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.

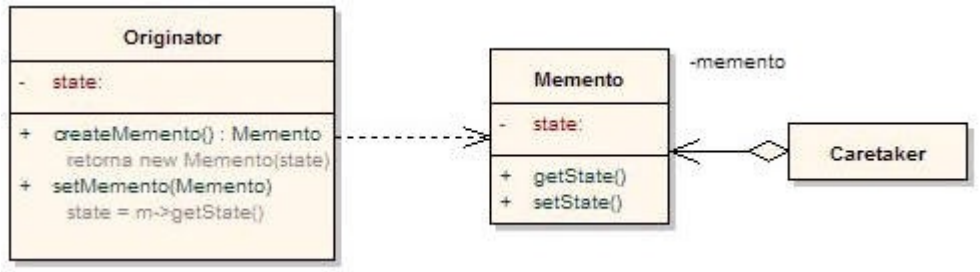
Solución

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

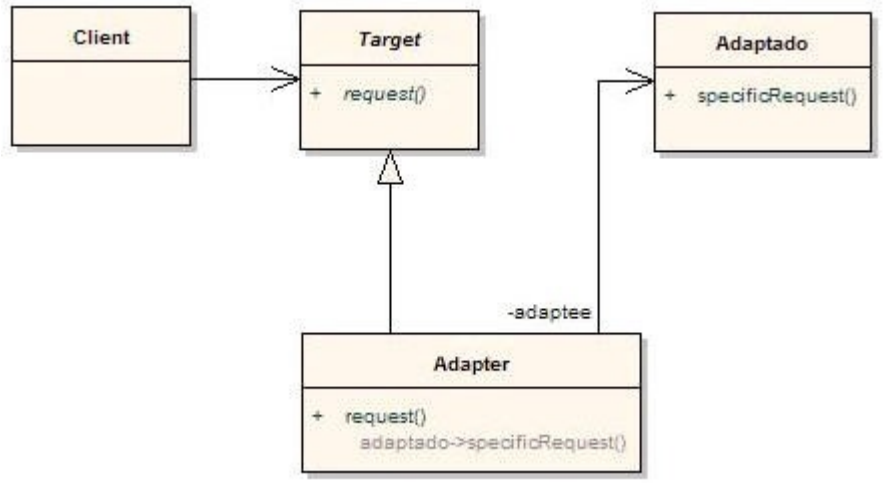
Este patrón se debe utilizar cuando:

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.