Patrones de Diseño

Patrón "MEMENTO"

Contenido

1. Introducción, componentes y objetivos.
2. Ejemplos y explicación.
3. Conclusiones y preguntas.

Introducción

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.

El patron "Memento":

3 componente principales.

- Originator (Origen):
  Este es el objeto cuyo estado se desea guardar. Puede crear un objeto "Memento" que contenga una copia del estado actual del "Originator".

- Memento:
  El objeto "Memento" almacena una copia inmutable del estado del "Originator". El "Originator" tiene acceso a los datos dentro del "Memento", pero otros objetos no deberían poder modificarlo.

- Caretaker (Cuidador):
  Este componente es responsable de mantener y administrar una colección de objetos "Memento". Puede solicitar un "Memento" al "Originator" para guardar su estado actual en la colección de "Mementos". Luego, puede devolver un "Memento" a "Originator" para restaurar su estado a uno anterior si es necesario.

Objetivos

Deshacer y Rehacer:

Los usuarios pueden retroceder a estados anteriores y, si es necesario, avanzar nuevamente.

Separación de Responsabilidades:

Se separa la responsabilidad de manejar el estado y el historial de cambios de la lógica principal de los objetos.

Conservación del Encapsulamiento:

Permite guardar y restaurar el estado de un objeto sin violar el principio de encapsulamiento.

Historial y Auditoría:

En ciertos sistemas, puede utilizarse para mantener un historial completo de cambios realizados en los objetos.

Objetivos

Flexibilidad

Capturar y restaurar el estado de un objeto en diferentes puntos en el tiempo de manera eficiente y adaptable.

Mejora de la Experiencia del Usuario:

Permitir deshacer acciones no deseadas o recuperar trabajos anteriores.

Aislamiento de Operaciones de Estado:

Permite aislar las operaciones de gestión de estado, lo que hace que el código sea más limpio y claro.

Ejemplo sin el Patrón:

```python
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()}")
```

Ejemplo con el Patrón:

```python
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
```

```python
# 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.
```

Conclusiones

- Permitir la captura, almacenamiento y restauración.
- Mecanismo para deshacer cambios y volver a estados anteriores de un objeto sin violar su encapsulación.
- Mejora la mantenibilidad, la extensibilidad y la usabilidad del software.

Al ofrecer enfoques reutilizables y eficientes se construyen sistemas robustos y flexibles.

¡Gracias!

¿Alguna pregunta?