Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Кафедра програмної інженерії

Презентація до практичної роботи №2

На тему:

Refactoring Methods

Виконала:

Студентка групи ПЗПІ 22-2

Моссур Д. Є.

Перевірив:

Доцент кафедри ПІ

Лещинський Володимир Олександрович

Харків 2025

Refactoring Methods

Мета роботи:

Метою даної роботи було ознайомлення з різними методами рефакторингу програмного коду та продемонструвати їх використання на реальних прикладах.

Хід роботи:

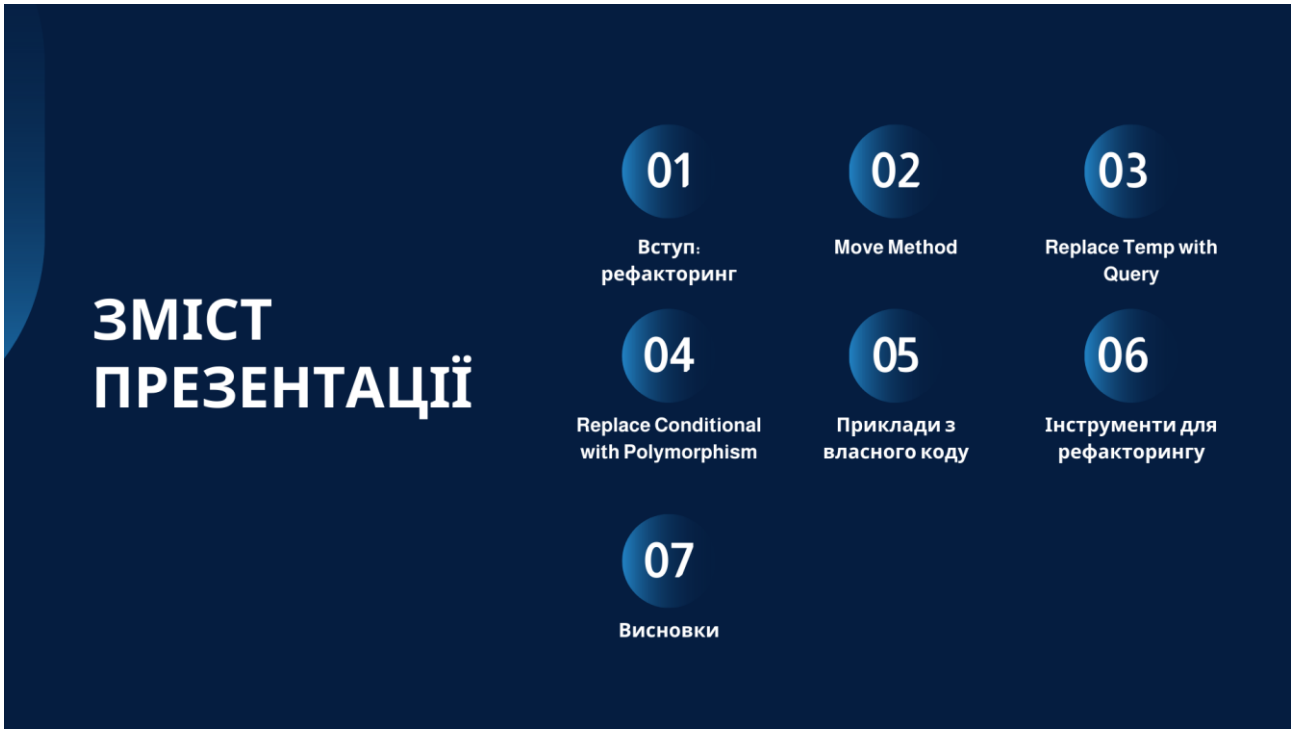
Таким чином, з книги «Чистий Код» Роберта Мартіна було обрано три методи рефакторингу та продемонстровано, як змінюється код після застосування цих методів. В ході виконання даного завдання було розглянуто три методи: Move Method, Replace Temp with Query, Replace Conditional with Polymorphism. Результати ознайомлено викладено в презентацію. Презентація наведена нижче у Додатку А.

Висновок:

Рефакторинг є невід'ємною частиною процесу розробки програмного забезпечення, спрямованою на покращення якості коду без зміни його функціональності. Він забезпечує читабельність, зрозумілість і підтримуваність коду. Дозволяє усувати технічний борг, оптимізувати продуктивність і зменшувати ймовірність помилок.

Додаток А

Презентація



ВСТУП: РЕФАКТОРИНГ

Рефакторинг — це процес покращення існуючого коду без зміни його функціональності.
Мета полягає в тому, щоб зробити код більш зрозумілим, зручним для підтримки та розширення.

Метою рефакторингу є:

- Поліпшення читабельності коду.
- Зменшення дублікатів та складності.
- Підвищення підтримуваності проєкту.
- Полегшення впровадження нових функцій.

ВСТУП: РЕФАКТОРИНГ

Короткий опис методів рефакторингу, які будуть розглянуті

- Move Method (Переміщення методу):

Використовується, коли метод краще розташувати в іншому класі або модулі для покращення зв'язності.

- Replace Temp with Query (Заміна тимчасової змінної на запит):

Дає змогу замінити локальні змінні на методи, що робить код більш декларативним.

- Replace Conditional with Polymorphism (Заміна умовної логіки поліморфізмом):

Дозволяє усунути складну умовну логіку, використовуючи наслідування та поліморфізм.

ОБРАНІ МЕТОДИ РЕФАКТОРИНГУ

1. MOVE METHOD
2. REPLACE TEMP WITH QUERY
3. REPLACE CONDITIONAL WITH POLYMORPHISM

MOVE METHOD

Метод "Move Method" використовується для переміщення методу з одного класу в інший, якщо він краще підходить за контекстом і функціональністю іншому класу. Це покращує зв'язність коду і зменшує залежність між класами.

Проблема, яку вирішує метод:

1. Метод використовує більше даних іншого класу, ніж даних свого власного.
2. Знижується зв'язність класів, що ускладнює підтримку та розширення.
3. Код стає заплутаним через використання методів або полів іншого класу.

MOVE METHOD

Приклад використання методу рефакторингу для покращення коду

Код до рефакторингу:

```python
class Customer:
    def __init__(self, name, subscription):
        self.name = name
        self.subscription = subscription

    def subscription_cost(self):
        if self.subscription.level == "Premium":
            return 50
        elif self.subscription.level == "Standard":
            return 30
        else:
            return 10

class Subscription:
    def __init__(self, level):
        self.level = level
```

Код після рефакторингу:

```python
class Customer:
    def __init__(self, name, subscription):
        self.name = name
        self.subscription = subscription

    def subscription_cost(self):
        return self.subscription.calculate_cost()

class Subscription:
    def __init__(self, level):
        self.level = level

    def calculate_cost(self):
        if self.level == "Premium":
            return 50
        elif self.level == "Standard":
            return 30
        else:
            return 10
```

MOVE METHOD

Проблеми коду до рефакторингу:

- Метод subscription_cost більше залежить від даних класу Subscription, ніж Customer.
- Логіка підрахунку вартості належить саме класу Subscription.

Переваги застосування методу:

- Логіка тепер знаходиться в тому класі, якому вона належить (клас Subscription).
- Для внесення змін у логіку підрахунку потрібно змінювати лише метод calculate_cost.
- Зменшення дублювання - методи чітко виконують свої ролі в рамках своїх класів.
- Тестувати логіку обчислення вартості можна окремо від інших класів.

REPLACE TEMP WITH QUERY

Метод "Replace Temp with Query" передбачає заміну тимчасових змінних (локальних змінних у методах) на методи-запити. Це підвищує читабельність та дозволяє уникнути дублювання логіки.

Проблеми, яку вирішує метод:

1. Тимчасові змінні можуть знижувати читабельність, особливо коли вони використовуються для проміжних значень.
2. Логіка обчислення таких значень дублюється в різних місцях коду.
3. Ускладнення внесення змін: якщо потрібно змінити логіку, її потрібно шукати в декількох місцях.

REPLACE TEMP WITH QUERY

Приклад використання методу рефакторингу для покращення коду

До рефакторингу:

```python
class Order:
    def __init__(self, items):
        self.items = items

    def calculate_total(self):
        total_price = 0
        for item in self.items:
            total_price += item['price'] * item['quantity']
        if total_price > 100:
            discount = total_price * 0.1
        else:
            discount = 0
        return total_price - discount
```

Код після рефакторингу:

```python
class Order:
    def __init__(self, items):
        self.items = items

    def total_price(self):
        return sum(item['price'] * item['quantity'] for item in self.items)

    def discount(self):
        return self.total_price() * 0.1 if self.total_price() > 100 else 0

    def calculate_total(self):
        return self.total_price() - self.discount()
```

REPLACE TEMP WITH QUERY

Приклад використання методу рефакторингу для покращення коду

Проблеми коду до рефакторингу:

- Логіка обчислення total_price і знижки захована в методі calculate_total.
- Важко повторно використовувати ці обчислення в інших місцях.

Покращення після змін:

- Логіка винесена в окремі методи total_price і discount.
- Обчислення стали декларативними та легкими для повторного використання.
- Читабельність коду значно покращилась.

Переваги застосування методу:

1. Код став зрозумілішим, оскільки логіка тепер розташована в окремих методах.
2. Полегшена підтримка: для внесення змін потрібно змінити лише один метод.
3. Зниження ризику помилок, оскільки логіка не дублюється.
4. Підвищена модульність: окремі методи легко тестувати.

REPLACE CONDITIONAL WITH POLYMORPHISM

Метод "Replace Conditional with Polymorphism" передбачає заміну умовних операторів (if, else, switch) на поліморфізм через використання підкласів або реалізацію інтерфейсів. Це дозволяє уникнути складної умовної логіки та зробити код більш модульним і зрозумілим.

Проблема, яку вирішує метод:

1. Умовна логіка з численними гілками (if-else або switch) ускладнює розуміння та підтримку коду.
2. Додавання нової логіки вимагає змін в основному коді, порушуючи принцип відкритості/закритості (OCP) з SOLID.
3. Код стає менш гнучким і повторно використовуваним.

REPLACE CONDITIONAL WITH POLYMORPHISM

Код до рефакторингу:

```python
class Employee:
    def __init__(self, name, role):
        self.name = name
        self.role = role

    def calculate_salary(self):
        if self.role == "Manager":
            return 5000
        elif self.role == "Developer":
            return 4000
        elif self.role == "Intern":
            return 2000
        else:
            return 0
```

Код після рефакторингу:

```python
class Employee:
    def __init__(self, name):
        self.name = name

    def calculate_salary(self):
        raise NotImplementedError("This method should be implemented by subclasses")

class Manager(Employee):
    def calculate_salary(self):
        return 5000

class Developer(Employee):
    def calculate_salary(self):
        return 4000

class Intern(Employee):
    def calculate_salary(self):
        return 2000
```

REPLACE CONDITIONAL WITH POLYMORPHISM

Код для демонстрації:

```python
employees = [
    Manager("Alice"),
    Developer("Bob"),
    Intern("Charlie")
]

for employee in employees:
    print(f"{employee.name}: {employee.calculate_salary()}")
```

Результат відпрацювання коду:

```
ПРОБЛЕМЫ  ВЫХОДНЫЕ ДАННЫЕ  КОНСОЛЬ ОТЛАДКИ  ТЕРМИНАЛ
PS C:\уник\3_КУРС\1_семестр\АтРК\pz2> python pz2.py
Alice: 5000
Bob: 4000
Charlie: 2000
PS C:\уник\3_КУРС\1_семестр\АтРК\pz2>
```

REPLACE CONDITIONAL WITH POLYMORPHISM

ПЕРЕВАГИ ЗАСТОСУВАННЯ ДАНОГО МЕТОДУ:

- Покращення структури коду:

Логіка для кожної ролі інкапсульована в окремому класі, що робить код модульним.

- Легкість розширення:

Додавання нової ролі вимагає створення нового класу, без змін у вже існуючому коді (дотримання принципу відкритості/закритості).

- Читабельність:

Поліморфізм робить код легшим для розуміння, оскільки кожна роль має власну реалізацію.

- Зручність тестування:

Кожен клас можна тестувати окремо, що спрощує пошук і виправлення помилок.

ПРИКЛАД ЗАСТОСУВАННЯ МЕТОДУ REPLACE CONDITIONAL WITH POLYMORPHISM НА ПРАКТИЦІ

За приклад візьмемо програму, що була написана раніше для обчислення статистичних характеристик масиву чисел.

Основні проблеми:

- Методи реалізовані в одному класі, що не відповідає принципу SRP (Single Responsibility Principle).
- Відсутня можливість масштабування — для додавання нових операцій потрібно змінювати основний код.
- Код не модульний, а повторюваність логіки ускладнює підтримку.

```
Ссылок: 2
static double CalculateMean(int[] data)
{
    return data.Average();
}
Ссылок: 1
static double CalculateVariance(int[] data)
{
    double mean = CalculateMean(data);
    return data.Sum(x => Math.Pow(x - mean, 2)) / (data.Length - 1);
}
Ссылок: 1
static int CalculateMedian(int[] data)
{
    int[] sortedData = data.OrderBy(x => x).ToArray();
    int middle = sortedData.Length / 2;
    return (sortedData.Length % 2 == 0) ? (sortedData[middle - 1] + sortedData[middle]) / 2 : sortedData[middle];
}
Ссылок: 1
static int CalculateMode(int[] data)
{
    var groupedData = data.GroupBy(x => x);
    var maxFrequency = groupedData.Max(g => g.Count());
    return groupedData.First(g => g.Count() == maxFrequency).Key;
}
Ссылок: 1
static double[] CalculateQuantiles(int[] data, double[] quantiles)
{
    Array.Sort(data);
    int n = data.Length;

    double[] result = new double[quantiles.Length];
    for (int i = 0; i < quantiles.Length; i++)
    {
        double k = (n - 1) * quantiles[i];
        int index = (int)k;
        double fraction = k - index;
        result[i] = (1 - fraction) * data[index] + fraction * data[index + 1];
    }
    return result;
}
}
```

ПРИКЛАД ЗАСТОСУВАННЯ МЕТОДУ REPLACE CONDITIONAL WITH POLYMORPHISM НА ПРАКТИЦІ

```
// Інтерфейс для обчислення метрик
Ссылок: 5
interface IStatistic
{
    Ссылок: 4
    double Calculate(int[] data);
}

// Клас для середнього значення
Ссылок: 1
class Mean : IStatistic
{
    Ссылок: 2
    public double Calculate(int[] data) => data.Average();
}

// Клас для дисперсії
Ссылок: 1
class Variance : IStatistic
{
    Ссылок: 2
    public double Calculate(int[] data)
    {
        double mean = data.Average();
        return data.Sum(x => Math.Pow(x - mean, 2)) / (data.Length - 1);
    }
}

// Клас для медіани
Ссылок: 1
class Median : IStatistic
{
    Ссылок: 2
    public double Calculate(int[] data)
    {
        int[] sortedData = data.OrderBy(x => x).ToArray();
        int middle = sortedData.Length / 2;
        return (sortedData.Length % 2 == 0)
            ? (sortedData[middle - 1] + sortedData[middle]) / 2.0
            : sortedData[middle];
    }
}
```

Бачимо, що після застосування методу:

- Тепер логіка кожного обчислення винесена в окремий клас.
- Нову метрику можна додати через реалізацію інтерфейсу IStatistic, не змінюючи основний код.
- Кожен клас відповідає за обчислення лише одного показника.
- Код став зрозумілішим і легко підтримується.

ІНСТРУМЕНТИ ДЛЯ РЕФАКТОРИНГУ

- **JETBRAINS IDE (INTELLIJ IDEA, PYCHARM):**
- **РОЗШИРЕННЯ ДЛЯ VS CODE**
- **VISUAL STUDIO** — Вбудовані інструменти для рефакторингу.
- **ECLIPSE** — Потужні засоби рефакторингу для Java та інших мов, а також підтримка популярних плагінів.

ВИСНОВОК

Рефакторинг — це потужний інструмент для розробників, що дозволяє створювати якісний, надійний і зрозумілий код.

ВАЖЛИВІСТЬ

Рефакторинг є важливим, бо забезпечує читабельність, зрозумілість і підтримуваність коду. Також дозволяє усувати технічний борг, оптимізувати продуктивність і зменшувати ймовірність помилок.

ПЕРЕВАГИ РЕФАКТОРИНГУ

- Покращення структури коду.
- Зменшення дублювання.
- Легше розширення та підтримка проєкту.

В ході виконання даного завдання було розглянуто три методи: Move Method, Replace Temp with Query, Replace Conditional with Polymorphism.

ДЯКУЮ ЗА УВАГУ!