SPRINT Ciclo 4 G29 #1

Tema: Diseñadores Gráficos

Nombre: Edward Julián Sandoval

Fecha: 4/11/2020

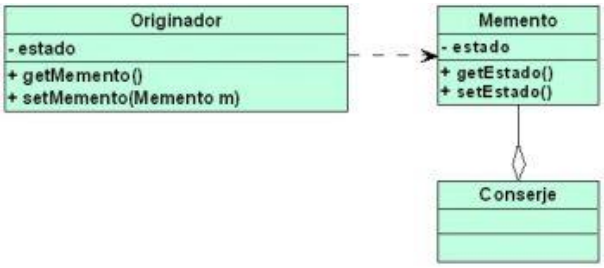
Grupo:#10

SPRINT #1

MEMENTO

Este método lo que hace es ayudar a que el usuario pueda guardar la información y/o los elementos que se realizaron en el objeto, esto con la intención de poder modificar y llevarlo al lugar donde este se encuentra guardado y lleve la última modificación que se hizo, de todas formas al guardar esto tendrá memoria de todas las modificaciones que se realizaron y este se guarde sin ningún problema.

Ejemplo #1:



Main.java:

```
package Memento;

public class Main
{
    public static void main(String[] args)
    {
        // Crear el objeto originador/creador
        Originator creador = new Originator("Pedro", "Gil Mena");

        // Crear el objeto gestor/vigilante del Memento
        Caretaker vigilante= new Caretaker();

        // Crear el Memento y asociarlo al objeto gestor
        vigilante.setMemento( creador.createMemento() );

        // Mostrar los datos del objeto
        System.out.println("Nombre completo: [" + creador.getNombre() + " " +
creador.getApellidos() + "]" );

        // Modificar los datos del objeto
        creador.setNombre("María");
        creador.setApellidos("Mora Miró");

        // Mostrar los datos del objeto
        System.out.println("Nombre completo: [" + creador.getNombre() + " " +
creador.getApellidos() + "]" );

        // Restaurar los datos del objeto
        creador.setMemento( vigilante.getMemento() );

        // Mostrar los datos del objeto
        System.out.println("Nombre completo: [" + creador.getNombre() + " " +
creador.getApellidos() + "]" );
    }
}
```

Originator.java:

```
package Memento;

public class Originator
{
    private String nombre;
    private String apellidos;

    // ---------------------------

    public Originator(String nombre, String apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
    }

    // ---------------------------

    public void setMemento(Memento m) {
        this.nombre = m.getNombre();
        this.apellidos = m.getApellidos();
    }

    // ---------------------------

    public Memento createMemento() {
        return new Memento(nombre, apellidos);
    }

    // ---------------------------

    public String getNombre() {
        return this.nombre;
    }

    // ---------------------------

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

    // ---------------------------

    public String getApellidos() {
        return this.apellidos;
    }

    // ---------------------------

    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }
}
```

Caretaker.java:

package Memento;

public class Memento

{private String nombre;

```
    private String apellidos;
    public Memento(String nombre, String apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;}
    public String getNombre() {
        return this.nombre;}
    public void setNombre(String nombre) {
        this.nombre = nombre;}
    public String getApellidos() {
        return this.apellidos;}
    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }
}
```

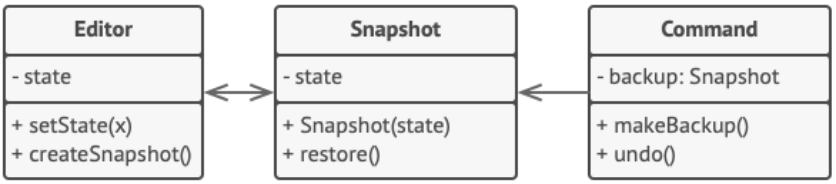
EXPLICACIÓN:

Al inicio del programa creamos un objeto de tipo Originator (que deseamos poder restaurar si se producen cambios) y otro de tipo Caretaker que utilizaremos a su vez para gestionar otro de tipo Memento (en el cual quedará registrado el estado anterior del primero de ellos, y que utilizaremos para restaurarlo).

A continuación se muestran los datos iniciales del objeto de tipo Originator, y tras modificarlos se volverán a restaurar al estado inicial.

Basándonos en este ejemplo, en la clase Caretaker también podríamos haber definido una lista en la que ir guardando una instancia de Memento por cada modificación que se realizase en el objeto de tipo Originator, para de dicho modo implementar la funcionalidad de restaurar los últimos cambios de uno en uno.

Ejemplo #2:




```
// El originador contiene información importante que puede
// cambiar con el paso del tiempo. También define un método para
// guardar su estado dentro de un memento, y otro método para
// restaurar el estado a partir de él.
class Editor is
    private field text, curX, curY, selectionWidth

    method setText(text) is
        this.text = text

    method setCursor(x, y) is
        this.curX = x
        this.curY = y

    method setSelectionWidth(width) is
        this.selectionWidth = width

    // Guarda el estado actual dentro de un memento.
    method createSnapshot():Snapshot is
        // El memento es un objeto inmutable; ese es el motivo
        // por el que el originador pasa su estado a los
        // parámetros de su constructor.
        return new Snapshot(this, text, curX, curY, selectionWidth)

// La clase memento almacena el estado pasado del editor.
class Snapshot is
    private field editor: Editor
    private field text, curX, curY, selectionWidth

    constructor Snapshot(editor, text, curX, curY, selectionWidth) is
        this.editor = editor
        this.text = text
        this.curX = x
        this.curY = y
        this.selectionWidth = selectionWidth

    // En cierto punto, puede restaurarse un estado previo del
    // editor utilizando un objeto memento.
    method restore() is
        editor.setText(text)
        editor.setCursor(curX, curY)
        editor.setSelectionWidth(selectionWidth)

// Un objeto de comando puede actuar como cuidador. En este
// caso, el comando obtiene un memento justo antes de cambiar el
// estado del originador. Cuando se solicita deshacer, restaura
// el estado del originador a partir del memento.
class Command is
    private field backup: Snapshot

    method makeBackup() is
        backup = editor.createSnapshot()
```

```
    method undo() is
        if (backup != null)
            backup.restore()
    // ...
```

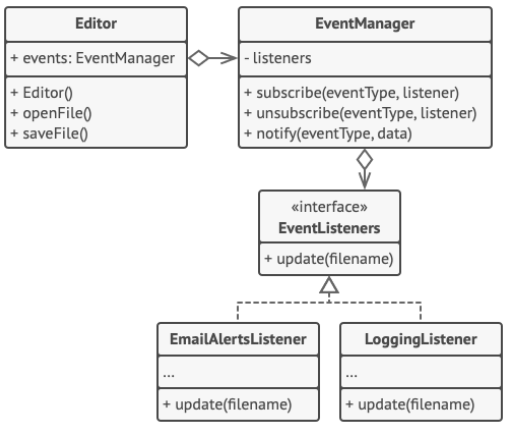
Los objetos de comando actúan como cuidadores. Buscan el memento del editor antes de ejecutar operaciones relacionadas con los comandos. Cuando un usuario intenta deshacer el comando más reciente, el editor puede utilizar el memento almacenado en ese comando para revertirse a sí mismo al estado previo.

La clase memento no declara ningún campo, consultor (getter) o modificador (setter) como público. Por lo tanto, ningún objeto puede alterar sus contenidos. Los mementos se vinculan al objeto del editor que los creó. Esto permite a un memento restaurar el estado del editor vinculado pasando los datos a través de modificadores en el objeto editor. Ya que los mementos están vinculados a objetos de editor específicos, puedes hacer que tu aplicación soporte varias ventanas de editor independientes con una pila centralizada para deshacer.

OBSERVER

Lo que hace este patrón de diseño es mediante un mecanismo de suscripción puede notificar a diferentes objetos sobre cualquier suceso o evento que suceda en este momento y para que lo observe esto con el fin de que no se pierda nada un ejemplo común de esto puede ser una suscripción en un canal de YouTube que lo que hacen las suscripciones notificar a las personas que otros videos ha hecho y recomendarnos si no notifica nada es porque el patrón de diseño observer no está funcionando.

Ejemplo #1:




```
// La clase notificadora base incluye código de gestión de
```

```
// suscripciones y métodos de notificación.
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        listeners.remove(eventType, listener)

    method notify(eventType, data) is
        foreach (listener in listeners.of(eventType)) do
            listener.update(data)

// El notificador concreto contiene lógica de negocio real, de
// interés para algunos suscriptores. Podemos derivar esta clase
```

```
// de la notificadora base, pero esto no siempre es posible en
// el mundo real porque puede que la notificadora concreta sea
// ya una subclase. En este caso, puedes modificar la lógica de
// la suscripción con composición, como hicimos aquí.
class Editor is
    public field events: EventManager
    private field file: File

    constructor Editor() is
        events = new EventManager()

    // Los métodos de la lógica de negocio pueden notificar los
    // cambios a los suscriptores.
    method openFile(path) is
        this.file = new File(path)
        events.notify("open", file.name)

    method saveFile() is
        file.write()
        events.notify("save", file.name)

// Aquí está la interfaz suscriptora. Si tu lenguaje de
// programación soporta tipos funcionales, puedes sustituir toda
// la jerarquía suscriptora por un grupo de funciones.

interface EventListener is
    method update(filename)

// Los suscriptores concretos reaccionan a las actualizaciones
// emitidas por el notificador al que están unidos.
class LoggingListener implements EventListener is
    private field log: File
    private field message

    constructor LoggingListener(log_filename, message) is
        this.log = new File(log_filename)
        this.message = message

    method update(filename) is
        log.write(replace('%s',filename,message))

class EmailAlertsListener implements EventListener is
    private field email: string

    constructor EmailAlertsListener(email, message) is
        this.email = email
        this.message = message

    method update(filename) is
        system.email(email, replace('%s',filename,message))

// Una  aplicación puede configurar notificadores y suscriptores
// durante el tiempo de ejecución.
class Application is
    method config() is
        editor = new Editor()

        logger = new LoggingListener(
            "/path/to/log.txt",
            "Someone has opened the file: %s")
        editor.events.subscribe("open", logger)
```

```
        emailAlerts = new EmailAlertsListener(
            "admin@example.com",
            "Someone has changed the file: %s")
        editor.events.subscribe("save", emailAlerts)
```

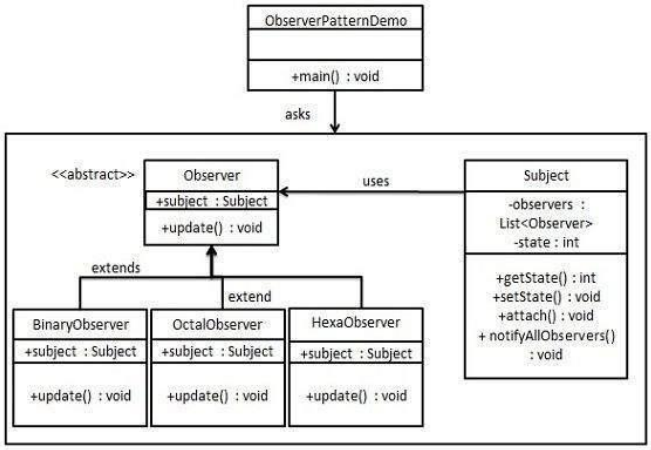
La lista de suscriptores se compila dinámicamente: los objetos pueden empezar o parar de escuchar notificaciones durante el tiempo de ejecución, dependiendo del comportamiento que desees para tu aplicación.

En esta implementación, la clase editora no mantiene la lista de suscripción por sí misma. Delega este trabajo al objeto ayudante especial dedicado justo a eso. Puedes actualizar ese objeto para que sirva como despachador centralizado de eventos, dejando que cualquier objeto actúe como notificador.

Añadir nuevos suscriptores al programa no requiere cambios en clases notificadoras existentes, siempre y cuando trabajen con todos los suscriptores a través de la misma interfaz.

Ejemplo #2:

El cambio de dólares determinadas divisas es decir que cuando esté haciendo el cambio donde terminado monto los observadores de las distintas divisas van a reaccionar para poder mostrar un mensaje.



App.java:

```
package com.mitocode;

import com.mitocode.observer.PesoARGObservador;

import com.mitocode.observer.PesoMXObservador;

import com.mitocode.observer.SolObservador;

import com.mitocode.observer.Subject;

public class App {

	public static void main(String[] args) {

		Subject subject = new Subject();
```

```
		new SolObservador(subject);
		new PesoARGObservador(subject);
		new PesoCOObservador(subject);
		System.out.println("Si desea cambiar 10 dólares obtendrá : ");
		subject.setEstado(10);
		System.out.println("-----------------");
		System.out.println("Si desea cambiar 100 dólares obtendrá : ");
		subject.setEstado(100);
	}
}
```

Observador.java:

```
package com.mitocode.observer;
public abstract class Observador {
	protected Subject sujeto;
	public abstract void actualizar();
}
```

PesoCOObservador.java:

```
package com.mitocode.observer;
public class PesoCOObservador extends Observador{
	private double valorCambio = 3.500;
	public PesoCOObservador(Subject sujeto) {
		this.sujeto = sujeto;
		this.sujeto.agregar(this);
	}

	@Override
	public void actualizar() {
		System.out.println("CO: " + (sujeto.getEstado() * valorCambio));
	}
```

```
}
```

PesoARGObservador.java:

```
package com.mitocode.observer;

public class PesoARGObservador extends Observador
	private double valorCambio = 29.86;
	public PesoARGObservador(Subject sujeto) {
		this.sujeto = sujeto;
		this.sujeto.agregar(this);
	}
	@Override
	public void actualizar() {
		System.out.println("ARG: " + (sujeto.getEstado() * valorCambio));
	}
}
```

Subject.java:

```
package com.mitocode.observer;

import java.util.ArrayList;
import java.util.List;
public class Subject {
	private List<Observador> observadores = new ArrayList<Observador>();
	private int estado;
	public int getEstado() {
		return estado;
	}
	public void setEstado(int estado) {
		this.estado = estado;
		notificarTodosObservadores();
	}
```

```
	public void agregar(Observador observador) {
		observadores.add(observador);
	}
	public void notificarTodosObservadores() {
		observadores.forEach(x -> x.actualizar());
	}

}
```