Лабораторная работа №5

«Реализация одного из поведенческих паттернов проектирования»

Цель работы: научиться применять поведенческие паттерны проектирования.

Продолжительность работы - 4 часа.

Содержание

Поведенческие паттерны

Вспомним вопросы, на которые отвечают паттерны:

Структурные - Какое будет устройство объекта?

Порождающие - Как создать объект?

Поведенческие - Как объект будет взаимодействовать с другими?

Поведенческие паттерны определяют алгоритмы и способы реализации взаимодействия различных объектов и классов.

Их ключевая задача - определить связи, взаимодействия и зависимости объектов, увеличивая гибкость и масштабируемость их использования.

Список поведенческих паттернов (GoF):

- Цепочка обязанностей (Chain of responsibility)
- Команда (Command)
- Интерпретатор (Interpreter)
- Итератор (Iterator)
- Посредник (Mediator)

- Хранитель (Memento)
- Наблюдатель (Observer)
- Состояние (State)
- Стратегия (Strategy)
- Шаблонный метод (Template method)
- Посетитель (Visitor)

Данные паттерны не создают новые объекты, не задают их структуру, а лишь определяют взаимодействие.

Паттерн Стратегия (Strategy)

Назначение:

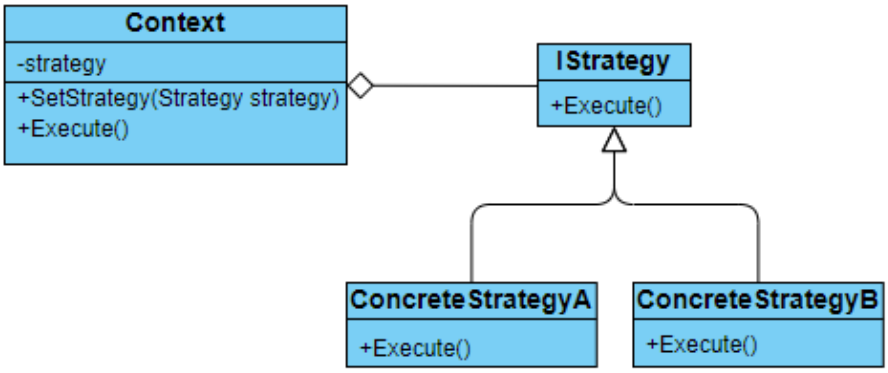
Паттерн Стратегия - поведенческий паттерн проектирования, предназначенный для определения семейства алгоритмов и обеспечения взаимозаменяемости каждого из них прямо во время исполнения программы.

Применимость:

Применять паттерн Стратегия целесообразно, когда:

- Вы используете разные варианты алгоритма внутри одного объекта.
- У вас есть множество классов, отличающихся только некоторым поведением (алгоритмом). В таком случае можно задать один главный класс, а поведения вынести в отдельные классы и использовать их по мере необходимости.
- Вам нужно менять поведение во время выполнения программы.
- Вам нужно скрыть реализацию поведения от других классов, а также от класса, использующего данный алгоритм.

Структура:



Участники:

Context - класс, содержащий в себе абстрактный экземпляр необходимой ему стратегии. При необходимости она может меняться на другую (так как они наследуются от одного интерфейса).

IStrategy - интерфейс, определяющий, как будут использоваться различные поведения.

ConcreteSrategy - конкретные классы, наследующиеся от Strategy, реализующие в себе необходимое поведение.

Результаты применения:

Достоинства:

- Изолирование поведения от остальных классов
- Уход от наследования к делегированию
- Изменения поведения во время выполнения
- Реализует принцип открытости-закрытости (система открыта для масштабирования за счет добавления новых классов и закрыта для изменения имеющегося алгоритма)

Недостатки:

- Усложняет структуру за счет дополнительных классов
- Клиент должен знать каждую стратегию, чтобы правильно ее применять

Примеры из жизни:

- Выбор кофе в кофемашине тоже схож с паттерном Стратегия. Клиент желает выпить кофе, поэтому обращается к кофемашине и в зависимости от своих пожеланий выбирает конкретный напиток. Кофемашина же в зависимости от выбора клиента готовит напиток, используя необходимую стратегию.
- Путешественник пытается добраться из пункта А в пункт В. Есть много вариантов, чтобы это сделать: пешком, на машине, на поезде, на самолете. Однако путник хочет найти баланс "время-стоимость", чтобы добраться как можно быстрее и потратив как можно меньше средств. Его выбор конкретной стратегии как раз и зависит от затрат и времени в пути.
- Вы пишите программу для выполнения какого-либо алгоритма сортировки или поиска. Однако, как вам должно быть известно каждый такой алгоритм обладает своей сложностью, которая влияет на скорость его выполнения. Также на скорость влияют входные данные (на каком-то массиве будет быстрее работать быстрая сортировка, на другом - шейкерная, на третьем с самого начала выполнится условие Айверсона и тогда сортировка пузырьком завершит свою работу раньше всех). Тогда в зависимости от массива вы вправе выбрать наилучшую стратегию для его сортировки.

Паттерн Состояние (State)

Назначение:

Паттерн Состояние - поведенческий шаблон проектирования, использующийся для изменения поведения объекта в зависимости от его состояния.
Поведение может измениться настолько, что может создаться впечатление, что изменился класс объекта.

Явной аналогией является конечный автомат.

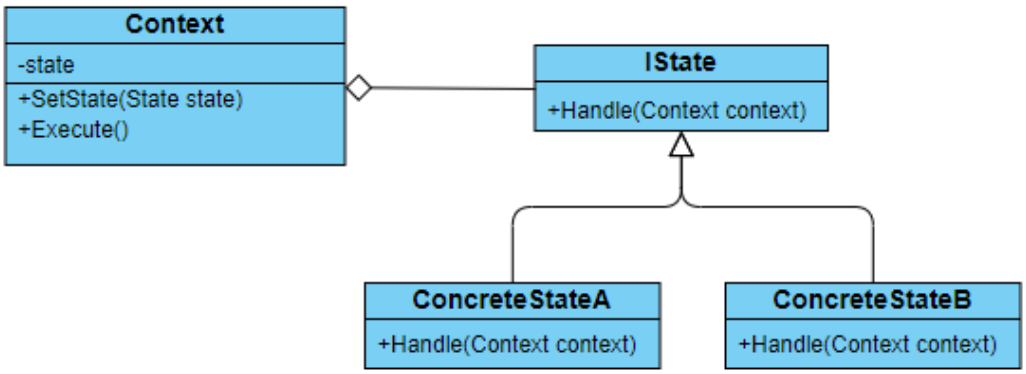
Применимость:

Применять паттерн Стратегия целесообразно, когда:

- у объекта существует несколько состояний, от которых зависит его поведение

- у объекта используются многочисленные условные конструкции, выбор которых зависит от текущего состояния объекта

Структура:



Участники:

Context - класс, содержащий в себе абстрактный экземпляр состояния, которое будет меняться в ходе выполнения программы.

State - интерфейс, реализующий каждое из конкретных состояний. Через данный интерфейс Context взаимодействует с конкретным состоянием (ConcreteState). Интерфейс должен содержать средства для обратной связи с объектом (например, события).

ConcreteState - конкретные классы, наследующиеся от State. Реализуют в себе необходимое поведение и условия при которых Context может переходить из текущего в другое.

Результаты применения:

Достоинства:

- Позволяет избавиться от множества условий
- Упрощает код контекста

Недостатки:

- Усложняет код при малом количестве условий

Примеры из жизни:

- Коробка передач автомобиля является хорошим примером паттерная состояния. В зависимости от включенной передачи изменяется поведение машины (едет вперед, назад или стоит).

- Примером Состояния является Машина Тьюринга, так как обладает конечным множеством состояний, от которых меняется поведение самой машина (изменение состояния, перехода влево или вправо, запись символа на ленту)

Пример кода для State:

В настоящее время можно заметить рост популярности сервисов и экосистем, предоставляющий возможность пользователям смотреть фильмы, заказывать еду, совершать покупки на своей платформе. Представим, что у нас существует такой сервис. У каждого пользователя он может находиться в трех состояниях: Без подписки, С подпиской и Пробный период.

Причем:

- Купить или отменить подписку можно в любой момент.
- Пробный период может быть активирован всего 1 раз, после его истечения пользователь должен либо купить подписку, либо отменить.
- В сервисе существуют 3 продукта: Просмотр фильмов, Заказ еды, Скачивание музыки на устройство. В зависимости от текущего состояния эти продукты предоставляются (или нет) в разном виде.
- Пробный период заканчивается, когда пользователь использует 3 любых продукта.

Абстрактный класс State:

Содержит абстрактные методы приобретения, отмены подписки, оформления пробного периода и все виды использования продуктов.

```
internal abstract class State
  {
    public abstract void GetTrialPeriod(Service service);
    public abstract void BuySubscription(Service service);
    public abstract void Unsubscribe(Service service);
    public abstract void WatchFilm();
    public abstract void OrderFood();
    public abstract void DownlandMusic();
  }
```

Классы наследники:

В классах наследниках реализуются методы State.

SubscriptionState

```
    public override void BuySubscription(Service service)
    {
      Console.WriteLine("Подписка уже куплена\n");
    }

    public override void GetTrialPeriod(Service service){ }
      // Пожалуй, не за чем использовать пробный период, когда
подписка уже активна

    public override void Unsubscribe(Service service)
    {
      Console.WriteLine("Подписка отменена!\n");
      service.state = new WithoutSubscriptionState();
      service.isNewUser = false;
    }

    // Наследуемые методы использования сервисов приложения
    public override void DownlandMusic()
    {
      Console.WriteLine("Музыка скачивается...");
      Console.WriteLine("Завершено!\n");
    }

    public override void OrderFood()
    {
      Console.WriteLine("Курьер доставит заказ в течение 10-ти
минут.\n");
    }

    public override void WatchFilm()
    {
      Console.WriteLine("Наслаждайтесь просмотром!\n");
    }
  }
```

WithoutSubscriptionState

```
internal class WithoutSubscriptionState : State
  {
    // Методы изменения состояния
    public override void BuySubscription(Service service)
    {
      service.state = new SubscriptionState();
    }

    public override void Unsubscribe(Service service)
    { }
```

```
    public override void GetTrialPeriod(Service service)
    {
      if(service.isNewUser == true){
        service.state = new TrialState();
      }
      else{
        Console.WriteLine("Вы уже использовали пробный
период\n");
      }
    }

    // Наследуемые методы использования сервисов приложения
    public override void OrderFood()
    {
      Console.WriteLine("Курьер доставит заказ в течение
часа.\n");
    }

    public override void DownlandMusic()
    {
      Console.WriteLine("Вы не можете скачать музыку на свое
устройство!\n");
    }

    public override void WatchFilm()
    {
      Console.WriteLine("*Реклама*");
      Console.WriteLine("Наслаждайтесь просмотром!");
      Console.WriteLine("*Реклама*\n");
    }
  }
```

TrialState

Данный класс содержит счетчик, который определяет сколько раз пользовались продуктами сервиса. При превышении лимита продукты блокируются.

```
internal class TrialState : State
  {
    // счетчик, сколько раз были использованы сервисы
приложения
    private int count = 0;
    // Наследуемые методы изменения состояния
    public override void BuySubscription(Service service)
    {
      Console.WriteLine("Подписка оформлена!");
      service.state = new SubscriptionState();
    }
```

```
    public override void GetTrialPeriod(Service service)
    {
      Console.WriteLine("Пробный период уже действует!\n");
    }
    public override void Unsubscribe(Service service)
    {
      service.state = new WithoutSubscriptionState();
      service.isNewUser = false;
    }

    // Наследуемые методы использования сервисов приложения
    public override void DownlandMusic()
    {
      if(count < 3){
        Console.WriteLine("Музыка скачивается...");
        Console.WriteLine("Завершено!");
        Console.WriteLine("Скачанная музыка станет недоступна
по завершению пробного периода.\n");
        count += 1;
      }
      else{
        Console.WriteLine("Пробный период завершен. Купите или
отмените подписку.\n");
      }
    }

    public override void OrderFood()
    {
      if (count < 3)
      {
        Console.WriteLine("Курьер доставит заказ в течение 15-
ти минут.\n");
        count += 1;
      }
      else
      {
        Console.WriteLine("Пробный период завершен. Купите или
отмените подписку.\n");
      }
    }
```

```
    public override void WatchFilm()
    {
      if (count < 3)
      {
        Console.WriteLine("Наслаждайтесь просмотром!\n");
        count += 1;
      }
      else
      {
        Console.WriteLine("Пробный период завершен. Купите или
отмените подписку.\n");
      }
    }
  }
```

Класс-контекст:

Хранит в себе текущее состояние, а также вызывает необходимые методы состояния.

```
 internal class Service
  {
    // приложение (Context) хранит в себе текущее состояние
    public State state { get; set; }

    // А также сохраняет информацию, новый ли это
пользователь, которое изменится при смене состояния
    public bool isNewUser;

    // Конструктор по умолчанию содержит в себе информацию:
    // Состояние подписки - без подписки
    // Пользователь новый
    public Service()
    {
      state = new WithoutSubscriptionState();
      isNewUser = true;
    }

    // Другой конструктор предусматривает, что состояние может
быть отличным от "Без подписки"
    // с начала пользования приложением
    public Service(State _state){
      state = _state;
      isNewUser = true;
    }
```

```
    // Методы изменения состояния подписки
    // Получить пробный период, причем получить его могут
только новые пользователи
    public void GetTrialPeriod(){
      if(isNewUser){
        state.GetTrialPeriod(this);
        Console.WriteLine("Пробная версия активирована!\n");
      }
      else{
        Console.WriteLine("Вы уже использовали пробную
версию.\n");
      }
    }

    public void BuySubscription(){
      state.BuySubscription(this);
      Console.WriteLine("Подписка куплена!\n");
    }

    public void Unsubscribe(){
      state.Unsubscribe(this);
      Console.WriteLine("Подписка отменена!\n");
    }

    // Далее методы вызова методов текущего состояния
    public void WatchFilm(){
      state.WatchFilm();
    }
    public void OrderFood(){
      state.OrderFood();
    }
    public void DownlandMusic(){
      state.DownlandMusic();
    }
  }
```

Программа:

```
 public class Program{

    public static void Main(){
      // Создаем новое приложение, по умолчанию создастся
новый пользователь с неактивной подпиской
      Service service = new Service();
      // Попробуем использовать музыкальный сервис и
видеохостинг без подписки
      service.DownlandMusic();
      service.WatchFilm();
      // Смотреть с рекламой не очень нравится, поэтому
активируем пробный период
      service.GetTrialPeriod();
```

```
        // И опробуем несколько сервисов
        service.OrderFood();
        service.OrderFood();
        service.WatchFilm();
        service.WatchFilm();
        // Пробный период закончился, необходимо отписаться... И
попробуем заново активировать пробный
        service.Unsubscribe();
        service.GetTrialPeriod();
        // Ничего не выходит, поэтому покупаем подписку и
наслаждаемся полным функционалом приложения
        service.BuySubscription();
        service.DownlandMusic();
        service.WatchFilm();
    }
  }
```

Результат выполнения:

```
Вы не можете скачать музыку на свое устройство!

*Реклама*
Наслаждайтесь просмотром!
*Реклама*

Пробная версия активирована!

Курьер доставит заказ в течение 15-ти минут.

Курьер доставит заказ в течение 15-ти минут.

Наслаждайтесь просмотром!

Пробный период завершен. Купите или отмените подписку.

Подписка отменена!

Вы уже использовали пробную версию.

Подписка куплена!

Музыка скачивается...
Завершено!

Наслаждайтесь просмотром!
```

Нюансы State и Strategy

- Как можно заметить, структура паттернов State и Strategy очень схожа. Однако они не стоит забывать о задачах каждого из паттернов: у Стратегии это определение однотипных взаимозаменяемых стратегий, у Состояния же - определения поведения в зависимости от состояния объекта.

- В Стратегии конкретные стратегии не знают о существовании друг друга. Они никак не взаимодействуют, Client лишь меняет их по необходимости.
  В Состоянии же конкретные состояния не только знают, что есть другие, но еще и сами могут менять состояние главного объекта.

- Оба паттерна используют композицию, чтобы менять поведение основного объекта, делегируя работу другим объектам.

- Состояние можно рассматривать, как надстройку над Стратегией

Задание:

Реализовать Машину Тьюринга и решить одну из задач с помощью паттерна Состояние.

Для удобства можно использовать на ленте трехзначный алфавит: 0, 1, x

Пример ленты: ..xx0001000101010xx..

Начинать и заканчивать работу необходимо на крайнем левом известном значении (отмечено подчеркиванием).

Вариант 1:

Реализовать сложение двух натуральных чисел, где последовательностью единиц обозначается само число, нулем - пробелом между двумя числами (11111 - 4, 111 - 2, 11 - 1, 0 - 1 (сделано специально так, чтобы можно было отличить ноль от пробела))

На вход: x11011x (2 + 1)

Выход: x1111xxx (3)

Вариант 2:

Лабораторная работа №6

«Реализация одного из поведенческих паттернов проектирования»

Цель работы: научиться применять поведенческие паттерны проектирования.

Продолжительность работы - 4 часа.

Содержание

Поведенческие паттерны

На предыдущей лабораторной работе вы уже познакомились с поведенческими паттернами проектирования. Как вы бы могли заметить, некоторые из них довольно похожи по структуре. Однако отличаются задачи, которые решают эти паттерны.

В этой лабораторной работе вы продолжите знакомство с паттернами поведения, но для начала вспомним их задачи.

Поведенческие паттерны определяют алгоритмы и способы реализации взаимодействия различных объектов и классов.

Их ключевая задача - определить связи, взаимодействия и зависимости объектов, увеличивая гибкость и масштабируемость их использования.

Список поведенческих паттернов (GoF):

- Цепочка обязанностей (Chain of responsibility)
- Команда (Command)
- Интерпретатор (Interpreter)
- Итератор (Iterator)
- Посредник (Mediator)
- Хранитель (Memento)

- Наблюдатель (Observer)
- Состояние (State)
- Стратегия (Strategy)
- Шаблонный метод (Template method)
- Посетитель (Visitor)

Паттерн Команда (Command)

Назначение:

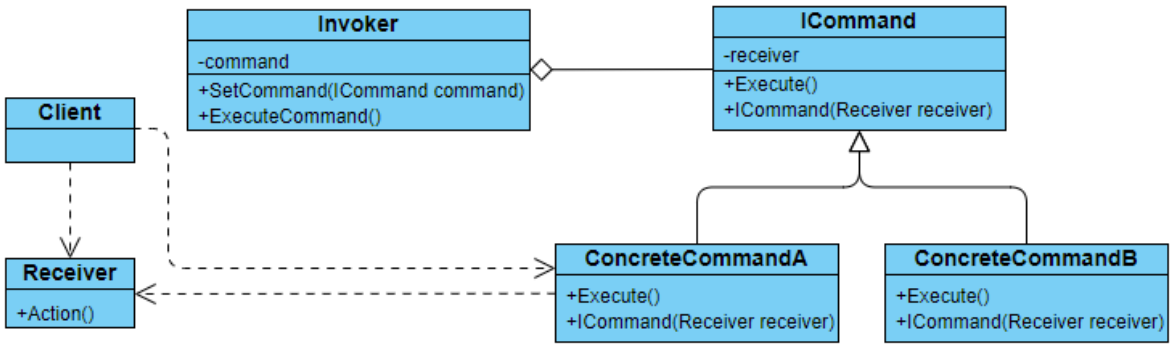
Паттерн Команда - поведенческий паттерн, позволяющий представлять запрос в виде объекта, что позволяет использовать его, как аргумент метода. С помощью этого запросы (команды) можно конфигурировать, ставить в очередь, протоколировать, а также поддерживать откат операций.

Применимость:

Применять паттерн Стратегия целесообразно, когда:

- существует необходимость в параметризации объектов (на разные команды - разное поведение)
- необходимо формировать очередь запросов, а также обеспечить отмену операций
- необходимо логировать изменения
- система должна быть основана на транзакциях

Структура:



Участники:

Command - интерфейс для выполнения операций.

ConcreteCommand - конкретная реализация команды. Реализует метод +Execute(), который вызывает соответствующие операции объекта класса Receiver.

Client - создает одну или несколько конкретных команд и устанавливает получателя - Receiver.

Invoker - вызывает команду для выполнения запроса.

Receiver - получатель, определяющий последовательность действий, которые должны выполниться в результате запроса.

Результаты применения:

Достоинства:

- Позволяет создавать очередь операций с возможной отменой и повтором
- С помощью команды можно комбинировать несколько операций, создавая все более сложные

Недостатки:

- усложняет код из-за множества дополнительных классов

Примеры из жизни:

- Пульт дистанционного управления телевизора представляет собой набор команд (переключить канал, прибавить громкость и тд). К тому же в некоторых телевизорах сохраняется очередь из команд с возможностью отмены.
- Калькулятор, хранящий в себе команды Add, Sub, Div, Mul, а также выполняющий операции Undo и Redo является примером "Команды"

Паттерн Посетитель (Visitor)

Назначение:

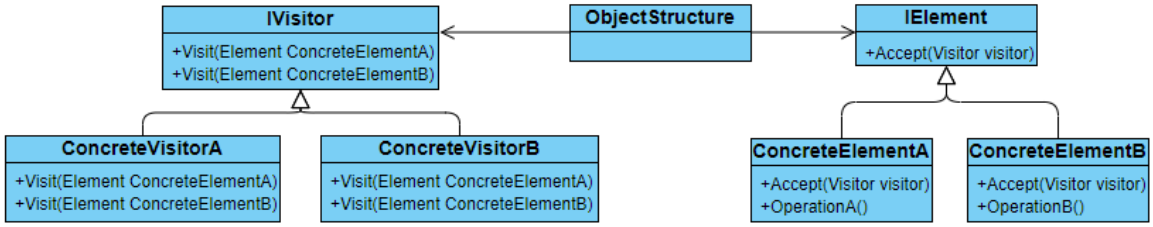
Посетитель - поведенческий паттерн, позволяющий определить новые операции для объектов классов без их изменения.

Применимость:

Применять паттерн Посетитель целесообразно, когда:

- необходимо сделать одну и ту же операцию над разнородными объектами.
- необходимо определить одинаковый метод для нескольких классов, не меняя их структуру.
- множество объектов изменяются редко, но новые операции нужно добавлять часто.

Структура:



Участники:

Visitor - интерфейс посетителя, который определяет методы для каждого объекта Element, причем для языков, поддерживающих перегрузку названия этих методов могут быть одинаковыми, но аргументы должны отличаться.

ConcreteVisitor - посетители, реализующие интерфейсы, определенные в Visitor.

Element - элемент определяет метод +Accept(Visitor visitor), который позволяет принимать посетителя.

ConcreteElement - конкретные элементы, реализующие метод +Accept(Visitor visitor). Задача метода - вызвать необходимый метод посещения, который соответствует данному элементу.

ObjectStructure - набор объектов типа Element. Иногда этот объект может быть заменен клиентом (Client, хранящий в себе коллекцию или любой другой сложный объект)

Результаты применения:

Достоинства:

- Упрощает добавление новых операций, которые применяются к сложным структурам

- Позволяет объединять в одном классе сходные операции.

Недостатки:

- Если иерархия элементов будет меняться, то использование паттерна будет под вопросом
- Разрушение инкапсуляции

Примеры из жизни:

- У преподавателя есть методички по нескольким предметам (например, математический анализ, дифференциальные уравнения и тд), а так же несколько групп студентов у которых он преподает. Соответственно перед занятием преподаватель должен выбрать необходимую страницу в методичке в зависимости от предмета и пройденного материала с конкретной группой.
- Работник колл-центра предлагает клиентам перейти на специальный тариф мобильного оператора. Причем это предложение зависит от того, сколько они тратят минут звонков, гигабайт трафика интернета, количество СМС и, конечно, средств на оплату по тарифу.

Пример кода для Visitor:

Допустим существует 2 приложения для хранения данных: Excel и SQLite. Заполняются таблицы в них по-разному: в первом случае нужно вписывать в поля, во втором - с помощью запросов INSERT.

Так же существует 2 вида пользователей с разными полями: студент (имя, предмет, оценка) и преподаватель (имя, кафедра, средний балл по дисциплине).

Необходимо создать сервис, который может заполнять данные пользователей в любом приложении для хранения данных.

Интерфейс IPerson (Element):

Содержит метод для выполнения какого-либо алгоритма в зависимости от принимаемого посетителя.

```
  // Интерфейс-Element, содержащий единственный метод
"Принятия" принимающий любого посетителя
  public interface IPerson
  {
    public void Accept(IVisitor visitor);
  }
```

Наследуемые от IPerson классы Student и Teacher (ConcreteElement):

```
public class Student : IPerson
  {
    // Пояснения полей и конструкторов избыточны
    public string Name { get; set; }
    public string Subject { get; set; }
    public int Mark { get; set; }
    public Student(){
      Name = "Ivan";
      Subject = "Math";
      Mark = 5;
    }
    public Student(string name, string subject, int mark){
      Name = name;
      Subject = subject;
      Mark = mark;
    }

    // метод принятия посетителя, которому передается сам
объект Студента
    public void Accept(IVisitor visitor)
    {
      visitor.Visit(this);
    }
  }

  public class Teacher : IPerson
  {
    public string Name { get; set; }
    public string Departament { get; set; }
    public double AvgMark { get; set; }
    public Teacher(){
      Name = "Ivan";
      Departament = "Dep of High Math";
      AvgMark = 3.45;
    }
    public Teacher(string name, string dep, int avgMark)
    {
      Name = name;
      Departament = dep;
      AvgMark = avgMark;
    }
    // метод принятия посетителя, которому передается сам
объект Преподавателя
    public void Accept(IVisitor visitor)
    {
      visitor.Visit(this);
    }
  }
```

Интерфейс IVisitor:

Содержит информацию о там "кого он будет посещать"

```
  // Интерфейс посетителя со всеми методами, кого он
собирается посещать
  public interface IVisitor
  {
    public void VisitStudent(Student student);
    public void VisitTeacher(Teacher teacher);
  }
```

Наследуемые от IVisitor классы SqlVisitor и ExcelVisitor (ConcreteVisitor):

```
// Класс, наследующий интерфейс IVisitor
  public class ExcelVisitor : IVisitor
  {
    // "Посещение" Студентов, или добавление элемента в
таблицу Excel
    public void Visit(Student student)
    {
      string table = "|    Name    |      Subject      |    Mark
|";
      string newValue = student.Name + "\t\t" +
student.Subject + "\t\t" + student.Mark.ToString();
      Console.WriteLine(table);
      Console.WriteLine(newValue);
Console.WriteLine("-------------------------");
    }
    // Аналогичное "Посещение" Преподавателей
    public void Visit(Teacher teacher)
    {
      string table = "|    Name    |    Departament    |
Average Mark    |";
      string newValue = teacher.Name + "\t\t" +
teacher.Departament + "\t\t" + teacher.AvgMark.ToString();
      Console.WriteLine(table);
      Console.WriteLine(newValue); Console.WriteLine("--------
-------------------------------");
    }
  }
```

```
// Класс, наследующий интерфейс IVisitor
  public class SqlVisitor : IVisitor
  {
    // "Посещение" Студентов, или создание SQL запроса
    public void Visit(Student student)
    {
      string newValue = $@"INSERT INTO Students (Name,
Subject, Mark) VALUES
('{student.Name}','{student.Subject}',{student.Mark})";
      Console.WriteLine(newValue);   Console.WriteLine("------
----------------------------------");
    }
    // Аналогичное "Посещение" Преподавателей
    public void Visit(Teacher teacher)
    {
      string newValue = $@"INSERT INTO Teachers (Name,
Departament, AvgMark) VALUES
('{teacher.Name}','{teacher.Departament}',{teacher.AvgMark})";
      Console.WriteLine(newValue);      Console.WriteLine("---
-----------------------------------");
    }
  }
```

Класс Database для хранения списка пользователей (ObjectStructure)

```
 public class Database
  {
    // Список, хранящийся в структуре
    public List<IPerson> people = new List<IPerson>();

    // Методы добавления новых людей и удаления
    public void Add(IPerson person){
      people.Add(person);
    }
    public void Remove(IPerson person){
      people.Remove(person);
    }

    // "Действие" совершаемое со всем людьми в базе данных
    public void Accept(IVisitor visitor){
      foreach(IPerson person in people){
        person.Accept(visitor);
      }
    }
  }
```

Программа:

```
public static class Program{
    static void Main()
    {
      // Создадим структруку
      Database database = new Database();
      // А также несколько студентов и преподавателей
      Student student1 = new Student{ Name = "Olga", Subject =
"Physics", Mark = 4};
      Student student2 = new Student { Name = "Oleg", Subject
= "P.E.", Mark = 5};
      Teacher teacher = new Teacher{ Name = "Petrov",
Departament = "History dep", AvgMark = 4.72};

      // И добавим их в структуру
      database.Add(student1);
      database.Add(student2);
      database.Add(teacher);

      // Вызов одного и того же метода с разными посетителями
даст различные результаты
      database.Accept(new ExcelVisitor());
      database.Accept(new SqlVisitor());
    }
  }
```

Результат выполнения:

```
|   Name   |     Subject     |    Mark   |

Olga            Physics         4

----------------------------------------

|   Name   |     Subject     |    Mark   |

Oleg            P.E.            5

----------------------------------------

|   Name   |   Departament   |    Average Mark   |

Petrov          History dep             4,72

----------------------------------------

INSERT INTO Students (Name, Subject, Mark) VALUES
('Olga','Physics',4)

----------------------------------------

INSERT INTO Students (Name, Subject, Mark) VALUES
('Oleg','P.E.',5)

----------------------------------------

INSERT INTO Teachers (Name, Departament, AvgMark) VALUES
('Petrov','History dep',4,72)

----------------------------------------
```

Нюансы Visitor и Command

- Компоновщик может выступать ObjectStructure в Посетителе, таким образом можно совершать какие-либо действия над всем деревом при помощи Посетителя.
- Посетитель в каком-то смысле является расширением Команды, так как позволяет совершать действия с несколькими видами получателей.
- Команда и Стратегия похожи по структуре, но отличаются применимостью: Команда превращает любые запросы в объекты, которые можно хранить, дополнять, удалять и т.д. Стратегия же предоставляет возможность заменять одни действия другими.