Паттерны:
Visitor(Посетитель),
Iterator,
Factory(Фабрика)

Паттерн Visitor(Посетитель)

Visitor(Посетитель)-поведенческий паттерн.

Согласно Википедии, поведенческие шаблоны (behavioral patterns) — шаблоны проектирования, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов.

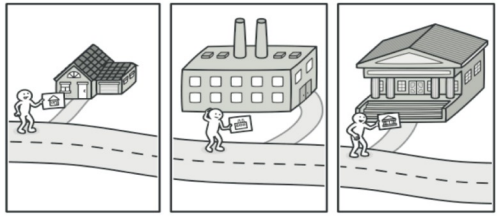
Проще говоря, поведенческие паттерны связаны с распределением обязанностей между объектами и описывают структуру и шаблоны для передачи сообщений / связи между компонентами.

Паттерн Visitor(Посетитель)

Посетитель добавляет новую функциональность к уже существующим классам, не изменяя исходный код.

При изменении visitor нет необходимости изменять обслуживаемые классы.

Паттерн Visitor(Посетитель)



У страхового агента приготовлены полисы для разных видов организаций.

Аналогия из жизни:

Представьте начинающего страхового агента, жаждущего получить новых клиентов. Он беспорядочно посещает все дома в округе, предлагая свои услуги. Но для каждого из посещаемых типов домов у него имеется особое предложение.

-Придя в дом к обычной семье, он предлагает оформить медицинскую страховку.

-Придя в банк, он предлагает страховку от грабежа.

-Придя на фабрику, он предлагает страховку предприятия от пожара и наводнения.

Паттерн Visitor(Посетитель)

Когда использовать данный паттерн?

-Когда имеется много объектов разнородных классов с разными интерфейсами, и требуется выполнить ряд операций над каждым из этих объектов

-Когда классам необходимо добавить одинаковый набор операций без изменения этих классов

-Когда часто добавляются новые операции к классам, при этом общая структура классов стабильна и практически не изменяется

Паттерн Visitor(Посетитель)

Реализация:

1. Добавьте метод accept(Visitor) в иерархию «элемент».
2. Создайте базовый класс Visitor и определите методы visit() для каждого типа элемента.
3. Создайте производные классы Visitor для каждой операции, исполняемой над элементами.
4. Клиент создаёт объект Visitor и передаёт его в вызываемый метод accept().

Паттерн Visitor(Посетитель)

```
4   // Элементы в которые будет приходить посетитель
5   class Monkey {
6       shout() {
7           console.log('Ooh oo aa aa!')
8       }
9   }
10
11  class Lion {
12      roar() {
13          console.log('Roaaar!')
14      }
15  }
16
17  class Dolphin {
18      speak() {
19          console.log('Tuut tuttu tuutt!')
20      }
21  }
22
```

Паттерн Visitor(Посетитель)

```
class Monkey {
    shout() {
        console.log('Ooh oo aa aa!')
    }

    accept(operation) {
        operation.visitMonkey(this)
    }
}

class Lion {
    roar() {
        console.log('Roaaar!')
    }

    accept(operation) {
        operation.visitLion(this)
    }
}

class Dolphin {
    speak() {
        console.log('Tuut tuttu tuutt!')
    }

    accept(operation) {
        operation.visitDolphin(this)
    }
}
```

```

// Посетители
class Visitor {
    visitMonkey(monkey) {
        throw new Error(`В ${this.constructor.name} не описан метод visitMonkey()`)
    }
    visitLion(lion) {
        throw new Error(`В ${this.constructor.name} не описан метод visitLion()`);
    }
    visitDolphin(dolphin) {
        throw new Error(`В ${this.constructor.name} не описан метод visitDolphin()`);
    }
}

```

Паттерн Visitor(Посетитель)

```
50  class voiceVisitor extends Visitor {
51      visitMonkey(monkey){
52          monkey.shout()
53      }
54      visitLion(lion){
55          lion.roar()
56      }
57      visitDolphin(dolphin){
58          dolphin.speak()
59      }
60  }
61
```

```
63  const monkey = new Monkey()
64  const lion = new Lion()
65  const dolphin = new Dolphin()
66  const voicer = new voiceVisitor();
67
68
69  // Пробуем первого посетителя
70  monkey.accept(voicer)     // Ooh oo aa aa!
71  lion.accept(voicer)       // Roaaar!
72  dolphin.accept(voicer)    // Tuut tutt tuutt!
73
74
```

Паттерн Visitor(Посетитель)

```
class jumpVisitor extends Visitor {
    visitMonkey(monkey) {
        console.log('Jumped 20 feet high! on to the tree!')
    }
    visitLion(lion) {
        console.log('Jumped 7 feet! Back on the ground!')
    }
    visitDolphin(dolphin) {
        console.log('Walked on water a little and disappeared')
    }
}

const jumper = new jumpVisitor();

monkey.accept(jumper)    // Jumped 20 feet high! on to the tree!
lion.accept(jumper)      // Jumped 7 feet! Back on the ground!
dolphin.accept(jumper)   // Walked on water a little and disappeared
```

Паттерн Visitor(Посетитель)

Недостатки

-Паттерн не оправдан, если иерархия элементов часто меняется.

-Может привести к нарушению инкапсуляции элементов.

Паттерн Iterator

Iterator — поведенческий шаблон проектирования.

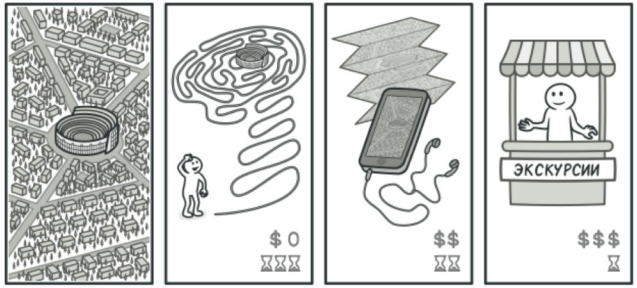
Паттерн Итератор предоставляет доступ к элементам объекта, не раскрывая способ их внутреннего представления.

Паттерн Iterator

Перебор элементов выполняется объектом итератора, а не самой коллекцией.

Особенностью полноценно реализованного итератора является то, что код, использующий итератор, может ничего не знать о типе итерируемого агрегата.

Паттерн Iterator



Варианты прогулок по Риму.

Аналогия из жизни

Вы планируете полететь в Рим и обойти все достопримечательности за пару дней. Но приехав, вы можете долго петлять узкими улочками, пытаясь найти Колизей.

Если у вас ограниченный бюджет — не беда. Вы можете воспользоваться виртуальным гидом, скачанным на телефон, который позволит отфильтровать только интересные вам точки. А можете плюнуть и нанять локального гида, который хоть и обойдётся в копеечку, но знает город как свои пять пальцев, и сможет посвятить вас во все городские легенды.

Таким образом, Рим выступает коллекцией достопримечательностей, а ваш мозг, навигатор или гид — итератором по коллекции. Вы, как клиентский код, можете выбрать один из итераторов, отталкиваясь от решаемой задачи и доступных ресурсов.

Паттерн Iterator

Когда использовать данный паттерн?

- Когда необходимо осуществить обход объекта без раскрытия его внутренней структуры;
- Когда имеется набор составных объектов, и надо обеспечить единый интерфейс для их перебора;
- Когда необходимо предоставить несколько альтернативных вариантов перебора одного и того же объекта;

Паттерн Iterator

Реализация:

Создайте общий интерфейс итераторов. Обязательный минимум — это операция получения следующего элемента коллекции. Но для удобства можно предусмотреть и другое. Например, методы для получения предыдущего элемента, текущей позиции, проверки окончания обхода и прочие.

Паттерн Iterator

```
18  // using Iterator
19  class IteratorClass {
20      constructor(data) {
21          this.index = -1;
22          this.data = data;
23          this.done = false;
24          this.checkDataType(data);
25      }
26
27      checkDataType(data) { // Проверяем получен массив или объект
28          if (Array.isArray(data)) { // Перебор массива
29              this.collectionLength = this.data.length - 1;
30              this.collection = this.data;
31              this.type = 'array';
32          } else if (!Array.isArray(data) && typeof data == 'object') { // Перебор объекта
33              this.collectionLength = Object.keys(this.data).length - 1;
34              this.collection = Object.values(this.data);
35              this.type = 'object';
36          } else {
37              throw new Error(`${this.constructor.name} получил data которую не может итерировать - ${typeof data}`)
38          }
39      }
```

Паттерн Iterator

```
40
41      next() {
42          if (this.index < this.collectionLength) {
43              this.index++;
44              return {key: Object.keys(this.data)[this.index], value: this.collection[this.index],
45                      done: false, index: this.index, type: this.type};
46          } else {
47              this.done = true;
48              return {done: true, msg: 'достигнут конец коллекции', index: this.index, type: this.type};
49          }
50      }
51
52      prev() {
53          if (this.done) {
54              this.done = false;
55              this.index = this.collectionLength + 1;
56          }
57          this.index--;
58          if (this.index > -1) {
59              return {key: Object.keys(this.data)[this.index], value: this.collection[this.index],
60                      done: false, index: this.index, type: this.type};
61          } else {
62              this.index = -1;
63              return {done: this.done, msg: 'достигнуто начало коллекции', index: this.index, type: this.type};
64          }
65      }
66  }
67
```

Паттерн Iterator

```
// usage Iterator

console.log('Итерируем массив:');
const gen = new IteratorClass(['Hi', 'Hello', 'Bye']);
console.log(gen.next()); // {key: "0", value: "Hi", done: false,...}
console.log(gen.next()); // {key: "1", value: "Hello", done: fal...}
console.log(gen.next()); // {key: "2", value: "Bye", done: false...}
console.log(gen.next()); // {done: true, msg: "достигнут конец к...}
console.log(gen.prev()); // {key: "2", value: "Bye", done: false...}
console.log(gen.prev()); // {key: "1", value: "Hello", done: fal...}


console.log('Итерируем объект:');
const obj = new IteratorClass({a: '1h', b: '2o', c: '3Bye'});
console.log(obj.next()); // {key: "a", value: "1h", done: false,...}
console.log(obj.next()); // {key: "b", value: "2o", done: false,...}
console.log(obj.next()); // {key: "c", value: "3Bye", done: fals...}
console.log(obj.next()); // {done: true, msg: "достигнут конец к...}


console.log('Итерируем строку:'); // выдаст ошибку, итератор не настроен на строки
const string = new IteratorClass('1hello');
console.log(string.next()); // Error: IteratorClass получил data которую не может итерировать - string
console.log(string.next());
```

Паттерн Iterator

Недостатки

Не оправдан, если можно обойтись простым циклом.

Паттерн Factory

Фабричный метод — это порождающий паттерн проектирования.

Основная идея в том, что мы создаем фабрику, которая может создавать нам обьекты.

И сразу же возникает вопрос, а почему мы не можем просто использовать оператор new, чтобы создавать обьекты? Есть ситуации, когда мы хотим скрыть снаружи реализацию создания обьекта и в этом случае нам поможет паттерн Factory.

Паттерн Factory

Аналогия из жизни

Если нам нужна гитара, мы можем выпилить деку, самостоятельно сделать струны из никеля, склеить корпус, сделать гриф, расставить лады и натянуть струны... А можем сходить в магазин и взять гитару, созданную на фабрике — в этом случае нам уже не требуется знать, что именно надо сделать, чтобы гитару создать.

Паттерн Factory

Когда использовать данный паттерн?

- Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код.

- Используйте фабрику, если создание объекта сложнее, чем 1-2 строки кода. Особенно полезно использовать этот шаблон, когда для создания объекта требуется применить расчёты или получить дополнительные данные.

Паттерн Factory

```
26
27  class Employee {
28    create (type) {
29      let employee;
30      if (type === 'fulltime') {
31        employee = new FullTime()
32      } else if (type === 'parttime') {
33        employee = new PartTime()
34      } else if (type === 'temporary') {
35        employee = new Temporary()
36      } else if (type === 'contractor') {
37        employee = new Contractor()
38      }
39      employee.type = type;
40      employee.say = function () {
41        console.log(`${this.type}: rate ${this.rate}/hour`)
42      }
43      return employee;
44    }
45  }
46
47
```

Паттерн Factory

```
2
3   class FullTime {
4       constructor () {
5         this.rate = '$12';
6       }
7     }
8
9     class PartTime {
10      constructor () {
11        this.rate = '$11';
12      }
13    }
14
15    class Temporary {
16      constructor () {
17        this.rate = '$10';
18      }
19    }
20
21    class Contractor {
22      constructor () {
23        this.rate = '$15';
24      }
25    }
```

Паттерн Factory

```
49
50  const factory = new Employee();
51  fulltime = factory.create('fulltime');
52  parttime = factory.create('parttime');
53  temporary = factory.create('temporary');
54  contractor = factory.create('contractor');
55
56  fulltime.say(); // fulltime: rate $12/hour
57  parttime.say(); // parttime: rate $11/hour
58  temporary.say(); // temporary: rate $10/hour
59  contractor.say(); // contractor: rate $15/hour
60
61
```

Паттерн Factory

Недостатки

Может создать дополнительную сложность в приложении там, где ее можно было бы избежать.