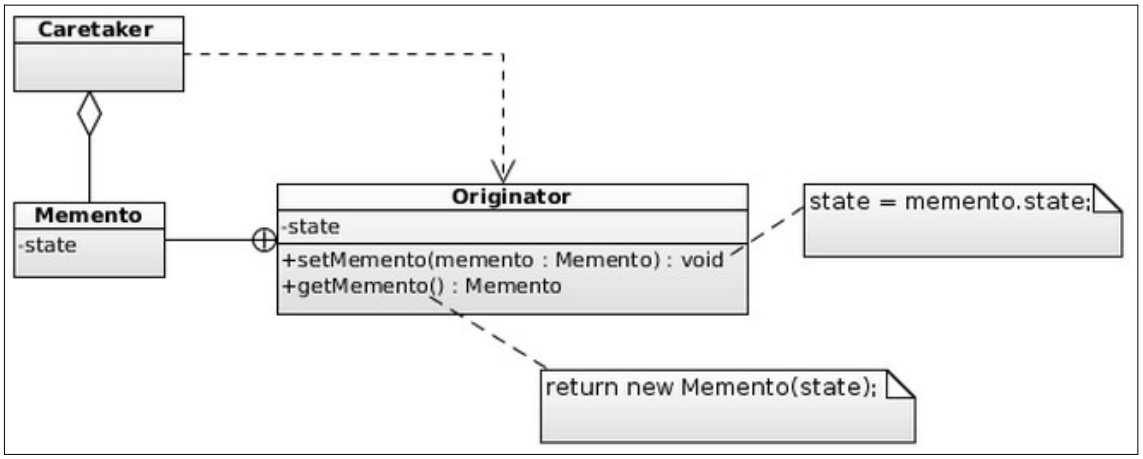
Memento

Diagrama de clases e interfaces



Intención

Proporcionar un mecanismo que, sin violar el principio de la encapsulación, permita capturar y externalizar el estado interno de un objeto, de manera que el objeto pueda ser restaurado a ese mismo estado en un momento futuro.

Motivación

A veces es necesario capturar el estado interno de un objeto en un cierto punto y tener la capacidad de restaurar el objeto a ese estado en un momento posterior. Este podría ser el caso de una aplicación que ante un error o fallo pudiese volver al estado anterior al problema. Supongamos que tal aplicación se trata de un editor gráfico en el que podemos componer una figura con distintos elementos gráficos primitivos. El editor debe permitirnos mover una figura a otra posición en la pantalla. Ahora bien ¿cómo podemos hacer para que la aplicación permita volver a una posición anterior?

- Una posibilidad sería que una supuesta clase EditorGrafico implementara la funcionalidad de deshacer (undo). La operación undo() sería posible gracias a que la clase EditorGrafico mantendría una lista o mapa con todas las

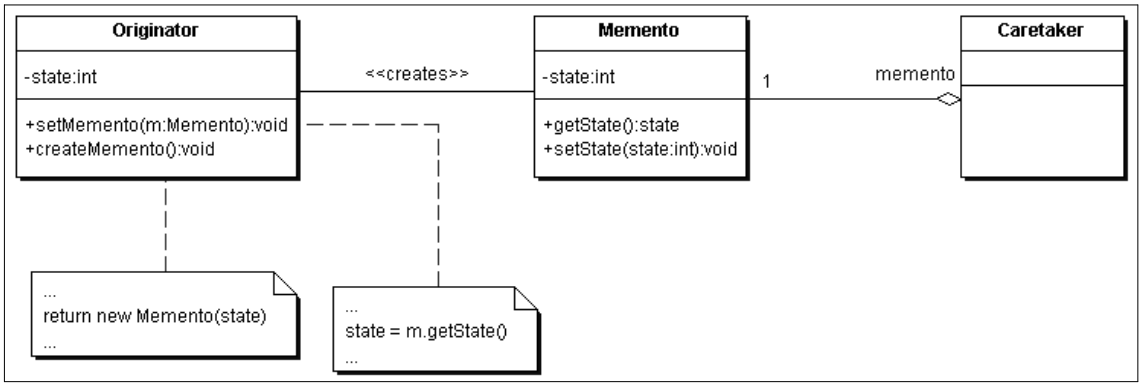
operaciones realizadas sobre cada figura. Sin embargo, aquí tenemos un problema: la clase EditorGrafico, con mucha probabilidad, se habrá vuelto más compleja y difícil de entender y de mantener.

- Otra opción es trasladar la funcionalidad de deshacer las operaciones realizadas a una nueva clase, por ejemplo llamada UndoManager. Esta clase consultaría el estado interno del EditorGrafico y lo "fotografiaría" cuando se le indicara. Esta opción parece que resuelve el problema del exceso de responsabilidades de EditorGrafico. No obstante, presenta dos inconvenientes importantes:
    - Podría ser impracticable que EditorGrafico pudiera proporcionar a UndoManager acceso explícito a todo su estado interno para poder ser restautado en un futuro.
    - Tal acceso violaría el principio de encapsulación/ocultación de la información de una clase.
- Otra opción (la mejor) es utilizar el patrón Memento, tal y como veremos a continuación.

Nota: Mediante una pila de objetos Command y otra de objetos Memento, se dispone de capacidad sin límite para operaciones “undo” y “redo”.

Implementación

El funcionamiento del patrón es el mostrado en el diagrama de clases siguiente:



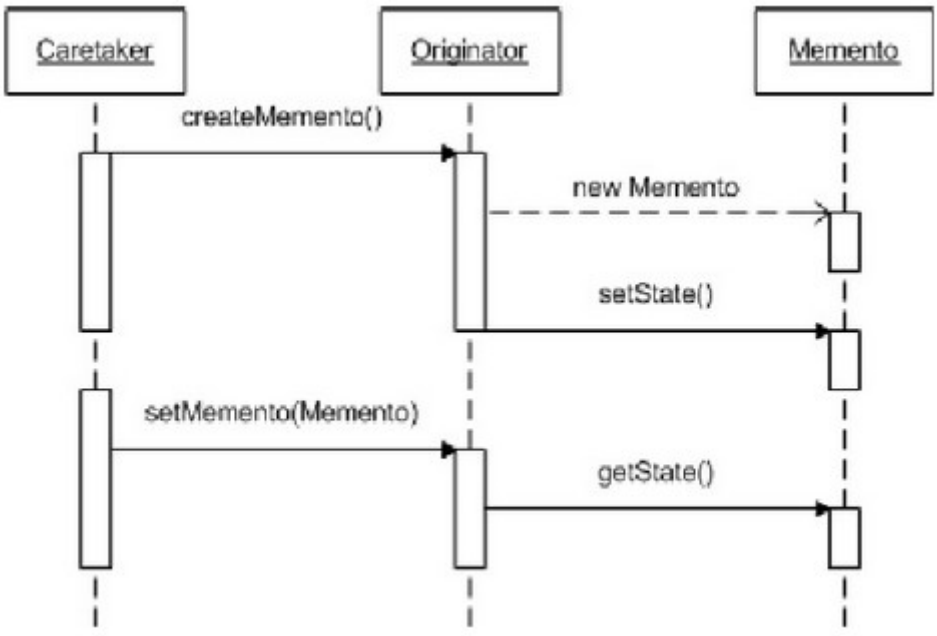
Las clases participantes en el patrón son las siguientes:

- **Memento:** Almacena una instantánea del estado interno del objeto Originator. La cantidad de estado que debe almacenar viene determinada por lo que necesite el Originator para poder restaurarse en un futuro a un estado anterior. El objeto Memento es totalmente pasivo, ya que su creación la lleva a cabo la clase Originator y, de manera similar, la restauración a un estado anterior es conducida por el propio Originator.

- **Originator**: Es la clase que interesará restaurar a un punto anterior en el tiempo. Es responsable del crear el objeto Mememento (el cual toma una instantánea del estado interno del Originator) y lo utilizará cuando sea necesario para restaurar su estado interno.

- **Caretaker**: Clase que solicita un objeto Memento a la clase Originator y los custodia hasta que llegue el momento de pasarlo de nuevo al Originator para que éste lo utilice para restaurar su estado interno (aunque puede ser que el Originator nunca llegue a necesitar restauración de estado). Hay que tener en cuenta que la clase Caretaker ni opera ni examina el contenido del objeto Memento.

La siguiente figura muestra un diagrama de secuencias del patrón:



Aplicabilidad y Ejemplos

El patrón Memento es adecuado cuando:

- Se necesita obtener una instantánea del estado interno de un objeto para poder restaurarlo a ese mismo estado más adelante.
- Una interfaz directa para obtener el estado del objeto expondría detalles de la implementación del objeto que violarían principios básicos de encapsulación de la información.

Veamos ahora algunos ejemplos.

Ejemplo 1 – Memento como clase privada estática del Originator

Se trata de un ejemplo muy elemental, aunque suficientemente ilustrativo de la esencia del patrón, en el que se utiliza una pila para almacenar diferentes instantáneas de un objeto Originator.

Comenzamos por la clase Originator. Según el patrón Memento, esta clase es la responsable de tomar una instantánea de su estado interno para poder restaurarse a ese estado en un momento posterior. En nuestro ejemplo, Originator sólo define un atributo de tipo entero llamado 'state', que será el estado interno a salvaguardar.

Notad que Originator mediante su método:

- getMemento() crea y devuelve un objeto Memento con el estado interno actual de Originator a la clase cliente que se lo solicite.
- setMemento(): Restablece su estado internando a partir del Memento recibido como argumento.

Lo más destacable en este ejemplo, es que la clase Originator incluye una clase privada estática par definir el objeto Memento, por lo que nadie salvo ella tiene acceso al Memento. La clase Memento se limita a guardar el parámetro recibido. De esta manera, la clase Originator podrá restaurar su estado a partir del Memento.

Originator.java

```
package comportamiento.memento.basico;

public class Originator {
    private int state;
    public void setState(int state) {
        this.state = state;
    }
    public int getState() {
        return state;
    }

    public void setMemento(Object object) {
      Memento memento = (Memento) object;
        state = memento.state;
    }

    public Object getMemento() {
        return new Memento(state);
    }

    private static class Memento {
        private int state;
        private Memento(int state) {
            this.state = state;
        }
    }
}
```

A continuación veamos el código para la clase CareTaker, la cual es realmente una clase cliente, que se limita a solicitar al Originator:

- Una instantánea de su estado (un Memento).
- La restauración a un estado previo, por lo que CareTaker tendrá que pasarle de nuevo el Memento.

Hay que advertir que en el ejemplo CareTaker está totalmente desacoplado del Memento (éste se encapsula en Object), de hecho, desconoce por completo su existencia.

Como se comento al principio del enunciado del ejemplo, se trata de disponer de una pila de puntos de restauración (checkpoints). Es por esto que CareTaker actúa de historial, permitiendo almacenar y recuperar objetos.

Notad que CareTaker requiere una referencia al Originator con tal de poder invocar a los métodos que éste ofrece: getMemento()/setMemento().

CareTaker.java

```
package comportamiento.memento.basico;


import java.util.Stack;


public class Caretaker {


    private Stack<Object> history = new Stack<Object>();
    private Originator originator;


    public Caretaker(Originator originator) {
        this.originator = originator;
    }


    public void save() {
        history.push(originator.getMemento());
    }
```

```
    public void restore() {
        originator.setMemento(history.pop());
    }

}
```

Por último, veamos una clase que muestre la utilidad del patrón:

MainClent.java

```
package comportamiento.memento.basico;

public class MainClient {
      public static void main(String[] args) {
            Originator originator = new Originator();
            Caretaker caretaker = new Caretaker(originator);
            originator.setState(1);
            caretaker.save();
            originator.setState(2);
            caretaker.save();
            originator.setState(3);
            caretaker.restore();
            caretaker.restore();
            System.out.println(originator.getState()); // imprime 1
      }
}
```

Salida:

Ejemplo 2 – Memento como clase independiente

El siguiente ejemplo es conceptualmente muy parecido al anterior, con la diferencia que la clase Memento esta vez no es clase interna de Originator. Esto produce que el CareTaker tenga acceso a todas las operaciones del Memento, lo cual podría comprometer el principio de encapsulación. Por tanto, una mejor práctica sería hacer que el Memento implementara al menos dos interfaces:

- Una interfaz con acceso normal para el Originator.
- Una interfaz restringida para el CareTaker.

Esto lo veremos en el tercer ejemplo.

Veamos el código para este ejemplo. Comenzamos por el Originator, que también esta vez está formado por un sólo atributo. Notad que los métodos createMemento() y setMemento() tienen visibilidad de paquete. La idea es que sólo la clase CareTaker, definida en el mismo paquete, pueda utilizarlos. En cambio, los métodos write() y print() son métodos públicos, accesible a cualquier clase, independientemente de su paquete.

Originator.java

```
package comportamiento.memento.basico2;

public class Originator {
	private String words = "";
	Memento createMemento() {
		return new Memento(words);
	}

	void setMemento(Memento memento) {
```

```
            words = memento.getSnapshot();
        }

        public void write(String words) {
            this.words += words;
        }

        public void print() {
            System.out.println("*****************************");
            System.out.println(words);
        }
}
```

Veamos la clase Memento, cuya sencillez hace innecesario comentario alguno.

Memento.java

```
package comportamiento.memento.basico2;

public class Memento {
        private String snapshot;

        Memento(String words) {
            snapshot = new String(words);
        }

        String getSnapshot() {
            return snapshot;
        }
}
```

Ahora veamos el código para el CareTaker. En este caso se utiliza una lista enlazada para para almacenar las instantáneas. En cualquier caso, su funcionamiento es muy similar al del ejemplo anterior.

CareTaker.java

```
package comportamiento.memento.basico2;


import java.util.LinkedList;


public class CareTaker {

	private Originator originator = null;

	private LinkedList<Memento> mementos = new
LinkedList<Memento>();


	public CareTaker(Originator originator) {

		this.originator = originator;

	}


	public void save() {

		Memento memento = originator.createMemento();

		mementos.add(memento);

	}


	public void putBack(int index) {

		if (index >= mementos.size()) {

			return;

		}

		Memento memento = (Memento) mementos.get(index);

		originator.setMemento(memento);

	}

}
```

Finalmente, veamos una clase cliente:

MainClient.java

```
package comportamiento.memento.basico2;


public class MainClient {
    static public void main(String[] args) {
        String crlf =
System.getProperties().getProperty("line.separator");

        Originator originator = new Originator();
        CareTaker caretaker = new CareTaker(originator);

        originator.write("Con diez cañones por banda," + crlf);
        originator.write("viento en popa, a toda vela," + crlf);
        caretaker.save(); // Primera instantánea

        originator.write("no corta el mar, sino vuela" + crlf);
        originator.write("un velero bergantín." + crlf);
        caretaker.save(); // Segunda instantánea

        originator.write("Bajel pirata que llaman," + crlf);
        originator.write("por su bravura, el Temido," + crlf);
        originator.write("en todo mar conocido, " + crlf);
        originator.write("del uno al otro confín. " + crlf);
        caretaker.save(); // Tercera instantánea
        originator.print();

        // Establecer el estado a la segunda instantanea
        caretaker.putBack(1);
        originator.print();
```

```
        // Establecer el estado a la primera instantanea
        caretaker.putBack(0);
        originator.print();

        // Establecer el estado a la tercera instantanea
        caretaker.putBack(2);
        originator.print();
    }
}
```

Salida:

```
*****************************
Con diez cañones por banda,
viento en popa, a toda vela,
no corta el mar, sino vuela
un velero bergantín.
Bajel pirata que llaman,
por su bravura, el Temido,
en todo mar conocido,
del uno al otro confín.

*****************************
Con diez cañones por banda,
viento en popa, a toda vela,
no corta el mar, sino vuela
un velero bergantín.

*****************************
Con diez cañones por banda,
viento en popa, a toda vela,

*****************************
Con diez cañones por banda,
viento en popa, a toda vela,
no corta el mar, sino vuela
un velero bergantín.
Bajel pirata que llaman,
por su bravura, el Temido,
en todo mar conocido,
del uno al otro confín.
```

Ejemplo 3 – Memento implementando dos interfaces

En este ejemplo implementamos una sencilla calculadora que sólo permite sumar dos cantidades, permitiendo deshacer la última operación realizada. La gracia del caso es que la clase Memento implementa dos interfaces:

- Una sin ningún método (interfaz de marcado) adecuada para el Caretaker, pues éste no tiene que conocer detalle alguno del Memento. Esta interfaz la podemos considerar como la "pública", la que no permite manipular el objeto Memento.
- Otra con los métodos, adecuada para que el Originator pueda restaurar su estado a partir del Memento. Esta interfaz la consideramos como la "privada", es decir, es una interfaz exclusiva para el Originator.

Por lo anterior, este ejemplo es el más purista de los vistos hasta el momento.

Comenzamos por ver la interfaz de marcado. Como acabamos de comentar, no define ningún método, por lo que la clase cliente que la reciba no podrá conocer nada sobre el Memento:

MementoToCaretaker.java

```
package comportamiento.memento.calculadora;


/**
 *  Memento appropriate interface to Caretaker
 */
public interface MementoToCareTaker {
      // no operations permitted for the caretaker
}
```

Ahora veamos la interfaz que utilizar el Originator cuando sea necesario restaurar su estado:

MementoToOriginator.java

```
package comportamiento.memento.calculadora;


/**
* Memento Interface to Originator
*
* This interface allows the originator to restore its state
*/
public interface MementoToOriginator {
	public int getFirstNumber();
	public int getSecondNumber();
}
```

Seguimos con la clase Memento. Notad que implementa las dos interfaces anteriores:

Memento.java

```
package comportamiento.memento.calculadora;


/**
 * Memento Object Implementation
 *
 * Note that this object implements both interfaces
 */
public class Memento implements MementoToCareTaker,
MementoToOriginator {


	private int firstNumber;
	private int secondNumber;


	public Memento(int firstNumber, int secondNumber) {
		this.firstNumber = firstNumber;
```

```
            this.secondNumber = secondNumber;
      }

      @Override
      public int getFirstNumber() {
            return firstNumber;
      }

      @Override
      public int getSecondNumber() {
            return secondNumber;
      }
}
```

Continuamos con la interfaz Originator. Notad que:

- El método backupLastCalculation() es el método encargado de crear el Memento con el estado actual del Originator.
- El método restorePreviousCalculation() es el método responsable de restaurar el Originator a su estado anterior.

Originator.java

```
package comportamiento.memento.calculadora;

/**
 * Originator Interface
 */
public interface Originator {

      // Create Memento
```

```
	public MementoToCareTaker backupLastCalculation();


	// setMemento
	public void restorePreviousCalculation(MementoToCareTaker
memento);


	// Actual Services Provided by the originator
	public int getCalculationResult();
	public void setFirstNumber(int firstNumber);
	public void setSecondNumber(int secondNumber);
}
```

Ahora veamos la clase de implementación para el Originator:

OriginatorImp.java

```
package comportamiento.memento.calculadora;


/**
 * Originator Implementation
 */
public class OriginatorImp implements Originator {


	private int firstNumber;
	private int secondNumber;


	@Override
	public MementoToCareTaker backupLastCalculation() {
		// create a memento object used for restoring two numbers
		return new Memento(firstNumber,secondNumber);
	}
```

```
	@Override
	public int getCalculationResult() {
		// result is adding two numbers
		return firstNumber + secondNumber;
	}

	@Override
	public void restorePreviousCalculation(MementoToCareTaker
memento) {
		this.firstNumber =
((MementoToOriginator)memento).getFirstNumber();
		this.secondNumber =
((MementoToOriginator)memento).getSecondNumber();
	}

	@Override
	public void setFirstNumber(int firstNumber) {
		this.firstNumber =  firstNumber;
	}

	@Override
	public void setSecondNumber(int secondNumber) {
		this.secondNumber = secondNumber;
	}
}
```

Finalmente, veamos el código para la clase cliente. Notad que la clase cliente realiza un “copia” del estado de Originator (obtiene un objeto Memento) a través de la interfaz MementoToCaretaker, por tanto, aunque quisiera no podría alterarlo de ninguna forma. En cambio, cuando la clase cliente quiere restaurar el Originator al estado anterior, sólo tiene una manera de hacerlo: invocando el método restorePreviousCalculator() del objeto Originator., pues él es el único que tiene acceso a la interfaz que permite manipular al objeto Memento.

MainClient.java

```
package comportamiento.memento.calculadora;

public class MainClient {

	public static void main(String[] args) {

		// program starts
		Originator calculator = new OriginatorImp();

		// assume user enters two numbers
		calculator.setFirstNumber(10);
		calculator.setSecondNumber(100);

		// find result
		System.out.println(calculator.getCalculationResult());

		// Store result of this calculation in case of error
		MementoToCareTaker memento =
calculator.backupLastCalculation();

		// user enters a number
		calculator.setFirstNumber(17);
		// user enters a wrong second number and calculates result
		calculator.setSecondNumber(-290);

		// calculate result
		System.out.println(calculator.getCalculationResult());

		// user hits CTRL + Z to undo last operation and see last
result
		calculator.restorePreviousCalculation(memento);
```

```
            // result restored
            System.out.println(calculator.getCalculationResult());
        }
}
```

Salida:

```
110
-273
```

Problemas específicos e implementación

Almacenar sólo cambios incrementales

Si diferentes objetos Memento, tanto en el momento de su creación como en el momento de enviarse al Originator para si restauración, siguen una secuencia predecible, entonces es posible llevar a cabo una implementación en la que sólo se guarden en los diversos objetos Memento los cambios diferenciales de todos ellos. Esto es una clara ventaja en cuanto a espacio de almacenamiento y tiempo de procesamiento.

Patrones relacionados

- Command: El patrón Command puede usar mementos para mantener el estado de las operaciones de deshacer (undo).
- Iterator: Se pueden utilizar objetos memento para iterar a través de objetos.