It encapsulates a request as an object, thereby letting you parameterize clients with different requests. It can help you queue or log requests, and supportundoable operations. **It is also known as action/transaction design pattern.**

**An Example Where It Can Be Used:**

Sometimes it's necessary to issue requests to objects without knowing anything

about the operation being requested or the receiver of the request. **For example,**

**user interface toolkits include objects like buttons and menus that carry out a**

**request in response to user input.**But the toolkit can't implement the request

explicitly in the button or menu, because **only applications that use the toolkit**

**know what should be done on which object**. As toolkit designers we have no way of

knowing the receiver of the request or the operations that will carry it out.

The Command pattern lets toolkit objects make requests of unspecified application

objects by turning the request itself into an object. This object can be stored

and passed around like other objects. The key to this pattern is an abstract Command class, which declares an interface for executing operations. In the simplest form this interface includes an abstract Execute operation. Concrete Command

Subclasses specify a receiver-action pair by storing the receiver as an

Instance variable and by implementing Execute to invoke the request. The receiver

has the knowledge required to carry out the request.

// A simple Java program to demonstrate

// implementation of Command Pattern using

// a remote control example.

// An interface for command

interface Command

{

public void execute();

}

// Light class and its corresponding command

// classes

class Light

{

public void on()

{

System.out.println("Light is on");

}

public void off()

{

System.out.println("Light is off");

}

}

class LightOnCommand implements Command

{

Light light;

// The constructor is passed the light it

// is going to control.

public LightOnCommand(Light light)

{

this.light = light;

}

public void execute()

{

light.on();

}

}

class LightOffCommand implements Command

{

Light light;

public LightOffCommand(Light light)

{

this.light = light;

}

public void execute()

{

light.off();

}

}

// Stereo and its command classes

class Stereo

{

public void on()

{

System.out.println("Stereo is on");

}

public void off()

{

System.out.println("Stereo is off");

}

public void setCD()

{

System.out.println("Stereo is set " +

"for CD input");

}

public void setDVD()

{

System.out.println("Stereo is set"+

" for DVD input");

}

public void setRadio()

{

System.out.println("Stereo is set" +

" for Radio");

}

public void setVolume(int volume)

{

// code to set the volume

System.out.println("Stereo volume set"

+ " to " + volume);

}

}

class StereoOffCommand implements Command

{

Stereo stereo;

public StereoOffCommand(Stereo stereo)

{

this.stereo = stereo;

}

public void execute()

{

stereo.off();

}

}

class StereoOnWithCDCommand implements Command

{

Stereo stereo;

//stores the reciever

public StereoOnWithCDCommand(Stereo stereo)

{

this.stereo = stereo;

}

public void execute()

{

stereo.on();

stereo.setCD();

stereo.setVolume(11);

}

}

// A Simple remote control with one button

class SimpleRemoteControl

{

Command slot; // only one button

public SimpleRemoteControl()

{

}

public void setCommand(Command command)

{

// set the command the remote will

// execute

slot = command;

}

public void buttonWasPressed()

{

slot.execute();

}

}

// Driver class

class RemoteControlTest

{

public static void main(String[] args)

{

SimpleRemoteControl remote =

new SimpleRemoteControl();

Light light = new Light();

Stereo stereo = new Stereo();

// we can change command dynamically

remote.setCommand(new

LightOnCommand(light));

remote.buttonWasPressed();

remote.setCommand(new

StereoOnWithCDCommand(stereo));

remote.buttonWasPressed();

remote.setCommand(new

StereoOffCommand(stereo));

remote.buttonWasPressed();

}

}

**Note:**

Now, see, here the toolkit equivalent object here is Remote. Now, toolkit **include objects like buttons and menus that carry out a request in response to user input.**

(Here, remote will have some buttons to carry out certain operations. Like, Light on, make stereo on with CD, make stereo off and all. Now, light and stereo are the objects on which these requests will be performed when toolkit makes a request. Now, think about it? If there’s a normal request, can toolkit have any control over the operations that will be performed by light and Stereo if those operations corresponding to the requests are handled by the light and stereo. No. Hence, Turn those requests as objects. Make them perform on Light and Stereo.

Now, notice the all commands implements the interface command:

interface Command

{

public void execute();

}

Now, All concrete command classes will doe their specific task in execute() function.

Note also that all concrete command classes has an object on which it will operate.

**Advantages:**

Makes our code extensible as we can add new commands without changing existing code.

Reduces coupling the invoker and receiver of a command.

**Disadvantages:**

Increase in the number of classes for each individual command