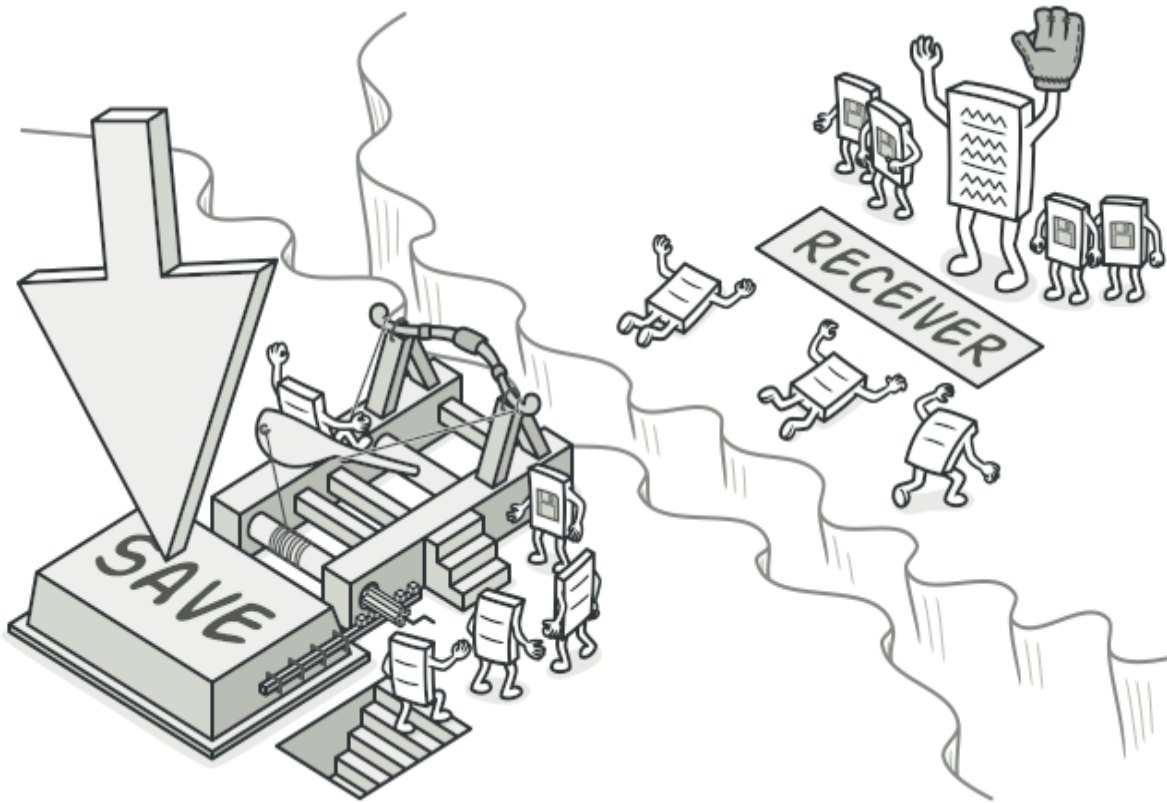

COMMAND

Complejidad: ★★ ★

Popularidad: ★ ☆ ☆

También conocido como **Action** o **Transaction**



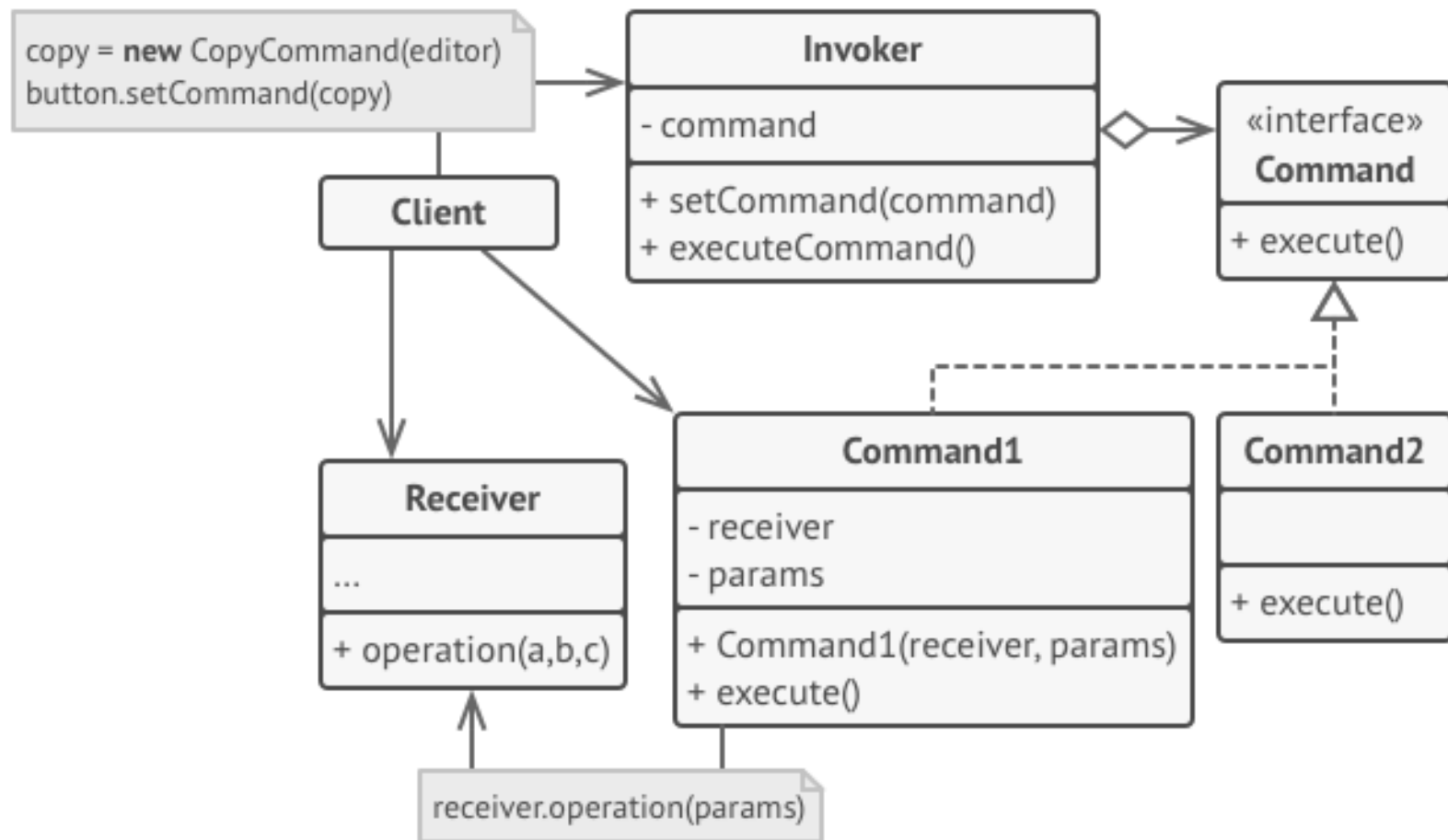
El comando es un patrón de diseño de comportamiento que convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud.

Esta transformación le permite parametrizar métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y respaldar operaciones que no se pueden deshacer.

1.

Estructura





1

La clase del remitente (**invoker**) es responsable de iniciar las solicitudes.

Esta clase debe tener un campo para almacenar una referencia a un objeto de comando.

El remitente activa ese comando en lugar de enviar la solicitud directamente al receptor. Hay que tener en cuenta que el remitente no es responsable de crear el objeto de comando.

Por lo general, obtiene un pre comando creado desde el cliente a través del constructor.

2

La interfaz **Command** generalmente declara un solo método para ejecutar el comando.

3

Los **comandos concretos** implementan varios tipos de solicitudes. No se supone que un comando concreto realice el trabajo por sí solo, sino que pase la llamada a uno de los objetos de lógica de negocios. Sin embargo, en aras de simplificar el código, estas clases pueden fusionarse.

Los parámetros necesarios para ejecutar un método en un objeto receptor se pueden declarar como campos en el comando concreto.

Se puede hacer que los objetos de comando sean inmutables permitiendo solo la inicialización de estos campos a través del constructor.

5

El **Cliente** crea y configura objetos de comando concretos.

El cliente debe pasar todos los parámetros de la solicitud, incluida una instancia del receptor, al constructor del comando. Después de eso, el comando resultante puede estar asociado con uno o varios remitentes

4

La clase **Receiver** contiene cierta lógica empresarial. Casi cualquier objeto puede actuar como un receptor. La mayoría de los comandos solo manejan los detalles de cómo se pasa una solicitud al receptor, mientras que el receptor mismo hace el trabajo real.

Analogía en el Mundo Real

En un amable camarero se le acerca y rápidamente toma su pedido, escribiéndolo en una hoja de papel.

El camarero va a la cocina y pega la orden en la pared. Después de un tiempo, la orden llega al chef, quien lo lee y cocina la comida en consecuencia.

El cocinero coloca la comida en una bandeja junto con el pedido. El camarero descubre la bandeja, comprueba el pedido para asegurarse de que todo esté como lo quería y lo trae todo a su mesa.

El pedido en papel sirve como un comando. Permanece en la cola hasta que el chef esté listo para servirlo.

El pedido contiene toda la información relevante requerida para cocinar la comida. Le permite al chef comenzar a cocinar de inmediato en lugar de correr aclarando los detalles del pedido directamente de usted.



2.

Consecuencias al Implementar



✓ **Single Responsibility Principle.** Puede desacoplar clases que invocan operaciones de clases que realizan estas operaciones.

✓ **Open/Close Principle.** Puede introducir nuevos comandos en la aplicación sin romper el código del cliente existente.

✓ Se puede implementar **deshacer / rehacer**.

✓ Se puede implementar la ejecución diferida de operaciones.


✓ Se puede ensamblar un conjunto de comandos simples en uno complejo

✗ El código puede volverse más complicado ya que se está introduciendo una capa completamente nueva entre remitentes y receptores

⚡ Se usa este patrón de diseño cuando desee parametrizar objetos con operaciones.

⚡ El patrón Command puede convertir una llamada a un método específico en un objeto independiente. Este cambio abre muchos usos interesantes: puede pasar comandos como argumentos de método, almacenarlos dentro de otros objetos, cambiar comandos vinculados en tiempo de ejecución, etc.

 Se usa este patrón de diseño cuando desee poner en cola las operaciones, programar su ejecución o ejecutarlas de forma remota.

 Al igual que con cualquier otro objeto, un comando se puede serializar, lo que significa convertirlo en una cadena que se puede escribir fácilmente en un archivo o una base de datos. Más tarde, la cadena se puede restaurar como el objeto de comando inicial. Así, puede retrasar y programar la ejecución del comando. ¡Pero aún hay más! Del mismo modo, puede poner en cola, registrar o enviar comandos a través de la red.

⚡ Se usa este patrón de diseño cuando desee implementar operaciones reversibles.

⚡ Aunque hay muchas formas de implementar deshacer / rehacer, el patrón de comando es quizás el más popular de todos.

3.

Relación con Otros Patrones de Diseño



Chain Responsibility, Command, Mediator y Observer abordan varias formas de conectar a los remitentes y receptores de solicitudes:

- ❖ **Chain Responsibility** pasa una solicitud secuencialmente a lo largo de una cadena dinámica de receptores potenciales hasta que uno de ellos la maneja.
- ❖ **Command** establece conexiones unidireccionales entre remitentes y receptores.
- ❖ **Mediator** elimina las conexiones directas entre remitentes y receptores, obligándolos a comunicarse indirectamente a través de un objeto mediador.
- ❖ **Observer** permite a los receptores suscribirse dinámicamente y darse de baja de recibir solicitudes.

Puede usar **Command** y **Memento** juntos al implementar "deshacer". En este caso, los comandos son responsables de realizar varias operaciones sobre un objeto de destino, mientras que los recuerdos guardan el estado de ese objeto justo antes de que se ejecute un comando.

Prototype puede ayudar cuando necesita guardar copias de comandos en el historial.

Se puede tratar a **Visitor** como una versión poderosa del patrón de **Command**. Sus objetos pueden ejecutar operaciones sobre varios objetos de diferentes clases.

Command, y **Strategy** pueden ser similares porque puede usar ambos para parametrizar un objeto con alguna acción. Sin embargo, tienen intenciones muy diferentes.

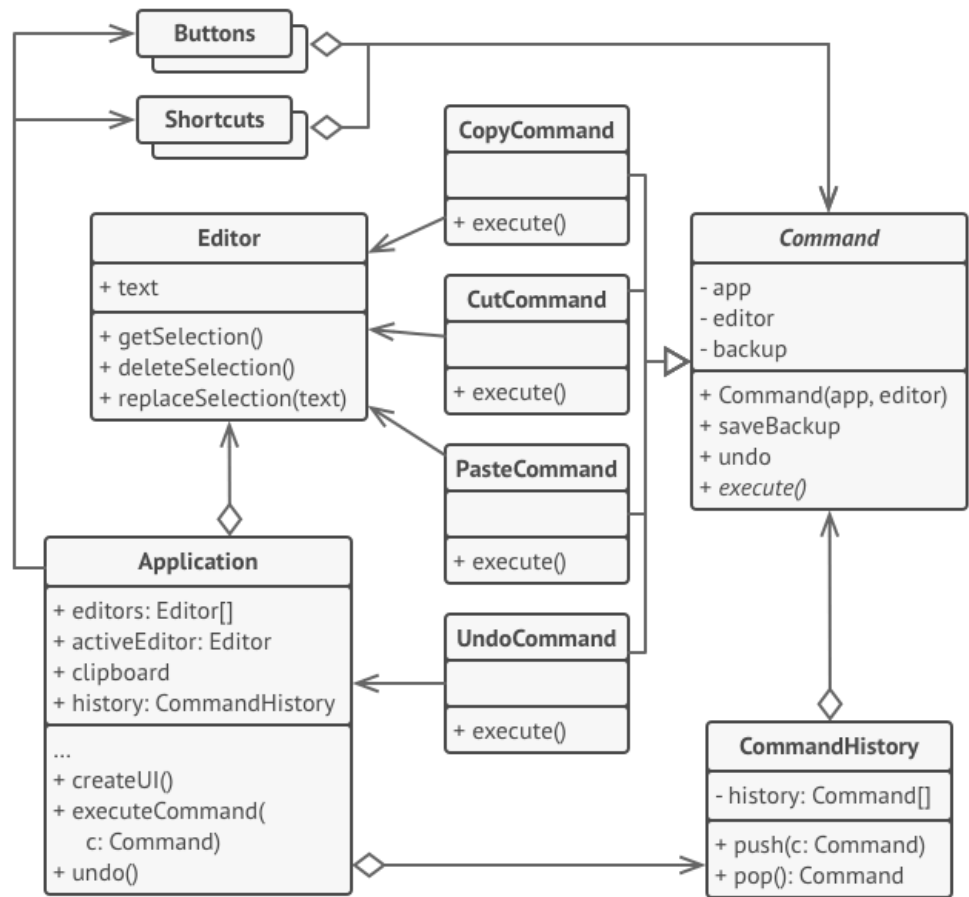
- Se puede usar **Command** para convertir cualquier operación en un objeto. Los parámetros de la operación se convierten en campos de ese objeto. La conversión le permite diferir la ejecución de la operación, ponerla en cola, almacenar el historial de comandos, enviar comandos a servicios remotos, etc.
- **Strategy** generalmente describe diferentes formas de hacer lo mismo, permitiéndole intercambiar estos algoritmos dentro de una sola clase de contexto.

4.

Ejemplo de Implementación



En este ejemplo, el patrón Command ayuda a rastrear el historial de operaciones ejecutadas y hace posible revertir una operación si es necesario.



Operaciones que se pueden deshacer en un editor de texto.

Los comandos que resultan en cambiar el estado del editor (por ejemplo, cortar y pegar) hacen una copia de seguridad del estado del editor antes de ejecutar una operación asociada con el comando.

Después de ejecutar un comando, se coloca en el historial de comandos (una pila de objetos de comando) junto con la copia de seguridad del estado del editor en ese punto.

Más tarde, si el usuario necesita revertir una operación, la aplicación puede tomar el comando más reciente del historial, leer la copia de seguridad asociada del estado del editor y restaurarla.

El código del cliente (elementos de la GUI, historial de comandos, etc.) no está acoplado a clases de comandos concretas porque funciona con comandos a través de la interfaz de comandos.

Este enfoque le permite introducir nuevos comandos en la aplicación sin romper ningún código existente.

```
// The base command class defines the common interface for all
// concrete commands.
abstract class Command is
    protected field app: Application
    protected field editor: Editor
    protected field backup: text

    constructor Command(app: Application, editor: Editor) is
        this.app = app
        this.editor = editor

    // Make a backup of the editor's state.
    method saveBackup() is
        backup = editor.text

    // Restore the editor's state.
    method undo() is
        editor.text = backup

    // The execution method is declared abstract to force all
    // concrete commands to provide their own implementations.
    // The method must return true or false depending on whether
    // the command changes the editor's state.
    abstract method execute()
```

```
// The concrete commands go here.
class CopyCommand extends Command is
    // The copy command isn't saved to the history since it
    // doesn't change the editor's state.
    method execute() is
        app.clipboard = editor.getSelection()
        return false

class CutCommand extends Command is
    // The cut command does change the editor's state, therefore
    // it must be saved to the history. And it'll be saved as
    // long as the method returns true.
    method execute() is
        saveBackup()
        app.clipboard = editor.getSelection()
        editor.deleteSelection()
        return true

class PasteCommand extends Command is
    method execute() is
        saveBackup()
        editor.replaceSelection(app.clipboard)
        return true

// The undo operation is also a command.
class UndoCommand extends Command is
    method execute() is
        app.undo()
        return false
```

```
// The global command history is just a stack.
class CommandHistory is
  private field history: array of Command

  // Last in...
  method push(c: Command) is
    // Push the command to the end of the history array.

  // ...first out
  method pop():Command is
    // Get the most recent command from the history.

// The editor class has actual text editing operations. It plays
// the role of a receiver: all commands end up delegating
// execution to the editor's methods.
class Editor is
  field text: string

  method getSelection() is
    // Return selected text.

  method deleteSelection() is
    // Delete selected text.

  method replaceSelection(text) is
    // Insert the clipboard's contents at the current
    // position.
```

```

// The application class sets up object relations. It acts as a
// sender: when something needs to be done, it creates a command
// object and executes it.
class Application is
    field clipboard: string
    field editors: array of Editors
    field activeEditor: Editor
    field history: CommandHistory

    // The code which assigns commands to UI objects may look
    // like this.
    method createUI() is
        // ...
        copy = function() { executeCommand(
            new CopyCommand(this, activeEditor)) }
        copyButton.setCommand(copy)
        shortcuts.onKeyPress("Ctrl+C", copy)

        cut = function() { executeCommand(
            new CutCommand(this, activeEditor)) }
        cutButton.setCommand(cut)
        shortcuts.onKeyPress("Ctrl+X", cut)

        paste = function() { executeCommand(
            new PasteCommand(this, activeEditor)) }
        pasteButton.setCommand(paste)
        shortcuts.onKeyPress("Ctrl+V", paste)
        undo = function() { executeCommand(
            new UndoCommand(this, activeEditor)) }
        undoButton.setCommand(undo)
        shortcuts.onKeyPress("Ctrl+Z", undo)

    // Execute a command and check whether it has to be added to
    // the history.
    method executeCommand(command) is
        if (command.execute)
            history.push(command)

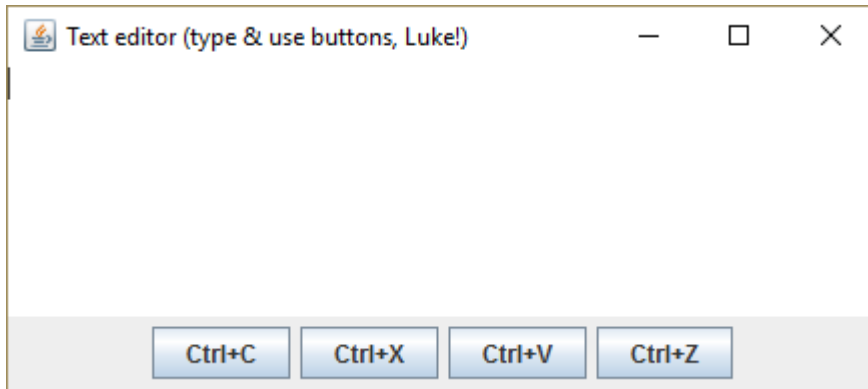
    // Take the most recent command from the history and run its
    // undo method. Note that we don't know the class of that
    // command. But we don't have to, since the command knows
    // how to undo its own action.
    method undo() is
        command = history.pop()
        if (command != null)
            command.undo()

```

GitHub

https://github.com/RefactoringGuru/design-patterns-java/tree/master/src/refactoring_guru/command/example

Repositorio donde se realiza este ejemplo a profundidad



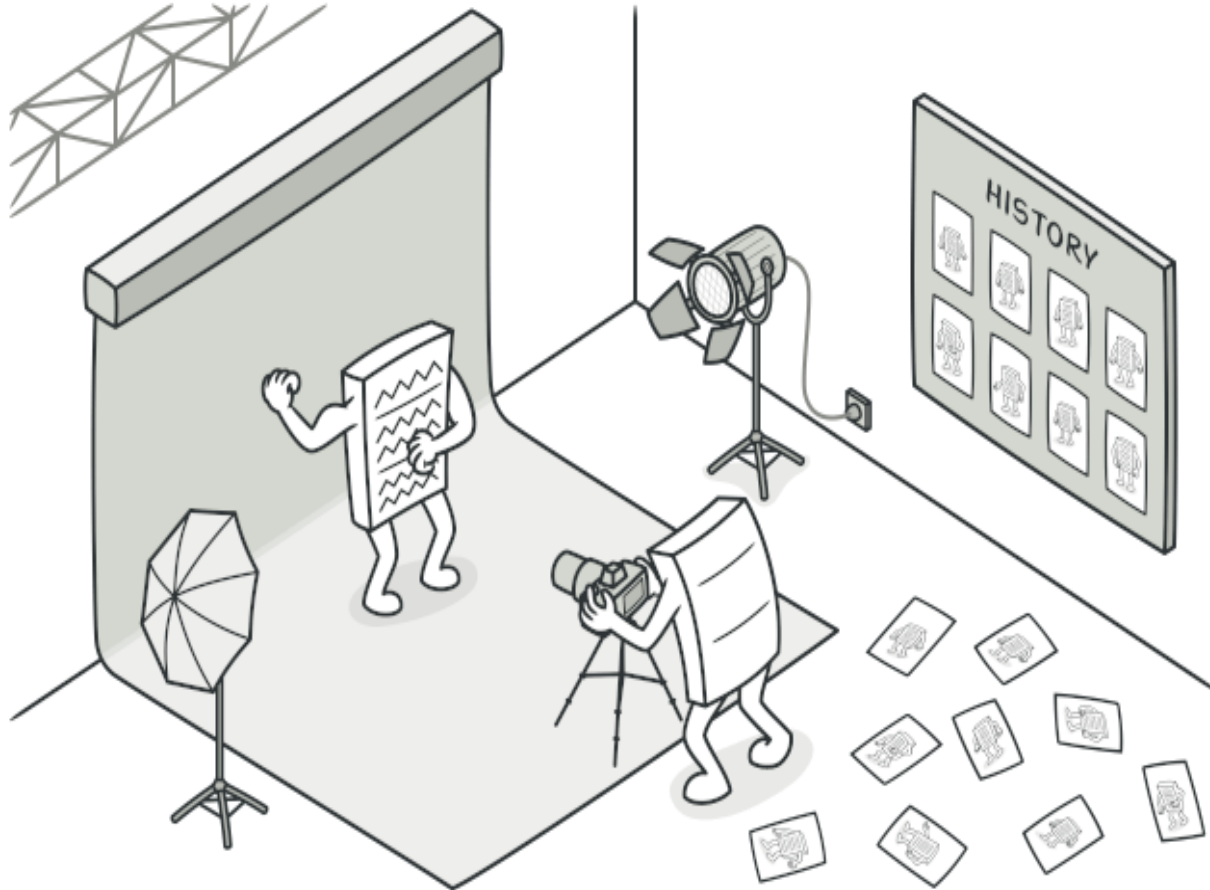


MEMENTO

Complejidad: ★★ ★

Popularidad: ★ ☆ ☆

También conocido como **Snapshot**



Es un patrón de diseño de comportamiento que permite guardar y restaurar el estado anterior de un objeto sin revelar los detalles de su implementación.

Es decir, permite tomar *snapshots* del estado de un objeto y restaurarlo en el futuro.

No compromete la estructura interna del objeto con el que trabaja, así como los datos guardados dentro de las instantáneas.

1.

Estructura



1

La clase **Originator** puede producir *snapshots* de su propio estado, así como restaurar su estado a partir de instantáneas cuando sea necesario.

2

Memento es un objeto de valor que actúa como un *snapshot* del estado del originador. Es una práctica común hacer que el memento sea inmutable y pasarle los datos solo una vez, a través del constructor.

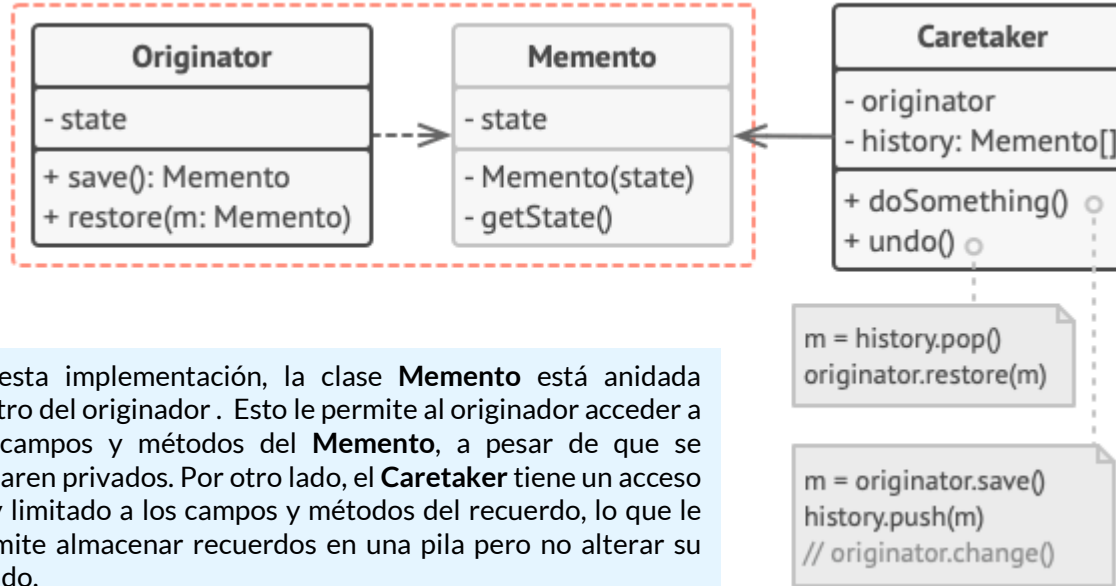
3

Caretaker no solo sabe "cuándo" y "por qué" capturar el estado del originador, sino también cuándo debe restaurarse el estado.

El **Caretaker** puede realizar un seguimiento de la historia del originador almacenando una pila de recuerdos. Cuando el autor tiene que viajar atrás en la historia, el cuidador toma el recuerdo más alto de la pila y lo pasa al método de restauración del autor.

4

En esta implementación, la clase **Memento** está anidada dentro del originador. Esto le permite al originador acceder a los campos y métodos del **Memento**, a pesar de que se declaren privados. Por otro lado, el **Caretaker** tiene un acceso muy limitado a los campos y métodos del recuerdo, lo que le permite almacenar recuerdos en una pila pero no alterar su estado.

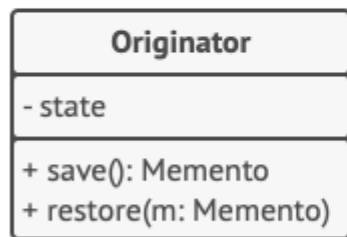


Implementación basada en una interfaz intermedia

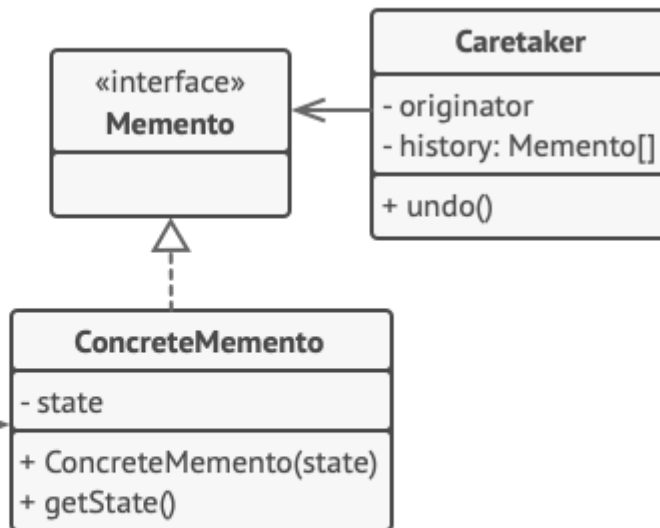
Existe una implementación alternativa, adecuada para lenguajes de programación que no admiten clases anidadas

1

En ausencia de clases anidadas, puede restringir el acceso a los campos del recuerdo estableciendo una convención según la cual los **caretakers** pueden trabajar con un **memento** solo a través de una interfaz intermedia declarada explícitamente, que solo declararía métodos relacionados con los metadatos del **memento**..



```
cm = (ConcreteMemento) m
state = cm.getState()
```



2

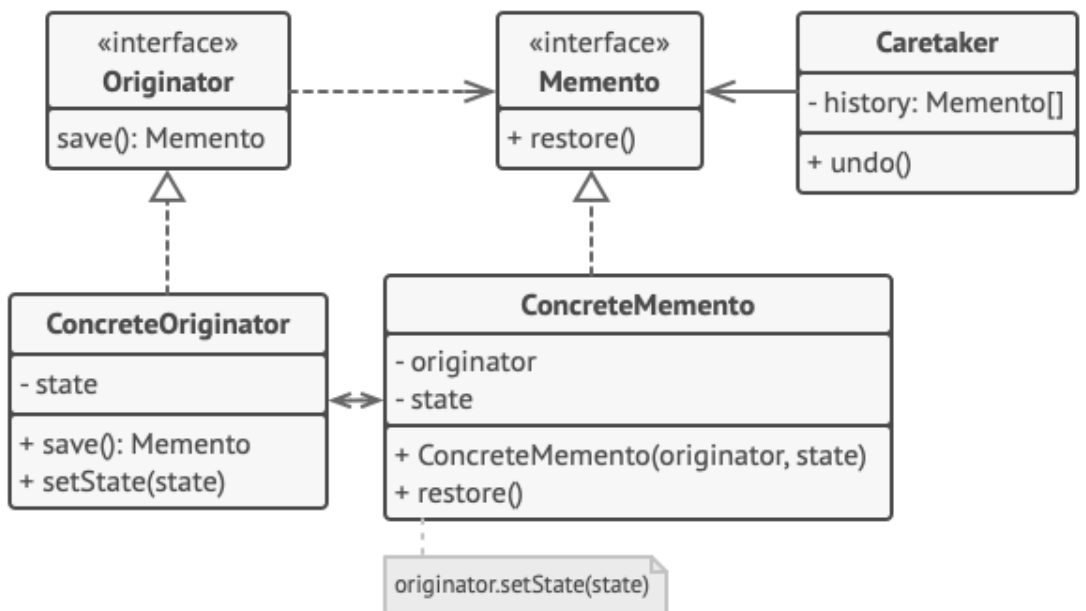
Por otro lado, los **originadores** pueden trabajar con un objeto de *memento* directamente, accediendo a campos y métodos declarados en la clase de **Memento**. La desventaja de este enfoque es que debe declarar a todos los miembros del *memento* público.

Implementación con encapsulación estricta

Es útil cuando no desea dejar la más mínima posibilidad de que otras clases accedan al estado del originador a través del memento.

1 Esta implementación permite tener múltiples tipos de *originadores* y *mementos*. Cada *originador* trabaja con una clase de *memento* correspondiente. Ni los *originadores* ni los *mementos* exponen su estado a nadie.

2 Los *caretakers* ahora tienen restricciones explícitas para cambiar el estado almacenado en los mementos. Además, la clase de **caretaker** se vuelve independiente del *originador* porque el método de restauración ahora se define en la clase de *memento*.



3 Cada *memento* se vincula con el autor que lo produjo. El autor se pasa al constructor del *memento*, junto con los valores de su estado. Gracias a la estrecha relación entre estas clases, un *memento* puede restaurar el estado de su *originador*, dado que este último ha definido los establecedores apropiados.

2.

Consecuencias al Implementar




✓ Puede producir snapshots/mementos del estado del objeto sin violar su encapsulación.


✓ Puede simplificar el código del originador dejando que el caretaker mantenga el historial del estado del originador.

✗ La aplicación puede consumir mucha RAM si los clientes crean recuerdos con demasiada frecuencia.

✗ Los *caretaker* deben rastrear el ciclo de vida del originador para poder destruir *mementos* obsoletos.

✗ No se puede garantizar que el estado dentro del *memento* permanezca intacto.

 Se usa este patrón de diseño cuando se desea generar instantáneas del estado del objeto para poder restaurar un estado anterior del objeto.

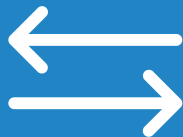
 El patrón Memento le permite hacer copias completas del estado de un objeto, incluidos los campos privados, y almacenarlas por separado del objeto. Si bien la mayoría de las personas recuerda este patrón gracias al caso de uso "deshacer", También es indispensable cuando se trata de transacciones (es decir, si necesita revertir una operación por error).

⚡ Use el patrón cuando el acceso directo a los campos / captadores / definidores del objeto viola su encapsulación.

⚡ El Memento hace que el objeto sea responsable de crear una instantánea de su estado. Ningún otro objeto puede leer la instantánea, haciendo que los datos de estado del objeto original sean seguros y protegidos.

3.

Relación con Otros Patrones de Diseño



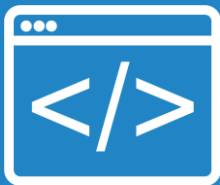
Se puede usar **Command** y **Memento** juntos al implementar "deshacer". En este caso, los comandos son responsables de realizar varias operaciones sobre un objeto de destino, mientras que los *mementos* guardan el estado de ese objeto justo antes de que se ejecute un comando.

A veces, **Prototype** puede ser una alternativa más simple a **Memento**. Esto funciona si el objeto, cuyo estado desea almacenar en el historial, es bastante sencillo y no tiene enlaces a recursos externos, o si los enlaces son fáciles de restablecer.

Se puede usar **Memento** junto con **Iterator** para capturar el estado de iteración actual y revertirlo si es necesario.

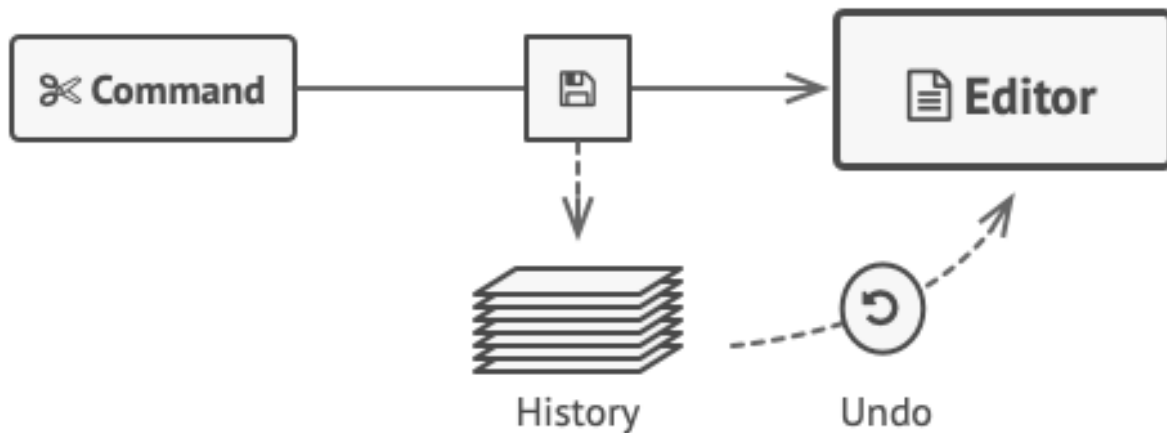
4.

Ejemplo de Implementación



En la creación una aplicación de editor de texto simple. Se decide permitir a los usuarios deshacer cualquier operación realizada en el texto, una característica tan común hoy en día.

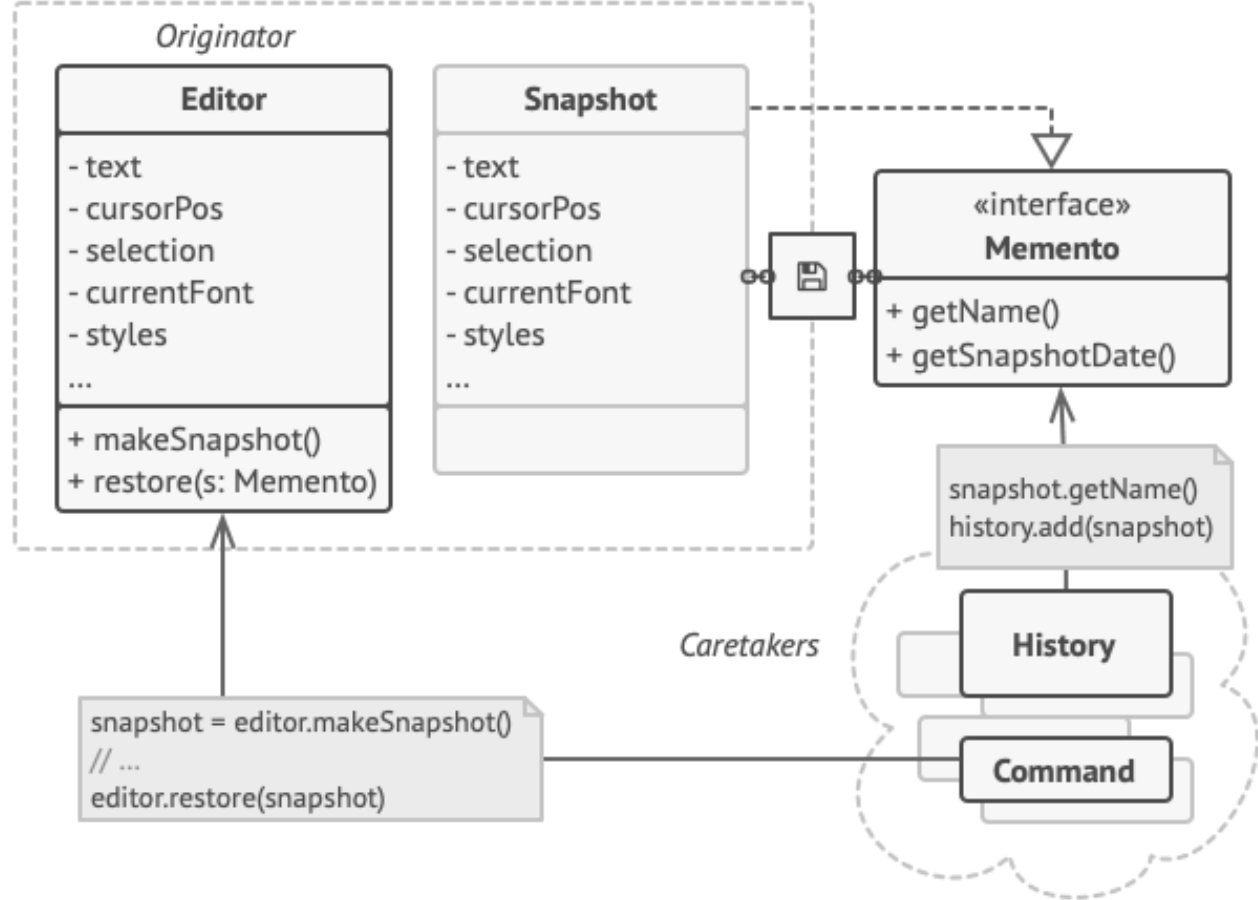
Antes de realizar cualquier operación, la aplicación registra el estado de todos los objetos y la guarda en algún almacenamiento. Más tarde, cuando un usuario decide revertir una acción, la aplicación obtiene la última instantánea del historial y la usa para restaurar el estado de todos los objetos.



Antes de ejecutar una operación, la aplicación guarda una instantánea del estado de los objetos, que luego se puede utilizar para restaurar los objetos a su estado anterior.

El patrón sugiere almacenar la copia del estado del objeto en un objeto especial llamado *memento*. El contenido del *memento* no es accesible para ningún otro objeto, excepto el que lo produjo.

Otros objetos deben comunicarse con *mementos* utilizando una interfaz limitada que puede permitir obtener los metadatos de la instantánea (tiempo de creación, el nombre de la operación realizada, etc.), pero no el estado del objeto original contenido en la instantánea.



El autor tiene acceso completo al recuerdo, mientras que el cuidador solo puede acceder a los metadatos.

Podemos crear una clase *Historial* separada para actuar como el *caretaker*.

Una pila de *mementos* almacenados dentro del *caretaker* crecerá cada vez que el editor esté a punto de ejecutar una operación. Incluso podría representar esta pila dentro de la interfaz de usuario de la aplicación, mostrando el historial de operaciones realizadas previamente a un usuario.

Cuando un usuario activa el deshacer, el historial toma el recuerdo más reciente de la pila y lo devuelve al editor, solicitando una reversión. Como el editor tiene acceso completo al recuerdo, cambia su propio estado con los valores tomados del recuerdo.



EJEMPLOS USANDO COMMAND & MEMENTO

```
// The originator holds some important data that may change over
// time. It also defines a method for saving its state inside a
// Memento and another method for restoring the state from it.
class Editor is
    private field text, curX, curY, selectionWidth

    method setText(text) is
        this.text = text

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

    method setSelectionWidth(width) is
        this.selectionWidth = width

    // Saves the current state inside a Memento.
    method createSnapshot():Snapshot is
        // Memento is an immutable object; that's why the
        // originator passes its state to the Memento's
        // constructor parameters.
        return new Snapshot(this, text, curX, curY, selectionWidth)
```

```
// The Memento class stores the past state of the 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 = curX
        this.curY = curY
        this.selectionWidth = selectionWidth

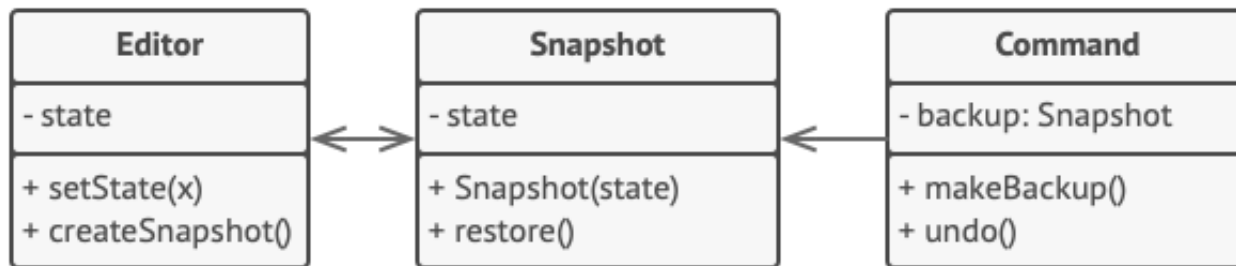
    // At some point, a previous state of the editor can be
    // restored using a Memento object.
    method restore() is
        editor.setText(text)
        editor.setCursor(curX, curY)
        editor.setSelectionWidth(selectionWidth)

    // A Command object can act as a caretaker. In that case, the
    // Command gets a Memento just before it changes the
    // originator's state. When undo is requested, it restores the
    // originator's state from a 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 *caretakers*.

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 usar el *memento* almacenado en ese comando para volver al estado anterior.

La clase memento no declara ningún campo público, captador o definidor.

Por lo tanto, ningún objeto puede alterar su contenido. Los *mementos* están vinculados al objeto editor que los creó. Esto permite que un *memento* restaure el estado del editor vinculado al pasar los datos a través de setters en el objeto del editor.

Como los *mementos* están vinculados a objetos específicos del editor, puede hacer que su aplicación admita varias ventanas de editor independientes con una pila de deshacer centralizada.

GitHub

https://github.com/RefactoringGuru/design-patterns-java/tree/master/src/refactoring_guru/memento/example

```
Output - Memento Example (run) x
run:
Undoing: Move by X:44 Y:150
Undoing: Move by X:-10 Y:-121
Undoing: Move by X:62 Y:-38
Undoing: Move by X:-102 Y:71
Undoing: Move by X:-87 Y:76
Undoing: Move by X:110 Y:-20
Undoing: Move by X:-4 Y:-93
```

