Das Composite-Entwurfsmuster

Dieses git-Projekt demonstriert das zugrundeliegende Prinzip des Composite-Entwurfsmusters.

Das Composite-Entwurfsmuster wird verwendet, um hierarchische Strukturen zu modellieren, in denen es sinnvoll ist, Einzelobjekte und zusammengesetzte Objekte einheitlich behandeln zu können. Ein häufiges Anwendungsbeispiel, das auch im vorliegenden Projekt verwendet wird, ist ein Dateisystem, das Dateien und Ordner organisiert.

Das Muster ermöglicht es, rekursive Operationen auf einer Baumstruktur auszuführen, unabhängig davon, ob ein Element ein Leaf (hier: eine Datei) oder ein Composite (hier: ein Ordner) ist. Der Nutzer kann dann im vorliegenden Beispiel etwa die Gesamtgröße aller Dateien berechnen lassen, die in einem beliebigen Ordner einschließlich aller Unterordner enthalten sind.

Wichtige Elemente des Composite-Entwurfsmusters

1. **Component**: Die abstrakte Basisklasse oder Schnittstelle, die die gemeinsamen Methoden für Leaves und Composites definiert.
2. **Leaf**: Einzelobjekte, die keine untergeordneten Elemente enthalten.
3. **Composite**: Zusammengesetzte Objekte, die sowohl Leaves als auch andere Composite-Objekte enthalten können. Die Methoden rufen rekursiv die entsprechenden Methoden der enthaltenen Komponenten auf.

Der Client interagiert mit den Objekten über die Schnittstelle, ohne sich darum kümmern zu müssen, ob es sich um ein Leaf oder ein Composite handelt. Dies ermöglicht eine transparente Nutzung der Hierarchie.

Implementierung

Verzeichnisstruktur

Eine mögliche Projektstruktur für das Beispiel:

```
project/
├── src/
│   ├── node.py          # Basisklasse (Component)
│   ├── file.py          # Datei-Klasse (Leaf)
│   ├── folder.py        # Ordner-Klasse (Composite)
├── main.py              # Beispielanwendung
```

Wie implementiert man das Composite-Muster?

1. Definiere eine abstrakte Komponente

Erstelle eine abstrakte Basisklasse oder Schnittstelle, die gemeinsame Methoden für Einzel- und Composite-Objekte definiert. Diese Klasse sollte Methoden wie `display()`, `count_items()` oder andere operationale Methoden enthalten.

```python
from abc import ABC, abstractmethod

class Component(ABC):
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def operation(self):
        pass
```

2. Implementiere die Leaf-Klasse

Erstelle eine konkrete Klasse, die die abstrakte Basisklasse erweitert und keine Kinder enthält. Diese Klasse repräsentiert ein Einzelobjekt, z. B. eine Datei.

```python
class Leaf(Component):
    def __init__(self, name, size):
        super().__init__(name)
        self.size = size

    def operation(self):
        return f"Leaf {self.name} with size {self.size}"
```

3. Implementiere die Composite-Klasse

Erstelle eine Klasse, die andere Komponenten (Leaves oder weitere Composite-Objekte) enthalten kann. Sie sollte Methoden zum Hinzufügen, Entfernen und Durchlaufen der Kinder implementieren.

```python
class Composite(Component):
    def __init__(self, name):
        super().__init__(name)
        self.children = []

    def add(self, component):
        self.children.append(component)

    def remove(self, component):
        self.children.remove(component)

    def operation(self):
        result = f"Composite {self.name} contains:\n"
        for child in self.children:
            result += f"  {child.operation()}\n"
        return result
```

4. Teste den Code

Weil Leaves und Composite-Objekte beide von der gemeinsamen Basisklasse erben, können sie einheitlich behandelt werden. Test dies ggf. anhand des folgenden Codes:

```
class Composite(Component):
    def __init__(self, name):
        super().__init__(name)
        self.children = []

    def add(self, component):
        self.children.append(component)

    def remove(self, component):
        self.children.remove(component)

    def operation(self):
        result = f"Composite {self.name} contains:\n"
        for child in self.children:
            result += f"  {child.operation()}\n"
        return result
```