



## Chapter 18- Command Design Pattern

Copyright ©2015 by Linda Marshall and Vreda Pieterse. All rights reserved.

### Contents

<b>18.1</b>	<b>Introduction</b>	<b>2</b>
<b>18.2</b>	<b>Command Design Pattern</b>	<b>2</b>
18.2.1	Identification	2
18.2.2	Problem	2
18.2.3	Structure	2
18.2.4	Participants	2
<b>18.3</b>	<b>Comand Pattern Explained</b>	<b>3</b>
18.3.1	Related Patterns	3
<b>18.4</b>	<b>Example</b>	<b>3</b>
18.4.1	TV Remote	3
<b>18.5</b>	<b>Exercises</b>	<b>6</b>
	<b>References</b>	<b>8</b>

## 18.1 Introduction

This Lecture Note introduces the Command design pattern. The pattern will be illustrated using a TV remote as an example.

## 18.2 Command Design Pattern

### 18.2.1 Identification

Name	Classification	Strategy
Command	Behavioural	Delegation
<b>Intent</b>		
<i>Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations.</i> ([1]:263)		

### 18.2.2 Problem

Used to modify existing interfaces to make it work after it has been designed.

### 18.2.3 Structure

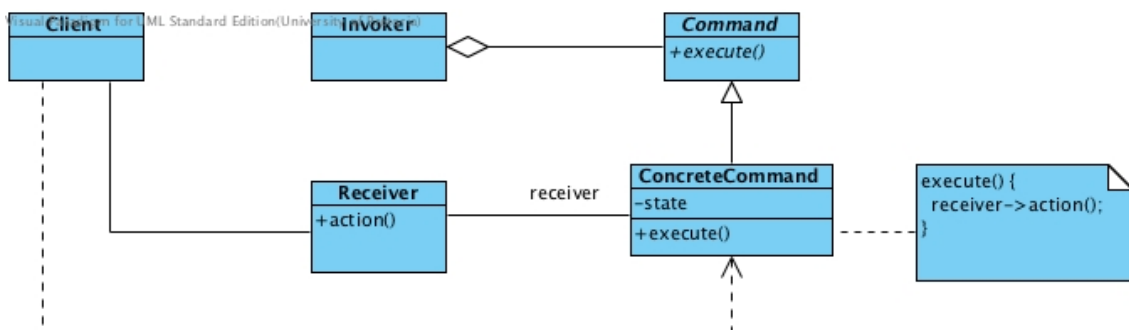


Figure 1: The structure of the Command Design Pattern

### 18.2.4 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 (Application)

- 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.

# 18.3 Comand Pattern Explained

## 18.3.1 Related Patterns

### Chain of Responsibility:

Makes use of Command to represent requests as objects.

### Composite:

MacroCommands can be implemented when combining command with Composite.

### Memento:

Makes use of Command to keep state the command requires to undo its effect.

### Prototype:

A command that must be copied before being placed on the history list acts as a Prototype.

# 18.4 Example

## 18.4.1 TV Remote

This example will model a TV remote. The remote has two buttons, one to flip through channels and one to switch the TV on and off.

Listing 1: Implementation of a TV remote

```
#include <iostream>

using namespace std;

class TV {
public:
    static void action(char* s) { cout<<s<<endl;};
};

class Command {
```

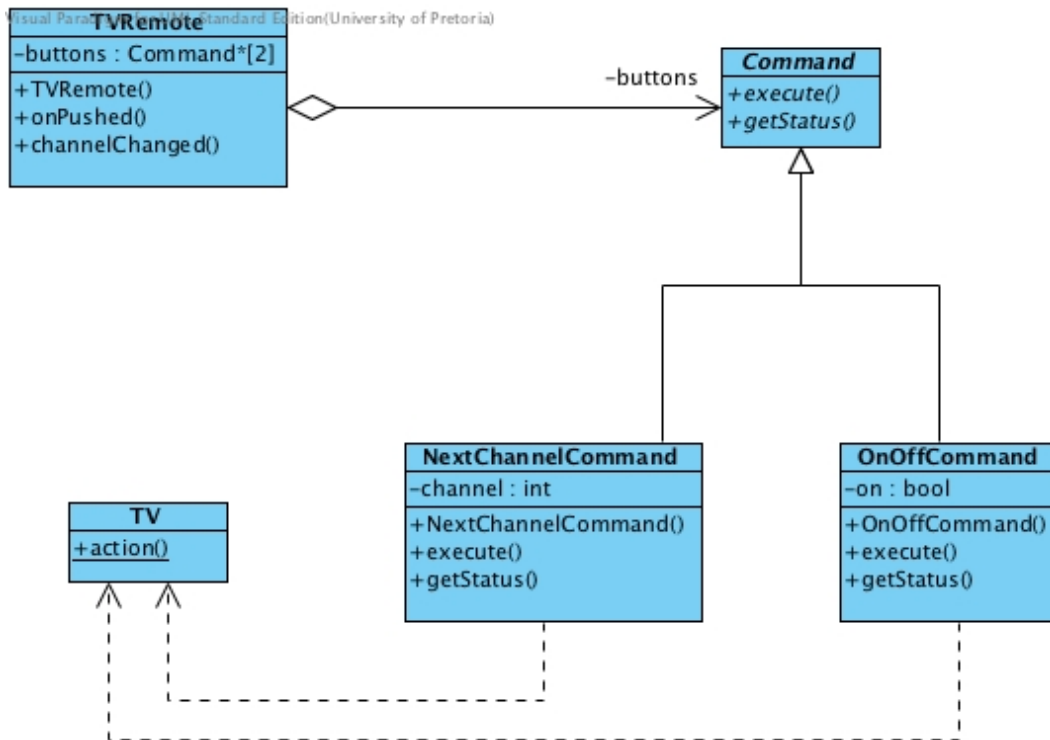


Figure 2: TV remote example

```

public:
    virtual void execute() = 0;
    virtual char* getStatus() = 0;
};

class OnOffCommand : public Command {
public:
    OnOffCommand(bool s) : on(s) {};
    void execute() {
        on = on?false:true;
        TV::action(getStatus());
    };
    char* getStatus() {
        char* str = new char[20];
        strcpy(str, "The device is ");
        strcat(str, (on==false?" off ":" on"));
        return str;
    };
private:
    bool on;
};

class NextChannelCommand : public Command {
public:
    NextChannelCommand() {
        channel = 1;
  
```

```

};
void execute() {
    channel = (channel==5)?1:++channel;
    TV::action(getStatus());
};
char* getStatus() {
    char* str = new char[20];
    strcpy(str, "The device is on channel ");
    switch (channel) {
        case 1: strcat(str, "1"); break;
        case 2: strcat(str, "2"); break;
        case 3: strcat(str, "3"); break;
        case 4: strcat(str, "4"); break;
        case 5: strcat(str, "5"); break;
    }
    return str;
};

private:
    int channel;
};

class TVRemote { // Invoker
public:
    TVRemote(){
        buttons[0] = new OnOffCommand(false);
        buttons[1] = new NextChannelCommand();
    };

    void onPushed(){
        buttons[0]->execute();
    };

    void channelChanged() {
        buttons[1]->execute();
    };
private:
    Command* buttons[2];
};

int main(){
    TVRemote* tvr = new TVRemote;
    tvr->onPushed();
    tvr->channelChanged();
    tvr->channelChanged();
    tvr->channelChanged();
}

```

```

    tvr->onPushed ();
    tvr->channelChanged ();
    tvr->channelChanged ();

    return 0;
}

```

Sample output for the program is given by:

Listing 2: TV Remote Sample Output

## 18.5 Exercises

1. Consider the following code that illustrated the command pattern. Draw the UML class diagram.

Listing 3: Implementation of the LightSwitch

```

class Fan
{
    public:
        void startRotate() { cout << "Fan_is_rotating" << endl;}
        void stopRotate() { cout << "Fan_is_not_rotating" << endl;}
};

class Light
{
    public:
        void turnOn( ) { cout << "Light_is_on_" << endl; }
        void turnOff( ) { cout << "Light_is_off" << endl; }
};

class Command
{
    public:
        virtual void execute ( ) = 0;
};

class LightOnCommand : public Command
{
    public:
        LightOnCommand (Light* L) { myLight = L;}
        void execute( ) { myLight -> turnOn( ); }
    private:
        Light* myLight;
};

class LightOffCommand : public Command

```

```

{
    public:
        LightOffCommand (Light* L) { myLight = L;}
        void execute( ) { myLight -> turnOff( ); }
    private:
        Light* myLight;
};

class FanOnCommand : public Command
{
    public:
        FanOnCommand (Fan* f) { myFan = f;}
        void execute( ) { myFan -> startRotate( );}
    private:
        Fan* myFan;
};

class FanOffCommand : public Command
{
    public:
        FanOffCommand (Fan* f) { myFan = f;}
        void execute( ) {myFan -> stopRotate( );}
    private:
        Fan* myFan;
};

class Switch
{
    public:
        Switch(Command* up, Command* down)
        {
            upCommand = up;
            downCommand = down;
        }

        void flipUp( ) { upCommand -> execute( );};
        void flipDown( ) {downCommand -> execute ( );};

    private:
        Command* upCommand;
        Command* downCommand;
};

int main()
{
    Light* testLight = new Light( );
    Fan* testFan = new Fan();
}

```

```

LightOnCommand* testLiOnCmnd = new LightOnCommand(testLight);
LightOffCommand* testLiOffCmnd = new LightOffCommand(testLight);
FanOnCommand* testFaOnCmnd = new FanOnCommand(testFan);
FanOffCommand* testFaOffCmnd = new FanOffCommand(testFan);

Switch* lightSwitch = new Switch(testLiOnCmnd, testLiOffCmnd);
Switch* fanSwitch = new Switch(testFaOnCmnd, testFaOffCmnd);

lightSwitch -> flipUp();
lightSwitch -> flipDown();
fanSwitch -> flipUp();
fanSwitch -> flipDown();

/** As opposed to
testLight -> turnOn();
testLight -> turnOff();
testFan -> startRotate();
testFan -> stopRotate();
*/

return 0;
}

```

## References

- [1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1994.