

Padrão GoF: Memento

Pamela Maria da Silva
Engenharia de Sistemas II

Memento

Porque utilizar?

O nome Memento tem origem no latim e significa "lembrar" ou "recordar". Ele é frequentemente associado à ideia de manter uma lembrança ou registro de algo.

Assim como o nome indica, a grande vantagem do padrão Memento é permitir que um objeto possa retornar a estados anteriores de forma simples e segura, sem quebrar o encapsulamento.



Características

Separação de Responsabilidades

01.

Originator

O objeto que terá o estado salvo.

02.

Memento

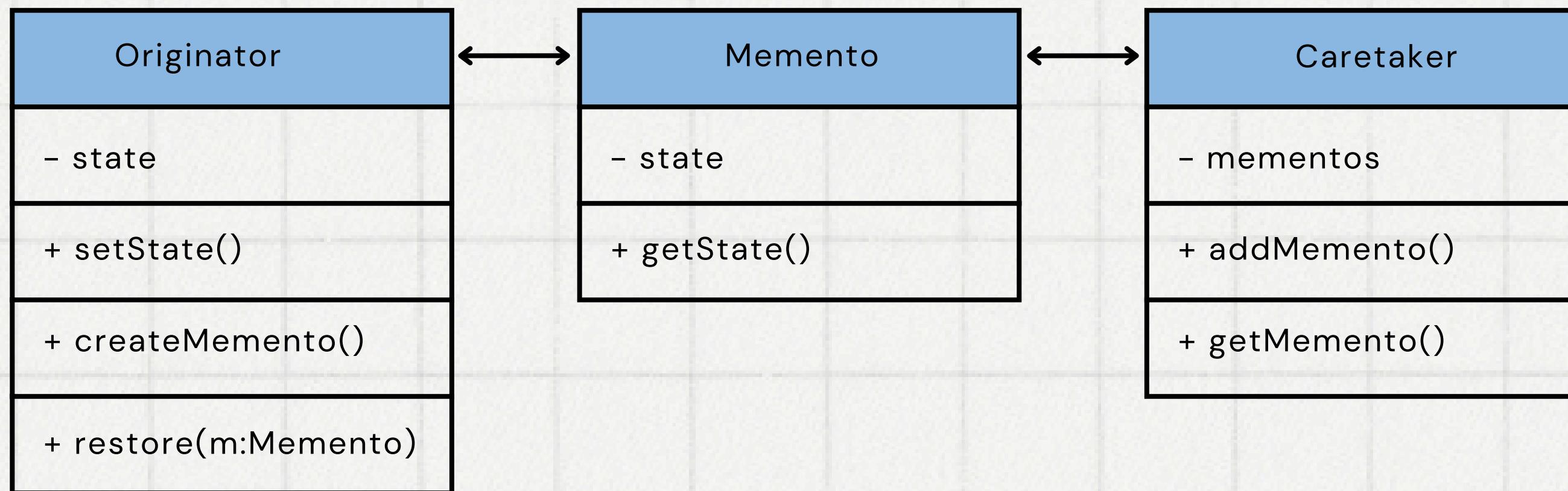
Contém o estado do Originator e permite sua recuperação posterior.

03.

Caretaker

Guarda e gerencia os Mementos sem saber o que há neles. Ele solicita que o Originator crie ou restaure um estado quando necessário.

Diagrama de Classes



Originator

```
● ● ●  
class TextEditor:  
    def __init__(self):  
        self._content = ""  
  
    def write(self, text: str):  
        self._content += text  
  
    def get_content(self) -> str:  
        return self._content  
  
    def save(self) -> Memento:  
        return Memento(self._content)  
  
    def restore(self, memento: Memento):  
        self._content = memento.get_state()
```

Memento

```
class Memento:  
    def __init__(self, state: str):  
        self._state = state  
  
    def get_state(self) -> str:  
        return self._state, []
```

Caretaker

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

    def push(self, memento: Memento):
        self._mementos.append(memento)

    def pop(self) -> Memento:
        if not self._mementos:
            return None
        return self._mementos.pop()
```

Uso prático

```
● ● ●

if __name__ == "__main__":
    editor = TextEditor()
    history = History()

    editor.write("Olá")
    history.push(editor.save())
    print("Conteúdo atual:", editor.get_content()) # Conteúdo atual: Olá

    editor.write(", mundo!")
    history.push(editor.save())
    print("Conteúdo atual:", editor.get_content()) # Conteúdo atual: Olá, mundo!

    editor.restore(history.pop())
    print("Após desfazer:", editor.get_content()) # Após desfazer: Olá

    editor.restore(history.pop())
    print("Após desfazer novamente:", editor.get_content()) # Após desfazer novamente:
        return None
    return self._mementos.pop()
```

Padrão GoF: Strategy

Eduardo Vinícius Puton
Engenharia de Sistemas II

Strategy

Finalidade

O Strategy é um padrão de projeto comportamental sugerido para algoritmos que apresentam diferentes variações de comportamento de uma mesma funcionalidade.





Qual é o problema resolvido pelo Strategy

Usando como exemplo um algoritmo que tem o intuito de fazer um cálculo matemático com 2 números. Esse cálculo pode ser uma adição, subtração ou multiplicação. O código é o mesmo para todas as variações do cálculo, a única coisa que muda é o operador.

A classe que guarda o algoritmo pode separar os comportamentos por condicionais, um if ou um switch case. O padrão Strategy sugere que esses comportamentos sejam separados por classes e não por condicionais. Essa segmentação facilita a manutenção e futura adição de variações da função.

Estrutura



- **Interface Strategy:** Define os métodos comuns que todos os comportamentos (Estratégias) devem implementar.
- **Classes Concretas (Estratégias):** Implementam a interface Strategy e definem algoritmos ou comportamentos específicos. Cada classe representa uma variação do comportamento.
- **Classe Contexto:** Mantém uma referência para a interface Strategy e delega a ela a execução do comportamento. A classe Contexto não conhece a implementação específica da estratégia, apenas sabe que ela segue a interface definida.
- **Classe Cliente:** Responsável por selecionar a estratégia apropriada e associá-la ao Contexto.

Exemplo em código

Interface

```
● ● ●  
public interface Operacao {  
    1 usage  2 implementations  
    int calcular(int a, int b);  
}
```

Exemplo em código

Classe Concreta (Adição)

```
● ● ●  
public class operacaoAdicao implements Operacao {  
    1 usage  
    @Override  
    public int calcular(int a, int b) {  
        return a + b;  
    }  
}
```

Exemplo em código

Classe Concreta (Subtração)

```
● ● ●  
public class operacaoSubtracao implements Operacao {  
    1 usage  
    @Override  
    public int calcular(int a, int b) {  
        return a - b;  
    }  
}
```

Exemplo em código

```
public class Calculadora {  
    2 usages  
    private Operacao operacao;  
    2 usages  
    public void setOperacao(Operacao operacao) {  
        this.operacao = operacao;  
    }  
    2 usages  
    public int calcular(int a, int b) {  
        return operacao.calcular(a, b);  
    }  
}
```

Classe Contexto

- institui a função que possui as variáveis
- **operacao**: Variável do tipo Operação (Interface) criada;
- **setOperacao**: Método para definir a operação, através da interface;
- **calcular**: Método que usa a classe concreta.

Exemplo em código

Classe Cliente

```
public class Main {  
    public static void main(String[] args) {  
  
        Calculadora calculadora = new Calculadora();  
        Scanner scanner = new Scanner(System.in);  
        System.out.print("Digite o primeiro número: ");  
        int primeiroNumero = scanner.nextInt();  
        System.out.print("Digite o segundo número: ");  
        int segundoNumero = scanner.nextInt();  
  
        System.out.print("Escolha a operação (adicao, subtracao): ");  
        String operacao = scanner.next();
```

Exemplo em código

Classe Cliente

```
if (operacao.equalsIgnoreCase( anotherString: "adicao")) {  
    calculadora.setOperacao(new operacaoAdicao());  
} else if (operacao.equalsIgnoreCase( anotherString: "subtracao")) {  
    calculadora.setOperacao(new operacaoSubtracao());  
} else {  
    System.out.println("Operação inválida.");  
    scanner.close();  
    return;  
}  
  
int resultado = calculadora.calcular(primeiroNumero, segundoNumero);  
System.out.println("Resultado: " + resultado);  
scanner.close();
```

Prós

- Reduz o acoplamento entre o código principal e algoritmos específicos.
- Facilita a expansão e manutenção do código, já que novas funcionalidades podem ser implementadas sem alterar o código principal.

Contra

- Em um algoritmo que possui pouquíssimas variações, usar o padrão Strategy pode ser desnecessário, pois ele adiciona classes e interfaces extras.



**Muito
obrigado!**

Referências

- Refactoring Guru. (n.d.). Strategy Design Pattern. Acessado em 08 nov. 2024 <https://refactoring.guru/pt-br/design-patterns/strategy>
- Softplan. (2020). Descomplicando o Strategy. Acessado em 08 nov. 2024 <https://www.softplan.com.br/tech-writers/descomplicando-o-strategy/>
- Shvets, A. (2019). Dive Into Design Patterns.
- REFACTORING GURU. Padrão de Projeto Memento. Disponível em: <https://refactoring.guru/pt-br/design-patterns/memento>. Acesso em: 10 nov. 2024.
- ROBERTO, Jones. Design Patterns – Parte 20: Memento. Medium, 19 out. 2019. Disponível em: <https://medium.com/@jonesroberto/design-patterns-parte-20-memento-6b30ad75b12f>. Acesso em: 10 nov. 2024.