2018/19

Patrón de diseño Command

AYAD MERCER LAAOUISSI JONES

Contenido

Patrón de diseño Command

Este patrón permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación. Se encapsula la petición como un objeto.

Concepto

En general hay cuatro términos invocador, cliente, command y receptor. El objeto command puede invocar un método del receptor de una forma que ha sido especificado a esa clase receptor. El receptor empieza a procesar el trabajo. El objeto command es pasado por separado al objeto invocador para invocar el comando. El objeto cliente sostiene el objeto invocador y el objeto command. El cliente es solo quien realiza la decisión – que comando debe ejecutar – y entonces pasa el comando al objeto invocador (para esa ejecución).

Los propósitos son los siguientes:

1. Encapsular un mensaje como un objeto, con lo que permite gestionar colas o registro de mensaje y deshacer operaciones.
2. Soportar restaurar el estado a partir de un momento dado.
3. Ofrecer una interfaz común que permita invocar las acciones de forma uniforme y extender el sistema con nuevas acciones de forma más sencilla.

Este patrón presenta una forma sencilla de implementar un sistema basado en comandos facilitando su uso.

Ejemplo mundo real

Cuando dibujas algo con un lápiz, puedes necesitar deshacer (borrar y rehacer) algunas partes para hacerlo mejor.

Ejemplo computador mundial

Un ejemplo en el mundo real en el escenario de pintura se aplica en Microsoft Paint. Puedes utilizar el menú o atajos de teclado para rehacer o deshacer en estos contextos. Si deseas realizar una aplicación que soporte deshacer, múltiples deshacer u operaciones similares, entonces este patrón puede ser tu salvador.

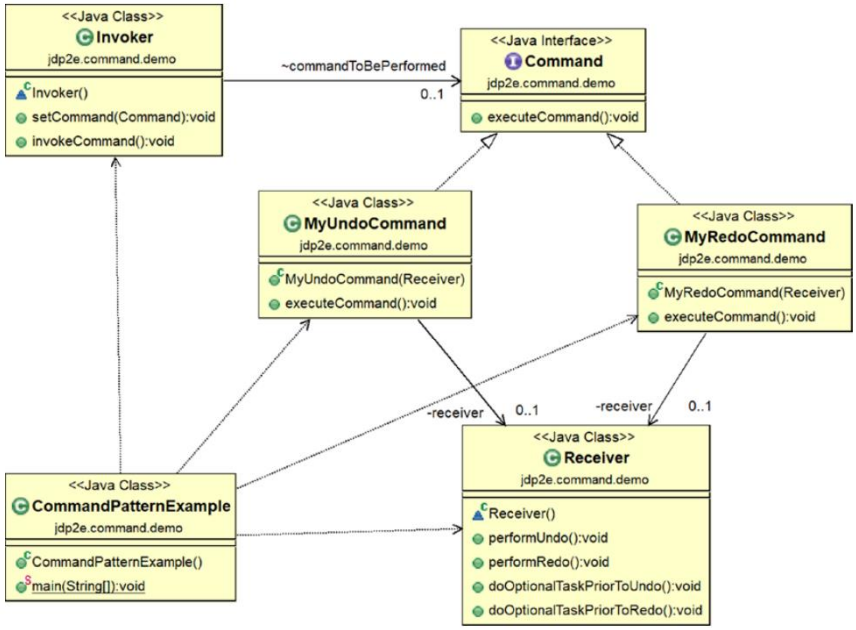
El patrón de diseño command es muy adecuado para manejar GUI interacciones. Funciona tan bien que Microsoft ha integrado en Windows Presentation Foundation (WPF) stack. La pieza más importante es la interfaz ICommand del sistema. Cualquier clase que implemente la interfaz ICommand puede ser usada para controlar eventos del teclado o ratón por los controles comunes WPF. Estos enlaces pueden hacerse mediante XAML o detrás en código.

Ilustración

Consideramos el siguiente ejemplo. Indica los mismos nombres que hemos utilizado en nuestra explicación para facilitar el entendimiento a la hora de ver los ejemplos.

Ejemplo CommandPatternExample.java

- *En la siguiente figura podemos observar el diagrama de clases.*



- *Command y sus tipos de comando*

```
interface Command
{
    //Typically this method does not take any argument.
    //Some of the reasons are:
    //1.We supply all the information when it is created.
    //2.Invoker may reside in different address space.etc.
    void executeCommand();
}

class MyUndoCommand implements Command
{
    private Receiver receiver;
    public MyUndoCommand(Receiver receiver)
    {
        this.receiver=receiver;
    }
    @Override
    public void executeCommand()
    {
        //Perform any optional task prior to UnDo
        receiver.doOptionalTaskPriorToUndo();
        //Call UnDo in receiver now
        receiver.performUndo();
    }
}
class MyRedoCommand implements Command
{
    private Receiver receiver;
    public MyRedoCommand(Receiver receiver)
    {
        this.receiver=receiver;
    }
    @Override
    public void executeCommand()
    {
        //Perform any optional task prior to ReDo
        receiver.doOptionalTaskPriorToRedo();
        //Call ReDo in receiver now
        receiver.performRedo();
    }
}
```

- *Receptor*

```
class Receiver
{
    public void performUndo()
    {
        System.out.println("Performing an undo command in Receiver.");
    }
    public void performRedo()
    {
        System.out.println("Performing an redo command in Receiver.");
    }
    /*Optional method-If you want to perform
     any prior tasks before undo operations.*/
    public void doOptionalTaskPriorToUndo()
    {
        System.out.println("Executing -Optional Task/s prior to  execute undo command.");
    }
    /*Optional method-If you want to perform
     any prior tasks before redo operations*/
    public void doOptionalTaskPriorToRedo()
    {
        System.out.println("Executing -Optional Task/s prior to  execute redo command.");
    }

}
```

- *Invocador*

```
class Invoker
{
    Command commandToBePerformed;
    //Alternative approach:
    //You can also initialize the invoker with a command object
    /*public Invoker(Command command)
    {
        this.commandToBePerformed = command;
    }*/

    //Set the command
    public void setCommand(Command command)
    {
        this.commandToBePerformed = command;
    }
    //Invoke the command
    public void invokeCommand()
    {
        commandToBePerformed.executeCommand();
    }
}
```

- *Salida por consola "Cliente"*

```
 93 public class CommandPatternExample {
 94
 95    public static void main(String[] args) {
 96        System.out.println("***Command Pattern Demo***\n");
 97        /*Client holds both the Invoker and Command Objects*/
 98        Receiver intendedReceiver = new Receiver();
 99        MyUndoCommand undoCmd = new MyUndoCommand(intendedReceiver);
100        //If you use parameterized constructor of Invoker
101        //use the following line of code.
102        //Invoker invoker = new Invoker(undoCmd);
103        Invoker invoker = new Invoker();
104        invoker.setCommand(undoCmd);
105        invoker.invokeCommand();
106
107        MyRedoCommand redoCmd = new MyRedoCommand(intendedReceiver);
108        invoker.setCommand(redoCmd);
109        invoker.invokeCommand();
110    }
111 }
112
```

Problems | Javadoc | Declaration | Console

```
<terminated> CommandPatternExample [Java Application] D:\GIPP2016EclipseMars\GIPP2016EclipseMars\JDKS\jdk1.8.
***Command Pattern Demo***

Executing -Optional Task/s prior to  execute undo command.
Performing an undo command in Receiver.
Executing -Optional Task/s prior to  execute redo command.
Performing an redo command in Receiver.
```

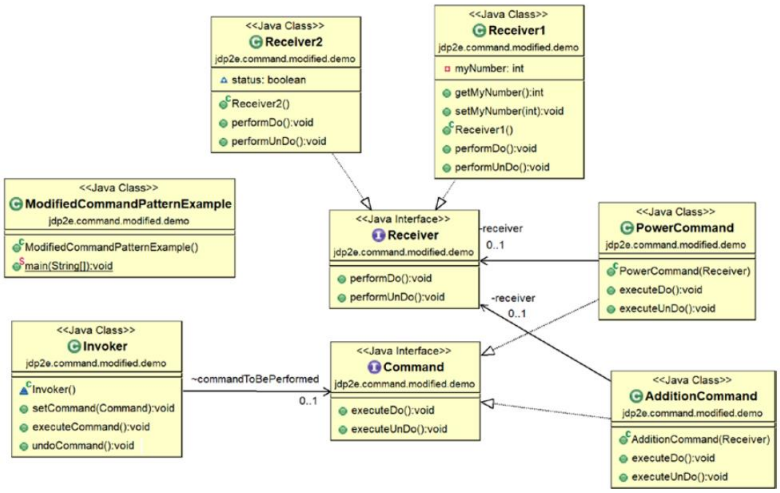
- *Funcionamiento*

En este ejemplo utilizamos el patrón de diseño command para ejecutar el contenido de uno de los dos tipos de comandos que le indicamos al invocador. Nuestra interfaz Command ejecuta el contenido sin conocer realmente el contenido de esta operación.

Lo primero que hacemos en este ejemplo es pasar el receptor a uno de los dos tipos de comando (MyUndoCommand o de tipo MyRedoCommand). Al invocador le pasamos este tipo de comando (MyUndoCommand o de tipo MyRedoCommand). El último paso es invocar el comando invokeCommand() del objeto invoker que realizara la acción dependiendo del tipo de comando que le hemos pasado al invocador.

Ejemplo ModifiedCommandPatternExample.java

- *En la siguiente figura podemos observar el diagrama de clases.*



- *Command y sus tipos de comando*

```
interface Command
{
    void executeDo();
    void executeUnDo();
}
class AdditionCommand implements Command
{
    private Receiver receiver;
    public AdditionCommand(Receiver receiver)
    {
        this.receiver = receiver;
    }
    @Override
    public void executeDo()
    {
        receiver.performDo();
    }
    @Override
    public void executeUnDo()
    {
        receiver.performUnDo();
    }
}
class PowerCommand implements Command
{
    private Receiver receiver;
    public PowerCommand(Receiver receiver)
    {
        this.receiver = receiver;
    }
    @Override
    public void executeDo()
    {
        receiver.performDo();
    }
    @Override
    public void executeUnDo()
    {
        receiver.performUnDo();
    }
}
```

- *Receptor "Interfaz"*

```
interface Receiver
{
    //It will add 2 with a number or switch on the m/c
    void performDo();
    //It will subtract 2 from a number or switch off the m/c
    void performUnDo();
}
```

- *Receptor 1*

```
class Receiver1 implements Receiver
{
    private int myNumber;

    public int getMyNumber()
    {
        return myNumber;
    }
    public void setMyNumber(int myNumber)
    {
        this.myNumber = myNumber;
    }
    public Receiver1()
    {
        myNumber = 10;
        System.out.println("Receiver1 initialized with " + myNumber);
        System.out.println("The objects of receiver1 cannot set beyond "+ myNumber);
    }
    @Override
    public void performDo()
    {
        System.out.println("Received an addition request.");
        //int presentNumber = this.myNumber;
        //this.myNumber = this.myNumber + 2;
        int presentNumber = getMyNumber();
        setMyNumber(presentNumber + 2);
        System.out.println(presentNumber +" + 2 ="+ this.myNumber);
    }
    @Override
    public void performUnDo()
    {
        System.out.println("Received an undo addition request.");
        int presentNumber = this.myNumber;
        //We started with number 10.We'll not decrease further.
        if (presentNumber > 10)
        {
            //this.myNumber = this.myNumber - 2;
            setMyNumber(this.myNumber - 2);
            System.out.println(presentNumber +" - 2 ="+ this.myNumber);
            System.out.println("\t Undo request processed.");
        }
        else
        {
            System.out.println("Nothing more to undo...");
        }
```

- *Receptor 2*

```
class Receiver2 implements Receiver
{
    boolean status;

    public Receiver2()
    {
        System.out.println("Receiver2 initialized ");
        status=false;
    }
    @Override
    public void performDo()
    {
        System.out.println("Received a system power on request.");
        if( status==false)
        {
            System.out.println("System is starting up.");
            status=true;
        }
        else
        {
            System.out.println("System is already running.So, power on request is ignored.");
        }

    }
    @Override
    public void performUnDo()
    {
        System.out.println("Received a undo request.");
        if( status==true)
        {
            System.out.println("System is currently powered on.");
            status=false;
            System.out.println("\t Undo request processed.System is switched off now.");
        }
        else
        {
            System.out.println("System is switched off at present.");
            status=true;
            System.out.println("\t Undo request processed.System is powered on now.");

        }
    }
}
```

- *Invocador*

```
class Invoker
{
    Command commandToBePerformed;
    public void setCommand(Command command)
    {
        this.commandToBePerformed = command;
    }
    public void executeCommand()
    {
        commandToBePerformed.executeDo();
    }
    public void undoCommand()
    {
        commandToBePerformed.executeUnDo();
    }
}
```

- *Cliente*

```
public class ModifiedCommandPatternExample {
    public static void main(String[] args) {

        System.out.println("***Command Pattern Q&As***");
        System.out.println("***A simple demo with undo supported operations***\n");
        /*Client holds  both the Invoker and Command Objects*/

        //Testing receiver -Receiver1
        System.out.println("-----Testing operations in Receiver1-----");
        Receiver intendedreceiver = new Receiver1();
        Command currentCmd = new AdditionCommand(intendedreceiver);

        Invoker invoker = new Invoker();
        invoker.setCommand(currentCmd);
        System.out.println("*Testing single do/undo operation*");
        invoker.executeCommand();
        invoker.undoCommand();
        System.out.println("_______");
        System.out.println("**Testing a series of do/undo operations**");
        //Executed the command 2 times
        invoker.executeCommand();
        //invoker.UndoCommand();
        invoker.executeCommand();
        //Trying to undo 3 times
        invoker.undoCommand();
        invoker.undoCommand();
        invoker.undoCommand();

        System.out.println("\n-----Testing operations in Receiver2-----");
        intendedreceiver = new Receiver2();
        currentCmd = new PowerCommand(intendedreceiver);
        invoker.setCommand(currentCmd);

        System.out.println("*Testing single do/undo operation*");
        invoker.executeCommand();
        invoker.undoCommand();
        System.out.println("_______");
        System.out.println("**Testing a series of do/undo operations**");
        //Executed the command 2 times
        invoker.executeCommand();
        invoker.executeCommand();
        //Trying to undo 3 times
        invoker.undoCommand();
        invoker.undoCommand();
        invoker.undoCommand();
```

- *Salida por consola*

```
***Command Pattern Q&As***
***A simple demo with undo supported operations***

-----Testing operations in Receiver1-----
Receiver1 initialized with 10
The objects of receiver1 cannot set beyond 10
*Testing single do/undo operation*
Received an addition request.
10 + 2 =12
Received an undo addition request.
12 - 2 =10
        Undo request processed.
_______
**Testing a series of do/undo operations**
Received an addition request.
10 + 2 =12
Received an addition request.
12 + 2 =14
Received an undo addition request.
14 - 2 =12
        Undo request processed.
Received an undo addition request.
12 - 2 =10
        Undo request processed.
Received an undo addition request.
Nothing more to undo...

-----Testing operations in Receiver2-----
Receiver2 initialized
*Testing single do/undo operation*
Received a system power on request.
System is starting up.
Received a undo request.
System is currently powered on.
        Undo request processed.System is switched off now.
_______
**Testing a series of do/undo operations**
Received a system power on request.
System is starting up.
Received a system power on request.
System is already running.So, power on request is ignored.
Received a undo request.
System is currently powered on.
        Undo request processed.System is switched off now.
Received a undo request.
```

- *Funcionamiento*

En este ejemplo tenemos dos tipos de receptor que implementan la interfaz receptor. El método executCommand() podra añadir 2 al valor o encender una máquina y el undoCommand() podrá quitar 2 cuando el valor no es menor a 10 o apagar la máquina.

Tiene la misma estructura que en el ejercicio anterior la única diferencia es que en esta tenemos distintos tipos de receptores que al ejecutar el comando va a conseguir hacer una cosa u otra, y al rehacer conseguirá rehacer a su estado anterior. El mismo funcionamiento cuando utilizamos en nuestro ordenador los comandos rehacer a un valor anterior o deshacer.

Funcionamiento del patrón de diseño en los ejemplos

Los ejemplos anteriores lo podemos conseguir gracias a que indicamos que Command es una interfaz y las clases que implementen esta interfaz deberán escribir sus métodos. De esta manera podemos pasar al invocador cualquier objeto que implementa esta interfaz, nuestro tipo de comando deberá implementa esta interfaz. Al pasar al invocador cualquier objeto que implementa esta interfaz podemos hacer uso de los métodos que tenemos en la interfaz y de este modo se consigue que ejecute contenido Command sin conocer realmente el contenido de esta operación, debido a que el contenido estará en el tipo de comando que hemos pasado al invocador.

La utilización de la interfaz es para poder conseguir multiples comandos o receptores (como en el caso ModifiedCommandPatternExample.java).