Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Отчет по курсу «Объектно-ориентированная разработка»

Студент: Д. Д. Наумов
Преподаватель: Н. П. Аносова
Группа: М8О-206М
Дата:
Оценка:
Подпись:

Москва, 2022

Постоновка задачи

Задание: Реализовать основные паттерны проектирования:

- MVC
- Abstract Factory
- Adapter
- Builder
- Chain of Responsibility
- Command
- Composite
- Facade
- Iterator
- Mediator
- Memento
- Observer
- Prototype
- Proxy
- Singleton
- State
- Strategy
- Visitor

1 Посетитель

Посетитель — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

Паттерн Посетитель предлагает разместить новое поведение в отдельном классе, вместо того чтобы множить его сразу в нескольких классах. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого нужно передавать эти объекты в методы посетителя.

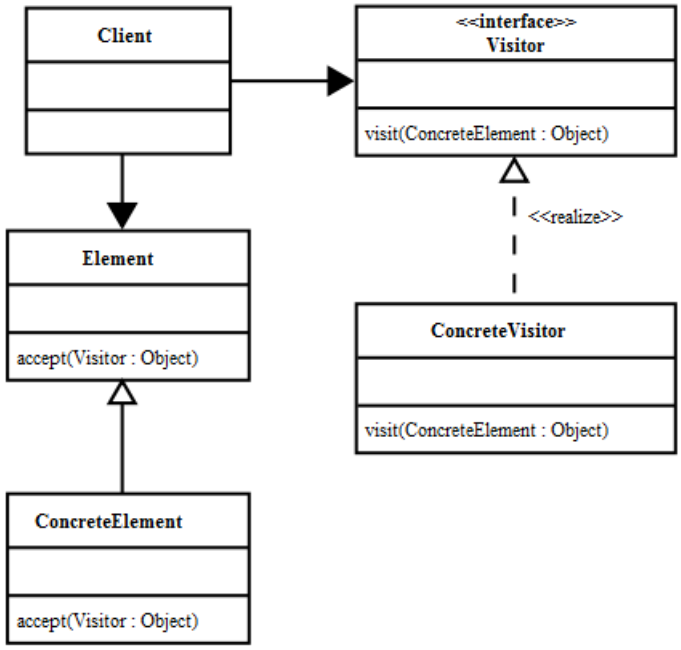

Рис. 1: Схема паттерна

2 Применимость

- Когда нужно выполнить какую-то операцию над всеми элементами сложной структуры объектов, например, деревом.

  Посетитель позволяет применять одну и ту же операцию к объектам различных классов.

- Когда над объектами сложной структуры объектов надо выполнять некоторые не связанные между собой операции, но нельзя «засорять» классы такими операциями.

  Посетитель позволяет извлечь родственные операции из классов, составляющих структуру объектов, поместив их в один класс-посетитель. Если структура объектов является общей для нескольких приложений, то паттерн позволит в каждое приложение включить только нужные операции.

- Когда новое поведение имеет смысл только для некоторых классов из существующей иерархии.

  Посетитель позволяет определить поведение только для этих классов, оставив его пустым для всех остальных.

3 Шаги реализации

1. Создать интерфейс посетителя и объявите в нём методы «посещения» для каждого класса элемента, который существует в программе.

2. Написть интерфейс элементов или объявить абстрактный метод принятия посетителей в базовом классе иерархии элементов.

3. Реализовать методы принятия во всех конкретных элементах. Они должны переадресовывать вызовы тому методу посетителя, в котором тип параметра совпадает с текущим классом элемента.

4. Иерархия элементов должна знать только о базовом интерфейсе посетителей. С другой стороны, посетители будут знать обо всех классах элементов.

5. Для каждого нового поведения создать конкретный класс посетителя. Приспособить это поведение для работы со всеми типами элементов, реализовав все методы интерфейса посетителей.

6. Клиент будет создавать объекты посетителей, а затем передавать их элементам, используя метод принятия.

4 Преимущества и недостатки

Плюсы	Минусы
Упрощает добавление операций, работающих со сложными структурами объектов	Паттерн не оправдан, если иерархия элементов часто меняется.
Объединяет родственные операции в одном классе	Может привести к нарушению инкапсуляции элементов
Посетитель может накапливать состояние при обходе структуры элементов	

5 Код

```
#include <iostream>
#include <array>

class ConcreteComponentA;
class ConcreteComponentB;

class Visitor {
 public:
  virtual void VisitConcreteComponentA(const ConcreteComponentA *element) const = 0;
  virtual void VisitConcreteComponentB(const ConcreteComponentB *element) const = 0;
};


class Component {
 public:
  virtual ~Component() {}
  virtual void Accept(Visitor *visitor) const = 0;
};


class ConcreteComponentA : public Component {

 public:
  void Accept(Visitor *visitor) const override {
    visitor->VisitConcreteComponentA(this);
  }
  std::string ExclusiveMethodOfConcreteComponentA() const {
    return "A";
  }
};
```

```
31
32 class ConcreteComponentB : public Component {
33
34  public:
35   void Accept(Visitor *visitor) const override {
36     visitor->VisitConcreteComponentB(this);
37   }
38   std::string SpecialMethodOfConcreteComponentB() const {
39     return "B";
40   }
41 };
42
43 class ConcreteVisitor1 : public Visitor {
44  public:
45   void VisitConcreteComponentA(const ConcreteComponentA *element) const override {
46     std::cout << element->ExclusiveMethodOfConcreteComponentA() << " + ConcreteVisitor1
         \n";
47   }
48
49   void VisitConcreteComponentB(const ConcreteComponentB *element) const override {
50     std::cout << element->SpecialMethodOfConcreteComponentB() << " + ConcreteVisitor1\n
         ";
51   }
52 };
53
54 class ConcreteVisitor2 : public Visitor {
55  public:
56   void VisitConcreteComponentA(const ConcreteComponentA *element) const override {
57     std::cout << element->ExclusiveMethodOfConcreteComponentA() << " + ConcreteVisitor2
         \n";
58   }
59   void VisitConcreteComponentB(const ConcreteComponentB *element) const override {
60     std::cout << element->SpecialMethodOfConcreteComponentB() << " + ConcreteVisitor2\n
         ";
61   }
62 };
63
64 void ClientCode(std::array<const Component *, 2> components, Visitor *visitor) {
65   for (const Component *comp : components) {
66     comp->Accept(visitor);
67   }
68 }
69
70 int main() {
71   std::array<const Component *, 2> components = {new ConcreteComponentA, new
         ConcreteComponentB};
72   std::cout << "The client code works with all visitors via the base Visitor interface
         :\n";
73   ConcreteVisitor1 *visitor1 = new ConcreteVisitor1;
```

```
74    ClientCode(components, visitor1);
75    std::cout << "\n";
76    std::cout << "It allows the same client code to work with different types of
          visitors:\n";
77    ConcreteVisitor2 *visitor2 = new ConcreteVisitor2;
78    ClientCode(components, visitor2);
79
80    for (const Component *comp : components) {
81      delete comp;
82    }
83    delete visitor1;
84    delete visitor2;
85
86    return 0;
87  }
```

Github: https://github.com/dandachok/Patterns.git

6 Выводы

Паттерны проектирования — это один из инструментов разработчика, который помогает ему сэкономить время и сделать более качественное решение. Даже теоретические знания шаблонов проектирования помогают понять чужой код гораздо быстрее и, соответственно, они необходимы в работе над проектом даже если у вас нет коммерческого опыта работы.