МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Кафедра «Програмна інженерія»

ЗВІТ

до практичного завдання №2

з дисципліни «Аналіз та рефакторинг коду»

на тему «Методи рефакторингу коду програмного забезпечення»

Виконала:
ст. гр. ПЗПІ-23-7
Никифорова Анастасія

Перевірив:
ст. викладач катедри ПІ
Сокорчук Ігор Петрович

Харків 2025

ІСТОРІЯ ЗМІН

№	Дата звернення	Версія звіту	Опис змін та виправлень
1	18.01.26	1.1	Створено розділ «Мета роботи» та зібрано основні рекомендації Google Style Guide.
2	18.01.26	1.1	Додано розділ «Хід роботи» з прикладами правил і форматування.
3	18.01.26	1.1	Описано принципи рефакторингу та узагальнено результати.
4	18.01.26	1.1	Додано розділи «Висновки» та «Використані джерела». Звіт підготовлено до подання.

Мета роботи

Метою даної практичної роботи є ознайомлення з основними методами рефакторингу коду програмного забезпечення та набуття практичних навичок їх застосування на прикладах власного програмного коду, написаного мовою програмування Kotlin.

У ході виконання роботи було обрано та реалізовано такі методи рефакторингу, як Extract Method, Rename Method / Variable та Simplify Conditional Expression, з метою покращення читабельності, структурованості та підтримуваності коду без зміни його функціональної поведінки.

Хід роботи

Вибір та обґрунтування теми

Тема рефакторингу коду була обрана у зв’язку з її важливістю для сучасної розробки програмного забезпечення. У процесі розробки програмних систем програмний код з часом ускладнюється, що негативно впливає на його читабельність, підтримуваність та можливість подальшого розширення. Рефакторинг дозволяє покращити внутрішню структуру коду без зміни його зовнішньої поведінки, що є важливим етапом життєвого циклу програмного забезпечення.

Для виконання практичної роботи були обрані методи Extract Method, Rename Method / Variable та Simplify Conditional Expression, оскільки вони є базовими та найбільш поширеними методами рефакторингу. Застосування методу Extract Method дозволяє зменшити складність великих функцій шляхом виділення окремих логічних блоків у самостійні методи. Метод Rename Method / Variable спрямований на підвищення зрозумілості коду завдяки використанню змістовних імен змінних та методів. Метод Simplify Conditional Expression дає змогу спростити умовні конструкції, зменшити вкладеність та покращити логічну структуру програмного коду.

Обрані методи добре підходять для демонстрації можливостей рефакторингу на прикладах коду, написаного мовою програмування Kotlin, та дозволяють наочно показати покращення якості програмного забезпечення без зміни його функціональності.

Метод 1 — Extract Method

У процесі аналізу програмного коду було виявлено метод, який містив велику кількість логіки та виконував декілька різних дій одночасно. Такий підхід ускладнює читабельність коду, робить його важким для розуміння та подальшого супроводу. Наявність довгих методів також ускладнює тестування окремих частин функціональності та підвищує ймовірність виникнення помилок під час внесення змін.

```
data class Product(val name: String, val price: Double, val quantity: Int)

fun generateOrderReport(products: List<Product>, promoCode: String?): String {
    var total = 0.0
    for (product in products) {
        total += product.price * product.quantity
    }

    var discount = 0.0
    if (promoCode != null) {
        if (promoCode == "SALE10" && total > 100) {
            discount = total * 0.10
        } else if (promoCode == "STUDENT") {
            discount = total * 0.15
        }
    }

    val finalPrice = total - discount

    val report = StringBuilder()
    report.appendLine("ORDER REPORT")
    for (product in products) {
        report.appendLine(
            "${product.name} x${product.quantity} = ${product.price * product.quantity}"
        )
    }
    report.appendLine("Total: $total")
    report.appendLine("Discount: $discount")
    report.appendLine("Final price: $finalPrice")

    return report.toString()
```

Рисунок 1.1 – Код до рефакторингу

```
fun generateOrderReport(products: List<Product>, promoCode: String?): String
    val total = calculateTotal(products)
    val discount = calculateDiscount(promoCode, total)
    val finalPrice = total - discount
    return buildReport(products, total, discount, finalPrice)
}

private fun calculateTotal(products: List<Product>): Double {
    return products.sumOf { it.price * it.quantity }
}

private fun calculateDiscount(promoCode: String?, total: Double): Double {
    if (promoCode == null) return 0.0
    return when (promoCode) {
        "SALE10" -> if (total > 100) total * 0.10 else 0.0
        "STUDENT" -> total * 0.15
        else -> 0.0
    }
}
```

```
private fun buildReport(
    products: List<Product>,
    total: Double,
    discount: Double,
    finalPrice: Double
): String {
    return buildString {
        appendLine("ORDER REPORT")
        products.forEach {
            appendLine("${it.name} x${it.quantity} = ${it.price * it.quantity}")
        }
        appendLine("Total: $total")
        appendLine("Discount: $discount")
        appendLine("Final price: $finalPrice")
    }
}
```

Рисунок 1.2 – Код після рефакторингу

До рефакторингу:

У початковому варіанті коду реалізовано метод generateOrderReport, який виконує формування звіту про замовлення. Даний метод одночасно відповідає за декілька різних задач: обчислення загальної вартості товарів, визначення розміру знижки залежно від промокоду, а також формування текстового звіту. У межах одного методу поєднано логіку обчислень, умовну логіку та роботу з рядками.

Такий підхід призводить до збільшення розміру методу та ускладнює його сприйняття. Змішування різних типів логіки в одному методі негативно впливає на читабельність коду та ускладнює його подальший супровід і тестування.

Після рефакторингу:

Після застосування методу рефакторингу Extract Method основний метод generateOrderReport було спрощено шляхом винесення окремих логічних частин у допоміжні методи. Обчислення загальної вартості товарів було перенесено до методу calculateTotal, логіку визначення знижки — до методу calculateDiscount, а формування текстового звіту — до методу buildReport.

У результаті основний метод виконує лише координацію викликів допоміжних методів, що значно покращує зрозумілість коду. Кожен допоміжний метод відповідає за одну конкретну задачу, що відповідає принципу єдиної відповідальності. Така структура полегшує читання коду, його тестування та подальше розширення без ризику зміни існуючої функціональності.

Метод 2 — Rename Method / Variable

Під час аналізу програмного коду було виявлено використання неінформативних назв методів та змінних, які не відображали їхнього реального призначення. Такі назви ускладнюють розуміння логіки програми, збільшують час на аналіз коду та можуть призводити до помилок під час його використання або модифікації.

```
fun calc(a: Double, b: Int, c: Boolean): Double {
    var x = a * b
    if (c) {
        x = x * 0.9
    }
    return x
}
```

Рисунок 1.3 – Код до рефакторингу

```
fun calculateFinalPrice(unitPrice: Double, quantity: Int, hasDiscount: Boolean): Double {
    var totalPrice = unitPrice * quantity
    if (hasDiscount) {
        totalPrice *= 0.9
    }
    return totalPrice
}
```

Рисунок 1.4 – Код після рефакторингу

До рефакторингу:
Метод містив параметри та змінні з короткими та незрозумілими назвами, що не давало чіткого уявлення про їх призначення. Для розуміння логіки обчислень необхідно було детально аналізувати тіло методу.

Після рефакторингу:
Назви методу та змінних були змінені на зрозумілі та такі, що відображають їхнє функціональне призначення. Завдяки цьому код став легшим для читання та сприйняття, а його логіка стала зрозумілою без додаткових пояснень.

Застосування методу Rename Method / Variable дозволило підвищити зрозумілість програмного коду та зменшити потребу у додаткових коментарях. Покращилась читабельність коду, що спрощує його підтримку та подальший розвиток.

Метод 3 — Simplify Conditional Expression

Під час аналізу програмного коду було виявлено складну умовну конструкцію з декількома вкладеними операторами if-else. Така структура ускладнює читання коду, збільшує його складність та робить логіку програми менш наочною. У разі розширення або модифікації умов зростає ризик допущення помилок.

```
fun getShippingCost(price: Double, isMember: Boolean, isInternational: Boolean): Double {
    var cost = 0.0

    if (isInternational) {
        if (isMember) {
            if (price > 100) {
                cost = 0.0
            } else {
                cost = 10.0
            }
        } else {
            cost = 25.0
        }
    } else {
        if (isMember) {
            cost = 0.0
        } else {
            cost = 5.0
        }
    }

    return cost
}
```

Рисунок 1.5 – Код до рефакторингу

У межах застосування методу Simplify Conditional Expression було проаналізовано логіку умовних операторів та усунено надмірну вкладеність. Складну структуру з декількома вкладеними умовами було замінено на більш просту та зрозумілу умовну конструкцію з використанням логічних виразів.

```
fun getShippingCost(price: Double, isMember: Boolean, isInternational: Boolean): Double {
    return when {
        isInternational && isMember && price > 100 -> 0.0
        isInternational && isMember -> 10.0
        isInternational && !isMember -> 25.0
        !isInternational && isMember -> 0.0
        else -> 5.0
    }
}
```

Рисунок 1.6 – Код після рефакторингу

До рефакторингу:
Логіка визначення вартості доставки була реалізована за допомогою великої

кількості вкладених умовних операторів, що ускладнювало сприйняття та аналіз коду.

Після рефакторингу:
Умовну логіку було спрощено шляхом використання конструкції when, яка дозволяє наочно відобразити всі можливі варіанти умов у компактному та зрозумілому вигляді.

Застосування методу Simplify Conditional Expression дозволило зменшити складність умовної логіки, покращити читабельність коду та полегшити його подальший супровід. Спрощена структура умов знижує ймовірність помилок під час внесення змін.

Висновки

У ході виконання практичної роботи було розглянуто та застосовано основні методи рефакторингу програмного коду відповідно до підходів, запропонованих М. Фаулером. На прикладах коду, написаного мовою програмування Kotlin, було продемонстровано можливість покращення внутрішньої структури програмного забезпечення без зміни його функціональної поведінки.

У процесі роботи було застосовано методи Extract Method, Rename Method / Variable та Simplify Conditional Expression, які дозволили зменшити складність коду, підвищити його читабельність та зробити логіку програми більш зрозумілою. Виконаний рефакторинг сприяв покращенню підтримуваності коду, полегшенню його тестування та подальшого розвитку.

Отримані результати підтверджують доцільність використання методів рефакторингу під час розробки програмного забезпечення, оскільки вони сприяють підвищенню якості коду та зменшенню ризику виникнення помилок у процесі внесення змін.

Використані джерела

1. Fowler M. *Refactoring: Improving the Design of Existing Code*. — Addison-Wesley, 2018.
2. Fowler M. Refactoring Catalog. — Офіційний сайт.
3. Oracle. *Java Documentation*. — Офіційна документація.
4. JetBrains. *Kotlin Documentation*. — Офіційна документація мови програмування Kotlin.
5. Clean Code concepts and best practices. — Навчальні матеріали з якісного програмування.

ДОДАТОК А

Посилання на відео:

https://youtu.be/-PG3ASqevVw

ДОДАТОК Б

Слайди презентації:

Рисунок Б.1 – Слайд 1 Титульний

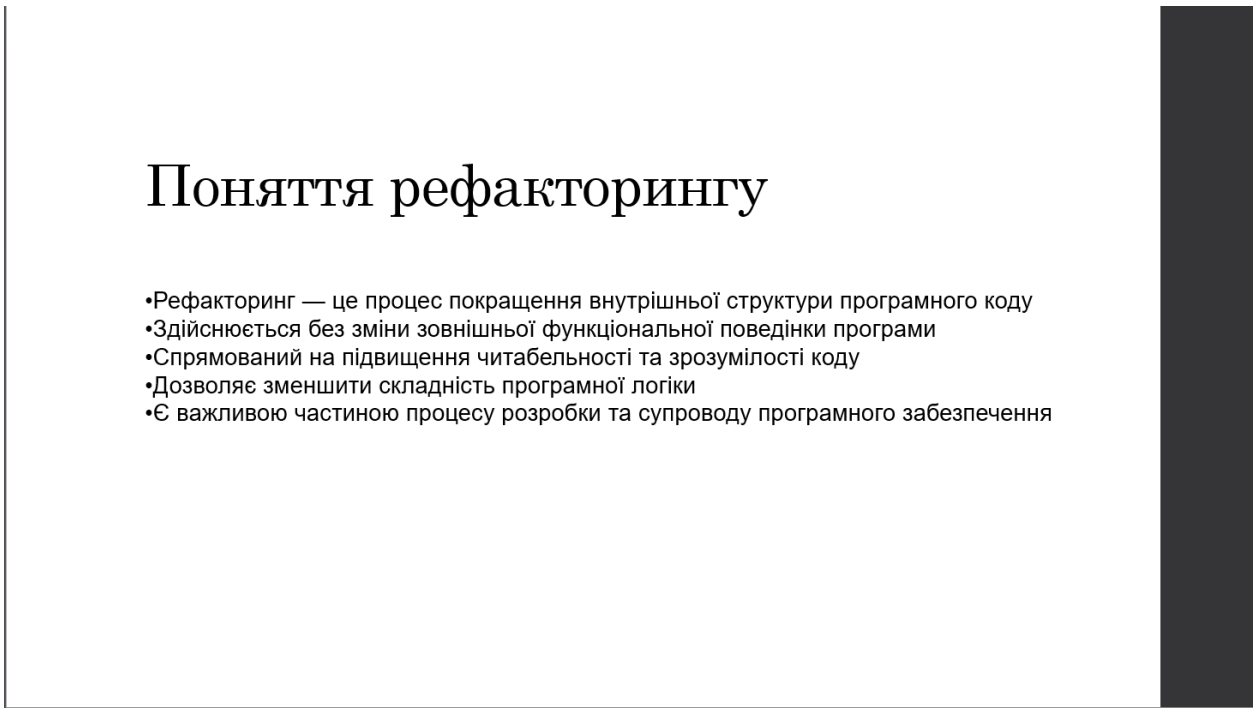

Рисунок Б.2 – Слайд 2 Поняття рефакторингу

Мета рефакторингу

- Покращення читабельності програмного коду
- Спрощення структури та логіки програми
- Полегшення супроводу та тестування коду
- Зменшення кількості помилок при внесенні змін
- Підготовка коду до подальшого розвитку та масштабування

Рисунок Б.3 – Слайд 3 Мета рефакторингу

Обрані методи рефакторингу.

- Extract Method
- Rename Method / Variable
- Simplify Conditional Expression
- Мова програмування: Kotlin

Рисунок Б.4 – Слайд 4 Обрані методи

Метод Extract Method

•Використовується для спрощення великих методів
•Дозволяє виділити логічні частини коду
•Зменшує складність та покращує структуру
•Підвищує читабельність і підтримуваність коду

Рисунок Б.5 – Слайд 5 Метод Extract Method

•Код до

```
data class Product(val name: String, val price: Double, val quantity: Int)

fun generateOrderReport(products: List<Product>, promoCode: String?): String {
    var total = 0.0
    for (product in products) {
        total += product.price * product.quantity
    }

    var discount = 0.0
    if (promoCode != null) {
        if (promoCode == "SALE10" && total > 100) {
            discount = total * 0.10
        } else if (promoCode == "STUDENT") {
            discount = total * 0.15
        }
    }

    val finalPrice = total - discount

    val report = StringBuilder()
    report.appendLine("ORDER REPORT")
    for (product in products) {
        report.appendLine(
            "${product.name} x${product.quantity} = ${product.price * product.quantity}"
        )
    }
    report.appendLine("Total: $total")
    report.appendLine("Discount: $discount")
    report.appendLine("Final price: $finalPrice")

    return report.toString()
}
```

•Код після

```
fun generateOrderReport(products: List<Product>, promoCode: String?): String
    val total = calculateTotal(products)
    val discount = calculateDiscount(promoCode, total)
    val finalPrice = total - discount
    return buildReport(products, total, discount, finalPrice)
}

private fun calculateTotal(products: List<Product>): Double {
    return products.sumOf { it.price * it.quantity }
}

private fun calculateDiscount(promoCode: String?, total: Double): Double {
    if (promoCode == null) return 0.0
    return when (promoCode) {
        "SALE10" -> if (total > 100) total * 0.10 else 0.0
        "STUDENT" -> total * 0.15
        else -> 0.0
    }
}
```

Extract Method — приклад

•Код до: один великий метод
•Код після: декілька допоміжних методів
•Логіка програми не змінена
•Код став більш структурованим

Рисунок Б.6 – Слайд 6 Приклад

Rename Method / Variable — приклад

•Код до

```
fun calc(a: Double, b: Int, c: Boolean): Double {
    var x = a * b
    if (c) {
        x = x * 0.9
    }
    return x
}
```

```
fun calculateFinalPrice(unitPrice: Double, quantity: Int, hasDiscount: Boolean): Double {
    var totalPrice = unitPrice * quantity
    if (hasDiscount) {
        totalPrice *= 0.9
    }
    return totalPrice
}
```

•Код після

- До рефакторингу:
- Метод: calc
- Змінні: a, b, x
- Після рефакторингу:
- Метод: calculateFinalPrice
- Змінні: unitPrice, quantity, totalPrice
- Логіка програми не змінена
- Покращено зрозумілість коду

Рисунок Б.7 – Слайд 7 Метод Rename Method / Variable

Simplify Conditional Expression (приклад)

- До рефакторингу:
- Складні вкладені if / else
- Важко сприймати логіку
- Після рефакторингу:
- Використання конструкції when
- Логіка стала наочною та зрозумілою
- Функціональність не змінена

•Код до

```
fun getShippingCost(price: Double, isMember: Boolean, isInternational: Boolean): Double {
    var cost = 0.0

    if (isInternational) {
        if (isMember) {
            if (price > 100) {
                cost = 0.0
            } else {
                cost = 10.0
            }
        } else {
            cost = 25.0
        }
    } else {
        if (isMember) {
            cost = 0.0
        } else {
            cost = 5.0
        }
    }

    return cost
}
```

```
fun getShippingCost(price: Double, isMember: Boolean, isInternational: Boolean): Double {
    return when {
        isInternational && isMember && price > 100 -> 0.0
        isInternational && isMember -> 10.0
        isInternational && !isMember -> 25.0
        !isInternational && isMember -> 0.0
        else -> 5.0
    }
}
```

•Код після

Рисунок Б.8 – Слайд 8 Метод Simplify Conditional Expression

Переваги застосування рефакторингу

•Покращення читабельності програмного коду
•Зменшення складності та вкладеності логіки
•Полегшення тестування та супроводу
•Зменшення кількості помилок при модифікації коду
•Підвищення якості програмного забезпечення

Рисунок Б.9 – Слайд 9 Переваги застосування рефакторингу

Інструменти для рефакторингу

•IntelliJ IDEA
•Підтримка мови програмування Kotlin
•Вбудовані інструменти рефакторингу
•Автоматичне перейменування та виділення методів

Рисунок Б.10 – Слайд 10 Інструменти для рефакторингу

Висновки

•Рефакторинг покращує якість програмного коду
•Підвищує читабельність та підтримуваність
•Не змінює функціональну поведінку програми
•Полегшує подальший розвиток програмного забезпечення

Рисунок Б.11 – Слайд 11 Висновки