Министерство Образования Республики Молдова

Технический Университет Молдовы

Департамент программной инженерии и автоматики

**Отчет**

Лабораторная работа №3

Тема: «Decorator»

Выполнил: TI-197-Герца Николай

Проверил: Булдумаг Олег

Кишинев-2021

**Суть паттерна:**

- Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

**Что будет происходить:**

- Создаю базовый класс TaxiType, который является базовым для любого вызова такси. Затем создаю класс для такси-комфорт и более конкретно указываю параметры поездки. Затем добавляю два декоратора: опции, которые можно применить к своему такси. Каждый декоратор принимает тип такси и добавляет к его стоимости, в следствии чего меняется итоговая стоимость

**Диаграмма UML:**



**Реализация:**

class TaxiType {

  constructor() {

    this.price = 2;

    this.type = "Economy";

  }

  getPrice() {

    return this.price;

  }

  getType() {

    return this.type;

  }

}

class Comfort extends TaxiType {

  constructor() {

*super*();

    this.price = 3;

    this.type = "Comfort";

  }

}

class OptionAnimal {

  constructor(*type*) {

    this.type = *type*;

  }

  getPrice() {

    return this.type.getPrice() + 1;

  }

  getType() {

    return `${this.type.getType()} with animal`;

  }

}

class OptionSmoke {

  constructor(*type*) {

    this.type = *type*;

  }

  getPrice() {

    return this.type.getPrice() + 2;

  }

  getType() {

    return `${this.type.getType()} with smoke`;

  }

}

let econom = new TaxiType();

console.log(econom.getType());

console.log(econom.getPrice());

econom = new OptionAnimal(econom);

console.log(econom.getPrice(), econom.getType());

let comfort = new Comfort();

comfort = new OptionAnimal(comfort);

console.log(comfort.getPrice(), comfort.getType());

comfort = new OptionSmoke(comfort);

console.log(comfort.getPrice(), comfort.getType());



**Применение:**

- когда вам нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует;

- когда нужно много подклассов;

- когда нельзя расширить обязанности объекта с помощью наследования.

**Преимущества**

- большая гибкость, чем у наследования;

- позволяет добавлять обязанности на лету;

- можно добавлять несколько новых обязанностей сразу;

- позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.

**Недостатки:**

- трудно конфигурировать многократно обёрнутые объекты;

- обилие крошечных классов.