        this.player = player;
        playCmd = new PlayCommand(player);
        recordCmd = new RecordCommand(player);
        forwardCmd =new ForwardCommand(player);
        backwardCmd = new BackwardCommand(player);
        stopCmd = new StopCommand(player);
        pauseCmd = new PauseCommand(player);
    }

    public void play(){
        playCmd.execute();
    }
    public void record(){
        recordCmd.execute();
    }
    public void backward(){
        backwardCmd.execute();
    }
    public void forward(){
        forwardCmd.execute();
    }
    public void stop(){
        stopCmd.execute();
    }
    public void pause(){
        pauseCmd.execute();
    }
}
```



```
class Client{
    public void testCommand() {
        Keypad keypad = new Keypad(new SonyAudioPlayer());
        keypad.play();
        keypad.stop();
    }
}
```

Extension: Macro command set

- A macro is a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence.
- A macro command set is pre-defined sequence which contains certain commands in specified order.
- Command pattern is easy to implemented macro command set.
- Macro command can be implemented by Aggregate and Iterator pattern.

Let’s go to next…