Министерство образования и науки Российской Федерации Федеральное государственное автономное образовательное учреждение высшего профессионального образования «Севастопольский государственный университет»

ОТЧЕТ

о выполнении лабораторной работы № 1 по дисциплине

«Рефакторинг ПО»

Выполнил:

ст. гр. ИС/б-22-1-о

Гюнтер М. Ю .

Проверил:

ст. преподаватель кафедры

“Информационные системы”

Петраков В. А.

Севастополь,2025

# Цель работы: исследовать эффективность составления методов при рефакторинга программного кода. Получить практические навыки применения приемов рефакторинга методов.

# Ход работы

## Исходный программный код, покрытый модульными тестами

Напишем на языке Java код, содержащий фрагменты, предполагающие рефакторинг с помощью 7 приёмов составления методов (листинги ‎2.1-‎2.6), а также покрытый модульными тестами, которые гарантируют работоспособность кода после проведения рефакторинга (листинг ‎2.7).

Листинг ‎2.1 – Класс Order

package refactoring.model;

public class Order {

private final String id;

private final int quantity;

private final double unitPrice;

private final double weightKg;

public Order(String id, int quantity, double unitPrice, double weightKg) {

this.id = id;

this.quantity = quantity;

this.unitPrice = unitPrice;

this.weightKg = weightKg;

}

public String getId() { return id; }

public int getQuantity() { return quantity; }

public double getUnitPrice() { return unitPrice; }

public double getWeightKg() { return weightKg; }

@Override

public String toString() {

return "Order{" + id + ", q=" + quantity + ", price=" + unitPrice + "}";

}

}

Листинг ‎2.2 – Класс OrderService, к которому применимы методы рефакторинга Extract Method и Split Temp Variable

package refactoring.service;

import refactoring.helper.InlineHelper;

import refactoring.model.Order;

import refactoring.util.DiscountUtil;

import refactoring.util.ShippingUtil;

import refactoring.util.TaxUtil;

import java.util.List;

public class OrderService {

private final DiscountUtil discountUtil = new DiscountUtil();

private final ShippingUtil shippingUtil = new ShippingUtil();

private final TaxUtil taxUtil = new TaxUtil();

private final InlineHelper inlineHelper = new InlineHelper();

// REFACTORING CANDIDATE: Метод является слишком длинным; В коде

// метода можно выделить отдельные логические фрагменты, и вынести их в

// отдельные методы с помощью Extract Method

public double calculateInvoiceTotal(List<Order> orders) {

String banner = inlineHelper.banner();

double subtotal = 0.0;

double totalDiscount = 0.0;

double totalShipping = 0.0;

for (Order o : orders) {

// REFACTORING CANDIDATE: Split Temporary Variable (Временная переменная

// 'linePrice' используется для

// хранения разных по назначению результатов выражений)

double linePrice = o.getQuantity() \* o.getUnitPrice();

subtotal += linePrice;

double discount = discountUtil.computeDiscount(o);

totalDiscount += discount;

// REFACTORING CANDIDATE: Split Temporary Variable (Временная переменная

// 'linePrice' используется для

// хранения разных по назначению результатов выражений)

linePrice = shippingUtil.shippingForOrder(o);

totalShipping += linePrice;

}

double taxable = subtotal - totalDiscount;

double tax = taxUtil.computeTax(taxable, 0.1);

double total = subtotal - totalDiscount + totalShipping + tax;

if (banner == null) {

System.out.println("no banner");

}

return total;

}

}

Листинг ‎2.3 – Класс ShippingUtil, к которому применим метод рефакторинга Extract Method

package refactoring.util;

import refactoring.model.Order;

public class ShippingUtil {

public double shippingForOrder(Order order) {

// REFACTORING CANDIDATE: Extract Method (Код для вычисления стоимости заказа

// можно вынести в отдельный метод calcOrderPrice)

double price = order.getQuantity() \* order.getUnitPrice();

double base = 5.0;

double perKg = 2.0;

// REFACTORING CANDIDATE: Extract Method (Код для проверки полученной стоимости

// для определения возможности бесплатной доставки можно вынести в отдельный

// метод isFreeShippingEligable)

if (price > 200) {

return 0.0;

}

return base + perKg \* order.getWeightKg();

}

}

Листинг ‎2.4 – Класс DiscountUtil, к которому применимы методы рефакторинга Replace Temp with Query и Extract Variable

package refactoring.util;

import refactoring.model.Order;

public class DiscountUtil {

public double computeDiscount(Order order) {

// REFACTORING CANDIDATE: Replace Temp with Query (Временную переменную

// 'price' можно заменить вызовом метода calcOrderPrice)

double price = order.getQuantity() \* order.getUnitPrice();

// REFACTORING CANDIDATE: Extract Variable (Условие price > 100

// удобно заменить на переменную isPriceBig)

if (price > 100) {

return price \* 0.10;

// REFACTORING CANDIDATE: Extract variable (Условие price > 50 && price <= 100

// удобно заменить на переменную isPriceMedium)

} else if (price > 50 && price <= 100) {

return price \* 0.05;

} else {

return 0.0;

}

}

}

Листинг ‎2.5 – Класс TaxUtil, к которому применимы методы рефакторинга Remove Assignments to Parameters и Inline Temp

package refactoring.util;

public class TaxUtil {

// REFACTORING CANDIDATE: Remove Assignments to Parameters (Происходит

// присвоение параметру amount нового значения)

public double computeTax(double amount, double rate) {

// REFACTORING CANDIDATE: Inline Temp (Переменная constant не несёт смысловой

// нагрузки, поэтому можно встроить её значение в связанное выражение)

double constant = 1.0;

amount = amount + constant;

if (rate < 0) {

rate = 0;

}

return amount \* rate;

}

}

Листинг ‎2.6 – Класс InlineHelper, к которому применим метод рефакторинга Inline Method

package refactoring.helper;

public class InlineHelper {

// REFACTORING CANDIDATE: Inline Method — trivial delegating method

public String banner() {

return generateBanner();

}

private String generateBanner() {

return "=== ORDER INVOICE ===";

}

}

Листинг ‎2.7 – Класс RefactoringTests, содержащий модульные тесты, проверяющие работоспособность каждого из разработанных методов

package refactoring;

import org.junit.jupiter.api.Test;

import refactoring.model.Order;

import refactoring.service.OrderService;

import refactoring.util.TaxUtil;

import refactoring.util.DiscountUtil;

import refactoring.util.ShippingUtil;

import refactoring.helper.InlineHelper;

import java.util.List;

import static org.junit.jupiter.api.Assertions.\*;

public class RefactoringTests {

@Test

void testDiscountUtil() {

DiscountUtil du = new DiscountUtil();

Order small = new Order("A", 1, 10.0, 1.0); // price 10 -> discount 0

Order medium = new Order("B", 5, 12.0, 2.0); // price 60 -> 5% -> 3.0

Order big = new Order("C", 10, 20.0, 5.0); // price 200 -> 10% -> 20.0

assertEquals(0.0, du.computeDiscount(small), 1e-9);

assertEquals(3.0, du.computeDiscount(medium), 1e-9);

assertEquals(20.0, du.computeDiscount(big), 1e-9);

}

@Test

void testShippingUtil() {

ShippingUtil su = new ShippingUtil();

Order o1 = new Order("S1", 1, 10.0, 3.0); // price 10 -> shipping = 5 + 2\*3 = 11

Order o2 = new Order("S2", 20, 11.0, 5.0); // price 220 -> free shipping 0

assertEquals(11.0, su.shippingForOrder(o1), 1e-9);

assertEquals(0.0, su.shippingForOrder(o2), 1e-9);

}

@Test

void testTaxUtil\_assignsParameterBehavior() {

TaxUtil tu = new TaxUtil();

// computeTax adds 1.0 to amount; this is part of current behavior and must be preserved by tests

double result = tu.computeTax(99.0, 0.1); // (99+1)\*0.1 = 10.0

assertEquals(10.0, result, 1e-9);

// negative rate becomes 0

assertEquals(0.0, tu.computeTax(10.0, -0.5), 1e-9);

}

@Test

void testOrderService\_calculateInvoiceTotal() {

OrderService service = new OrderService();

Order o1 = new Order("O1", 2, 30.0, 2.0); // price 60 -> discount 3 (5%)

Order o2 = new Order("O2", 5, 50.0, 10.0); // price 250 -> discount 25 (10%), free shipping

List<Order> orders = List.of(o1, o2);

// manual compute:

double subtotal = 60 + 250; // 310

double discounts = 3 + 25; // 28

// shipping: o1 -> 5 + 2\*2 =9 ; o2 -> free 0 => 9

double shipping = 9.0;

// taxable = subtotal - discounts = 282

// tax util adds +1 to taxable: taxable+1 = 283; tax rate 0.1 => 28.3

double tax = 283 \* 0.1;

double expected = subtotal - discounts + shipping + tax; // 310 - 28 + 9 + 28.3 = 319.3

assertEquals(expected, service.calculateInvoiceTotal(orders), 1e-9);

}

@Test

void testInlineHelper\_banner() {

InlineHelper ih = new InlineHelper();

assertEquals("=== ORDER INVOICE ===", ih.banner());

}

}

Результат тестирования разработанных методов приведен на рисунке ‎2.1.

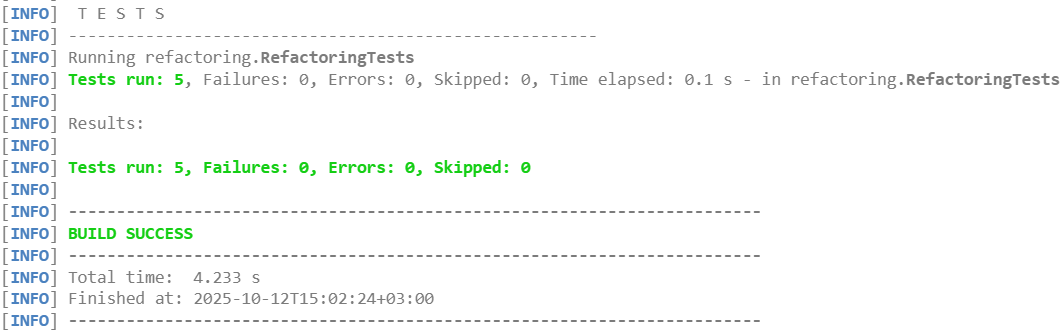


Рисунок ‎2.1 – Результат тестирования разработанных методов

## Рефакторинг исходного программного кода с использованием подхода Составления методов

Был проведён рефакторинг метода calcInvoiceTotal() класса OrderService (листинг ‎2.8).

Листинг ‎2.8 – Класс OrderService, к которому были применены методы рефакторинга Extract Method и Split Temp Variable

package refactoring.service;

import refactoring.model.Order;

import refactoring.util.DiscountUtil;

import refactoring.util.ShippingUtil;

import refactoring.util.TaxUtil;

import java.util.List;

public class OrderService {

private final DiscountUtil discountUtil = new DiscountUtil();

private final ShippingUtil shippingUtil = new ShippingUtil();

private final TaxUtil taxUtil = new TaxUtil();

private double calcOrderPrice(Order o) {

return o.getQuantity() \* o.getUnitPrice();

}

private double calcInvoiceOrderPrice(List<Order> orders) {

double result = 0.0;

for (Order o : orders) {

double orderPrice = this.calcOrderPrice(o);

result += orderPrice;

}

return result;

}

private double calcInvoiceDiscount(List<Order> orders) {

double result = 0.0;

for (Order o : orders) {

double discount = discountUtil.calcDiscount(o);

result += discount;

}

return result;

}

private double calcInvoiceShippingPrice(List<Order> orders) {

double result = 0.0;

for (Order o : orders) {

double shippingPrice = shippingUtil.calcShippingPrice(o);

result += shippingPrice;

}

return result;

}

private double calcInvoiceTax(double invoiceOrderPrice, double invoiceDiscount) {

final double taxable = invoiceOrderPrice - invoiceDiscount;

double tax = taxUtil.calcTax(taxable, 0.1);

return tax;

}

public double calcInvoiceTotal(List<Order> orders) {

double invoiceOrderPrice = this.calcInvoiceOrderPrice(orders);

double invoiceDiscount = this.calcInvoiceDiscount(orders);

double invoiceShippingPrice = this.calcInvoiceShippingPrice(orders);

double tax = this.calcInvoiceTax(invoiceOrderPrice, invoiceDiscount);

double result = invoiceOrderPrice - invoiceDiscount + invoiceShippingPrice + tax;

return result;

}

}

В ходе проведённого рефакторинга был применён подход Split Temporary Variable, в соответствии с которым переменная linePrice была разделена на переменные orderPrice и shippingPrice для отдельного хранения цены заказа и цены доставки. Кроме того, был также применен подход рефакторинга Extract Method, в ходе которого из метода calcInvoiceTotal() были выделены методы calcPrice(), calcInvoiceOrderPrice(), calcInvoiceDiscount(), calcInvoiceShippingPrice(), calcInvoiceTax(), calcInvoiceTotal().

Был проведён рефакторинг метода calcShippingPrice() класса ShippingUtil (листинг ‎2.9).

Листинг ‎2.9 – Класс ShippingUtil, к которому был применен метод рефакторинга Extract Method

package refactoring.util;

import refactoring.model.Order;

public class ShippingUtil {

private double calcOrderPrice(Order o) {

return o.getQuantity() \* o.getUnitPrice();

}

private boolean isFreeShippingEligible(double price) {

return price > 200;

}

public double calcShippingPrice(Order order) {

double price = this.calcOrderPrice(order);

double base = 5.0;

double perKg = 2.0;

if (isFreeShippingEligible(price)) {

return 0.0;

}

return base + perKg \* order.getWeightKg();

}

}

В ходе проведённого рефакторинга был применён подход Extract Method, в соответствии с которым были выделены методы calcOrderPrice() и isFreeShippingEligible().

Был проведён рефакторинг метода calcDiscount() класса DiscountUtil (листинг ‎2.10).

Листинг ‎2.10 – Класс DiscountUtil, к которому были применены методы рефакторинга Replace Temp with Query и Extract Variable

package refactoring.util;

import refactoring.model.Order;

public class DiscountUtil {

private double orderPrice(Order o) {

return o.getQuantity() \* o.getUnitPrice();

}

public double calcDiscount(Order order) {

final boolean isPriceBig = orderPrice(order) > 100;

final boolean isPriceMedium = orderPrice(order) > 50 && orderPrice(order) <= 100;

if (isPriceBig) {

return orderPrice(order) \* 0.10;

} else if (isPriceMedium) {

return orderPrice(order) \* 0.05;

} else {

return 0.0;

}

}

}

В ходе проведённого рефакторинга был применён подход Replace Temp with Query, в соответствии с которым переменная price была заменена вызовом метода orderPrice(). Кроме того, был также применен подход рефакторинга Extract Variable, в ходе которого условия price > 100 и price > 50 && price <= 100 были заменены поясняющими переменными isPriceBig и isPriceMedium.

Был проведён рефакторинг метода calcTax() класса TaxUtil (листинг ‎2.11).

Листинг ‎2.11 – Класс TaxUtil, к которому были применены методы рефакторинга Remove Assignments to Parameters и Inline Temp

package refactoring.util;

public class TaxUtil {

public double calcTax(double amount, double rate) {

double taxBase = amount + 1.0;

if (rate < 0) {

rate = 0;

}

return taxBase \* rate;

}

}

В ходе проведённого рефакторинга был применён подход Remove Assignments to Parameters, в соответствии с которым была создана локальная переменная taxbase, которая хранит значение переданного параметра. Кроме того, был также применен подход рефакторинга Inline Temp, в ходе которого очевидная переменная constant была встроена в выражение, присваиваемое taxBase.

Был проведён рефакторинг метода banner() класса InlineHelper (листинг ‎2.12).

Листинг ‎2.12 – Класс InlineHelper, к которому был применен метод рефакторинга Inline Method

package refactoring.helper;

public class InlineHelper {

public String banner() {

return "=== ORDER INVOICE ===";

}

}

В ходе проведённого рефакторинга был применён подход Inline Method, в соответствии с которым метод generateBanner() был встроен возвращаемое выражние метода banner().

Результат тестирования методов после проведения рефакторинга приведен на рисунке ‎2.2.

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок ‎2.2 – Результат тестирования методов после рефакторинга

По результатам тестирования можно судить об успешном рефакторинге программного кода программы без нарушения её ожидаемого поведения.

# Вывод:

В ходе выполнения лабораторной работы были изучены и применены основные приёмы рефакторинга, связанные с составлением и структурированием методов: «Выделение метода» (Extract Method), «Встраивание метода» (Inline Method), «Замена временной переменной вызовом метода» (Replace Temp with Query), «Введение поясняющей переменной» (Introduce Explaining Variable) и другие.

В результате проведения рефакторинга исходный программный код был преобразован таким образом, что его структура стала более логичной и читаемой. Длинные и трудные для восприятия методы были разбиты на несколько коротких, имеющих осмысленные имена и выполняющих чётко определённые функции. Это позволило повысить понятность, сопровождаемость и переиспользуемость кода без изменения его функционального поведения.