

Patrón Memento

*Hillary Caituiro Monge
Departamento de Ingeniería Eléctrica y de Computadoras
Universidad de Puerto Rico, Recinto Universitario de Mayagüez
Mayagüez, Puerto Rico 00681-9042
Noviembre del 2001
hillarycm@hotmail.com*

Resumen

El patrón de comportamiento Memento, un modelo para capturar y externalizar el estado interno de un objeto para restaurarlo más tarde sin violar el encapsulamiento. Puede ser usado en combinación con los Patrones Command e Iterator.

1. Introducción

Un patrón describe un problema frecuente y su solución, además de dar una solución un patrón debe nombrar, explicar y evaluar un diseño recurrente en sistemas OO. Los patrones conducen a diseños más simples y fáciles de usar. Los patrones de comportamiento caracterizan el modo en que interactúan las clases u objetos. Para seleccionar y usar un patrón debemos considerar el modo en que los patrones resuelven nuestro problema de diseño.

Muchas veces se quiere “retornar hacia atrás” retomando el mismo estado de ese instante al mismo tiempo, para lograrlo es necesario guardar el estado que se tenía en ese instante. Algunas personas quisieran poder ir al pasado y retomar su vida desde algún instante, en las mismas condiciones que tenían entonces.

Para obtener el estado interno de un objeto, este tiene que hacerlo público, exponiendo su estructura interna y haciendo visibles sus secretos, esto va en contra del principio del encapsulamiento.

Para dar solución a este problema existe el patrón *Memento*. *Memento* un patrón de comportamiento, que permite almacenar el estado interno de un objeto preservando el principio del encapsulamiento.

2. Clasificación

Patrón de comportamiento. Un patrón de comportamiento caracteriza el modo en que interactúan las clases u objetos.

3. Intención

Memento guarda parte o todo el estado interno de un objeto, para que este objeto pueda ser restaurado más tarde al estado guardado por *Memento*. Esta operación debe ocurrir sin romper el principio del encapsulamiento. [1]

4. 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.

Consideremos por ejemplo una aplicación de composición de figuras geométricas, donde el usuario hace sucesivas modificaciones a una composición, graficando nuevas líneas, círculos y rectángulos. Después de cierto tiempo, el usuario logra una composición “casi perfecta”, pero decide alcanzar la perfección, así que pinta una línea y esta no le sale como él esperaba. Definitivamente el usuario quisiera regresar al instante en que su “creación” era una obra de arte. Para dar solución a este problema, antes de que el usuario agregue una nueva figura geométrica a la composición, se debería guardar el estado de la composición y entonces siempre se tendría la posibilidad de regresar hacia atrás y restaurar la composición a su estado anterior.

Para lograr esto, sería necesario guardar la lista de figuras geométricas y el orden en que se encuentran en la composición, con información específica de cada una de ellas. En el caso de un círculo tendríamos que guardar la posición (x, y), el radio, color y relleno, para un rectángulo la posición (x, y), el ancho, el largo, color y relleno. Para lograr esto tenemos tres alternativas: la primera alternativa consiste en obtener la lista de figuras de la composición y obtener su estado, esto sería muy complejo y además va en contra del principio de encapsulamiento; la segunda alternativa, es que la composición se encargue de ir guardando su estado interno cada vez, esta no es una buena alternativa, la clase sería muy compleja y estaría asumiendo responsabilidades que no le corresponden; la tercera alternativa es la mejor, composición (Originator) crea un objeto (Memento) y almacena su estado interno en él, la aplicación (Caretaker) mantiene una lista de los objetos (Memento), de tal manera que cuando el usuario realice una operación de “deshacer”, la aplicación restaure el estado de la composición (Originator), con lo almacenado por el objeto Memento.

5. Aplicabilidad

El patrón *Memento* es aplicable cuando:

- Todo o parte del estado de un objeto debe ser guardado para ser restaurado más tarde.
- Cuando una interfaz directa para obtener el estado de un objeto exponga detalles de su implementación.

6. Estructura

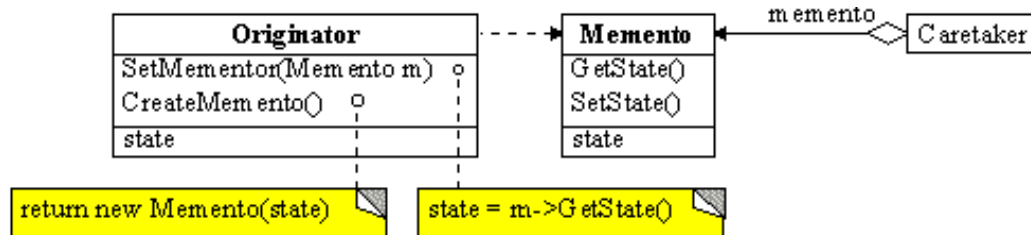


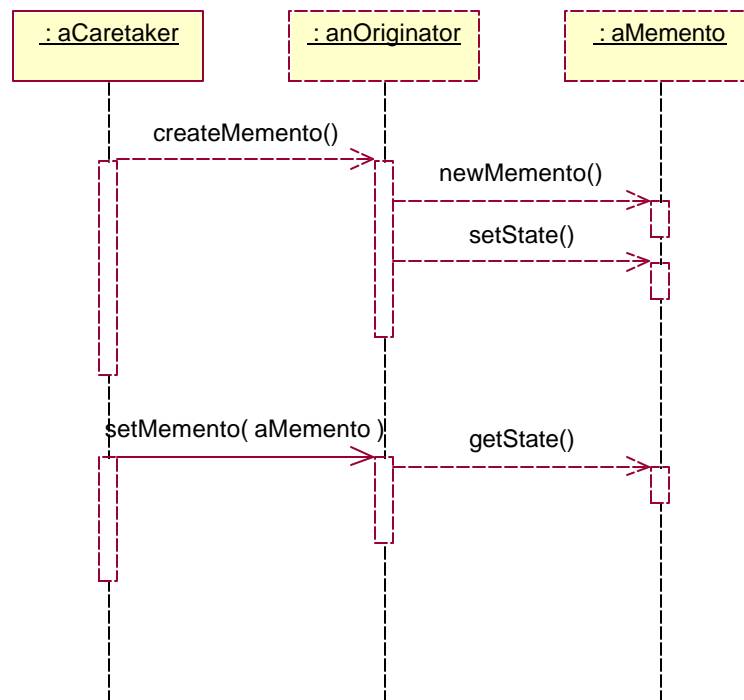
Figura 1. Estructura del Patrón de Diseño *Memento*.

7. Participantes

- **Memento.**
 - Almacena el estado interno de un objeto *Originator*. El *Memento* puede almacenar mucho 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, al menos en teoría.

- **Originator.**
 - *Originator* crea un objeto *Memento* conteniendo una fotografía de su estado interno.
 - *Originator* usa a *Memento* para restaurar su estado interno.
- **Caretaker**
 - Es responsable por mantener a salvo a *Memento*.
 - No opera o examina el contenido de *Memento*.

8. Colaboraciones



9. Consecuencias

- *Originator* crea un *Memento* y el mismo almacena su estado interno, de esta manera no es necesario exponer el estado interno como atributos de acceso público, preservando así la encapsulación.
- Si *Originator* tendría que almacenar y mantener a salvo una o muchas copias de su estado interno, sus responsabilidades crecerían y serían más complejas, se desviaría de su propósito disminuyendo la coherencia. Usar *Mementos* hace que *Originator* sea mucho más sencillo y coherente.
- El uso frecuente de *Mementos* para almacenar estados internos de gran tamaño, podría resultar costoso y perjudicar el rendimiento del sistema.
- *Caretaker* al no conocer detalles del estado interno de *Originator*, no tiene idea de cuanto espacio y tiempo se necesita para almacenar el estado interno de *Originator* en un *Memento* y restaurar su estado interno a partir de un *Memento*. Por lo que no puede hacer predicciones de tiempo ni de espacio.

- *Memento* debería proporcionar una interfaz privada, a la que sólo *Originator* puede acceder. Esta interfaz incluye la creación, el almacenamiento y recuperación del estado interno de *Originator*. Además una interfaz pública que permita la destrucción de *Memento*.

10. Implementación

Para implementar este patrón, debemos considerar que el *Memento* debe proporcionar una interfaz accesible sólo por *Originator*, en la mayoría de los lenguajes de programación, esta figura no es posible, sin embargo en C++ se puede hacer que *Originator* sea una clase amiga de *Memento* tal como se aprecia en la Figura 2, y con esto lograr que tenga acceso a la interfaz privada de *Memento*.

```
private:
    friend class Originator;
```

Figura 2. *Originator* declarado como clase amiga de *Memento* en C++ .

```
// Clase Originator.
class Originator {
    private State state;

    // crear un Memento
    public Memento createMemento() {
        Memento memento = new Memento();
        memento.setState( state );
        return memento;
    }

    // restaurar el estado de Originator a partir de un Memento
    public void setMemento( Memento m ){
        state = m.getState();
    }
}
```

Orig
y pro

nto(),

```

// Clase Memento
class Memento {
    private State state;

    // constructor. Sólo Originator debería poder crear un Memento
    Memento() {
    }

    // almacena estado en Memento. Accesible sólo por Originator
    public void setState( State state )
    {
        this.state = state;
    }

    // obtiene el estado de Memento. Accesible sólo por Originator
    public String getState()
    {
        return state;
    }
}

```

Figura 5. Implementación de *Memento* en Java.

Memento proporciona un método `setState(State)` para almacenar el estado interno de *Originator* y un método `getState()` para que *Originator* recupere su estado interno. En Java no es posible hacer que solo *Originator* acceda es esta interfaz.

11. Código ejemplo^[2]

```

public class Originator{
    private String name;
    private int value;

    public void setName(String name){
        this.name = name;
    }

    public void setValue(int value){
        this.value = value;
    }

    public String toString() {
        return name + ": " + value;
    }

    public void setMemento(Memento m){
        name = m.getName();
        value = m.getValue();
    }

    public Memento createMemento() {
        return new Memento(name, value);
    }
}

```

```

public class Memento
{
    String name;
    int value;

    public Memento(String name, int value)
    {
        this.name = name;
        this.value = value;
    }

    public String getName()
    {
        return name;
    }

    public int getValue()
    {
        return value;
    }
}

ArrayList caretaker = new ArrayList();

Originator o = new Originator();
o.setName("Barry");
o.setValue(10);
caretaker.add(o.createMemento());
System.out.println(o.toString());
o.setValue(20);
System.out.println(o.toString());
o.setMemento((Memento)caretaker.get(caretaker.size()-1));
System.out.println(o.toString());
}
}

```

Figura 8. Código ejemplo de *Caretaker* en Java.

12. Ejemplo no-software “Moviendo un ejercito”

Este ejemplo describe el procedimiento utilizado por un comité de oficiales de alto rango, encargado del desplazamiento de un ejercito, donde solo el oficial de logística conoce información secreta sobre el movimiento y lo guarda en una caja de seguridad que es solicitada y protegida por un guardia, quién la mantiene y devuelve cuando es necesario, este no puede acceder u obtener el contenido de la caja de seguridad.

COMITE DE OFICIALES DE ALTO RANGO (CLIENT)

- Decide donde mover cada día un batallón

- Debe poder mover al ejército y deshacer los movimiento

OFICIAL DE LOGISTICA (ORIGINATOR)

- Mantiene pista de cada cambio en la ubicación
- Calcula la distancia movida
- Responsable de conseguir información de los movimientos realizados.

CAJA DE SEGURIDAD (MEMENTO)

- Mantiene información necesitada para localizar el ejercito
- Accesada solamente por el OFICIAL DE LOGISTICA

GUARDIA (CARETAKER)

- Protege LA CAJA DE SEGURIDAD
- Pide la CAJA DE SEGURIDAD del OFICIAL DE LOGISTICA, la mantiene por un tiempo, entonces la envía de regreso (cuando es requerido)
- No puede obtener información de la CAJA DE SEGURIDAD

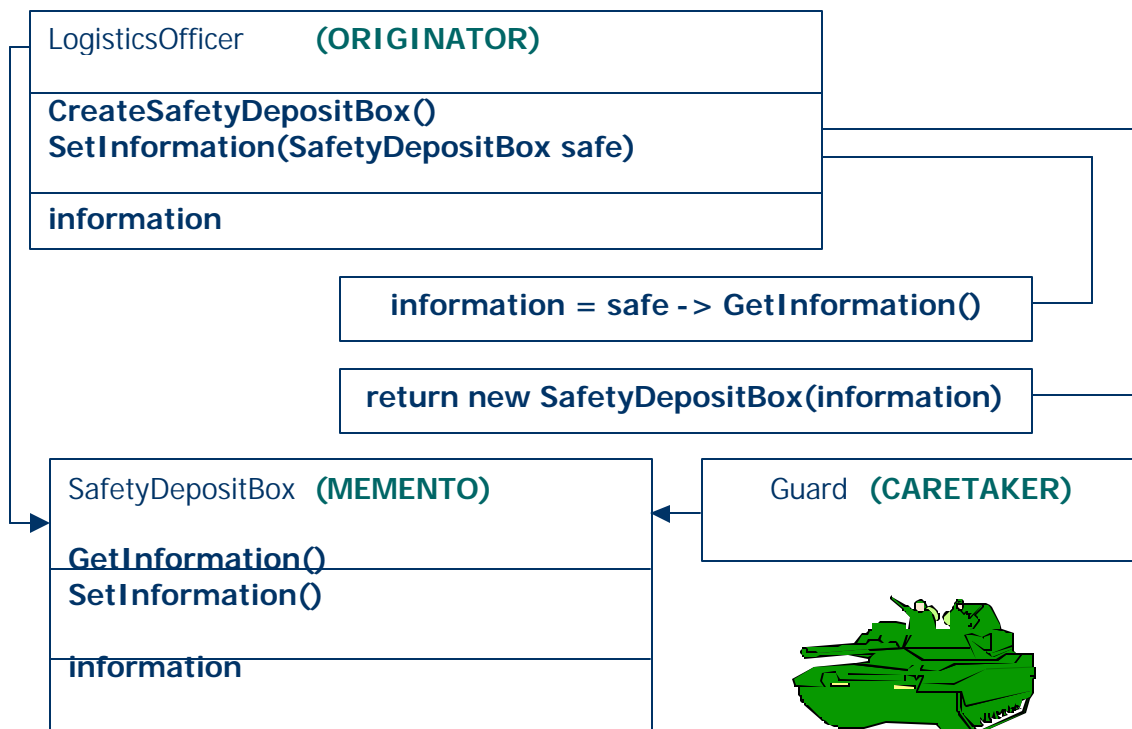


Figura 9. Moviendo un ejército.

13. Ejemplo no-software “Reparando los frenos de un auto” [3]

En mecanismos de “hágalo usted mismo”, cuando se van a reparar los frenos de un auto, se tiene el siguiente escenario. Las cubiertas de los frenos, son removidos de ambos lados, exponiendo los frenos de la derecha y de la izquierda. Pero solo el freno de un lado es desarmado, y el otro cara sirve como referencia (*Memento*) de como las partes del freno van juntas. Solamente después de que el trabajo haya sido completado en un lado el otro lado es desarmado. Cuando la Segunda cara es desarmada, la primera cara actúa como el *Memento*.

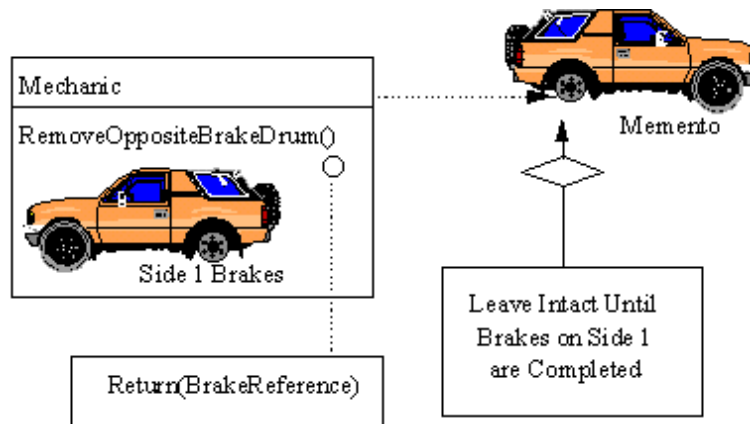


Figura 10: Diagrama de Objetos para *Memento* usando el ejemplo del freno

14. Usos conocidos

El Patrón de Diseño *Memento* se utiliza para situaciones donde se requiera una operación de restauración “UNDO”, como en los graficadores y procesadores de texto.

15. Patrones relacionados

- *Command* (233): Puede usar “*Mementos*” para guardar el estado de operaciones restaurables.
- *Iterator* (257): “*Mementos*” puede ser usado con *Iterator* para buscar colecciones para estados específicos.

16. Conclusión

Memento es una buena alternativa de diseño, para guardar el estado interno de un objeto preservando la encapsulación, aunque en la práctica no es posible preservar la encapsulación debido a que no es posible implementar en la mayoría de los lenguajes de programación orientados a objetos.

17. References

1. Erich Gamma, et. al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
2. Barry L. Geipel, *Design Patterns for Java*, <http://www.geipelnet.com/dpj/examples.htm>.
3. Michael Duell, *Non-Software Examples of Software Design Patterns*, "Object Magazine, Vol. 7, No. 5, July 1997, pp. 52-57.
4. *Pattern Depot*, <http://www.patterndepot.com/put/8/memento.pdf>
5. Jason Long, *Memento Pattern*, http://kachina.kennesaw.edu/~csis4650/Pattern_Lectures/memento/mem_lect.htm.