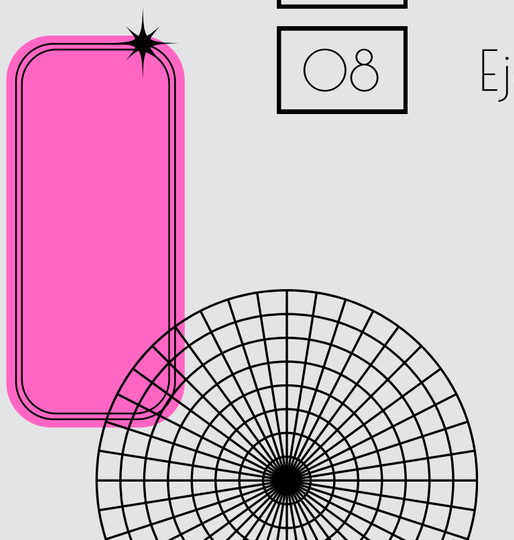
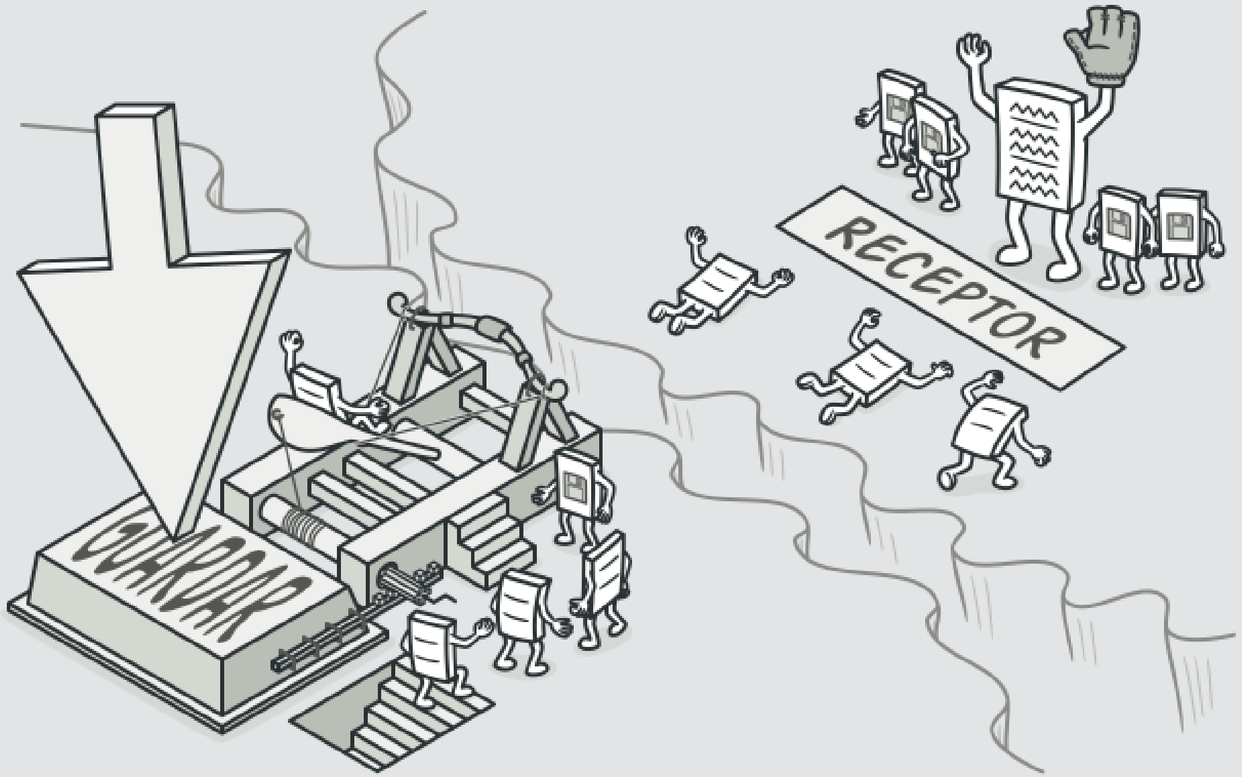
PATRONES DE DISEÑO

Command y memento

Equipo 8

Contenido



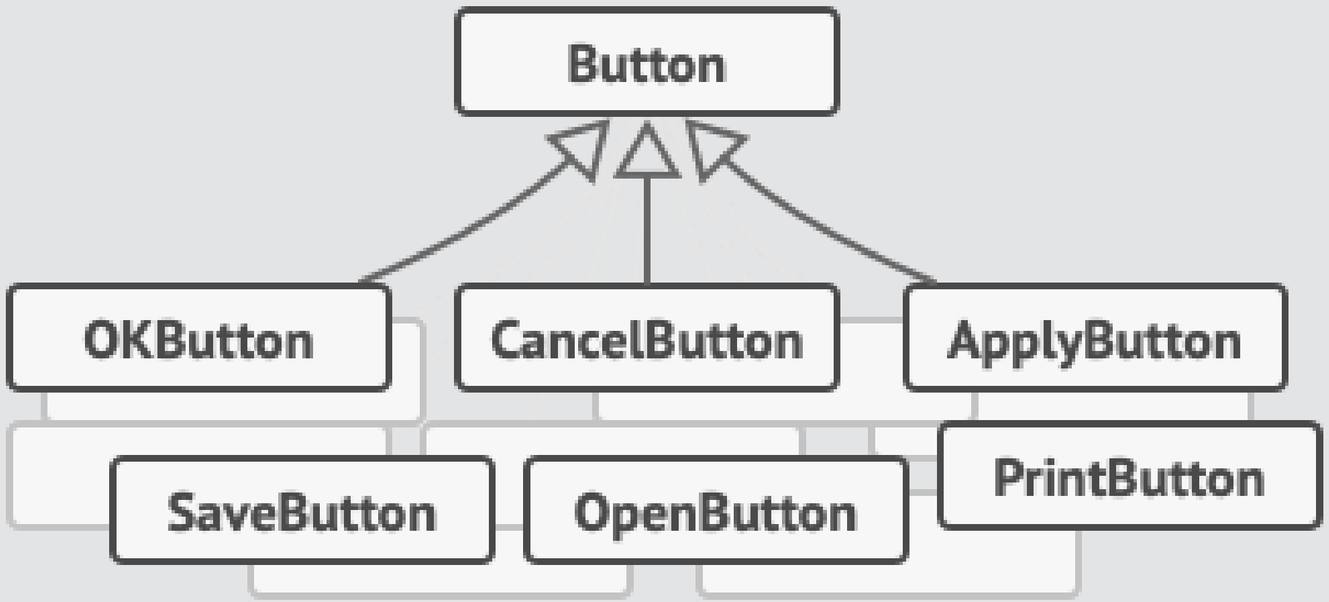


Command

Alias: Comando, Orden, Action, Transaction.

"El patrón Command es un patrón de diseño de comportamiento a nivel objeto. Cada instancia de un comando encapsula la lógica específica de una acción y se asocia con un objeto receptor particular." [2, p .253]

Problema



Dentro de un programa de cómputo podemos llegar a contener elementos que pueden llegar a ser casi idénticos, pero la funcionalidad es lo que los separa el uno del otro, por ejemplo, tenemos una clase Botón dentro de un programa que puede ser usado en cualquier momento de esta.

Ahora supongamos que tenemos muchos botones como: “Guardar”, “Cerrar”, “Cancelar”, “Imprimir”, etc. Todos son botones, pero no todos hacen lo mismo y aquí es donde se presenta el problema.

Problema

Podríamos pensar que la solución correcta es hacer varias subclases para cada tipo de operación que se requiera al usar el botón, pero mientras más botones tengamos estaríamos terminando con demasiadas subclases que corren el riesgo de estropearse si se modifica la clase Botón.

Solución

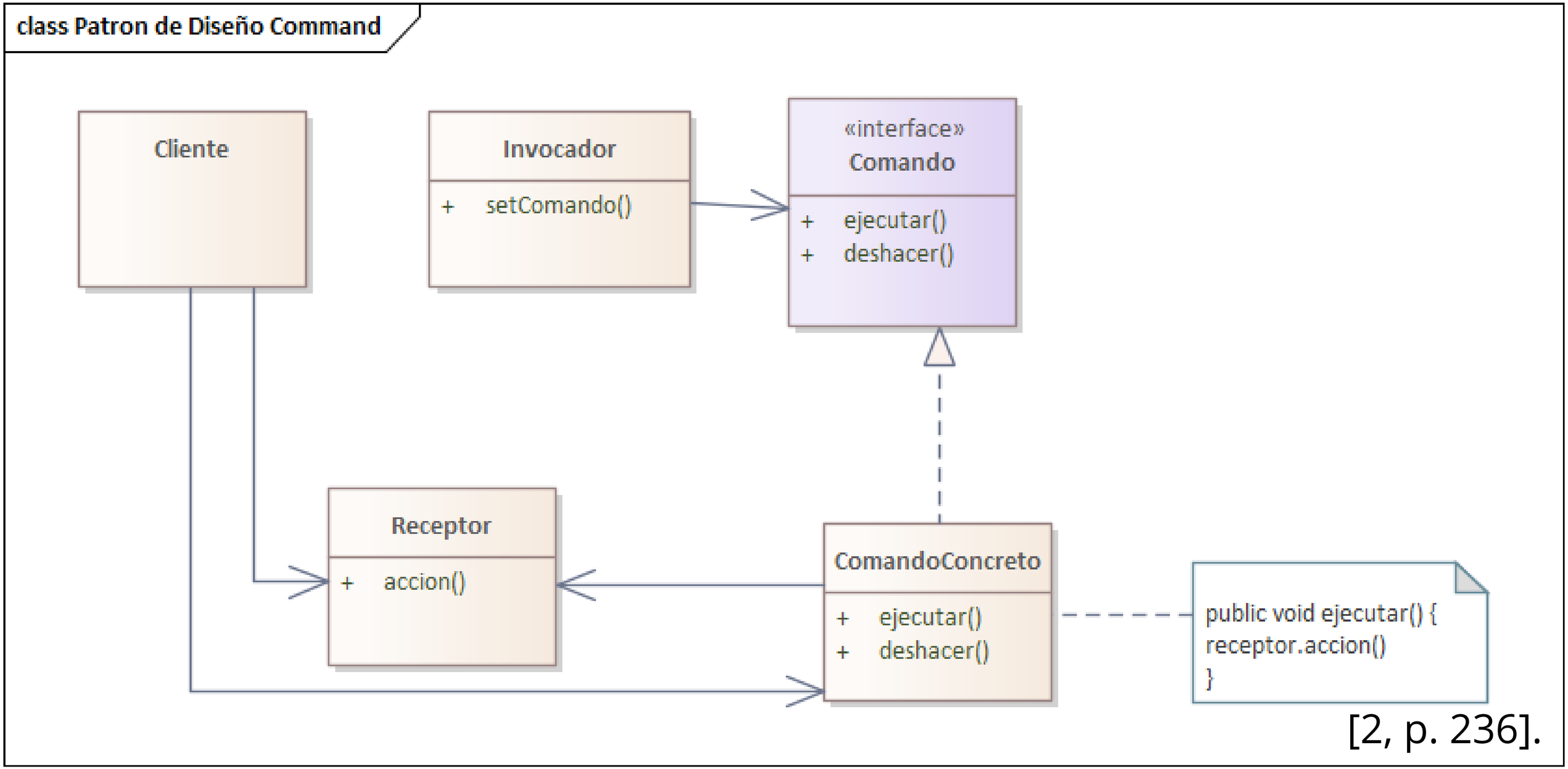
- El patrón Command **encapsula** la lógica de cada acción en objetos independientes llamados "comandos", evitando la creación de subclases para comportamientos específicos. Cada comando contiene la información necesaria para ejecutar su acción, como el receptor y los argumentos.

- La estructura incluye el **Cliente**, que solicita la ejecución de una operación; el **Invocador**, que asigna la solicitud a un comando concreto; el **Comando**, que define una interfaz común para todas las subclases de comandos; y el **Receptor**, que ejecuta la acción asociada. Este enfoque promueve la desacoplación entre remitente y receptor, brindando flexibilidad y extensibilidad al sistema.

Participantes y responsabilidades

- **Comando:** Clase que ofrece un interfaz para la ejecución de órdenes. Define los métodos ejecutar() y deshacer() que se implementarán en cada clase concreta.
- **ComandoConcreto:** Define un enlace entre una acción y un receptor. El invocador realiza una solicitud llamando a ejecutar() y ComandoConcreto la lleva a cabo llamando a una o más acciones en el Receptor.
- **Método ejecutar:** Invoca las acciones en el receptor necesarias para cumplir con la solicitud.
- **Cliente:** Es responsable de crear un ComandoConcreto y configurar su receptor.
- **Invocador:** Contiene un comando y en algún momento le pide al comando que lleve a cabo una solicitud llamando a su método ejecutar().
- **Receptor:** Sabe cómo realizar los trabajos necesarios para llevar a cabo la solicitud. Cualquier clase puede actuar como receptor.

Diagrama de clases

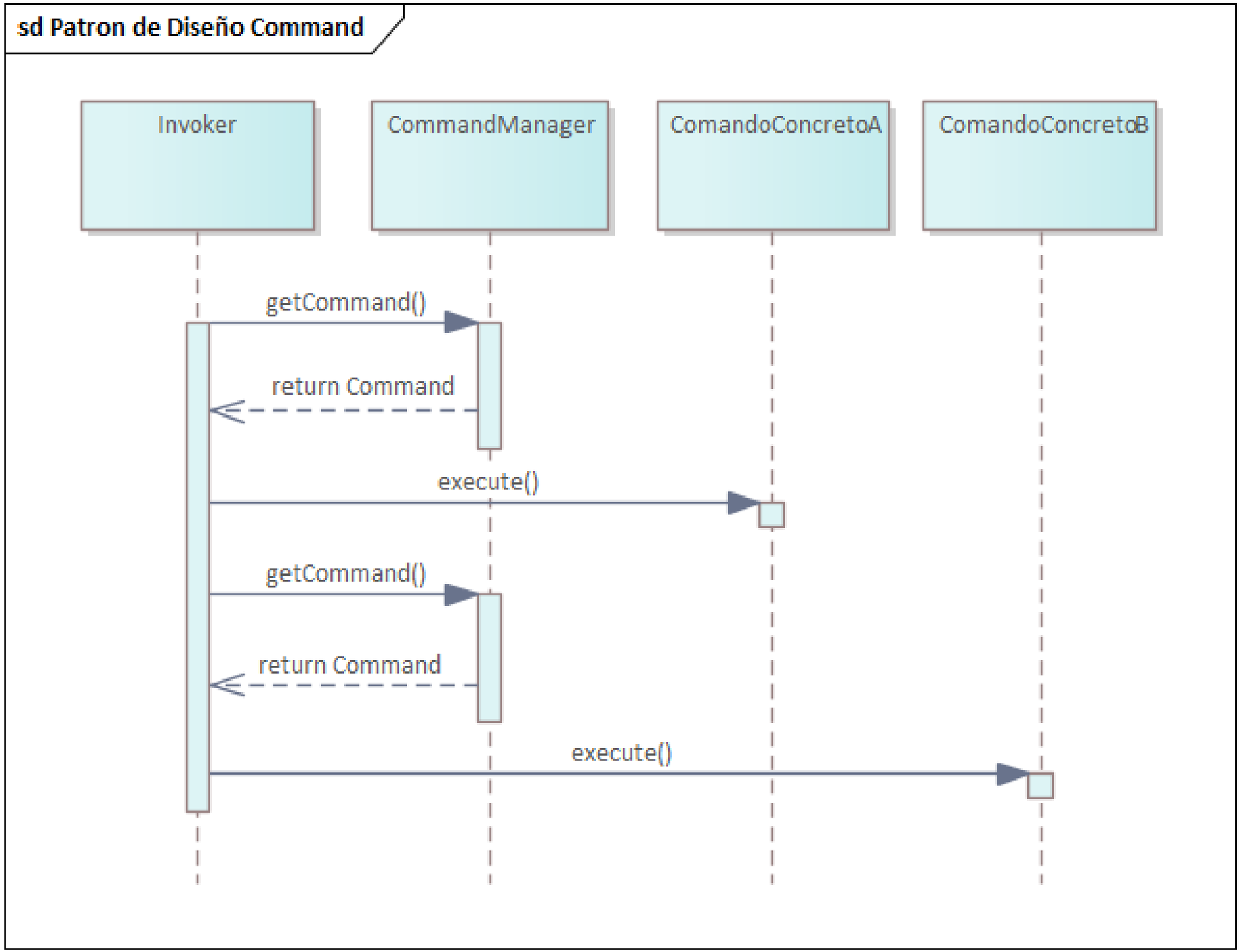


[2, p. 236].

Colaboraciones

- El invoker obtiene un Comando del CommandManager.
- El invoker ejecuta el comando.
- El invoker obtiene otro Comando del CommandManager.
- El invoker ejecuta el comando.

Diagrama de secuencia



[9].

Usos conocidos

- MacApp: Popularizó la noción de órdenes para implementar operaciones que podían deshacerse. [2]

- VLC Media Player: Utiliza el patrón Command para controlar la reproducción de audio y video. Los comandos se encapsulan en objetos Command y se envían al reproductor para su ejecución. [7]

- Los editores de texto suelen utilizar el patrón Command para implementar las acciones de los botones de la barra de herramientas. . [8]

Consecuencias

Ventajas

- Principio de **responsabilidad única**
- **Separación de interes**
- **Encapsula** lo que varía
- Programa para **interfaces,** no **implementaciones**
- Abierto para **extensión,** cerrado para modificaciones
- Macros
- Función de deshacer
- Registro de solicitudes

Desventajas

- El código puede **complicarse**, ya que estás introduciendo una nueva capa entre emisores y receptores.
- Posible impacto en el **rendimiento**: Debido a la creación y gestión de objetos adicionales.

Principios de POO aplicados

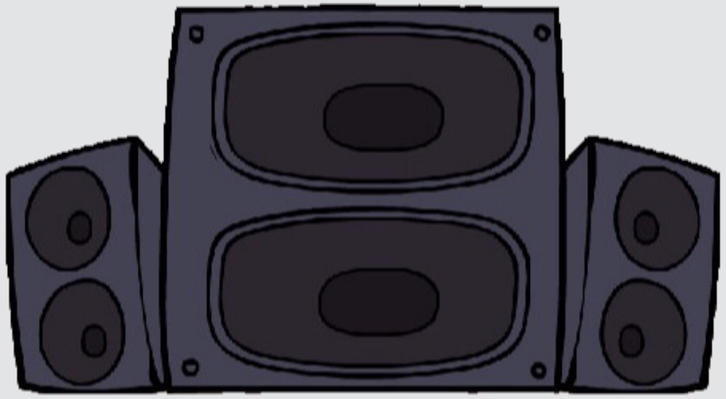
- Abstracción
- Encapsulación
- Polimorfismo

Patrones relacionados

El patrón Composite se puede utilizar junto con el patrón Command para construir estructuras jerárquicas de comandos.

El patrón Chain of Responsibility se puede utilizar para manejar solicitudes de comandos de manera secuencial.

Ejemplo de implementación



class CD_Control Remoto_Command Pattern

Diagrama de clases

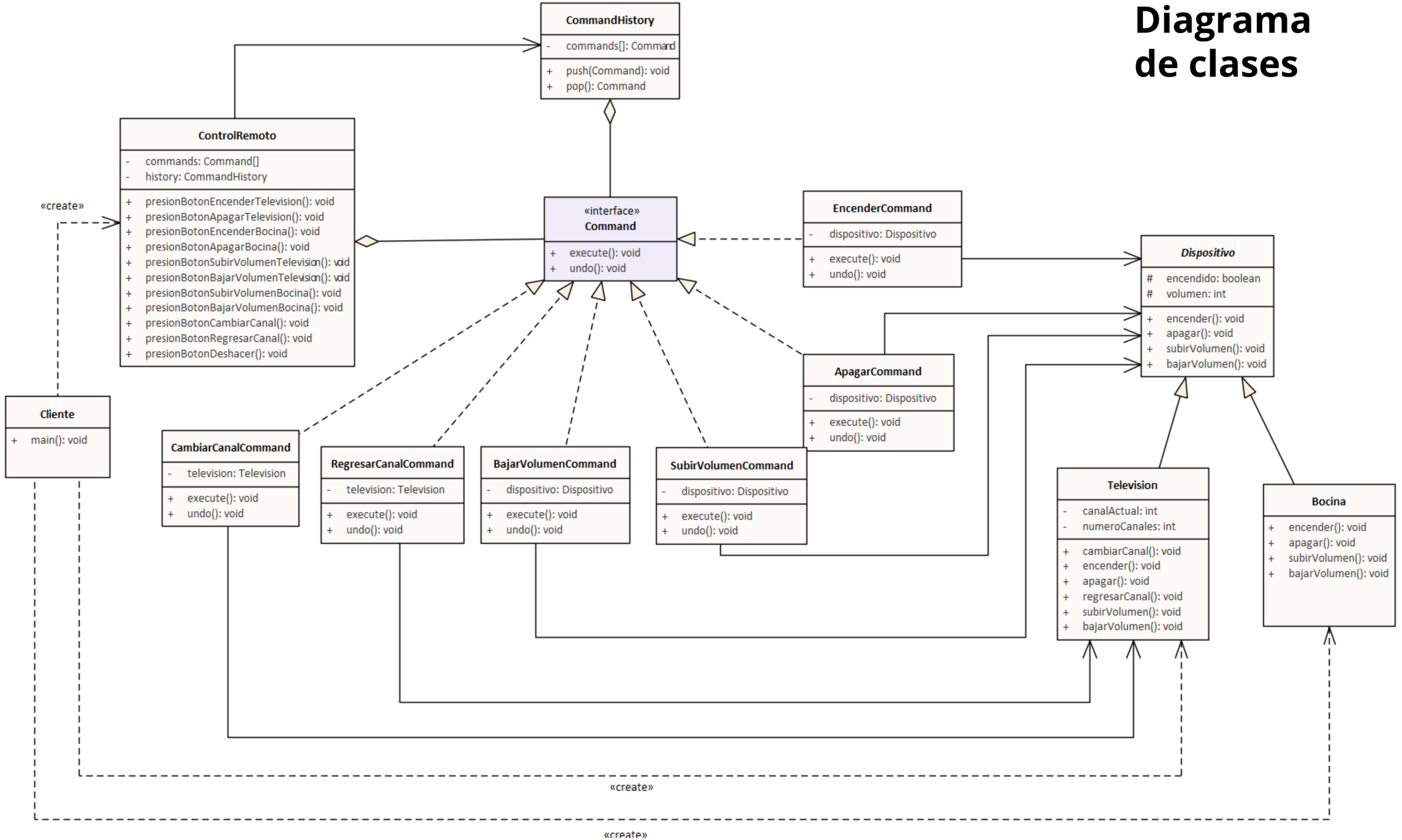
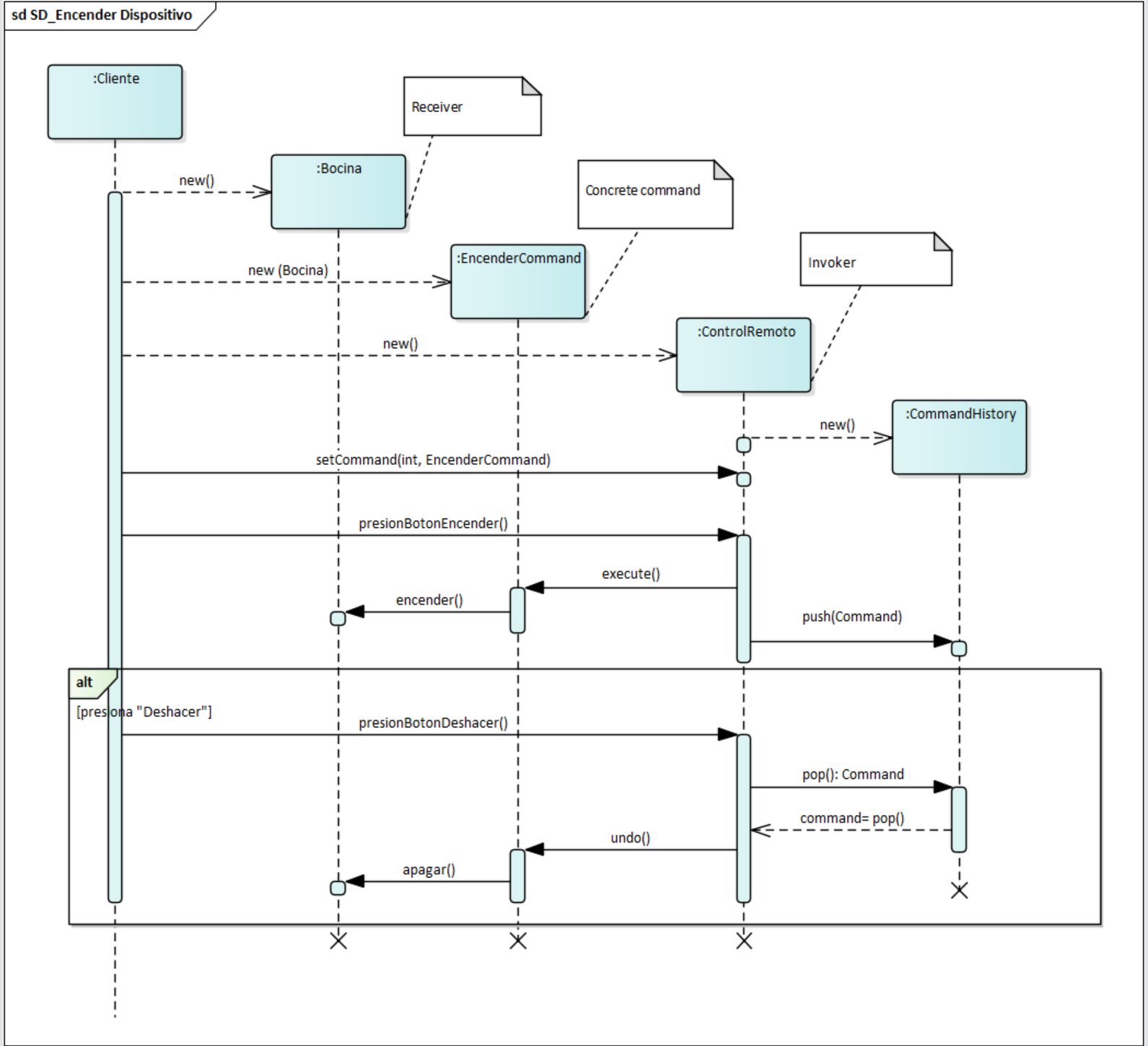


Diagrama de secuencia



```java
public class Bocina extends Dispositivo {

    public Bocina() {
        encendido = false;
        volumen = 0;
    }

    @Override
    public void encender() {
        encendido = true;
        System.out.println( x: "Se encendió la bocina");
    }

    @Override
    public void apagar() {
        encendido = false;
        System.out.println( x: "Se apagó la bocina");
    }

    @Override
    public void subirVolumen() {
        if (volumen < 100) {
            volumen += 1;
            System.out.println("Se subió el volumen de la televisión a: " + volumen);
        } else {
            System.out.println( x: "No se puede subir más el volumen de la bocina");
        }
    }

    @Override
    public void bajarVolumen() {
        if (volumen > 0) {
            volumen-= 1;
            System.out.println("Se bajó el volumen de la bocina a: " + volumen);
        } else {
            System.out.println( x: "No se puede bajar más el volumen de la bocina");
        }
    }
}
```

Reciever

Command

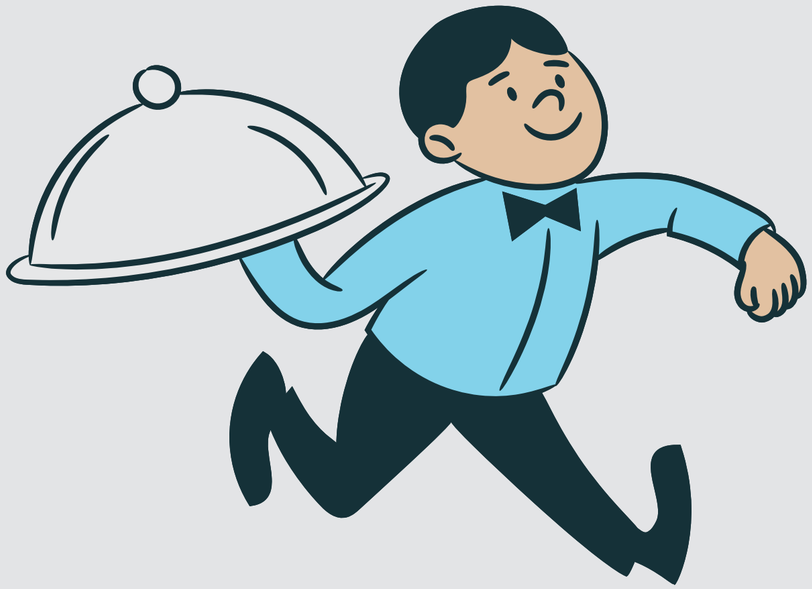
```java
public interface Command {

    public void execute();

    public void undo();
}
```

```java
public class EncenderCommand implements Command {

    public Dispositivo dispositivo;

    public EncenderCommand(Dispositivo dispositivo) {
        this.dispositivo = dispositivo;
    }

    @Override
    public void execute() {
        dispositivo.encender();
    }

    @Override
    public void undo() {
        dispositivo.apagar();
    }
}
```

```java
public class ControlRemoto {

    public Command[] command;
    CommandHistory history = new CommandHistory();

    public ControlRemoto() {
        command = new Command[10];
    }

    public void setCommand(int i, Command command) {
        this.command[i] = command;
    }

    public void presionBotonEncenderTelevision() {
        command[0].execute();
        history.push(command[0]);
    }

    public void presionBotonApagarTelevision() {
        command[1].execute();
        history.push(command[1]);
    }

    public void presionBotonEncenderBocina() {
        command[2].execute();
        history.push(command[2]);
    }
```

Invoker

```java
public void presionBotonDeshacer() {
    history.pop().undo();
}
```

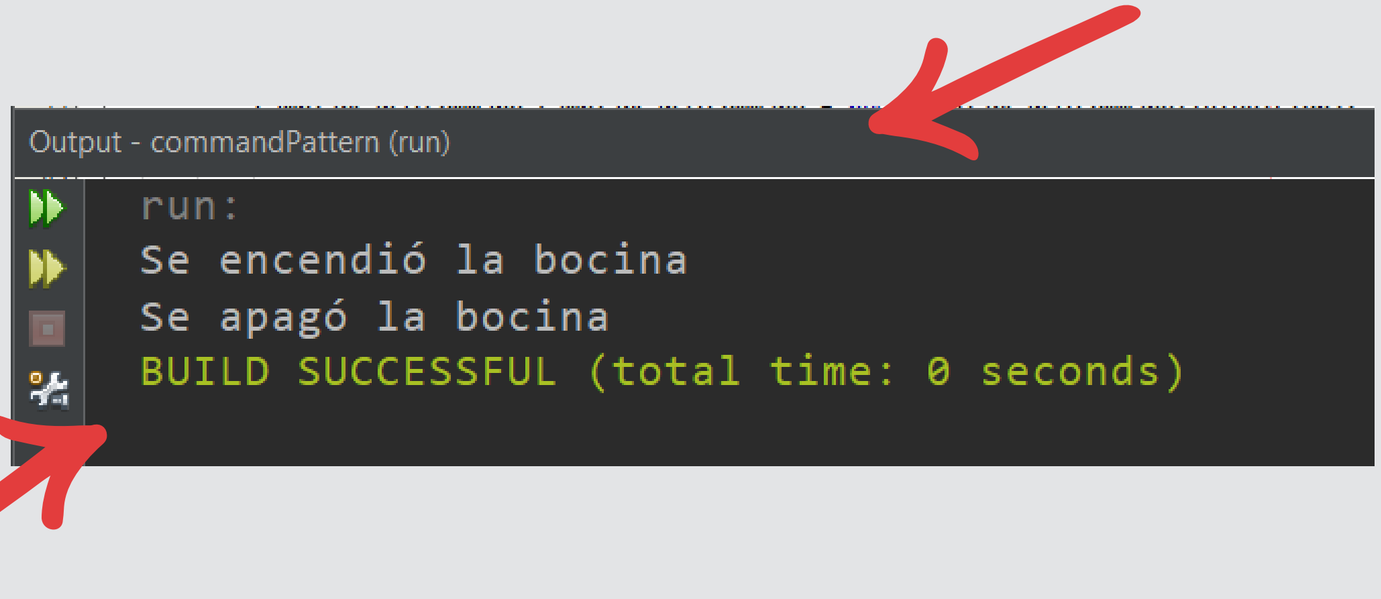


Client

```java
public class Cliente {

    public static void main(String[] args) {

        Bocina bocina = new Bocina();
        Television television = new Television(noCanales: 7);
        ControlRemoto controlRemoto = new ControlRemoto();

        EncenderCommand encenderCommandParaTV = new EncenderCommand(dispositivo: television);
        ApagarCommand apagarCommandParaTV = new ApagarCommand(dispositivo: television);
        EncenderCommand encenderCommandParaBocina = new EncenderCommand(dispositivo: bocina);
        ApagarCommand apagarCommandParaBocina = new ApagarCommand(dispositivo: bocina);
        SubirVolumenCommand subirVolumenCommandParaTV = new SubirVolumenCommand(dispositivo: television);
        BajarVolumenCommand bajarVolumenCommandParaTV = new BajarVolumenCommand(dispositivo: television);
        SubirVolumenCommand subirVolumenCommandParaBocina = new SubirVolumenCommand(dispositivo: bocina);
        BajarVolumenCommand bajarVolumenCommandParaBocina = new BajarVolumenCommand(dispositivo: bocina);
        CambiarCanalCommand cambiarCanalCommand = new CambiarCanalCommand(television);
        RegresarCanalCommand regresarCanalCommand = new RegresarCanalCommand(television);

        controlRemoto.setCommand(i: 0, command: encenderCommandParaTV);
        controlRemoto.setCommand(i: 1, command: apagarCommandParaTV);
        controlRemoto.setCommand(i: 2, command: encenderCommandParaBocina);
        controlRemoto.setCommand(i: 3, command: apagarCommandParaBocina);
        controlRemoto.setCommand(i: 4, command: subirVolumenCommandParaTV);
        controlRemoto.setCommand(i: 5, command: bajarVolumenCommandParaTV);
        controlRemoto.setCommand(i: 6, command: subirVolumenCommandParaBocina);
        controlRemoto.setCommand(i: 7, command: bajarVolumenCommandParaBocina);
        controlRemoto.setCommand(i: 8, command: cambiarCanalCommand);
        controlRemoto.setCommand(i: 9, command: regresarCanalCommand);

        controlRemoto.presionBotonEncenderBocina();
        controlRemoto.presionBotonDeshacer();

    }
}
```

Salida obtenida

presiona “Encender bocina”



presiona “Deshacer”

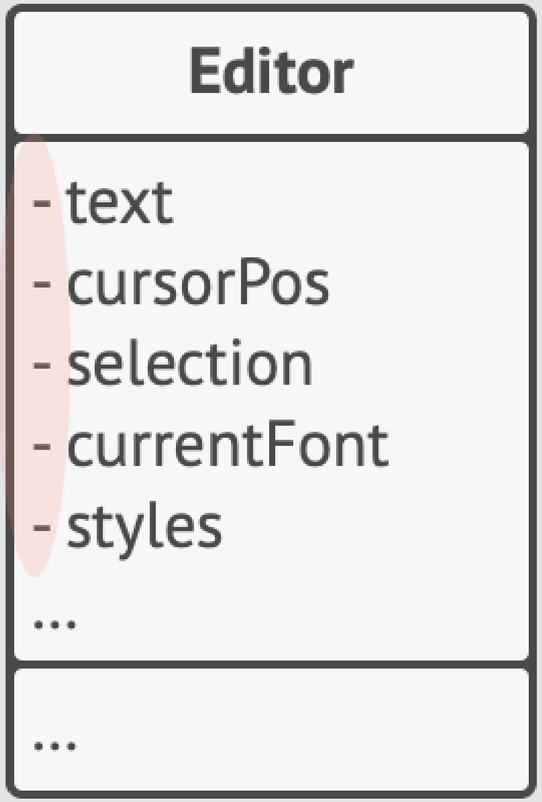
Memento

Alias: Recuerdo, instantánea, snapshot.

Es un patrón de **comportamiento** y con un alcance a nivel de **objetos** que permite guardar y restaurar el estado previo de un objeto sin violar su encapsulación.

private = (privado)
no se puede copiar

public = (público)
inseguro



Problema

En ocasiones es necesario registrar el estado de un objeto, permitiendo recuperar sus estados anteriores, pero los objetos encapsulan todo o la mayor parte de su estado haciendo que sea inaccesible a otros objetos e imposible de guardar de forma externa, además no se debe exponer el estado ya que violaría la encapsulación comprometiendo la confiabilidad.

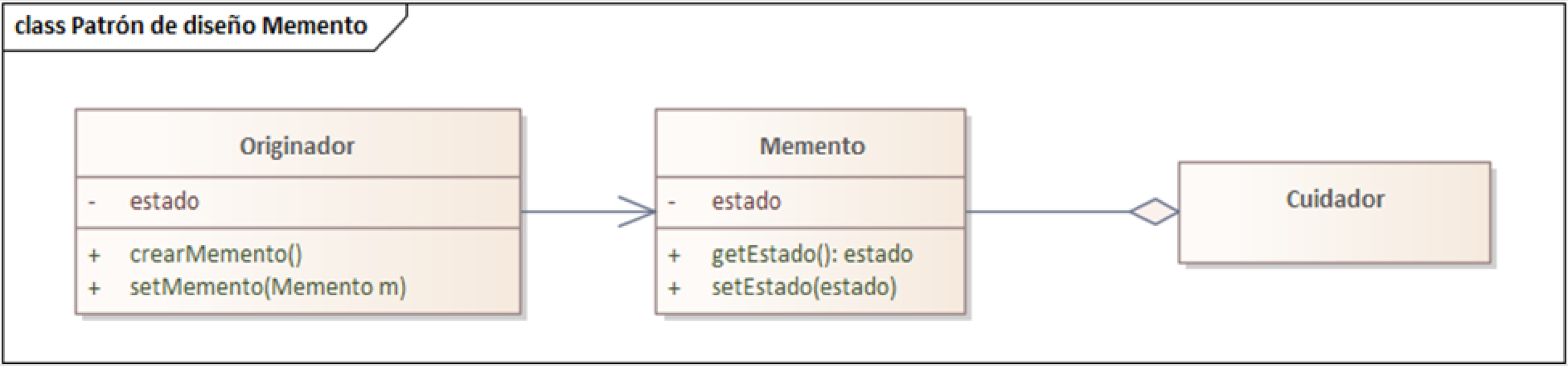
Solución

El patrón delega la creación de instantáneas de estado al propietario de ese estado llamado objeto **originador**, la copia del estado es realizada en un objeto especial llamada **memento** siendo accesible el estado de este solo por la clase originador. Para poder almacenar los mementos se hace uso del objeto **cuidador**.

Participantes y responsabilidades

- **Originador**: Crea un memento que contiene una instantánea de su estado interno en ese momento, además de poder utilizar a memento para restaurar su estado interno.
- **Memento**: Almacena el estado interno del objeto originador además de proteger contra el acceso de otros objetos.
- **Cuidador**: Es responsable de gestionar los mementos sin operar ni examinar su contenido.

Diagrama de clases

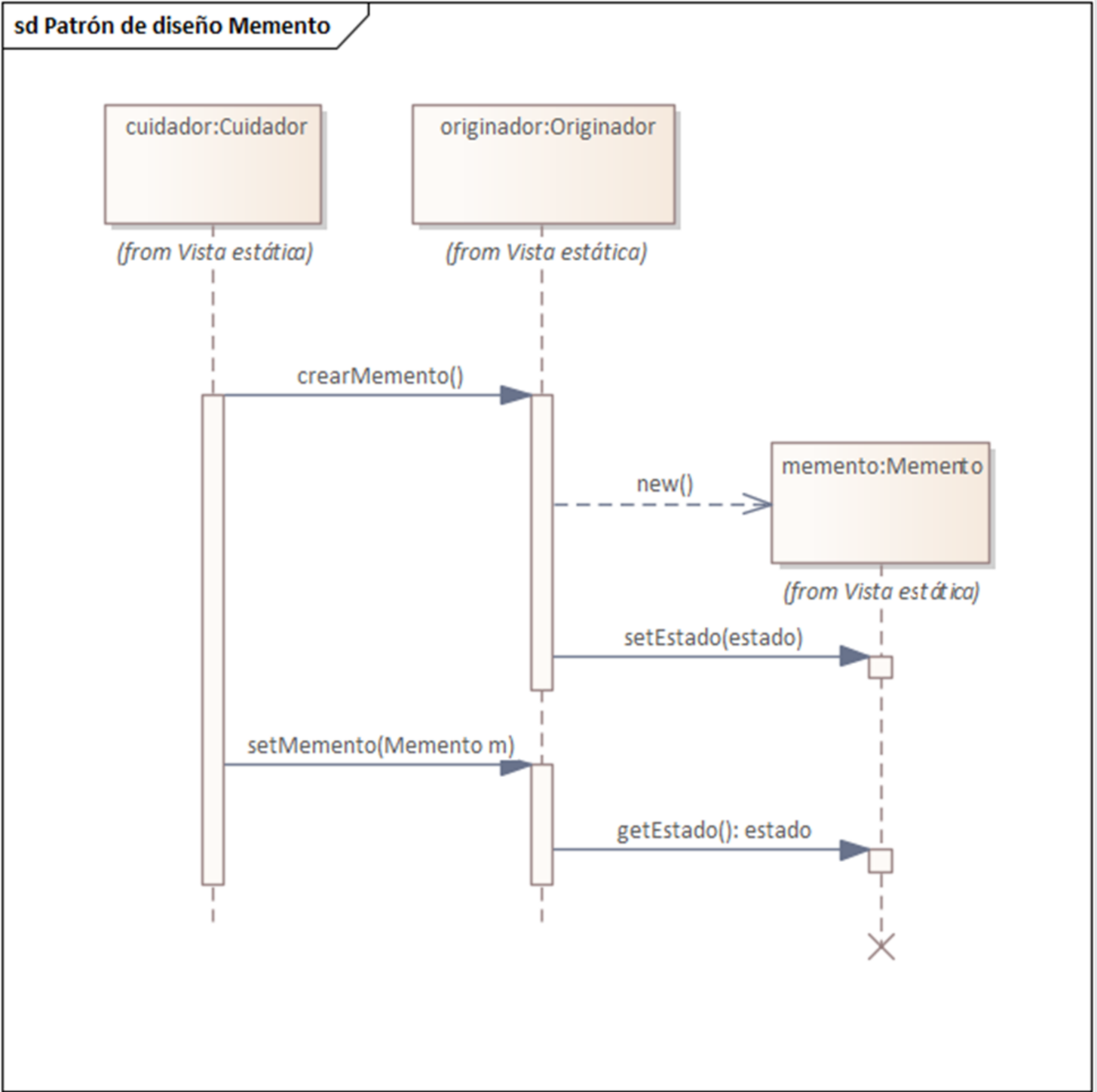


[2, p. 285]

Colaboraciones

- Un **cuidador** solicita un **memento** a un **originador**, lo retiene por un tiempo y se lo devuelve al **originador**.
- A veces, el **cuidador** no devuelve el **memento** al **originador**, porque es posible que el **originador** nunca necesite volver a un estado anterior.
- Los **mementos** no utilizan a otra clase. Sólo el **originador** que creó un **memento** asignará o recuperará su estado.

Diagrama de secuencia



[2, p. 286].

Usos conocidos

- “En las herramientas de Microsoft Office como Word o Power Point” [5].
- “En sistemas de edición de imagen como Photoshop” [5].
- “En sistemas transaccionales donde las operaciones puedan revertirse” [6].

Consecuencias

Ventajas

- “Mantener el estado guardado fuera del objeto clave ayuda a mantener la **cohesión**” [1, p. 637].
- “Puedes producir instantáneas del estado del objeto sin violar su **encapsulación**” [4, p. 357].
- “Puede simplificar el código de la originadora permitiendo que la cuidadora mantenga el historial del estado de la originadora” [4, p. 358].

Desventajas

- “La aplicación puede consumir mucha memoria RAM si los clientes crean mementos muy a menudo” [4, p. 358].
- “Las cuidadoras deben rastrear el ciclo de vida de la originadora para poder destruir mementos obsoletos” [4, p. 358].
- “Guardar y restaurar estados pueden llevar mucho tiempo” [1, p. 637].

Patrones relacionados

Command
El patrón command puede hacer uso de mementos para mantener el estado de las operaciones que se pueden hacer.

Iterator
Se puede capturar el estado de la iteración actual y reanudarla en caso de ser necesario.

Ejemplo de implementación

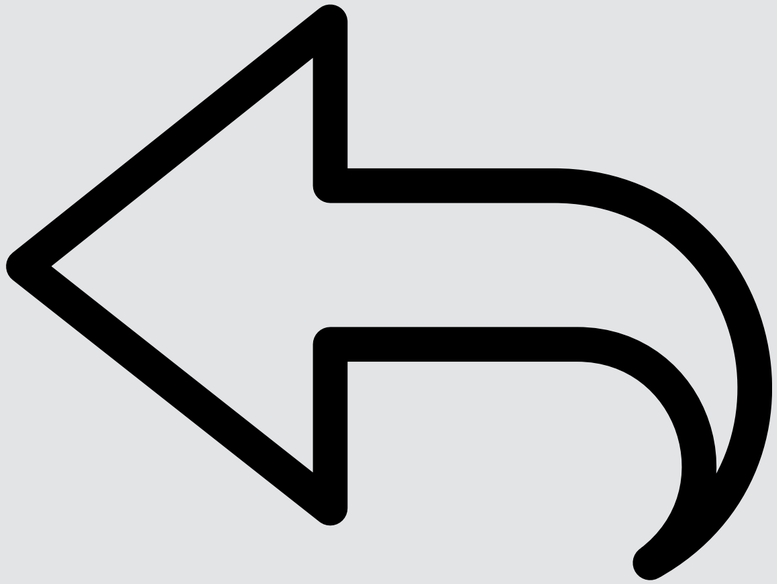


Diagrama de clases

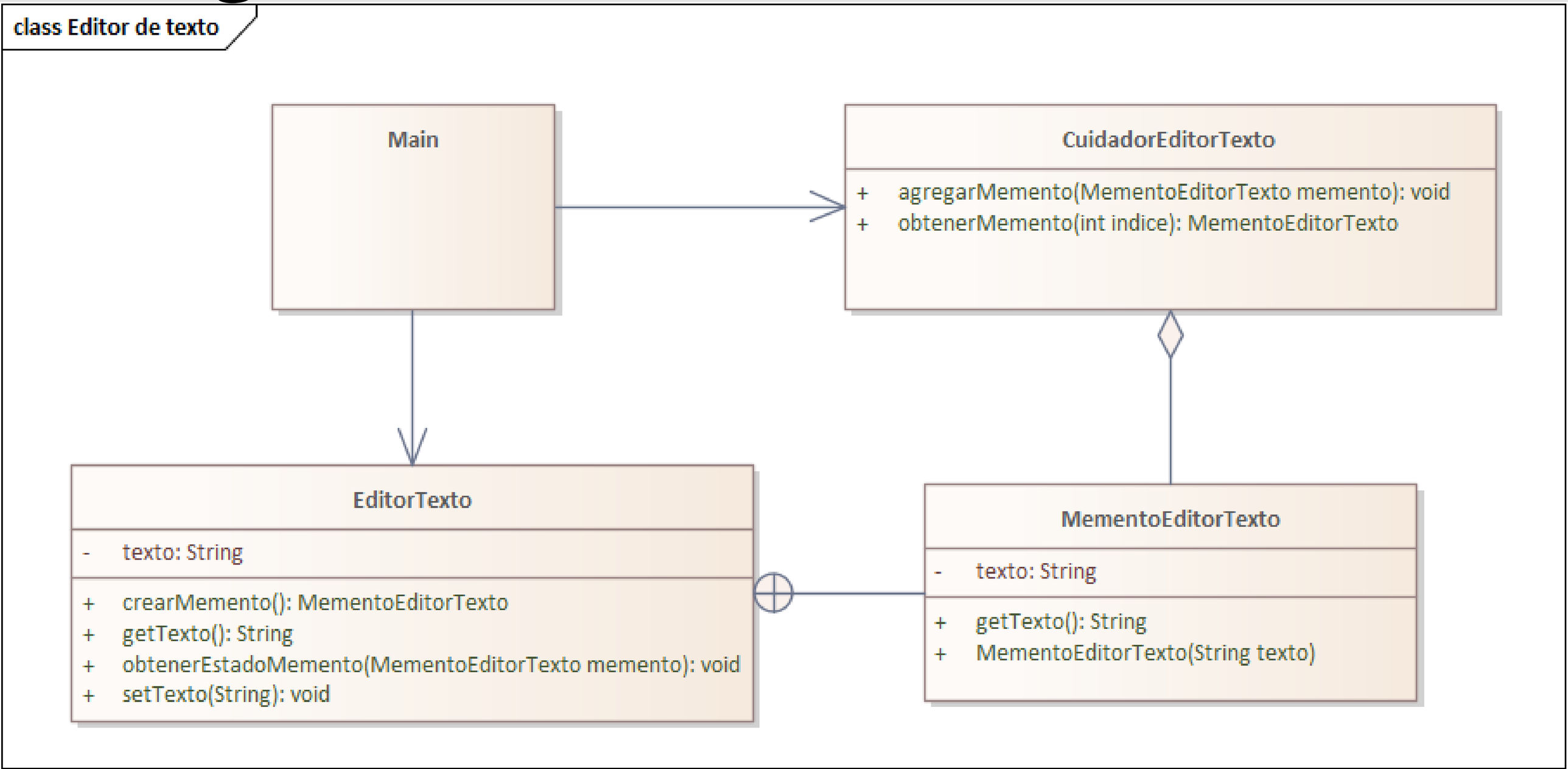
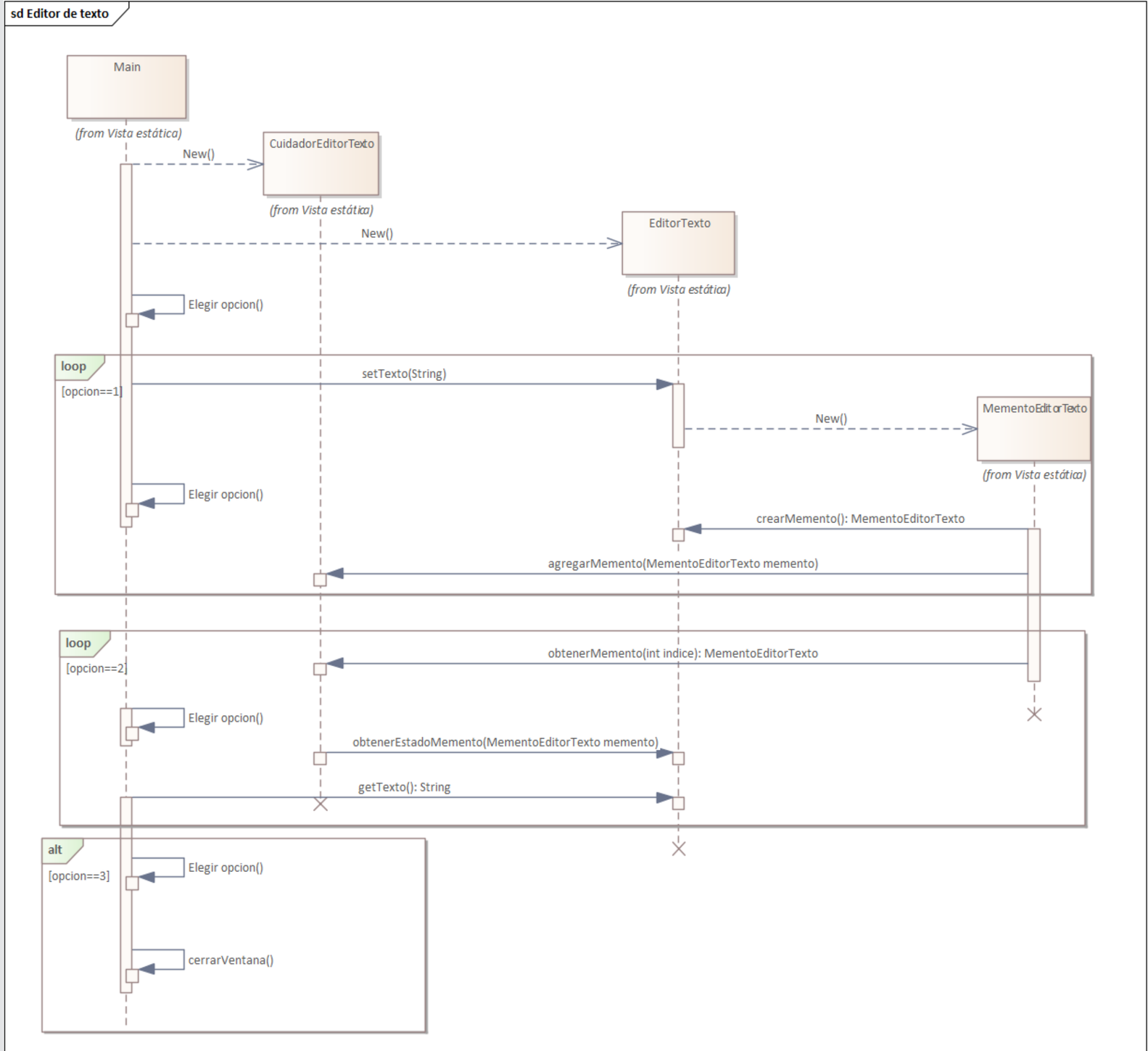


Diagrama de secuencia



Originador

```java
public class EditorTexto {
    private String texto;

    public String getTexto() {
        return texto;
    }

    public void setTexto(String texto) {
        this.texto = texto;
    }

    public MementoEditorTexto crearMemento(){
        return new MementoEditorTexto(texto);
    }

    public void obtenerEstadoMemento(MementoEditorTexto memento){
        this.texto=memento.getTexto();
    }
```

```java
public class MementoEditorTexto{
    private String texto;

    public MementoEditorTexto(String texto){
        this.texto=texto;
    }

    public String getTexto() {
        return texto;
    }
}
```

Memento

Cuidador

```
public class CuidadorEditorTexto {
    private List<MementoEditorTexto> mementos=new ArrayList<>();

    public void agregarMemento(MementoEditorTexto memento){
        this.mementos.add( e:memento);
    }

    public MementoEditorTexto obtenerMemento(int indice){
        return this.mementos.get( index:indice);
    }

}
```

```java
public class Main {
    public static void main(String[] args) {
        EditorTexto editorTexto=new EditorTexto();
        CuidadorEditorTexto cuidadorEditorTexto=new CuidadorEditorTexto();
        Scanner leer=new Scanner( in:System.in);
        String texto="";
        int opcion=0;
        int indice=0;
        editorTexto.setTexto(texto);
        cuidadorEditorTexto.agregarMemento( memento:editorTexto.crearMemento());
        do{
            if(indice<0||opcion==0){
                System.out.println( x:"Elige una opcion");
                System.out.println( x:"[1]Comenzar a escribir");
                System.out.println( x:"[3]Salir");
            }else{
                System.out.println( x:"Elige una opcion");
                System.out.println( x:"[1]Seguir escribiendo");
                System.out.println( x:"[2]Restaurar al ultimo estado");
                System.out.println( x:"[3]Salir");
            }
            opcion = Integer.parseInt( s:leer.nextLine());
```

```
switch(opcion){
    case 1:
        System.out.println( x:"Escribe y presiona enter al terminar: ");
        System.out.println( x:texto);
        texto+=leer.nextLine();
        texto+=" ";
        editorTexto.setTexto(texto);
        cuidadorEditorTexto.agregarMemento( memento:editorTexto.crearMemento());
        indice++;
    break;
    case 2:
        if(indice>=0){
            editorTexto.obtenerEstadoMemento( memento:cuidadorEditorTexto.obtenerMemento(indice));
            texto=editorTexto.getTexto();
            System.out.println( x:texto);
            indice--;
        }else{
            System.out.println( x:"No hay versiones anteriores");
        }
    break;
    case 3:
        System.out.println( x:"Hasta luego");
    break;
    default:
        System.out.println( x:"Opcion invalida");
    break;
```

Salida obtenida

```
Elige una opcion
[1]Comenzar a escribir
[3]Salir
1
Escribe y presiona enter al terminar:

Primera linea de prueba
Elige una opcion
[1]Seguir escribiendo
[2]Restaurar al ultimo estado
[3]Salir
1
Escribe y presiona enter al terminar:
Primera linea de prueba
Segunda linea de prueba
Elige una opcion
[1]Seguir escribiendo
[2]Restaurar al ultimo estado
[3]Salir
1
Escribe y presiona enter al terminar:
Primera linea de prueba Segunda linea de prueba
Tercera linea de prueba
Elige una opcion
[1]Seguir escribiendo
[2]Restaurar al ultimo estado
[3]Salir
2
```

```
Primera linea de prueba Segunda linea de prueba Tercera linea de prueba
Elige una opcion
[1]Seguir escribiendo
[2]Restaurar al ultimo estado
[3]Salir
2
Primera linea de prueba Segunda linea de prueba
Elige una opcion
[1]Seguir escribiendo
[2]Restaurar al ultimo estado
[3]Salir
2
Primera linea de prueba
Elige una opcion
[1]Seguir escribiendo
[2]Restaurar al ultimo estado
[3]Salir
2

Elige una opcion
[1]Comenzar a escribir
[3]Salir
3
Hasta luego
```

Referencias

- [1] K. Sierra, B. Bates, E. Freeman y E. Robson, Head First Design Patterns. O'Reilly Media, Inc., 2004.
- [2] R. Helm, E. Gamma, J. Vlissides y R. Johnson, Design patterns: Elements of reusable object-oriented software. Reading, Mass: Addison-Wesley, 1995.
- [3] V. Sarcar, Java Design Patterns: A Hands-On Experience with Real-World Examples. Apress, 2018.
- [4] A. Shvets, Sumérgete en los patrones de diseño. 2021.
- [5] “Memento”. Mi granito de java. Accedido el 20 de abril de 2024. [En línea]. Disponible: https://migranitodejava.blogspot.com/2011/06/memento.html
- [6] R. Fairushyn. “Mastering the Memento Design Pattern in C#/.NET”. LinkedIn: Log In or Sign Up. Accedido el 20 de abril de 2024. [En línea]. Disponible: https://www.linkedin.com/pulse/mastering-memento-design-pattern-cnet-roman-fairushyn-ojedf/
- [7] “Build software better, together”. GitHub. Accedido el 21 de abril de 2024. [En línea]. Disponible: Videolan. (2024). VLC Media Player. https://github.com/topics/vlc-media-player
- [8] “Design Patterns: Strategy”. Microsoft Learn: Build skills that open doors in your career. Accedido el 21 de abril de 2024. [En línea]. Disponible: Microsoft. (2024). Common Design Patterns. https://learn.microsoft.com/en-us/shows/visual-studio-toolbox/design-patterns-strategy
- [9] “Command”. Reactive Programming - arquitectura y desarrollo de software. Accedido el 22 de abril de 2024. [En línea]. Disponible: https://reactiveprogramming.io/blog/es/patrones-de-diseno/command