Patrones de Diseño:

El Patrón “MEMENTO”.

¿Qué es?:

El patrón de diseño Memento es un patrón de comportamiento en la programación orientada a objetos que se utiliza para capturar el estado interno de un objeto en un momento dado, de manera que este estado pueda ser restaurado en un futuro si es necesario. Este patrón permite implementar funciones de "deshacer" o "revertir" en una aplicación, permitiendo que los objetos vuelvan a estados anteriores.

En pocas palabras, el patrón Memento es una técnica que permite capturar y restaurar el estado de un objeto, facilitando deshacer acciones y volver a estados anteriores sin comprometer la encapsulación (permite restringir el acceso directo a los datos internos de un objeto y en su lugar proporciona métodos públicos para interactuar con esos datos de manera controlada.).

El patrón Memento consta de tres componentes principales:

1. **Originator (Origen):** Es el objeto cuyo estado interno se desea capturar, almacenar y, posteriormente, restaurar utilizando el patrón Memento. Es el punto focal del patrón, ya que es responsable de crear y utilizar los objetos Memento para mantener y restaurar su propio estado interno.
2. **Memento:** Es un objeto que actúa como una cápsula que almacena el estado interno del “Originator” en un momento dado, permitiendo que el objeto pueda ser restaurado a ese estado en el futuro. En otras palabras, el Memento es una representación inmutable del estado de un objeto en un punto específico en el tiempo (el "Originator" tiene acceso a los datos dentro del "Memento", pero otros objetos no deberían poder modificarlo).
3. **Caretaker (Cuidador):** El Caretaker en el patrón de diseño "Memento" es un objeto que gestiona y almacena objetos Memento, permitiendo que el Originator capture y restaure estados anteriores de manera controlada. El Caretaker ayuda a separar las responsabilidades de captura y restauración de estados del objeto Originator, contribuyendo así a una estructura más modular y encapsulada.

Aplicaciones:

El patrón Memento es útil en situaciones donde se desea implementar la capacidad de revertir o deshacer acciones en una aplicación. Algunos ejemplos de casos de uso incluyen:

1. **Editores de texto:** El patrón Memento puede usarse para guardar el historial de cambios en un documento y permitir que los usuarios deshagan o rehagan acciones.
2. **Aplicaciones de dibujo:** Puede utilizarse para permitir la reversión de trazos o elementos dibujados en una aplicación de dibujo.
3. **Editores de imágenes:** Permite deshacer y rehacer operaciones de edición como recortar, cambiar el tamaño, aplicar filtros, etc.
4. **Aplicaciones de videojuegos:** Puede ser utilizado para permitir que los jugadores deshagan o rehagan acciones en el juego.
5. **Aplicaciones financieras:** Para rastrear y revertir transacciones financieras.

Objetivos:

Estos objetivos contribuyen a crear aplicaciones más flexibles, mantenibles y amigables para el usuario. Pero tiene como objetivo principal permitir la captura, almacenamiento y restauración del estado interno de un objeto en un momento específico. Esto proporciona un mecanismo para deshacer cambios y volver a estados anteriores de un objeto sin violar su encapsulación.

1. **Deshacer y Rehacer:** Uno de los objetivos más prominentes de la implementación del patrón Memento es proporcionar la capacidad de deshacer y rehacer acciones en una aplicación. Los usuarios pueden retroceder a estados anteriores y, si es necesario, avanzar nuevamente. Esto es especialmente útil en aplicaciones donde se realizan cambios y se quiere permitir a los usuarios revertirlos.
2. **Separación de Responsabilidades:** Al usar el patrón Memento, se separa la responsabilidad de manejar el estado y el historial de cambios de la lógica principal de los objetos. Esto mejora la modularidad y facilita el mantenimiento, ya que los objetos no se sobrecargan con la lógica de manejo de estados.
3. **Conservación del Encapsulamiento:** El patrón Memento permite guardar y restaurar el estado de un objeto sin violar el principio de encapsulamiento. El "Originator" tiene acceso a los datos internos del "Memento", pero otros objetos no pueden acceder directamente a esos datos.
4. **Historial y Auditoría:** En ciertos sistemas, especialmente en aplicaciones financieras o de registro, el patrón Memento puede utilizarse para mantener un historial completo de cambios realizados en los objetos. Esto puede ser útil para fines de auditoría y seguimiento.
5. **Flexibilidad:** La implementación del patrón Memento brinda flexibilidad al permitir que los objetos mantengan múltiples puntos en el tiempo en forma de "Mementos". Esto puede ser valioso para explorar diferentes estados y volver atrás en el tiempo para tomar decisiones basadas en estados pasados. La flexibilidad en el patrón, se refiere a la capacidad de capturar y restaurar el estado de un objeto en diferentes puntos en el tiempo de manera eficiente y adaptable. Esta característica permite que el patrón Memento sea versátil y útil en una variedad de situaciones y escenarios.
6. **Mejora de la Experiencia del Usuario:** En aplicaciones de usuario, como editores de texto o software de diseño gráfico, el patrón Memento mejora la experiencia del usuario al permitir deshacer acciones no deseadas o recuperar trabajos anteriores.
7. **Aislamiento de Operaciones de Estado:** La implementación del patrón Memento permite aislar las operaciones de gestión de estado, lo que hace que el código sea más limpio y claro al separar estas operaciones del flujo principal del programa.

Fin del patrón:

En general, el patrón Memento brinda una solución estructurada y organizada para manejar los cambios y el historial de estados en una aplicación, lo que mejora la mantenibilidad, la extensibilidad y la usabilidad del software.

1. **Gestión de Historial:** Proporciona un método para mantener un historial de estados anteriores de un objeto. Esto es útil en aplicaciones donde se necesita la capacidad de deshacer y rehacer acciones.
2. **Restauración de Estados:** Permite que un objeto sea restaurado a un estado previo. Esto es especialmente valioso cuando se cometen errores o cuando se requiere volver a un punto anterior en el tiempo.
3. **Seguridad en Encapsulación:** El patrón Memento permite capturar el estado interno de un objeto sin exponer sus detalles internos. Esto mantiene la encapsulación y la integridad del diseño.
4. **Flexibilidad:** El patrón permite mantener múltiples estados en una estructura de datos, como una pila, lo que ofrece flexibilidad para navegar y restaurar entre diferentes momentos en el tiempo.

Ejemplo de la vida real:

Un ejemplo de la vida real en el que se podría aplicar el patrón de programación Memento es en una aplicación de edición de documentos, como un procesador de texto. Supongamos que se está trabajando en un software de procesamiento de texto y se desea implementar la funcionalidad de "deshacer" y "rehacer" cambios en el documento.

Así se podría aplicar el patrón Memento en esta situación:

1. **Originador (Document):** Este sería el objeto que representa el documento actual en el procesador de texto. Contendría el contenido actual del documento.

```python
class Document:
    def __init__(self, content):
        self.content = content

    def edit_content(self, new_content):
        self.content = new_content

    def create_memento(self):
        return DocumentMemento(self.content)

    def restore_memento(self, memento):
        self.content = memento.get_saved_content()

    def display_content(self):
        print(self.content)
```

2. **Memento (DocumentMemento):** Este objeto capturaría el estado actual del contenido del documento en un momento específico.

```python
class DocumentMemento:
    def __init__(self, content):
        self.saved_content = content

    def get_saved_content(self):
        return self.saved_content
```

3. **Cuidador (TextEditor):** Este sería el objeto que maneja la gestión de los estados guardados y la funcionalidad de "deshacer" y "rehacer".

```python
class TextEditor:
    def __init__(self):
        self.document = None
        self.mementos = []
        self.current_memento_index = -1

    def create_document(self, content):
        self.document = Document(content)

    def edit_document(self, new_content):
        if self.document:
            self.mementos = self.mementos[:self.current_memento_index + 1]
            self.mementos.append(self.document.create_memento())
            self.current_memento_index += 1
            self.document.edit_content(new_content)
```

```python
def undo(self):
    if self.current_memento_index > 0:
        self.current_memento_index -= 1
        self.document.restore_memento(self.mementos[self.current_memento

def redo(self):
    if self.current_memento_index < len(self.mementos) - 1:
        self.current_memento_index += 1
        self.document.restore_memento(self.mementos[self.current_memento

def display_document(self):
    self.document.display_content()
```

Ejemplo al aplicar el patrón VS el código sin la aplicación de patrón:

Con el patrón “Memento”

```python
class Memento:
    def __init__(self, state):
        self._state = state

    def get_state(self):
        return self._state


class Originator:
    def __init__(self, state):
        self._state = state

    def set_state(self, state):
        self._state = state

    def create_memento(self):
        return Memento(self._state)

    def restore_from_memento(self, memento):
        self._state = memento.get_state()


class Caretaker:
    def __init__(self):
        self._mementos = []

    def add_memento(self, memento):
        self._mementos.append(memento)

    def get_memento(self, index):
        return self._mementos[index]


if __name__ == "__main__":
    caretaker = Caretaker()
```

```
    originator = Originator("State1")
    print(f"Initial state: {originator._state}")

    memento1 = originator.create_memento()
    caretaker.add_memento(memento1)

    originator.set_state("State2")
    print(f"State after change: {originator._state}")

    memento2 = originator.create_memento()
    caretaker.add_memento(memento2)

    originator.restore_from_memento(caretaker.get_memento(0))
    print(f"State after restoring first memento: {originator._state}")
```

Sin el patrón “Memento”, el código quedaría algo así:

```
class Originator:
    def __init__(self, state):
        self._state = state

    def set_state(self, state):
        self._state = state

    def get_state(self):
        return self._state


if __name__ == "__main__":
    originator = Originator("State1")
    print(f"Initial state: {originator.get_state()}")

    saved_state = originator.get_state()

    originator.set_state("State2")
    print(f"State after change: {originator.get_state()}")

    originator.set_state(saved_state)
    print(f"State after restoring saved state: {originator.get_state()}")
```

En el primer ejemplo con el patrón Memento, se puede ver cómo se separa la responsabilidad de manejar los estados y su almacenamiento en objetos diferentes. En el segundo ejemplo sin el patrón Memento, el manejo de estados es más simple, pero carece de la capacidad de revertir cambios.

Ejemplo corto de programación:

```
class EditorState:
    def __init__(self, content):
        self.content = content

class Editor:
    def __init__(self):
        self.content = ""

    def type(self, words):
        self.content += words

    def save(self):
        return EditorState(self.content)

    def restore(self, state):
        self.content = state.content
```

```
# Uso del patrón Memento
editor = Editor()
editor.type("Hola, ")
editor.type("esto es una prueba.")

# Guardar el estado actual
saved_state = editor.save()

# Realizar más ediciones
editor.type(" Más texto agregado.")

# Restaurar el estado anterior
editor.restore(saved_state)

print(editor.content)  # Salida: Hola, esto es una prueba.
```