МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Кафедра «Програмна інженерія»

ЗВІТ

до практичного заняття №1 з дисципліни

«Аналіз та рефакторинг коду»

На тему: «Правила оформлення програмного коду»

Виконав:
ст. гр. ПЗПІ-22-7
Дудуков Олександр Сергійович

Прийняв:
ст. викладач кафедри ПІ
Сокорчук Ігор Петрович

Харків 2024

1 МЕТА

Навчитися рефакторингу програмного коду, закріпити основні правила оформлення коду.

2 ЗАВДАННЯ

Обрати мову програмування для прикладів коду. Створити презентацію на тему «Правила оформлення програмного коду».

3 ХІД РОБОТИ

Було обрано мову програмування Kotlin. У презентації (Додаток Б) наведено основні рекомендації щодо оформлення програмного коду з описами, а також приклад коду до і після застосування цих рекомендацій.

ВИСНОВКИ

Набуто навичок рефакторингу програмного коду, детально розглянуто основні правила оформлення коду.

Відео-презентація: https://youtu.be/fWGP0oppPB0

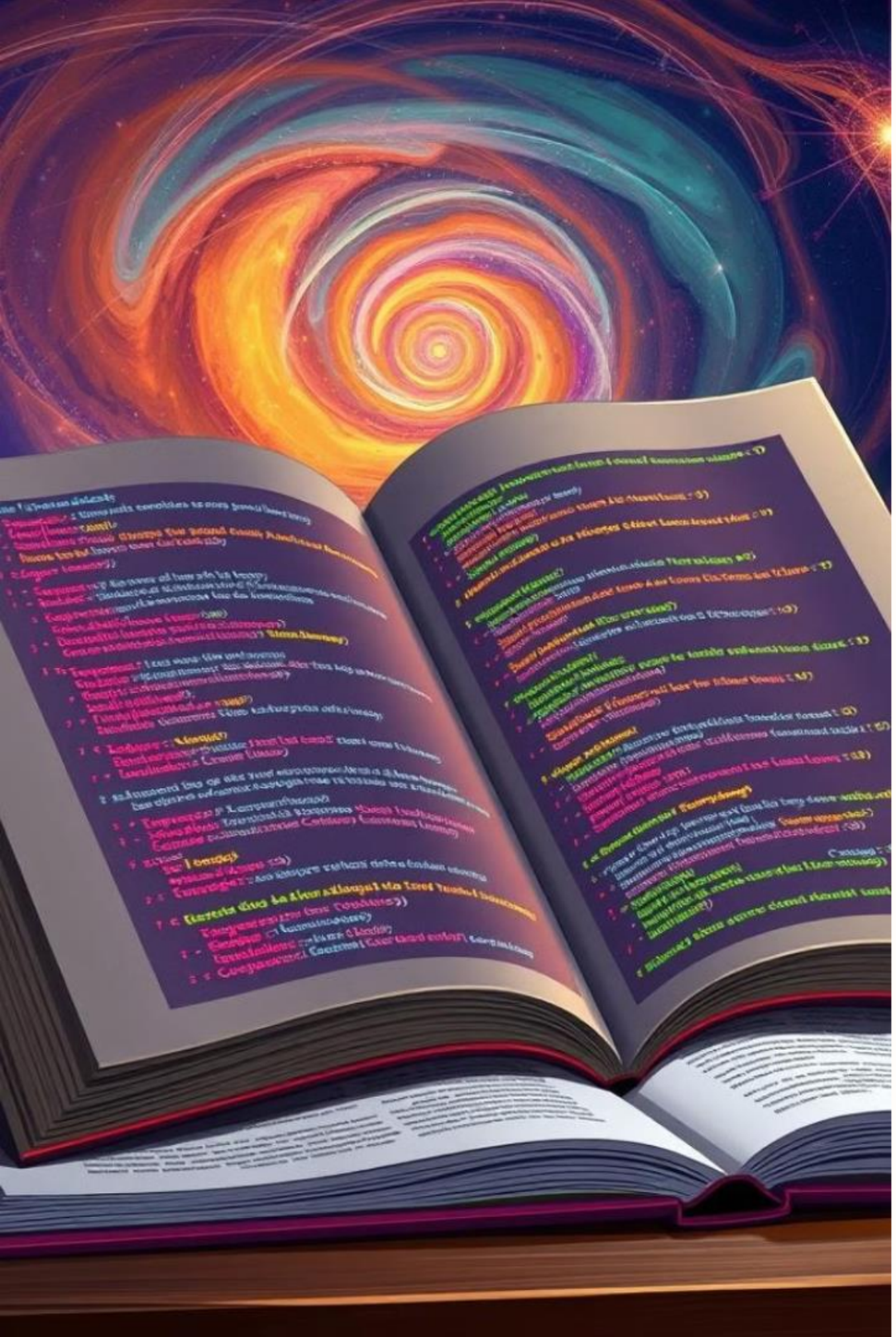
ДОДАТОК А

Програмний код, використаний як приклад у презентації.

```
class Calculator {
    // Додавання двох чисел
    fun add(a: Int, b: Int): Int {
        return a + b
    }

    // Віднімання двох чисел
    fun subtract(a: Int, b: Int): Int {
        return a - b
    }

    // Множення двох чисел
    fun multiply(a: Int, b: Int): Int {
        return a * b
    }

    // Ділення двох чисел
    fun divide(a: Int, b: Int): Int {
        if (b == 0) {
            throw IllegalArgumentException("Division by zero")
        }
        return a / b
    }
}
```

ДОДАТОК Б

Презентація на тему «Правила оформлення програмного коду».



Правила оформлення програмного коду на прикладі Kotlin

Дудуков Олександр Сергійович
ПЗПІ-22-7
08.10.2024

Вступ: чому важливе оформлення коду

1 Зрозумілий код

Легко читати, зрозуміти та опрацювати.

2 Ефективність роботи

Швидше знаходити помилки та вносити зміни.

3 Спільна робота

Зрозуміти код інших розробників та працювати в команді.

4 Довготривала підтримка

Легше підтримувати та розвивати проект з часом.

Стильові рекомендації: форматування, іменування, коментарі

Форматування

Використовуйте відступи для покращення читабельності (4 пробіли).

Застосовуйте перенос рядків для довгих рядків коду.

Іменування

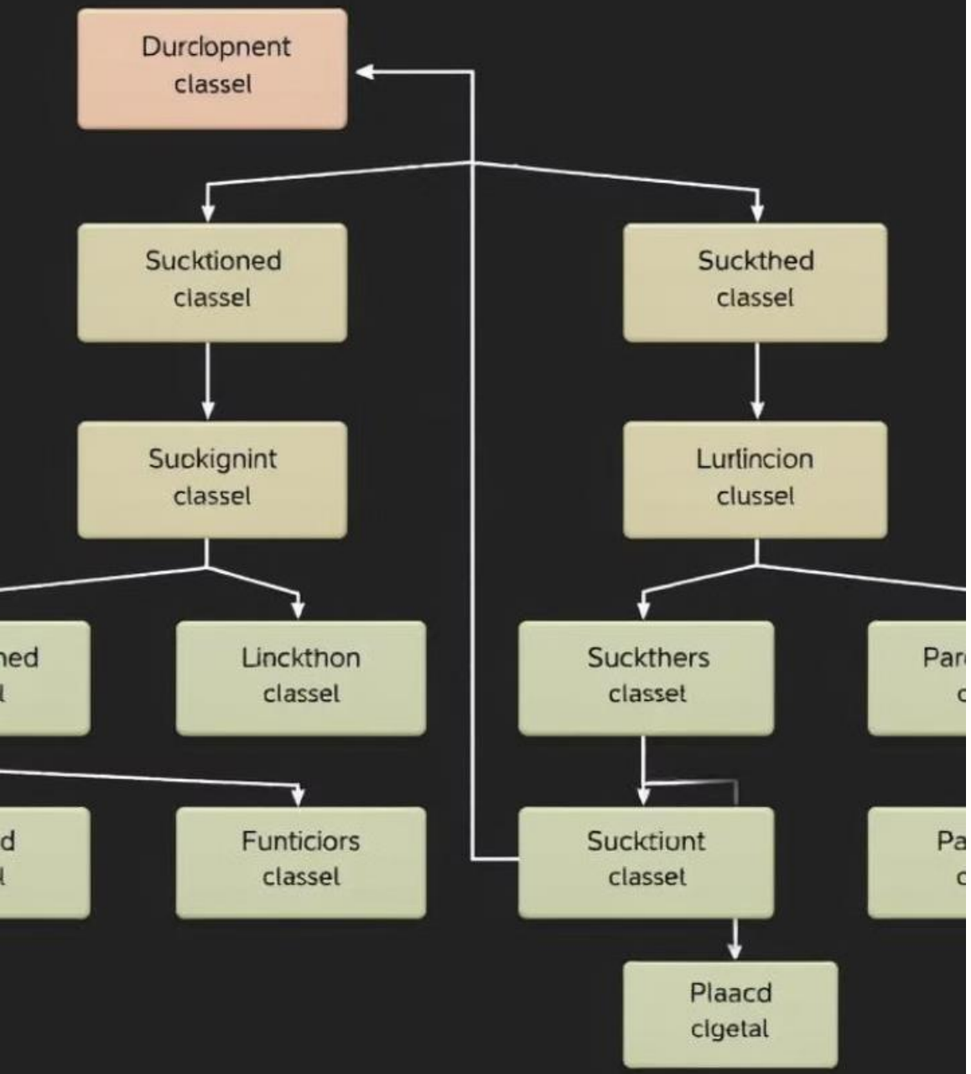
Використовуйте змістовні та зрозумілі назви для змінних, функцій, класів.

Дотримуйтесь стандартних конвенцій (CamelCase, snake_case).

Коментарі

Додайте коментарі для пояснення складних ділянок коду.

Використовуйте коментарі для пояснення логіки та намірів.



Структурні рекомендації: розділення на функції, класи, пакети

1

Функції

Розбивайте код на дрібні, самостійні функції з чітким призначенням.

2

Класи

Об'єднуйте пов'язані функції та дані в класи.

3

Пакеты

Створюйте пакети для організації коду за функціональним призначенням.

Практика чистого коду: принципи SOLID

S - Single Responsibility Principle

Кожен клас повинен відповідати за одну конкретну відповідальність.

O - Open/Closed Principle

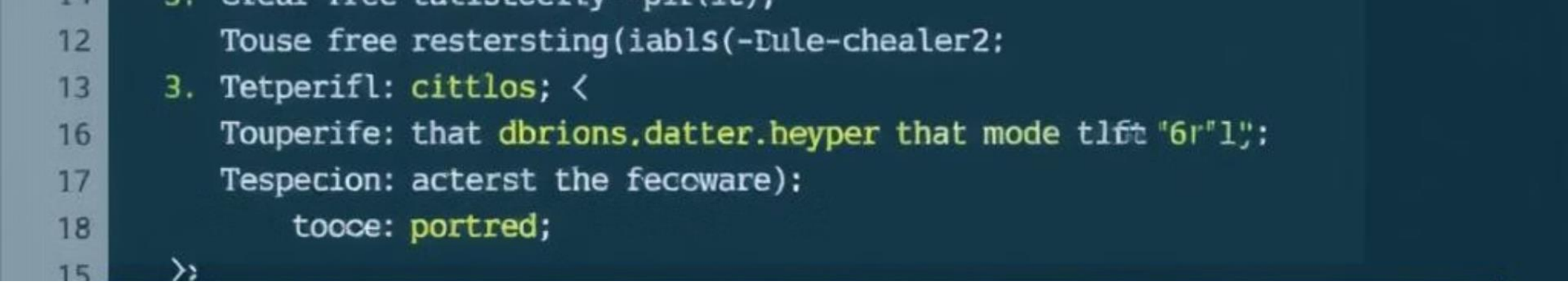
Класи та модулі повинні бути відкритими для розширення, але закритими для модифікації.

L - Liskov Substitution Principle

Підкласи повинні бути замінювані своїми батьківськими класами без порушення коректності програми.

I - Interface Segregation Principle

Інтерфейси повинні бути невеликими та специфічними.



Технічні рекомендації: використання стандартних бібліотек, обробка помилок

1

2

3

Стандартні бібліотеки

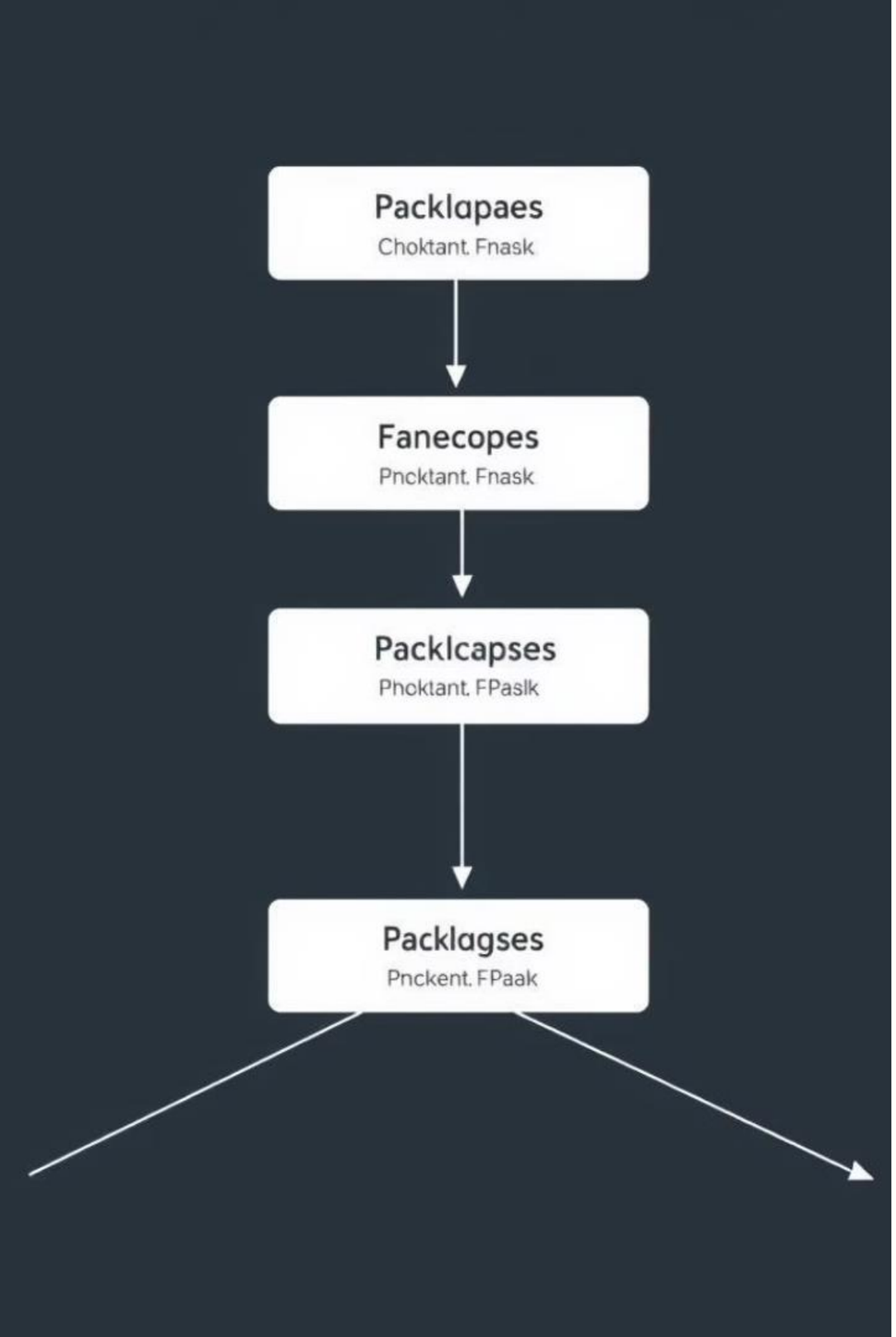
Використовуйте стандартні бібліотеки Kotlin для виконання загальних завдань.

Обробка помилок

Впровадьте чітку стратегію обробки помилок за допомогою try-catch блоку.

Виключення

Використовуйте виключення для позначення непередбачених ситуацій.



Організація імпортів та простору імен

Імпорти	Використовуйте явні імпорти для необхідних класів.
Простір імен	Організуйте код за допомогою просторів імен (package).
Унікальність	Переконайтесь, що всі імпорти є унікальними.

Приклади оформлення коду

Гарний приклад

```
class Calculator {

    fun add(number1: Int, number2: Int): Int {
        return number1 + number2
    }

    fun subtract(number1: Int, number2: Int): Int {
        return number1 - number2
    }
}
```

Пояснення:
•**Назви класів та методів**: Імена класу та функцій зрозумілі і пояснюють їх призначення (Calculator, add, subtract).
•**Стиль коду**: Дотримано стандартів іменування у camelCase, що відповідає конвенціям Kotlin.
•**Змістовність**: Кожен метод чітко відображає свою функціональність, що полегшує розуміння та підтримку коду.

Поганий приклад

```
class calc {
    fun a(n1: Int, n2: Int): Int {
        return n1 + n2
    }

    fun s(n1: Int, n2: Int): Int {
        return n1 - n2
    }
}
```

Пояснення:
•**Назви класів та методів**: Клас і функції мають неописові імена (calc, a, s), що робить код важким для розуміння.
•**Стиль**: Не дотримано конвенцій іменування змінних у стилі camelCase, що ускладнює підтримку коду.
•**Змістовність**: Імена методів не відображають їх призначення.

Приклади оформлення коду

Поганий приклад класу з дуже поганим оформленням коду

```
class calc{
 var n = 0
 var n2=0
 fun a(n1:Int,n2:Int):Int{n=n1+n2
 return n;}
 fun s(n1:Int,n2:Int):Int{n=n1-n2
 return n;}
 fun m(n1:Int,n2:Int):Int{return n1*n2}
fun d(n1:Int,n2:Int):Int{if(n2!=0){return n1/n2}else{ return 0}}
}
```

Пояснення:

•**Назви класу та змінних**: Клас названий "calc", що є занадто загальним і не відображає призначення. Імена змінних (n, n2) та функцій (a, s, m, d) неінформативні, через що важко зрозуміти їх призначення.

•**Форматування**: Відсутні відступи, незручне розміщення коду на одному рядку, що ускладнює його читання.

•**Стиль коду**: Використовуються непотрібні скорочення та абревіатури, порушені стандарти іменування. Також відсутні пробіли між операторами, що робить код важким для сприйняття.

•**Обробка помилок**: Функція d() для ділення не обробляє ситуації з діленням на нуль коректно (повертає 0, що є неочевидним для користувача).

•**Змістовність**: Код не інформує розробника, що відбувається всередині, і немає чітких коментарів або логічного розподілу на частини.

Приклади оформлення коду

Гарний приклад класу з правильним оформленням коду

Пояснення:

•**Назви класу та змінних**: Клас названий "Calculator", що чітко пояснює його призначення. Імена функцій (add, subtract, multiply, divide) зрозумілі і точно відображають їх функціональність. Змінна result використовується для зберігання результатів операцій, а її назва описова.

•**Форматування**: Використовуються чіткі відступи (4 пробіли), кожна операція на новому рядку. Це робить код легким для читання і розуміння.

•**Стиль коду**: Дотримано стандартів іменування у camelCase для функцій і змінних. Код добре структурований та читабельний.

•**Обробка помилок**: Функція divide() містить перевірку на ділення на нуль і кидає відповідний виняток, що робить обробку помилок передбачуваною.

•**Змістовність**: Кожна частина коду пояснює свою мету, а зрозумілі назви методів дозволяють легко підтримувати код у майбутньому.

```
class Calculator {

    // Змінні для збереження результату операції
    var result = 0

    // Функція для додавання двох чисел
    fun add(number1: Int, number2: Int): Int {
        result = number1 + number2
        return result
    }

    // Функція для віднімання двох чисел
    fun subtract(number1: Int, number2: Int): Int {
        result = number1 - number2
        return result
    }

    // Функція для множення двох чисел
    fun multiply(number1: Int, number2: Int): Int {
        return number1 * number2
    }

    // Функція для ділення двох чисел з перевіркою на ділення на нуль
    fun divide(number1: Int, number2: Int): Int {
        if (number2 == 0) {
            throw IllegalArgumentException("Division by zero is not allowed")
        }
        return number1 / number2
    }
}
```

Висновки та ключові моменти

Оформлення програмного коду є важливим фактором для його успіху.

Дотримання правил оформлення покращує читабельність, підтримуваність та ефективність коду.

Використовуйте наведені рекомендації для створення якісного та чистого коду в Kotlin.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Martin R. Clean Code: A Handbook of Agile Software Craftsmanship. Pearson Education, Limited.
2. Refactoring: Improving the Design of Existing Code. Addision-Wesley, 2018.