|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА** – **Российский технологический университет»**  **РТУ МИРЭА** | | | | |
| Институт ИКБ | |
| Кафедра защиты информации | |

|  |  |
| --- | --- |
| **ОТЧЁТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 5** | |
| **по дисциплине «Технологии и методы программирования»** | |
|  | |
| Выполнила студент группы БИСО-02-21 | Одинцова Ю.А. |
| Проверил | Лесько С.А. |

Москва 2023

**Практическая работа №5**

Ссылка:

<https://github.com/just-student-19/tmp_6_sem>

# Абстрактная фабрика

Абстрактная фабрика –– это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Листинг программы:

class AbstractProductA {

methodA() {

throw new Error('Ошибка в методе А!');

}

}

class ProductA1 extends AbstractProductA {

methodA() {

console.log('Продукт 1 метод A');

}

}

class ProductA2 extends AbstractProductA {

methodA() {

console.log('Продукт 2 метод A');

}

}

class AbstractProductB {

methodB() {

throw new Error('Ошибка в методе Б!');

}

}

class ProductB3 extends AbstractProductB {

methodB() {

console.log('Продукт 3 метод Б');

}

}

class ProductB4 extends AbstractProductB {

methodB() {

console.log('Продукт 4 метод Б');

}

}

class AbstractFactory {

createProductA() {

throw new Error('Ошибка создания продукта в А!');

}

createProductB() {

throw new Error('Ошибка создания продукта в Б!');

}

}

class ConcreteFactory1 extends AbstractFactory {

createProductA() {

return new ProductA1();

}

createProductB() {

return new ProductB3();

}

}

class ConcreteFactory2 extends AbstractFactory {

createProductA() {

return new ProductA2();

}

createProductB() {

return new ProductB4();

}

}

// Main

const factory1 = new ConcreteFactory1();

const productA1 = factory1.createProductA();

const productB3 = factory1.createProductB();

productA1.methodA();

productB3.methodB();

const factory2 = new ConcreteFactory2();

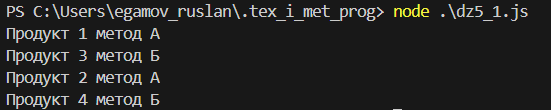
const productA2 = factory2.createProductA();

const productB4 = factory2.createProductB();

productA2.methodA();

productB4.methodB();

Скриншот выполнения программы:



**Строитель**  
  
 Строитель –– это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

Листинг программы:

class Product {

constructor() {

this.partA = "";

this.partB = "";

this.partC = "";

}

setPartA(partA) {

this.partA = partA;

}

setPartB(partB) {

this.partB = partB;

}

setPartC(partC) {

this.partC = partC;

}

show() {

console.log(`Части продукта: ${this.partA}, ${this.partB}, ${this.partC}`);

}

}

class Builder {

buildPartA() {

throw new Error('Ошибка в части А!');

}

buildPartB() {

throw new Error('Ошибка в части Б!');

}

buildPartC() {

throw new Error('Ошибка в части В!');

}

getProduct() {

throw new Error('Ошибка при получении!');

}

}

class SomeBuilder extends Builder {

constructor() {

super();

this.product = new Product();

}

buildPartA() {

this.product.setPartA("Часть A");

}

buildPartB() {

this.product.setPartB("Часть Б");

}

buildPartC() {

this.product.setPartC("Часть В");

}

getProduct() {

return this.product;

}

}

class Director {

setBuilder(builder) {

this.builder = builder;

}

construct() {

this.builder.buildPartA();

this.builder.buildPartB();

this.builder.buildPartC();

}

}

// Main

const builder = new SomeBuilder();

const director = new Director();

director.setBuilder(builder);

director.construct();

const product = builder.getProduct();

product.show();

Скриншот выполнения программы:



# Адаптер

Адаптер –– это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Листинг программы:

class Target {

request() {

}

}

class specAdapter {

specificRequest() {

console.log("Выполняем запрос!");

}

}

class Adapter extends Target {

constructor(adapter1) {

super();

this.adapter1 = adapter1;

}

request() {

this.adapter1.specificRequest();

}

}

// Main

const adapter1 = new specAdapter();

const adapter = new Adapter(adapter1);

adapter.request();

Скриншот выполнения программы:



# Посредник

Посредник –– это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс – посредник.

Листинг программы:

class Mediator {

sendMessage(message, colleague) {

}

}

class Colleague {

constructor(mediator = null) {

this.mediator = mediator;

}

receiveMessage(message) {

}

sendMessage(message) {

if (this.mediator) {

this.mediator.sendMessage(message, this);

}

}

setMediator(mediator) {

this.mediator = mediator;

}

}

class SomeColleague extends Colleague {

constructor(name, mediator = null) {

super(mediator);

this.name = name;

}

receiveMessage(message) {

console.log(`${this.name} получил: ${message}`);

}

}

class SomeMediator extends Mediator {

constructor() {

super();

this.colleagues = [];

}

addColleague(colleague) {

this.colleagues.push(colleague);

colleague.setMediator(this);

}

sendMessage(message, colleague) {

for (let col of this.colleagues) {

if (col !== colleague) {

col.receiveMessage(message);

}

}

}

}

// Main

const mediator = new SomeMediator();

const colleague1 = new SomeColleague('User1', mediator);

const colleague2 = new SomeColleague('User2', mediator);

mediator.addColleague(colleague1);

mediator.addColleague(colleague2);

colleague1.sendMessage('Hi User1!');

colleague2.sendMessage('Hello User2!');

Скриншот выполнения программы:

