Command Pattern

Introducción y nombre

Command. De comportamiento. Especifica una forma simple de separar la ejecución de un comando, del entorno que generó dicho comando.

Intención

Permite solicitar una operación a un objeto sin conocer el contenido ni el receptor real de la misma. Encapsula un mensaje como un objeto.

También conocido como

Comando, Orden, Action, Transaction.

Motivación

Este patrón suele establecer en escenarios donde se necesite encapsular una petición dentro de un objeto, permitiendo parametrizar a los clientes con distintas peticiones, encolarlas, guardarlas en un registro de sucesos o implementar un mecanismo de deshacer/repetir.

Solución

Dado que este patrón encapsula un mensaje como un objeto, este patrón debería ser usado cuando:

Se necesiten colas o registros de mensajes.

Tener la posibilidad de deshacer las operaciones realizadas.

Se necesite uniformidad al invocar las acciones.

Facilitar la parametrización de las acciones a realizar.

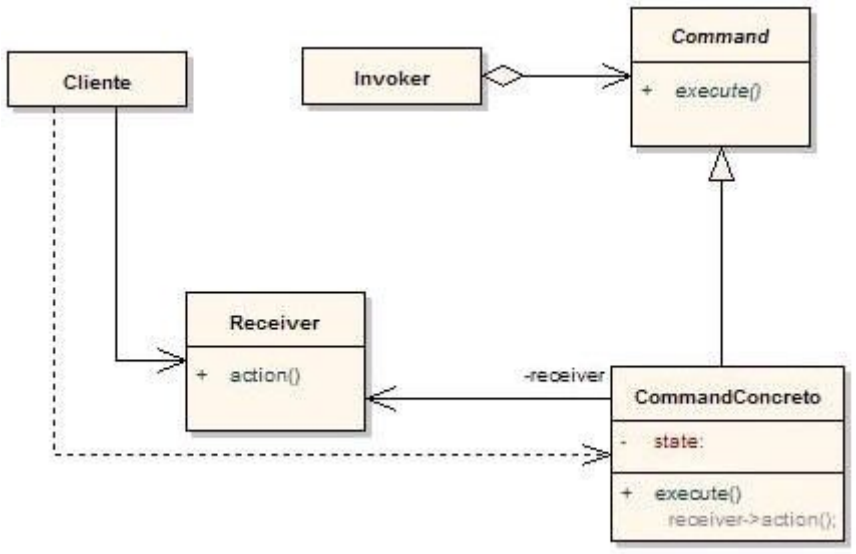
Independizar el momento de petición del de ejecución.

El parámetro de una orden puede ser otra orden a ejecutar.

Desarrollar sistemas utilizando órdenes de alto nivel que se construyen con operaciones sencillas (primitivas).

Se necesite sencillez al extender el sistema con nuevas acciones.

Diagrama UML



Participantes

Command: declara una interfaz para ejecutar una operación.

CommandConcreto: define un enlace entre un objeto “Receiver” y una acción. Implementa el método execute invocando la(s) correspondiente(s) operación(es) del “Receiver”.

Cliente: crea un objeto “CommandConcreto” y establece su receptor.

Invoker: le pide a la orden que ejecute la petición.

Receiver: sabe como llevar a cabo las operaciones asociadas a una petición. Cualquier clase puede hacer actuar como receptor.

Colaboraciones

El cliente crea un objeto “CommandConcreto” y especifica su receptor.

Un objeto “Invoker” almacena el objeto “CommandConcreto”.

El invocador envía una petición llamando al método execute sobre la orden.

El objeto “CommandConcreto”, invoca operaciones de su receptor para llevar a cabo la petición.

Consecuencias

Command desacoplado: el objeto que invoca la operación de aquél que sabe como realizarla.

Las órdenes son objetos manipulados y extendidos de forma natural.

Se pueden ensamblar órdenes en una orden compuesta.

Facilidad de adición de nuevos objetos Command.

Implementación

La clave de este patrón es una clase abstracta o interfaz Command que define una operación execute. Son las subclases concretas quienes implementan la operación y especifican el receptor de la orden. Para ello debemos tener en cuenta:

Permitir deshacer y repetir cuando sea necesario.

Evitar la acumulación de errores en el proceso de deshacer.

Los commands deberían invocar ordenes en el receptor.

Código de muestra

Escenario: una empresa maneja varios servidores y cada uno de ellos deben correr diversos procesos, como apagarse, prenderse, etc. Cada uno de estos procesos, a su vez, implican pequeños pasos como, por ejemplo, realizar una conexión a dicho servidor, guardar los datos en un log, etc.

Dado que cada servidor tiene su propia lógica para cada operación, se decidió crear una interfaz llamada IServer que deben implementar los servidores:

[code]

```
public interface IServer {

    public void apagate();

    public void prendete();

    public void conectate();

    public void verificaConexion();

    public void guardaLog();

    public void cerraConexion();
}
```

Los Servidores son:

```
public class BrasilServer implements IServer {

    public void apagate() {
        System.out.println("Apagando el servidor de Brasil");
    }

    @Override
    public void cerraConexion() {
        System.out.println("Cerrando conexion con el servidor de Brasil");
    }

    @Override
    public void conectate() {
        System.out.println("Conectando al servidor de Brasil");
    }

    @Override
    public void guardaLog() {
        System.out.println("Guardar Log de Brasil");
    }

    @Override
    public void prendete() {
        System.out.println("Prendiendo el servidor de Brasil");
    }

    @Override
    public void verificaConexion() {
        System.out.println("Comprobando la conexion de Brasil");
    }
}
```

[/code]

Obviamente en un caso real los métodos tendrían el algoritmo necesario para realizar tales operaciones. Hemos simplificado estos algoritmos y en su lugar realizamos una salida por consola en cada método.

[code]
```
public class USAServer implements IServer {

    @Override
    public void apagate() {
        System.out.println("Apagando el servidor de USA");
    }

    @Override
    public void cerraConexion() {
        System.out.println("Cerrando conexion con el servidor de USA");
    }

    @Override
    public void conectate() {
        System.out.println("Conectando al servidor de USA");
    }

    @Override
    public void guardaLog() {
        System.out.println("Guardar Log de USA");
    }

    @Override
    public void prendete() {
        System.out.println("Prendiendo el servidor de USA");
    }

    @Override
    public void verificaConexion() {
        System.out.println("Comprobando la conexion de USA");
    }
}

public class ArgentinaServer implements IServer {

    @Override
    public void apagate() {
        System.out.println("Apagando el servidor de Argentina");
    }

    @Override
    public void cerraConexion() {
        System.out.println("Cerrando conexion con el servidor de Argentina");
    }

    @Override
    public void conectate() {
        System.out.println("Conectando al servidor de Argentina");
    }

    @Override
    public void guardaLog() {
        System.out.println("Guardar Log de Argentina");
    }

    @Override
    public void prendete() {
        System.out.println("Prendiendo el servidor de Argentina");
    }

    @Override
    public void verificaConexion() {
        System.out.println("Comprobando la conexion de Argentina");
    }
```

```
}
```

Hasta aquí nada nuevo, sólo clases que implementan una interfaz y le dan inteligencia al método. Comenzaremos con el Command:

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

Y ahora realizamos las operaciones-objetos, es decir, los Command Concretos:

```
public class PrendeServer implements Command {

   private IServer servidor;

   public PrendeServer(IServer servidor) {
      this.servidor = servidor;
   }

   @Override
   public void execute() {
      servidor.conectate();
      servidor.verificaConexion();
      servidor.prendete();
      servidor.guardaLog();
      servidor.cerraConexion();
   }
}

public class ResetServer implements Command {

   private IServer servidor;

   public ResetServer(IServer servidor) {
      this.servidor = servidor;
   }

   @Override
   public void execute() {
      servidor.conectate();
      servidor.verificaConexion();
      servidor.guardaLog();
      servidor.apagate();
      servidor.prendete();
      servidor.guardaLog();
      servidor.cerraConexion();
   }
}

public class ApagarServer implements Command {

   private IServer servidor;

   public ApagarServer(IServer servidor) {
      this.servidor = servidor;
   }

   @Override
   public void execute() {
      servidor.conectate();
      servidor.verificaConexion();
      servidor.guardaLog();
      servidor.apagate();
      servidor.cerraConexion();
   }
}
```

Ahora realizaremos un invocador, es decir, una clase que simplemente llame al método execute:

```
public class Invoker {
   private Command command;

   public Invoker(Command command) {
      this.command = command;
   }

   public void run() {
      command.execute();
   }
}
```

Veamos como funciona el ejemplo:

```
public static void main(String[] args) {
      IServer server = new ArgentinaServer();

      Command command = new PrendeServer(server);

      Invoker serverAdmin = new Invoker(command);
      serverAdmin.run();
   }
```

[/code]

La salida por consola es:

Conectando al servidor de Argentina

Comprobando la conexión de Argentina

Prendiendo el servidor de Argentina

Guardar Log de Argentina

Cerrando conexión con el servidor de Argentina

Cuándo utilizarlo

Lo que permite el patrón Command es desacoplar al objeto que invoca a una operación de aquél que tiene el conocimiento necesario para realizarla. Esto nos otorga muchísima flexibilidad: podemos hacer, por ejemplo, que una aplicación ejecute tanto un elemento de menú como un botón para hacer una determinada acción. Además, podemos cambiar dinámicamente los objetos Command.