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.

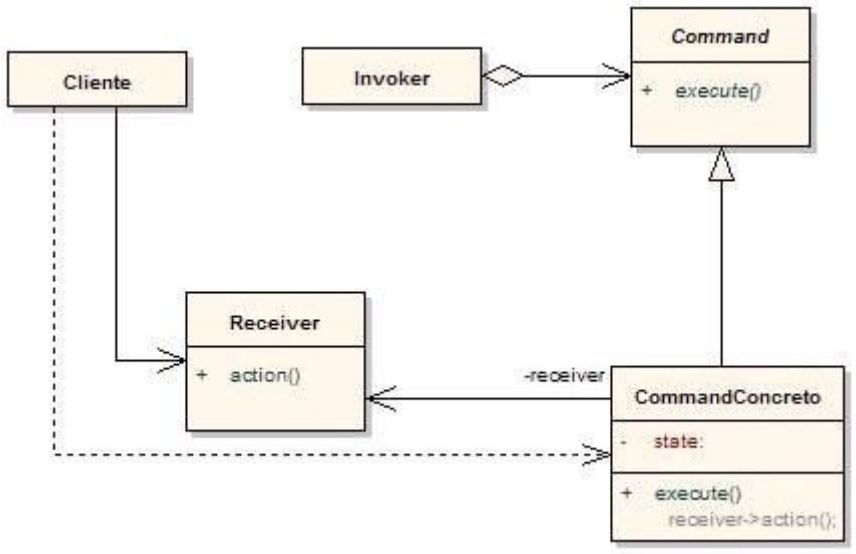
Motivación

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.

Chain of Responsibility Pattern

Introducción y nombre

Chain of Responsibility. De comportamiento. El patrón de diseño Chain of Responsibility permite establecer una cadena de objetos receptores a través de los cuales se pasa una petición formulada por un objeto emisor.

Intención

Como se dijo anteriormente se busca establecer una cadena de receptores, donde cualquiera de ellos puede responder a la petición en función de un criterio establecido. Busca evitar un montón de if – else largos y complejos en nuestro código, pero sobre todas las cosas busca evitar que el cliente necesite conocer toda nuestra estructura jerárquica y que rol cumple cada integrante de nuestra estructura.

También conocido como

Cadena de responsabilidad.

Motivación

En múltiples ocasiones, un cliente necesita que se realice una función, pero o no conoce al servidor concreto de esa función o es conveniente que no lo conozca para evitar un gran acoplamiento entre ambos.

Encadena los objetos receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto. Este evita que el cliente deba conocer toda nuestra estructura de clases, ya que cualquiera le resuelve el problema.

Solución

Se utiliza cuando:

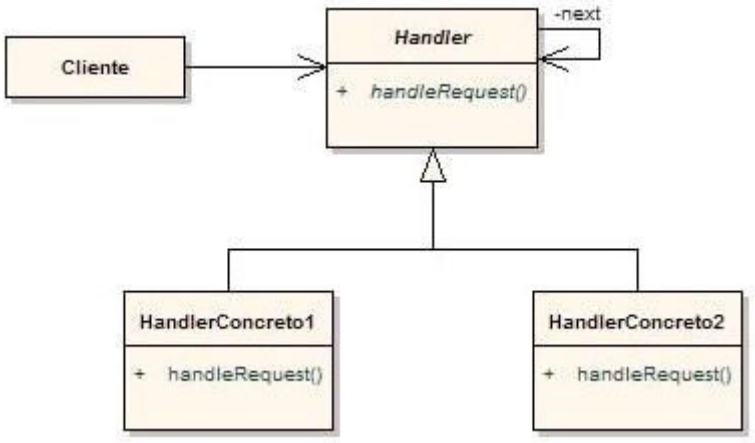
Las peticiones emitidas por un objeto deben ser atendidas por distintos objetos receptores.

No se sabe a priori cual es el objeto que me puede resolver el problema.

Cuando un pedido debe ser manejado por varios objetos.

El conjunto de objetos que pueden tratar una petición debería ser especificado dinámicamente.

Diagrama UML



Participantes

Handler: define una interfaz para tratar las peticiones. Implementa el enlace al sucesor.

HandlerConcreto: trata las peticiones de las que es responsable. Si puede manejar la petición, lo hace, en caso contrario la reenvía a su sucesor.

Cliente: inicializa la petición.

Colaboraciones

El cliente conoce a un gestor que es el que lanza la petición a la cadena hasta que alguien la recoge.

Consecuencias

Reduce el acoplamiento.

Añade flexibilidad para asignar responsabilidades a objetos.

No se garantiza la recepción.

Implementación

Todos los objetos receptores implementarán la misma interfaz o extenderán la misma clase abstracta. En ambos casos se proveerá de un método que permita obtener el sucesor y así el paso de la petición por la cadena será lo más flexible y transparente posible.

La idea es crear un sistema que pueda servir a diversas solicitudes de manera jerárquica.

Código de muestra

Escenario: estamos realizando el software para un banco y uno de los puntos más importantes es saber quién puede aprobar un crédito. Por lo tanto el banco define las siguientes reglas de negocio:

Si el monto no supera los $ 10.000 entonces el ejecutivo de cuenta pueda aprobar el préstamo.

Si el monto esta entre los $10.000 y $50.000 entonces la persona indicada para realizar la aprobación es el líder inmediato de dicho ejecutivo.

Si el monto se encuentra entre $ 50.000 y $100.000 entonces es el Gerente quién debe realizar dicha aprobación.

Por montos superiores a los $100.000 entonces la aprobación la realizará el Director.

Para este caso se ha decidido realizar un patrón Chain of Responsibility. Se decide crear una interface llamada IAprobador que debe implmentar toda clase que pertenezca a nuestra cadena de responsabilidades.

[code]

```
public interface IAprobador {
   public void setNext(IAprobador aprobador);
   public IAprobador getNext();
   public void solicitudPrestamo(int monto);
}
```

Ahora veamos las clases que son aprobadoras:

```
public class EjecutivoDeCuenta implements IAprobador {
```

```
    private IAprobador next;

    @Override
    public IAprobador getNext() {
        return next;
    }

    @Override
    public void solicitudPrestamo(int monto) {
        if (monto <= 10000) {
            System.out.println("Lo manejo yo, el ejecutivo de cuentas");
        } else {
            next.solicitudPrestamo(monto);
        }
    }

    @Override
    public void setNext(IAprobador aprobador) {
        next = aprobador;
    }
}
public class LiderTeamEjecutivo implements IAprobador {

    private IAprobador next;

    @Override
    public IAprobador getNext() {
        return next;
    }

    @Override
    public void solicitudPrestamo(int monto) {
        if (monto > 10000 && monto <= 50000) {
            System.out.println("Lo manejo yo, el lider");
        } else {
            next.solicitudPrestamo(monto);
        }
    }

    @Override
    public void setNext(IAprobador aprobador) {
        next = aprobador;
    }
}

public class Gerente implements IAprobador {

    private IAprobador next;

    @Override
    public IAprobador getNext() {
        return next;
    }

    @Override
    public void solicitudPrestamo(int monto) {
        if (monto > 50000 && monto <= 100000) {
            System.out.println("Lo manejo yo, el gerente");
        } else {
            next.solicitudPrestamo(monto);
        }
    }

    @Override
    public void setNext(IAprobador aprobador) {
        next = aprobador;
```

```
    }
}

public class Director implements IAprobador {

    private IAprobador next;

    @Override
    public IAprobador getNext() {
        return next;
    }

    @Override
    public void solicitudPrestamo(int monto) {
        if (monto >= 100000) {
            System.out.println("Lo manejo yo, el director");
        }
    }

    @Override
    public void setNext(IAprobador aprobador) {
        next = aprobador;
    }
}
```

Y, por último, el banco, que a fin de cuentas es quién decide las reglas del negocio.

```
public class Banco implements IAprobador {

    private IAprobador next;

    @Override
    public IAprobador getNext() {
        return next;
    }

    @Override
    public void solicitudPrestamo(int monto) {
        EjecutivoDeCuenta ejecutivo = new EjecutivoDeCuenta();
        this.setNext(ejecutivo);

        LiderTeamEjecutivo lider = new LiderTeamEjecutivo();
        ejecutivo.setNext(lider);

        Gerente gerente = new Gerente();
        lider.setNext(gerente);

        Director director = new Director();
        gerente.setNext(director);

        next.solicitudPrestamo(monto);
    }

    @Override
    public void setNext(IAprobador aprobador) {
        next = aprobador;
    }
}
```

Veamos como es su funcionamiento:

```
public class Main {
    public static void main(String[] args) {
        Banco banco = new Banco();
        banco.solicitudPrestamo(56000);
    }
```

}

Y el resultado por consola es: Lo manejo yo, el gerente

[/code]

Cuando utilizarlo

La motivación detrás de este patrón es crear un sistema que pueda servir a diversas solicitudes de manera jerárquica. En otras palabras, si un objeto que es parte de un sistema no sabe cómo responder a una solicitud, la pasa a lo largo del árbol de objetos. Como el nombre lo implica, cada objeto de dicho árbol puede tomar la responsabilidad y atender la solicitud.

Un ejemplo típico podría ser el lanzar un trabajo de impresión. El cliente no sabe siquiera qué impresoras están instaladas en el sistema, símplemente lanza el trabajo a la cadena de objetos que representan a las impresoras. Cada uno de ellos lo deja pasar, hasta que alguno, finalmente lo ejecuta.

Hay un desacoplamiento evidente entre el objeto que lanza el trabajo (el cliente) y el que lo realiza (impresora).

Abstract Factory Pattern

Introducción y nombre

Abstract Factory. Creacional.

Intención

Ofrece una interfaz para la creación de familias de productos relacionados o dependientes sin especificar las clases concretas a las que pertenecen.

También conocido como

Factoría Abstracta, Kit.

Motivación

El problema que intenta solucionar este patrón es el de crear diferentes familias de objetos. Su objetivo principal es soportar múltiples estándares que vienen definidos por las diferentes familias de objetos. Es similar al Factory Method, sólo le añade mayor abstracción.

Solución

Se debe utilizar este patrón cuando:

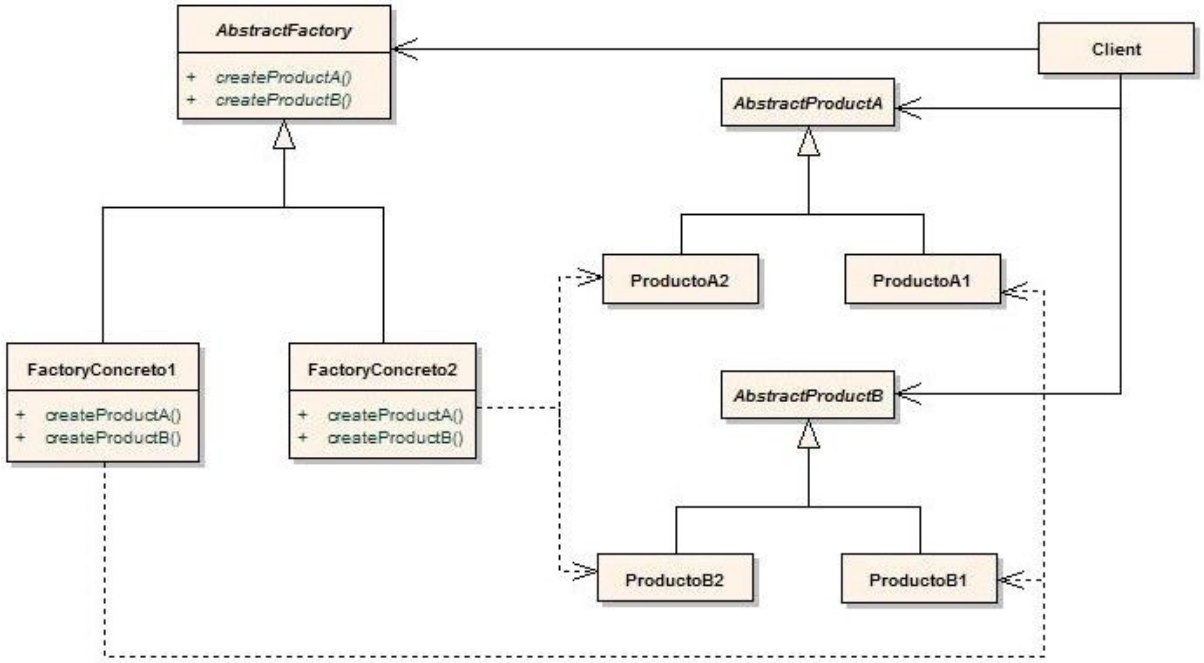
La aplicación debe ser independiente de como se crean, se componen y se representan sus productos.

Un sistema se debe configurar con una de entre varias familias de productos.

Una familia de productos relacionados están hechos para utilizarse juntos (hay que hacer que esto se cumpla).

Para ofrecer una librería de clases, mostrando sólo sus interfaces y no sus implementaciones.

Diagrama UML



Participantes

AbstractFactory: declara una interfaz para la creación de objetos de productos abstractos.

ConcreteFactory: implementa las operaciones para la creación de objetos de productos concretos.

AbstractProduct: declara una interfaz para los objetos de un tipo de productos.

ConcreteProduct: define un objeto de producto que la correspondiente factoría concreta se encargaría de crear, a la vez que implementa la interfaz de producto abstracto.

Colaboraciones

Client: utiliza solamente las interfaces declaradas en la factoría y en los productos abstractos.

Una única instancia de cada FactoryConcreto es creada en tiempo de ejecución. AbstractFactory delega la creación de productos a sus subclases FactoryConcreto.

Consecuencias

Se oculta a los clientes las clases de implementación: los clientes manipulan los objetos a través de las interfaces o clases abstractas.

Facilita el intercambio de familias de productos: al crear una familia completa de objetos con una factoría abstracta, es fácil cambiar toda la familia de una vez simplemente cambiando la factoría concreta.

Mejora la consistencia entre productos: el uso de la factoría abstracta permite forzar a utilizar un conjunto de objetos de una misma familia.

Como inconveniente podemos decir que no siempre es fácil soportar nuevos tipos de productos si se tiene que extender la interfaz de la Factoría abstracta.

Implementación

Veamos los puntos más importantes para su implementación:

Factorias como singletons: lo ideal es que exista una instancia de FactoryConcreto por familia de productos.

Definir factorías extensibles: añadiendo un parámetro en las operaciones de creación que indique el tipo de objeto a crear.

Para crear los productos se usa un Factory Method para cada uno de ellos.

Código de muestra

Hagamos de cuenta que tenemos dos familias de objetos:

1) La clase TV, que tiene dos hijas: Plasma y LCD.

2) La clase Color, que tiene dos hijas: Amarillo y Azul.

Más alla de todos los atributos/métodos que puedan tener la clase Color y TV, lo importante aquí es destacar que Color define un método abstracto:

[code]

```
public abstract void colorea(TV tv);

public abstract class TV {
   public abstract String getDescripcion();
}

public abstract class Color {
   public abstract void colorea(TV tv);
}

public class Amarillo extends Color {
   @Override
   public void colorea(TV tv) {
      System.out.println("Pintando de amarillo en el televisor " + tv.getDescripcion());
   }
}

public class Azul extends Color {
   @Override
   public void colorea(TV tv) {
      System.out.println("Pintando de azul en el televisor " + tv.getDescripcion());
   }
}

public class Plasma extends TV {
   @Override
   public String getDescripcion() {
      return "Plasma";
   }
}
public class LCD extends TV {
   @Override
   public String getDescripcion() {
      return "LCD";
   }
}
```

Escenario: nuestra empresa se dedica a darle un formato estético específico a los televisores LCD y Plasma. Se ha decidido que todos los LCD que saldrán al mercado serán azules y los plasma serán amarillos. Por este motivo se ha decidido realizar el patrón Abstract Factory:

```
public abstract class AbstractFactory {
   public abstract TV createTV();
```

```
    public abstract Color createColor();
}

public class FactoryLcdAzul extends AbstractFactory {
    @Override
    public Color createColor() {
        return new Azul();
    }

    @Override
    public TV createTV() {
        return new LCD();
    }
}

public class FactoryPlasmaAmarillo extends AbstractFactory {
    @Override
    public Color createColor() {
        return new Amarillo();
    }

    @Override
    public TV createTV() {
        return new Plasma();
    }
}
```

Y, por último, la clase Cliente de estas Factory:

```
public class EnsamblajeTV {
    public EnsamblajeTV(AbstractFactory factory) {
        Color color = factory.createColor();
        TV tv = factory.createTV();
        color.colorea(tv);
    }
}
```

Como se puede observar el código es bastante genérico. El secreto de este patrón ocurre en los Factory Concretos que es donde se definen las reglas del negocio.
Veamos esto en funcionamiento:

```
public class Main {
    public static void main(String[] args) {
        // Probando el factory LCD + Azul
        AbstractFactory f1 = new FactoryLcdAzul();
        EnsamblajeTV e1 = new EnsamblajeTV(f1);

        // Probando el factory Plasma + Amarillo
        AbstractFactory f2 = new FactoryPlasmaAmarillo();
        EnsamblajeTV e2 = new EnsamblajeTV(f2);
    }
}
```

El resultado por consola es:
Pintando de azul en el televisor LCD
Pintando de amarillo en el televisor Plasma

[/code]

Factory Method Pattern

Introducción y nombre

Factory Method. Creacional. Libera al desarrollador sobre la forma correcta de crear objetos.

Intención

Define la interfaz de creación de un cierto tipo de objeto, permitiendo que las subclases decidan que clase concreta necesitan instancias.

También conocido como

Virtual Constructor, Método de Factoría.

Motivación

Muchas veces ocurre que una clase no puede anticipar el tipo de objetos que debe crear, ya que la jerarquía de clases que tiene requiere que deba delegar la responsabilidad a una subclase.

Solución

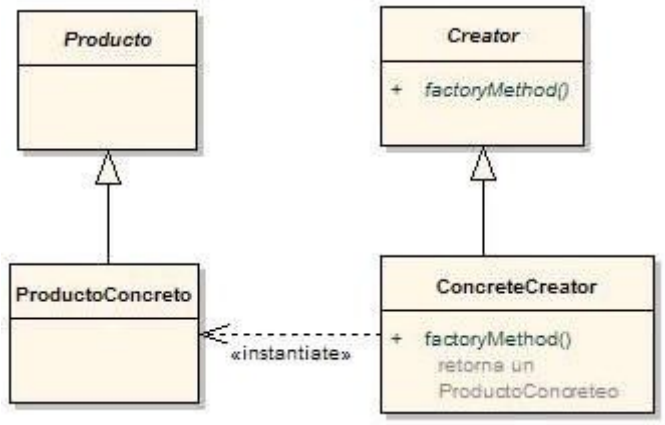
Este patrón debe ser utilizado cuando:

Una clase no puede anticipar el tipo de objeto que debe crear y quiere que sus subclases especifiquen dichos objetos.

Hay clases que delegan responsabilidades en una o varias subclases.

Una aplicación es grande y compleja y posee muchos patrones creacionales.

Diagrama UML



Participantes

Responsabilidad de cada clase participante:

Creator: declara el método de fabricación, que devuelve un objeto de tipo product. Puede llamar a dicho método para crear un objeto producto .

ConcretCreator: redefine el método de fabricación para devolver un objeto concretProduct.

Colaboraciones

ProductoConcreto: es el resultado final. El creador se apoya en sus subclases para definir el método de

fabricación que devuelve el objeto apropiado.

Consecuencias

Como ventaja se destaca que elimina la necesidad de introducir clases específicas en el código del creador. Solo maneja la interfaz Product, por lo que permite añadir cualquier clase ConcretProduct definida por el usuario. Otra ventaja: es más flexible crear un objeto con un Factory Method que directamente: un método factoría puede dar una implementación por defecto.

Un inconveniente es tener que crear una subclase de Creator en los casos en los que esta no fuera necesaria de no aplicar el patrón.

Implementación

Dificultades, técnicas y trucos a tener en cuenta al aplicar el PD

Implementación de la clase Creator

El método factoría es abstracto

El método factoría proporciona una implementación por defecto:

Permite extensibilidad: se pone la creación de objetos en una operación separada por si el usuario quiere cambiarla

Código de muestra

Asumamos que tenes una clase Triángulo y necesitamos crear un tipo de triángulo: escaleno, isosceles o equilatero. Para ello, esta la clase abstracta Triangulo y sus clases hijas:

[code]

```
public abstract class Triangulo {

    private int ladoA;
    private int ladoB;
    private int ladoC;

    public Triangulo(int ladoA, int ladoB, int ladoC) {
        this.ladoA = ladoA;
        this.ladoB = ladoB;
        this.ladoC = ladoC;
    }

    public abstract String getDescripcion();

    public abstract double getSuperficie();

    public abstract void dibujate();
}

public class Equilatero extends Triangulo {

    public Equilatero(int anguloA, int anguloB, int anguloC) {
        super(anguloA, anguloB, anguloC);
    }

    @Override
    public String getDescripcion() {
        return "Soy un Triangulo Equilatero";
    }
```

```
    @Override
    public double getSuperficie() {
        // Algoritmo para calcular superficie
        return 0;
    }

    @Override
    public void dibujate() {
        // Algoritmo para dibujarse
    }
}

public class Escaleno extends Triangulo {

    public Escaleno(int ladoA, int ladoB, int ladoC) {
        super(ladoA, ladoB, ladoC);
    }

    @Override
    public String getDescripcion() {
        return "Soy un Triangulo Escaleno";
    }

    @Override
    public double getSuperficie() {
        // Algoritmo para calcular superficie
        return 0;
    }

    @Override
    public void dibujate() {
        // Algoritmo para dibujarse
    }
}

public class Isosceles extends Triangulo {

    public Isosceles(int ladoA, int ladoB, int ladoC) {
        super(ladoA, ladoB, ladoC);
    }

    @Override
    public String getDescripcion() {
        return "Soy un Triangulo Isosceles";
    }

    public double getSuperficie() {
        // Algoritmo para calcular superficie
        return 0;
    }

    @Override
    public void dibujate() {
        // Algoritmo para dibujarse
    }
}
```

Para evitar que nuestros clientes deban conocer la estructura de nuestra jerarquía creamos un Factory de triángulos:

```
public class TrianguloFactory {
    public Triangulo createTriangulo(int ladoA, int ladoB, int ladoC) {
        if ((ladoA == ladoB) && (ladoA == ladoC)) {
            return new Equilatero(ladoA, ladoB, ladoC);
        } else if ((ladoA != ladoB) && (ladoA != ladoC) && (ladoB != ladoC)) {
            return new Escaleno(ladoA, ladoB, ladoC);
```

```
        } else {
            return new Isosceles(ladoA, ladoB, ladoC);
        }
    }
}
```

De esta forma, no sólo no tienen que conocer nuestra estructura de clases, sino que además tampoco es necesario que conozcan nuestra implementación (el conocimiento del algoritmo que resuelve que clases se deben crear).

Simplemente deberían crear un triángulo de la siguiente forma:

```
public class Main {
    public static void main(String[] args) {
        TrianguloFactory factory = new TrianguloFactory();

        Triangulo triangulo = factory.createTriangulo(10, 10, 10);
        System.out.println(triangulo.getDescripcion());
    }
}
[/code]
```

Cabe aclara que las clases Factory deberian ser Singleton, pero para evitar confundir al estudiante nos hemos concentrado sólo en este patrón.

Cuando utilizarlo

El escenario típico para utilizar este patrón es cuando existe una jerarquía de clases complejas y la creación de un objeto específico obliga al cliente a tener que conocer detalles específicos para poder crear un objeto. En este caso se puede utilizar un Factory Method para eliminar la necesidad de que el cliente tenga la obligación de conocer las todas especificaciones y variaciones posibles.

Por otro lado, imaginemos una aplicación grande, realizada por un grupo de desarrolladores que han utilizado muchos patrones creacionales. Esto hace que el grupo de programadores pierda el control sobre como se debe crear una clase. Es decir, cual es un Singleton o utiliza un Builder o un Prototype. Entonces se realiza un Factory Method para que instancie los objetos de una manera estandard. El desarrollador siempre llama a uno o varios Factory y se olvida de la forma en que se deben crear todos los objetos.

Patrones relacionados

Abstract Factory: suele ser implementada por varios Factory Method.

Prototype y Builder: Si bien hay casos donde parece que se excluyen mutuamente, en la mayoría de los casos suelen trabajar juntos.

Singleton: un Factory Method suele ser una clase Singleton.

Memento Pattern

Introducción y nombre

Memento. De Comportamiento. Permite capturar y exportar el estado interno de un objeto para que luego se peuda restaurar, sin romper la encapsulación.

Intención

Este patrón tiene como finalidad almacenar el estado de un objeto (o del sistema completo) en un momento dado de manera que se pueda restaurar en ese punto de manera sencilla. Para ello se mantiene almacenado el estado del objeto para un instante de tiempo en una clase independiente de aquella a la que pertenece el objeto (pero sin romper la encapsulación), de forma que ese recuerdo permita que el objeto sea modificado y pueda volver a su estado anterior.

También conocido como

Token.

Motivación

Muchas veces es necesario guardar el estado interno de un objeto. Esto debido a que tiempo después, se necesita restaurar el estado del objeto, al que previamente se ha guardado. Hoy en día, muchos aplicativos permiten el "deshacer" y "rehacer" de manera muy sencilla. Para ciertos aplicativos es casi una obligación tener estas funciones y sería impensado el hecho que no las posean. Sin embargo, cuando queremos llevar esto a código puede resultar complejo de implementar. Este patrón intenta mostrar una solución a este problema.

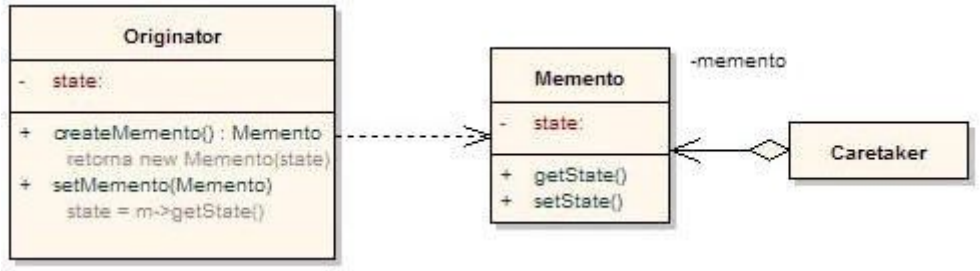
Solución

El patrón Memento se usa cuando:

Se necesite restaurar el sistema desde estados pasados.

Se quiera facilitar el hacer y deshacer de determinadas operaciones, para lo que habrá que guardar los estados anteriores de los objetos sobre los que se opere (o bien recordar los cambios de forma incremental).

Diagrama UML



Participantes

Caretaker

Es responsable por mantener a salvo a Memento.

No opera o examina el contenido de Memento.

Memento

Almacena el estado interno de un objeto Originator. El Memento puede almacenar todo o parte del estado interno de Originator.

Tiene dos interfaces. Una para Caretaker, que le permite manipular el Memento únicamente para pasarlo a otros

objetos. La otra interfaz sirve para que Originator pueda almacenar/restaurar su estado interno, sólo Originator puede acceder a esta interfaz.

Originator

Originator crea un objeto Memento conteniendo una fotografía de su estado interno.

Colaboraciones

Originator crea un Memento y el mismo almacena su estado interno.

Consecuencias

No es necesario exponer el estado interno como atributos de acceso público, preservando así la encapsulación.

Si el originador tuviera que almacenar y mantener a salvo una o muchas copias de su estado interno, sus responsabilidades crecerían y sería inmanejable.

El uso frecuente de Mementos para almacenar estados internos de gran tamaño, podría resultar costoso y perjudicar la performance del sistema.

Caretaker no puede hacer predicciones de tiempo ni de espacio.

Implementación

Si bien la implementación de un Memento no suele variar demasiado, cuando la secuencia de creación y restauración de mementos es conocida, se puede adoptar una estrategia de cambio incremental: en cada nuevo memento sólo se almacena la parte del estado que ha cambiado en lugar del estado completo.

Esta estrategia se aplica cuando memento se utiliza para mantener una lista de deshacer/rehacer.

Otra opción utilizada es no depender de índices en la colecciones y utilizar ciertos métodos no indexados como el .previus() que poseen algunas colecciones.

Código de muestra

Vamos a realizar un ejemplo de este patrón donde se busque salvar el nombre de una persona que puede variar a lo largo del tiempo.

[code]

```
public class Memento {
   private String estado;

   public Memento(String estado) {
      this.estado = estado;
   }

   public String getSavedState() {
      return estado;
   }
}

public class Caretaker {
   private List<Memento> estados = new ArrayList<Memento>();

   public void addMemento(Memento m) {
      estados.add(m);
   }

   public Memento getMemento(int index) {
```

```
        return estados.get(index);
    }
}

public class Persona {

    private String nombre;

    public Memento saveToMemento() {
        System.out.println("Originator: Guardando Memento...");
        return new Memento(nombre);
    }

    public void restoreFromMemento(Memento m) {
        nombre = m.getSavedState();
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
```

Vamos a probar este ejemplo:

```
public class Main {

    public static void main(String[] args) {
        Caretaker caretaker = new Caretaker();

        Persona p = new Persona();
        p.setNombre("Maxi");
        p.setNombre("Juan");

        caretaker.addMemento(p.saveToMemento());

        p.setNombre("Pedro");

        caretaker.addMemento(p.saveToMemento());

        p.setNombre("Diego");

        Memento m1 = caretaker.getMemento(0);
        Memento m2 = caretaker.getMemento(1);

        System.out.println(m1.getSavedState());
        System.out.println(m2.getSavedState());
    }
}

[/code]
```

La salida por consola es:

Originator: Guardando Memento...

Originator: Guardando Memento...

Juan

Pedro

Cuando utilizarlo

Este patrón debe ser utilizado cuando se necesite salvar el estado de un objeto y tener disponible los distintos estados históricos que se necesiten. Por ello mismo, este patrón es muy intuitivo para darse cuando debe ser utilizado.

Hoy en día una gran variedad de aplicaciones poseen las opciones de "deshacer" y "rehacer". Por ejemplo, las herramientas de Microsoft Office como Word, Power Point, etc. Es imposible pensar que ciertas herramientas no tengan esta opción, como el Photoshop. También IDEs de programación como Eclipse utilizan una opción de historial local. Una solución para este problema es el patrón Memento.

Adapter Pattern

Introducción y nombre

Adapter. Estructural. Busca una manera estandarizada de adaptar un objeto a otro.

Intención

El patrón Adapter se utiliza para transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda.

También conocido como

Adaptador. Wrapper (al patrón Decorator también se lo llama Wrapper, con lo cual es nombre Wrapper muchas veces se presta a confusión).

Motivación

Una clase Adapter implementa un interfaz que conoce a sus clientes y proporciona acceso a una instancia de una clase que no conoce a sus clientes, es decir convierte la interfaz de una clase en una interfaz que el cliente espera. Un objeto Adapter proporciona la funcionalidad prometida por un interfaz sin tener que conocer que clase es utilizada para implementar ese interfaz. Permite trabajar juntas a dos clases con interfaces incompatibles.

Solución

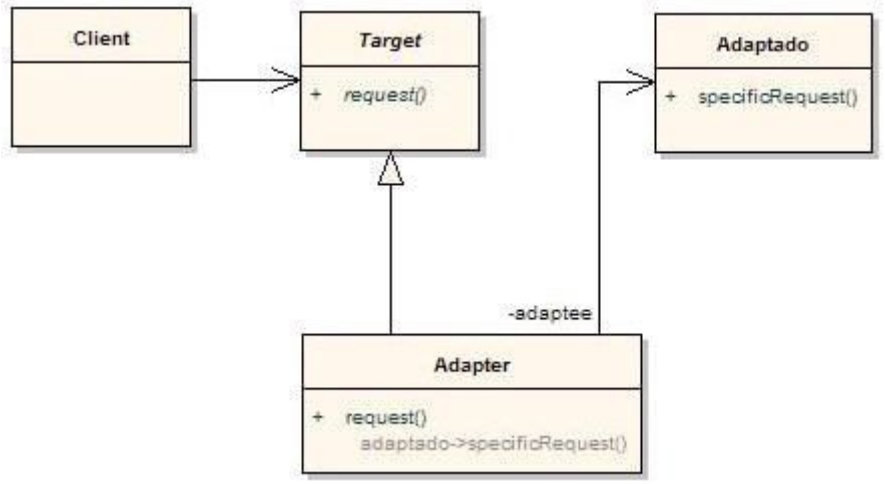
Este patrón se debe utilizar cuando:

Se quiere utilizar una clase que llame a un método a través de una interface, pero se busca utilizarlo con una clase que no implementa ese interface.

Se busca determinar dinámicamente que métodos de otros objetos llama un objeto.

No se quiere que el objeto llamado tenga conocimientos de la otra clase de objetos.

Diagrama UML



Participantes

Target: define la interfaz específica del dominio que Cliente usa.

Cliente: colabora con la conformación de objetos para la interfaz Target.

Adaptado: define una interfaz existente que necesita adaptarse

Adapter: adapta la interfaz de Adaptee a la interfaz Target

Colaboraciones

El Cliente llama a las operaciones sobre una instancia Adapter. De hecho, el adaptador llama a las operaciones de Adaptee que llevan a cabo el pedido.

Consecuencias

El cliente y las clases Adaptee permanecen independientes unas de las otras.

Puede hacer que un programa sea menos entendible.

Permite que un único Adapter trabaje con muchos Adaptees, es decir, el Adapter por sí mismo y las subclases (si es que la tiene). El Adapter también puede agregar funcionalidad a todos los Adaptees de una sola vez.

Implementación

Se debe tener en cuenta que si bien el Adapter tiene una implementación relativamente sencilla, se puede llevar a cabo con varias técnicas:

1) Creando una nueva clase que será el Adaptador, que extienda del componente existente e implemente la interfaz obligatoria. De este modo tenemos la funcionalidad que queríamos y cumplimos la condición de implementar la interfaz.

2) Pasar una referencia a los objetos cliente como parámetro a los costructores de los objetos adapter o a uno de sus métodos. Esto permite al objeto adapter ser utilizado con cualquier instancia o posiblemente muchas instancias de la clase Adaptee. En este caso particular, el Adapter tiene una implementación casi idéntica al patrón Decorator.

3) Hacer la clase Adapter una clase interna de la clase Adaptee. Esto asume que tenemos acceso al código de dicha clase y que es permitido la modificación de la misma.

4) Utilizar sólo interfaces para la comunicación entre los objetos.

Las opciones más utilizadas son la 1 y la 4.

Código de muestra

Vamos a plantear el siguiente escenario: nuestro código tiene una clase Persona (la llamamos PersonaVieja) que se utiliza a lo largo de todo el código y hemos importado un API que también necesita trabajar con una clase Persona (la llamamos PersonaNueva), que si bien son bastante similares tienen ciertas diferencias:

Nosotros trabajamos con los atributos nombre, apellido y fecha de nacimiento.

Sin embargo, la PersonaNueva tiene un solo atributo nombre (que es el nombre y apellido de la persona en cuestión) y la edad actual, en vez de la fecha de nacimiento.

Para esta situación lo ideal es utilizar el Adapter:

[code]

```
public interface IPersonaVieja {
   public String getNombre();
   public void setNombre(String nombre);

   public String getApellido();
   public void setApellido(String apellido);

   public Date getFechaDeNacimiento();
   public void setFechaDeNacimiento(Date fechaDeNacimiento);
}

public class PersonaVieja implements IPersonaVieja{
        private String nombre;
        private String apellido;
        private Date fechaDeNacimiento;

    @Override
        public String getNombre() {
            return nombre;
        }
    @Override
        public void setNombre(String nombre) {
            this.nombre = nombre;
        }
    @Override
        public String getApellido() {
            return apellido;
        }
    @Override
        public void setApellido(String apellido) {
            this.apellido = apellido;
        }
    @Override
        public Date getFechaDeNacimiento() {
            return fechaDeNacimiento;
        }
    @Override
        public void setFechaDeNacimiento(Date fechaDeNacimiento) {
            this.fechaDeNacimiento = fechaDeNacimiento;
        }
}

public interface IPersonaNueva {
   public String getNombre();

   public void setNombre(String nombre);
```

```
    public int getEdad();

    public void setEdad(int edad);
}

public class ViejaToNuevaAdapter implements IPersonaNueva {

    private IPersonaVieja vieja;

    public ViejaToNuevaAdapter(IPersonaVieja vieja) {
        this.vieja = vieja;
    }

    public int getEdad() {
        GregorianCalendar c = new GregorianCalendar();
        GregorianCalendar c2 = new GregorianCalendar();
        c2.setTime(vieja.getFechaDeNacimiento());
        return c.get(1) - c2.get(1);
    }

    @Override
    public String getNombre() {
        return vieja.getNombre() + " " + vieja.getApellido();
    }

    public void setEdad(int edad) {
        GregorianCalendar c = new GregorianCalendar();
        int anioActual = c.get(1);
        c.set(1, anioActual - edad);
        vieja.setFechaDeNacimiento(c.getTime());
    }

    @Override
    public void setNombre(String nombreCompleto) {
        String[] name = nombreCompleto.split(" ");
        String firstName = name[0];
        String lastName = name[1];
        vieja.setNombre(firstName);
        vieja.setApellido(lastName);
    }
}
```

Veremos como funciona este ejempplo:

```
public class Main {
    public static void main(String[] args) {
        PersonaVieja pv = new PersonaVieja();
        pv.setApellido("Perez");
        pv.setNombre("Maxi");
        GregorianCalendar g = new GregorianCalendar();
        g.set(2000, 01, 01);
        // Seteamos que nacio en el año 2000
        Date d = g.getTime();
        pv.setFechaDeNacimiento(d);

        IPersonaNueva adapter = new ViejaToNuevaAdapter(pv);

        System.out.println(adapter.getEdad());
        System.out.println(adapter.getNombre());

        adapter.setEdad(10);
        adapter.setNombre("Juan Perez");

        System.out.println(adapter.getEdad());
        System.out.println(adapter.getNombre());
    }
```

}

[/code]

Cuándo utilizarlo

Este patrón convierte la interfaz de una clase en otra interfaz que el cliente espera. Esto permite a las clases trabajar juntas, lo que de otra manera no podrían hacerlo debido a sus interfaces incompatibles.

Por lo general, esta situación se da porque no es posible modificar la clase original, ya sea porque no se tiene el código fuente de la clase o porque la clase es una clase de propósito general, y es inapropiado para ella implementar un interface par un propósito específico. En resumen, este patrón debe ser aplicado cuando debo transformar una estructura a otra, pero sin tocar la original, ya sea porque no puedo o no quiero cambiarla.