**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

**отчёт   
по учебной практике**

**по дисциплине «Объектно-ориентированное программирование»**

**Тема: Шаблоны проектирования в языке Java**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3311 |  | Сапронов К.Д. |
| Преподаватель |  | Павловский М.Г. |

Санкт-Петербург

2025

**Цель практики:**

Знакомство с шаблонами проектирования и их использование в языке Java

**Задание 1.** Привести примеры 3 шаблонов различных типов: порождающий, структурный и поведенческий. Для каждого описать назначение шаблона и в каких случаях его целесообразно применять, диаграммы классов примера шаблона с описанием классов и последовательности выполнения примера шаблона, код примера с комментариями и результат работы примера.

1. Порождающий – фабричный метод.

Назначение: Шаблон проектирования фабричный метод решает проблему создания различных групп продуктов, каждая из которых обладает некоторой спецификой.

Область применени:

• Необходимо создать систему, обладающую свойством расширяемости. • По некоторым соображениям предпочтительнее, чтобы решение о том, какой объект создавать, принималось на уровне подкласса, а не суперкласса.

• В силу тех или иных обстоятельств, известно, когда нужно создать объект, но неизвестно, какой именно.

• Необходимо иметь в распоряжении несколько перекрытых конструкторов с одинаковыми списками параметров, что недопустимо в языке Java. В этом случае можно прибегнуть к шаблону Factory Method, используя разные имена классов.

Диаграмма классов:

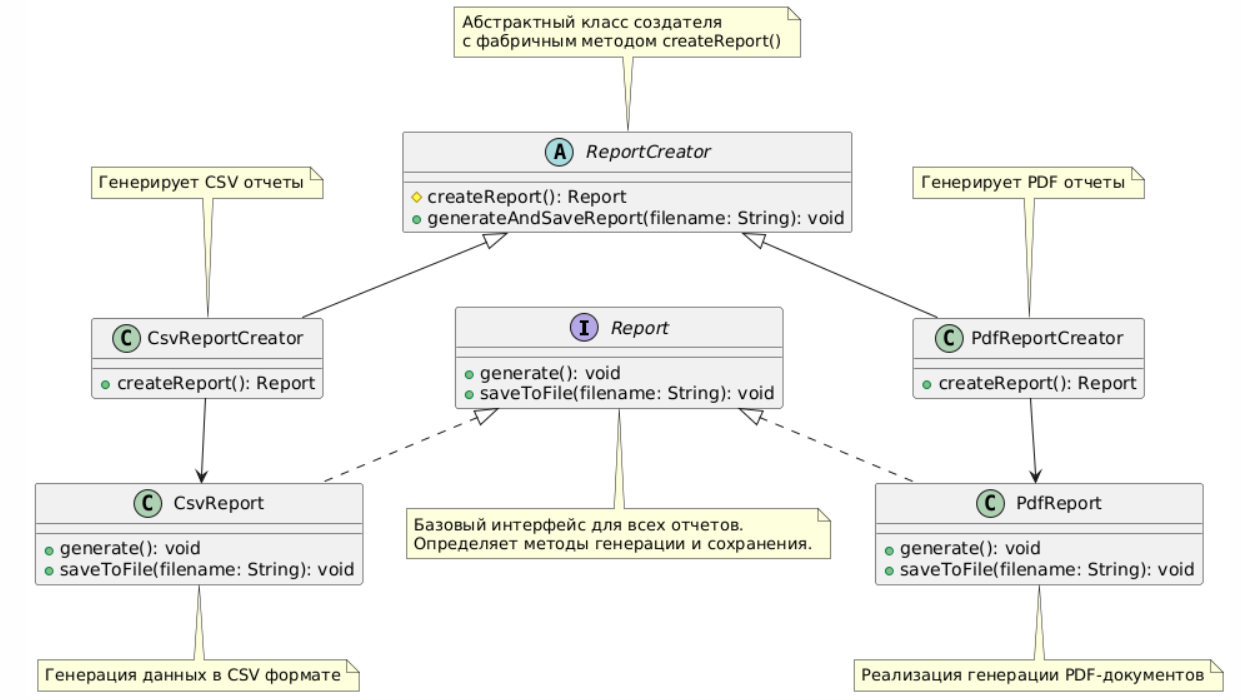
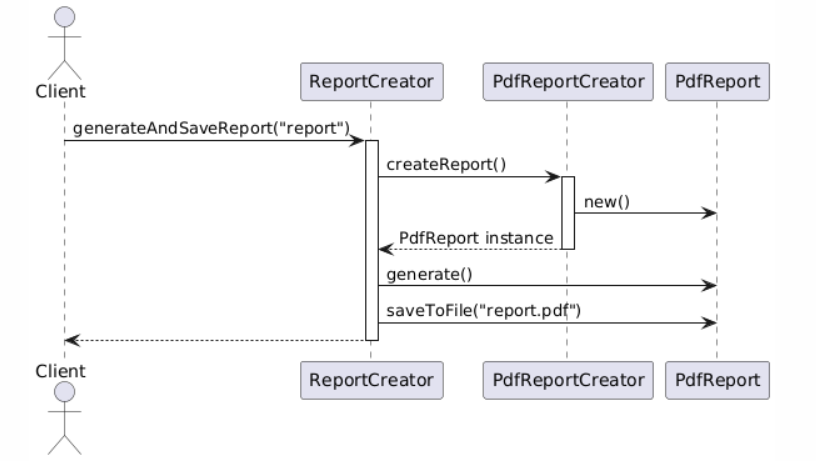


Диаграмма последовательности выполнения:

Код шаблона:

// Product interface

interface Report {

void generate();

void saveToFile(String filename);

}

// Concrete Products

class PdfReport implements Report {

public void generate() {

System.out.println("Generating PDF report with complex layout");

}

public void saveToFile(String filename) {

System.out.println("PDF saved to file " + filename + ".pdf");

}

}

class CsvReport implements Report {

public void generate() {

System.out.println("Generating CSV report with comma-separated data");

}

public void saveToFile(String filename) {

System.out.println("CSV saved to file " + filename + ".csv");

}

}

// Creator abstract class

abstract class ReportCreator {

// Factory method

abstract Report createReport();

public void generateAndSaveReport(String filename) {

Report report = createReport();

report.generate();

report.saveToFile(filename);

}

}

// Concrete Creators

class PdfReportCreator extends ReportCreator {

@Override

Report createReport() {

return new PdfReport();

}

}

class CsvReportCreator extends ReportCreator {

@Override

Report createReport() {

return new CsvReport();

}

}

// Client code

public class FactoryMethodExample {

public static void main(String[] args) {

ReportCreator creator = new PdfReportCreator();

creator.generateAndSaveReport("monthly\_report");

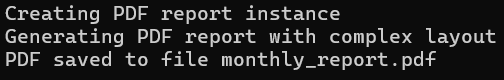
creator = new CsvReportCreator();

creator.generateAndSaveReport("data\_export");

}

}

Пример работы программы:



1. Структурный – декоратор

Назначение: Предоставление механизма для добавления или удаления функциональности компонентов без изменения их внешнего представления или функций.

Область применения:

• Необходимо осуществлять динамическое изменение свойств классов, причем незаметно для пользователя и не связываясь с ограничениями, присущими механизму наследования.

• Свойства могут подключаться к компоненту или отключаться от него во время работы системы.

• Имеется несколько независимых функций, которые нужно применять динамически и в любой комбинации

Диаграмма классов:

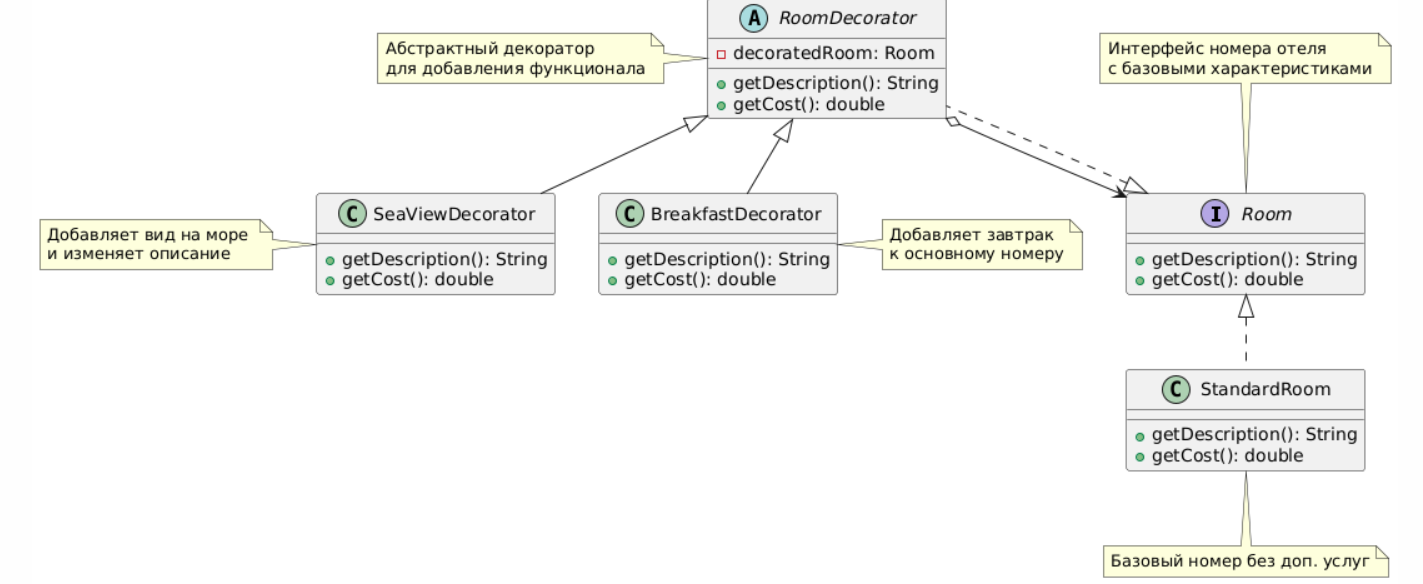
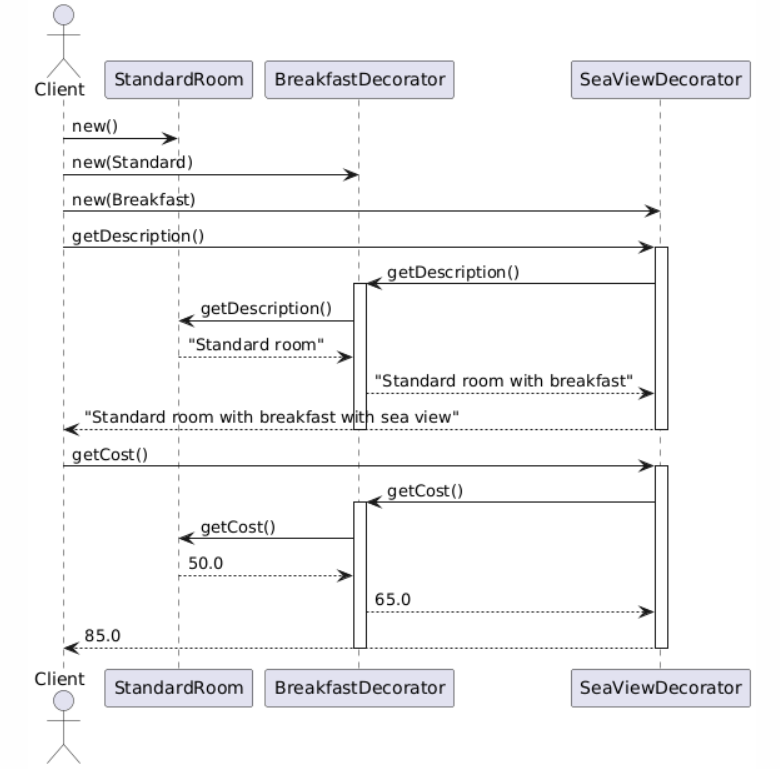


Диаграмма последовательности выполнения:



Код шаблона:

// Component interface

interface Room {

String getDescription();

double getCost();

}

// Concrete Component

class StandardRoom implements Room {

public String getDescription() {

return "Standard room";

}

public double getCost() {

return 50.0;

}

}

// Base Decorator

abstract class RoomDecorator implements Room {

protected Room decoratedRoom;

public RoomDecorator(Room room) {

this.decoratedRoom = room;

}

public String getDescription() {

return decoratedRoom.getDescription();

}

public double getCost() {

return decoratedRoom.getCost();

}

}

// Concrete Decorators

class BreakfastDecorator extends RoomDecorator {

public BreakfastDecorator(Room room) {

super(room);

}

@Override

public String getDescription() {

return decoratedRoom.getDescription() + " with breakfast";

}

@Override

public double getCost() {

return decoratedRoom.getCost() + 15.0;

}

}

class SeaViewDecorator extends RoomDecorator {

public SeaViewDecorator(Room room) {

super(room);

}

@Override

public String getDescription() {

return decoratedRoom.getDescription() + " with sea view";

}

@Override

public double getCost() {

return decoratedRoom.getCost() + 20.0;

}

}

// Client code

public class DecoratorExample {

public static void main(String[] args) {

Room room = new StandardRoom();

System.out.println(room.getDescription() + " for $" + room.getCost());

room = new BreakfastDecorator(room);

System.out.println(room.getDescription() + " for $" + room.getCost());

room = new SeaViewDecorator(room);

System.out.println(room.getDescription() + " for $" + room.getCost());

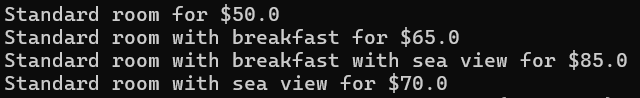
Room premiumRoom = new SeaViewDecorator(new StandardRoom());

System.out.println(premiumRoom.getDescription() + " for $" + premiumRoom.getCost());

}

}

Пример работы программы:



1. Поведенческий – стратегия

Назначение: Предназначен для определения группы классов, которые представляют собой набор возможных вариантов поведения. Позволяет переключаться между алгоритмами во время выполнения программы. Отделяет алгоритм от класса, в котором он используется.

Область применения:

• Когда есть несколько родственных классов, которые отличаются поведением. Можно задать один основной класс, а разные варианты поведения вынести в отдельные классы и при необходимости их применять.

• Когда необходимо обеспечить выбор из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий.

• Когда необходимо менять поведение объектов на стадии выполнения программы

• Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации

Диаграмма классов:

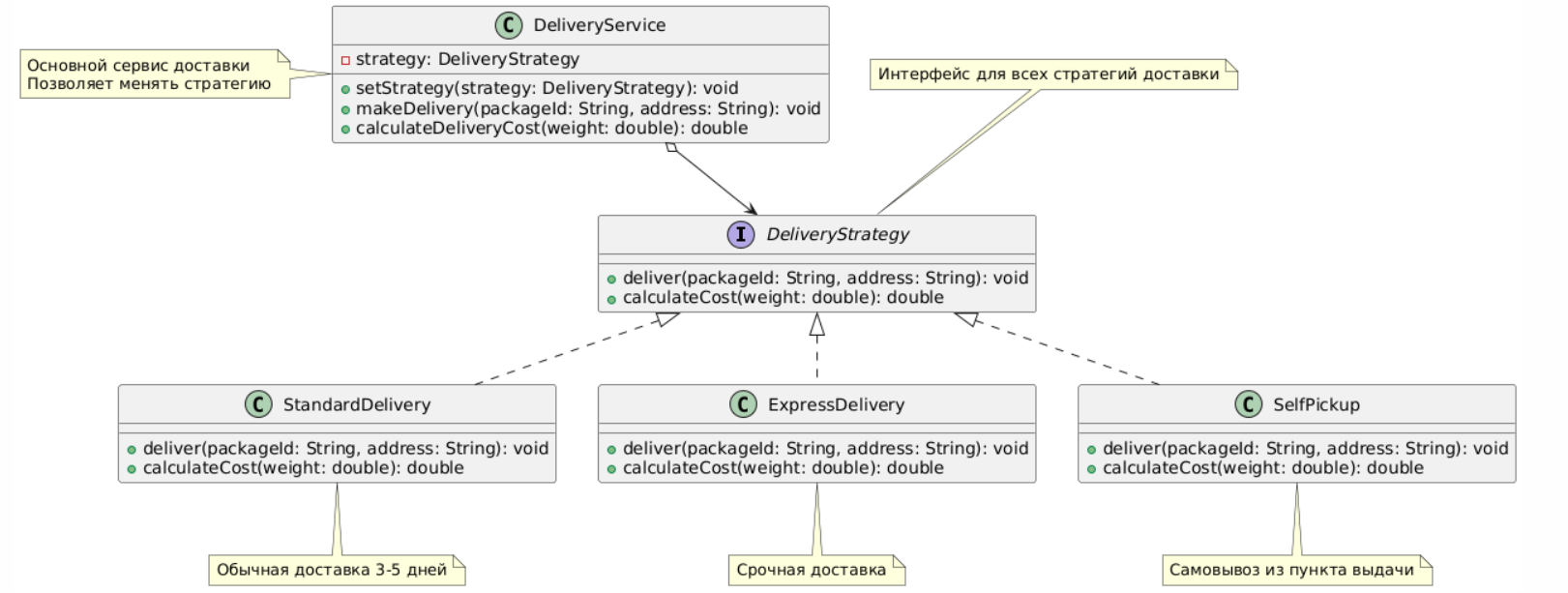
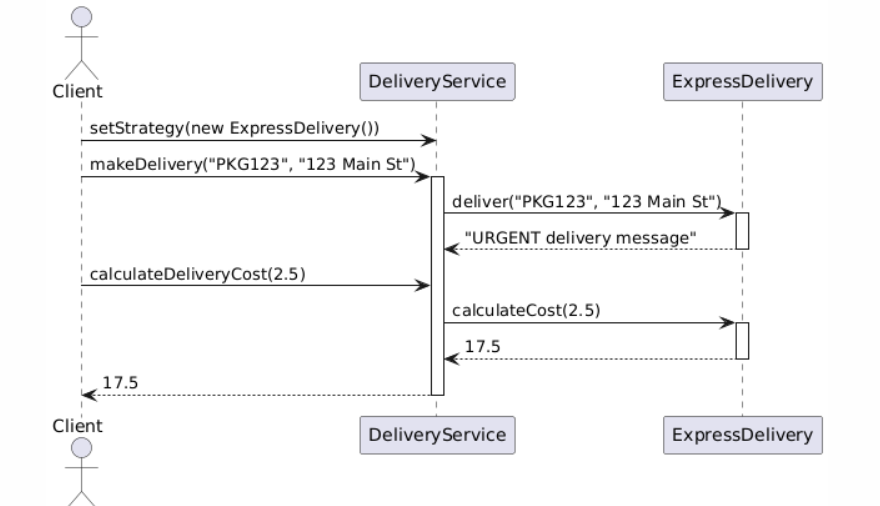


Диаграмма последовательности выполнения:



Код шаблона:

// Strategy interface

interface DeliveryStrategy {

void deliver(String packageId, String address);

double calculateCost(double weight);

}

// Concrete Strategies

class StandardDelivery implements DeliveryStrategy {

public void deliver(String packageId, String address) {

System.out.println("Delivering package " + packageId +

" to " + address + " (3-5 days)");

}

public double calculateCost(double weight) {

return weight \* 1.5;

}

}

class ExpressDelivery implements DeliveryStrategy {

public void deliver(String packageId, String address) {

System.out.println("URGENT delivery of package " + packageId +

" to " + address + " (1-2 days)");

}

public double calculateCost(double weight) {

return weight \* 3.0 + 10.0;

}

}

class SelfPickup implements DeliveryStrategy {

public void deliver(String packageId, String address) {

System.out.println("Package " + packageId +

" ready for pickup at " + address);

}

public double calculateCost(double weight) {

return 0.0;

}

}

// Context

class DeliveryService {

private DeliveryStrategy strategy;

public void setStrategy(DeliveryStrategy strategy) {

this.strategy = strategy;

}

public void makeDelivery(String packageId, String address) {

strategy.deliver(packageId, address);

}

public double calculateDeliveryCost(double weight) {

return strategy.calculateCost(weight);

}

}

// Client code

public class StrategyExample {

public static void main(String[] args) {

DeliveryService service = new DeliveryService();

// Standard delivery

service.setStrategy(new StandardDelivery());

service.makeDelivery("PKG123", "123 Main St");

System.out.println("Cost: $" + service.calculateDeliveryCost(2.5));

// Express delivery

service.setStrategy(new ExpressDelivery());

service.makeDelivery("PKG456", "456 Oak Ave");

System.out.println("Cost: $" + service.calculateDeliveryCost(2.5));

// Self pickup

service.setStrategy(new SelfPickup());

service.makeDelivery("PKG789", "Warehouse at 789 Industrial Rd");

System.out.println("Cost: $" + service.calculateDeliveryCost(2.5));

}

}

Пример работы программы:

