Министерство Образования Республики Молдова

Технический Университет Молдовы

Департамент программной инженерии и автоматики

**Отчет**

Лабораторная работа №3

Тема: «Facade. Proxy»

Выполнил: TI-197-Герца Николай

Проверил: Булдумаг Олег

Кишинев-2021

**Суть паттерна:**

• Facade - это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку. Его задача скрыть сложную логику за простым фасадом, то есть собрать различные сложные структуры, объединить их, и выдать простой способ манипуляции.

**Что будет происходить:**

• Будет главный класс, который будет производить регистрацию нового пользователя в системе такси. Вводим дополнительную абстракцию над методами в Registration, чтобы объединить их в один и на выходе вернуть один общий метод. У нас есть один удобный метод манипуляции, и мы не знаем, что находится под капотом и методов, которые находятся внутри.

**Диаграмма UML:**



**Реализация:**

class Registration {

  constructor(name, surname, phone, card) {

    this.name = name;

    this.surname = surname;

    this.phone = phone;

    this.card = card;

  }

  setName() {

    console.log(`Имя задано: ${this.name}`);

  }

  setSurname() {

    console.log(`Фамилия задана: ${this.surname}`);

  }

  setPhone() {

    console.log(`Телефон задан: ${this.phone}`);

  }

  setCard() {

    console.log(`Банковская карта задана: ${this.card}`);

  }

}

class SetUser {

  constructor(user) {

    this.user = user;

  }

  createUser() {

    this.user.setName();

    this.user.setSurname();

    this.user.setPhone();

    this.user.setCard();

  }

}

const registration = new SetUser(new Registration("Maxim", "Neescas", "37379010101", "23423123124124124"));

const user = registration.createUser();



**Применение:**

• Если в коде есть громоздкие реализации с запутанной логикой

• Много параметров и аргументов, влияющие на конечный результат

• Когда вы хотите разложить подсистему на отдельные слои.

**Преимущества и недостатки:**

+

• Изолирует клиентов от компонентов сложной подсистемы.

-

• Фасад рискует стать божественным объектом, привязанным ко всем классам программы.

**Суть паттерна:**

• Proxy. Паттерн, который вместо реальных объектов, предоставляет специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.

**Что будет происходить:**

• Авторизация. Пишем оригинальный класс, который содержит два метода входа и выхода в аккаунт. Затем создаем класс проверки. Класс принимает аккаунт и оборачивает ее в свою проверку. В методе login мы запрашивает идентефикацию пользователя и просим ввести логин и пароль. В случае правильного ввода, будет выполнен вход в аккаунт, в ином случае - вход не будет выполнен. На примере logout. Оборачиваем метод logout оригинального объекта в свой собственный, то есть создается прослойка между пользователем и оригинальным классом. Создали своего рода заместителя, с тем же интерфейсом, что и оригинал.

**Диаграмма UML:**



**Реализация:**

class AccountAccess {

  login() {

    console.log("Вы успешно вошли в аккаунт");

  }

  logOut() {

    console.log("Вы успешно вышли из аккаунта");

  }

}

class Security {

  constructor(account) {

    this.account = account;

  }

  login(username, password) {

    if (this.authenticateUsername(username) && this.authenticatePassword(password)) {

      this.account.login();

    } else {

      console.log("Неверные данные");

    }

  }

  authenticateUsername(username) {

    return username === "Maxim123";

  }

  authenticatePassword(password) {

    return password === "12345";

  }

  logOut() {

    this.account.logOut();

  }

}

const userOne = new Security(new AccountAccess());

userOne.login("Egor", "555");

userOne.login("Maxim123", "12345");

userOne.logOut();



**Применение:**

• Ленивая инициализация (виртуальный прокси). Когда у вас есть тяжёлый объект, грузящий данные из файловой системы или базы данных.

• Защита доступа (защищающий прокси). Когда в программе есть разные типы пользователей, и вам хочется защищать объект от неавторизованного доступа. Например, если ваши объекты — это важная часть операционной системы, а пользователи — сторонние программы (хорошие или вредоносные).

• Локальный запуск сервиса (удалённый прокси). Когда настоящий сервисный объект находится на удалённом сервере.

**Преимущества и недостатки:**

+

• Позволяет контролировать сервисный объект незаметно для клиента.

• Может работать, даже если сервисный объект ещё не создан.

• Может контролировать жизненный цикл служебного объекта.

-

• Усложняет код программы из-за введения дополнительных классов.

• Увеличивает время отклика от сервиса.