ПИ19-3 Деменчук Георгий

**Паттерны проектирования. Круг задач, решаемый структурными паттернами**

Структурные паттерны проектирования фокусируются на том, как создавать и организовывать объекты, чтобы получить более сложную структуру. Они помогают решать следующие задачи:

Компоновка объектов: Как создать структуру из объектов, чтобы получить нужную функциональность? Например, паттерн Декоратор позволяет динамически добавлять новую функциональность к объекту.

Разделение ответственностей: Как разбить функциональность на отдельные объекты, чтобы уменьшить связанность и улучшить расширяемость приложения? Например, паттерн Адаптер позволяет использовать существующий код с новым интерфейсом.

Обеспечение согласованности: Как создать структуру, которая будет работать совместно с другими структурами

**Паттерн Композиция (Composite)** - это структурный паттерн проектирования, который позволяет создавать иерархию объектов, чтобы обрабатывать их как единое целое. Паттерн Композиция позволяет клиенту работать с объектами одного типа, независимо от того, представляют ли они отдельный объект или совокупность объектов.

**Пример из жизни:**

Есть склад OZON, на складе много коробок, в коробках есть товары, также в коробке могут быть коробки поменьше.  
Нам необходимо вычислить стоимость товаров на складе.

Иерархию можно представить в виде дерева, где каждый элемент наследуется от класса Node с методом и абстрактным методом summ. Также Node объявляет методы add и remove для добавления объекта в иерархию.

from \_\_future\_\_ import annotations

from abc import ABC, abstractmethod

from typing import List

class Node(ABC):

@property

def parent(self) -> Node:

return self.\_parent

@parent.setter

def parent(self, parent: Node):

self.\_parent = parent

def add(self, node: Node) -> None:

pass

def remove(self, node: Node) -> None:

pass

def is\_box(self) -> bool:

return False

@abstractmethod

def summ(self) -> str:

pass

Далее мы наследуемся от Node и объявляем класс коробки (BoxNode) и товара (ItemNode), обязательно переопределяем метод summ

class ItemNode(Node):

def \_\_init\_\_(self, price) -> None:

super().\_\_init\_\_()

self.price = price

def summ(self) -> float:

return self.price

Товар обращается к своему полю цены в sum и возвращает цену

class BoxNode(Node):

def \_\_init\_\_(self) -> None:

self.\_children: List[Node] = []

def add(self, node: Node) -> None:

self.\_children.append(node)

node.parent = self

def remove(self, node: Node) -> None:

self.\_children.remove(node)

node.parent = None

def is\_box(self) -> bool:

return True

def summ(self) -> float:

result = 0

for child in self.\_children:

result += child.summ()

return result

Коробка в методе summ обращается ко всем товарам, лежащим в ней, суммирует их и возвращает результат выше

if \_\_name\_\_ == "\_\_main\_\_":

tree = BoxNode()

box1 = BoxNode()

box1.add(ItemNode(20))

box1.add(ItemNode(30))

box2 = BoxNode()

box2.add(ItemNode(50))

tree.add(box1)

tree.add(box2)

tree.add(ItemNode(150))

print(tree.summ())

Таким образом, можно обращаясь к родительскому элементу дерева (складу/коробке) получать общую цену, не зная о внутренней иерархии объектов и их отдельных ценах.

**Решаемые задачи:**

* Как создать структуру, в которой каждый элемент может иметь потомков, как простых, так и сложных?
* Как создать интерфейс, чтобы клиент мог обращаться как к простым элементам, так и к сложным структурам одинаково?

**Альтернативы:**

Паттерн Фасад (Facade): предоставляет простой интерфейс для сложной системы. Отличие от Композиции в том, что Фасад не создает новую структуру, а просто обеспечивает более простой интерфейс для существующей системы

**Паттерн посетитель (Visitor)** - это поведенческий паттерн проектирования, который позволяет добавлять новую функциональность к классам элементов структуры данных без изменения этих классов.

Обычно паттерн посетитель используется, когда у вас есть структура данных, состоящая из разнотипных элементов, и вы хотите реализовать различные операции над этими элементами, не изменяя сами элементы. Паттерн посетитель позволяет создать новый объект, называемый посетителем, который будет выполнять эти операции.

Практический пример:

У нас есть несколько вариаций БД (реплики и кеш), при этом для каждой необходимо реализовывать собственные методы инициализации

Следовательно, объявляем интерфейс, общий для всех БД с методами для каждой

// Супертип всех объектов в структуре

**interface** **DBElement** {

**void** **connect**(DBElementVisitor visitor);

}

// Супертип всех операций

**interface** **DBElementVisitor** {

**void** **visit**(Cache cache);

**void** **visit**(Replica replica);

**void** **visit**(PrimaryDB primaryDB);

}

Далее реализовываем наши классы

**class** **Cache** **implements** **DBElement** {

@Override

**public** **void** **connect** (DBElementVisitor visitor) {

visitor.visit(**this**);

}

}

**class** **Replica** **implements** **DBElement** {

@Override

**public** **void** **connect** (DBElementVisitor visitor) {

visitor.visit(**this**);

}

}

**Решаемые задачи:**

* Когда у вас есть структура данных, состоящая из разнотипных элементов, и вы хотите реализовать различные операции над этими элементами, не изменяя сами элементы. Паттерн посетитель позволяет создать новый объект, называемый посетителем, который будет выполнять эти операции.
* Когда у вас есть структура данных, которая часто меняется, и вы хотите избежать разрастания кода с множеством условных операторов, обрабатывающих различные типы элементов. Паттерн посетитель позволяет добавлять новые типы элементов без изменения существующего кода.
* Когда у вас есть несколько операций, которые нужно выполнять над элементами структуры данн

Альтернативы паттерну посетитель (Visitor) включают:

* Паттерн Шаблонный метод (Template Method) - определяет общую структуру алгоритма, но оставляет реализацию некоторых шагов субклассам.
* Паттерн Стратегия (Strategy) - определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми.
* Паттерн Команда (Command) - превращает запросы в объекты, позволяя передавать их как аргументы методам, ставить запросы в очередь, логировать их или отменять последние.
* Паттерн Цепочка обязанностей (Chain of Responsibility) - перенаправляет запросы по цепочке обработчиков, пока один из них не воспользуется им.