МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Звіт
з практичної роботи №2
із дисципліни «Аналіз та рефакторинг коду програмного забезпечення»
на тему “ Code Convention ”

Виконав:
ПЗПІ-22-1 Тимощук Денис

Перевірив:
доцент кафедри програмної інженерії, к.т.н.
Лещинський Володимир Олександрови.

Харків 2024

Мета роботи

Метою цієї роботи є ознайомлення з основними рекомендаціями щодо написання чистого, ефективного та підтримуваного коду на мові програмування Rust. Я прагнув не лише вивчити принципи кодування, але й продемонструвати їх на практиці через конкретні приклади. Це дозволило краще зрозуміти, як важливо дотримуватися стилю, структури та принципів оптимізації, щоб забезпечити високу якість програмного забезпечення.

Висновки

У результаті дослідження я зрозумів, що Rust є потужним інструментом для розробки, оскільки його унікальні особливості, такі як система власності, дозволяють уникати багатьох типових помилок у програмуванні. Я навчився, як важливо дотримуватися рекомендацій щодо написання коду, адже це не лише покращує його читабельність і підтримуваність, але й підвищує продуктивність і безпеку. Ця робота допомогла мені усвідомити, що якісний код є основою успішного програмного проєкту, і я відчуваю впевненість у своїх навичках роботи з Rust.

Додаток А

Скріншоти презинтації представленої на практичному занятті

Рисунок 1 – Титульна сторінка

Рисунок 2 – Вступ

Стильові рекомендації

Дотримуйтесь стильових правил мови Rust, щоб забезпечити чистоту та зрозумілість коду.

Основні принципи:
1. Використовуйте змїїний регістр (snake_case) для назв змінних і функцій.
2. Для назв типів і структур використовуйте капіталізовані назви (CamelCase).
3. Форматування коду за допомогою rustfmt для дотримання єдиного стилю.

```
5   struct Point {
6       x: i32,
7       y: i32,
8   }
```

3

Рисунок 3 – Стильові рекомендації

Структура коду

Використовуйте модулі та структури для організації коду.

Основні принципи:

1. Розбивайте великий код на модулі (mod).

2. Структуруйте код у файли та каталоги відповідно до функціональних областей.

```
5   mod geometry {
6       pub struct Circle {
7           pub radius: f64,
8       }
9
10      pub fn area(circle: &Circle) -> f64 {
11          3.14 * circle.radius * circle.radius
12      }
13  }
```

4

Рисунок 4 – Структура коду

Правила найменування змінних, функцій та класів

Пишіть тести та документуйте свій код.

Основні принципи:
1. Використовуйте модулі тестування для перевірки функціоналу.

2. Документуйте функції, щоб забезпечити їх розуміння іншими розробниками.

```
5   // Поганий приклад
6   let x = 10;
7   fn f() -> i32 { 10 }
8
9   // Гарний приклад
10  let user_age = 30;
11  fn calculate_area(radius: f64) -> f64 { 3.14 * radius * radius }
12
```

5

Рисунок 5 – Правила найменування змінних, функцій та класів

Принципи рефакторингу

Регулярно рефакторте код, щоб покращити його якість і продуктивність.

Основні принципи:
1. Використовуйте DRY (Don't Repeat Yourself).

2. Видаляйте або об'єднуйте дубльований код.

3. Використовуйте загальні типи й функції для повторюваних операцій.

```rust
// Поганий приклад
fn calculate_square(x: i32) -> i32 {
    x * x
}

fn calculate_cube(x: i32) -> i32 {
    x * x * x
}

// Гарний приклад
fn calculate_power(x: i32, power: u32) -> i32 {
    x.pow(power)
}
```

6

Рисунок 6 – Принципи рефакторингу

Обробка помилок

Використовуйте типи Result і Option для безпечної обробки помилок.

Основні принципи:

1. Обробляйте помилки через Result<T, E> замість паніки програми.

2. Використовуйте unwrap або expect лише там, де впевнені в результаті.

```rust
fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err(String::from("Cannot divide by zero"))
    } else {
        Ok(a / b)
    }
}
```

7

Рисунок 7 – Обробка помилок

Оптимізація продуктивності

Використовуйте парадигми, які найбільш підходять для задачі.

Основні принципи:
1. У Rust переважає системне та функціональне програмування.

2. Використовуйте замикання (closures) та ітератори для функціонального стилю.

```rust
let numbers = vec![1, 2, 3];
let doubled: Vec<i32> = numbers.iter().map(|x| x * 2).collect();
```

8

Рисунок 8 – Оптимізація продуктивності

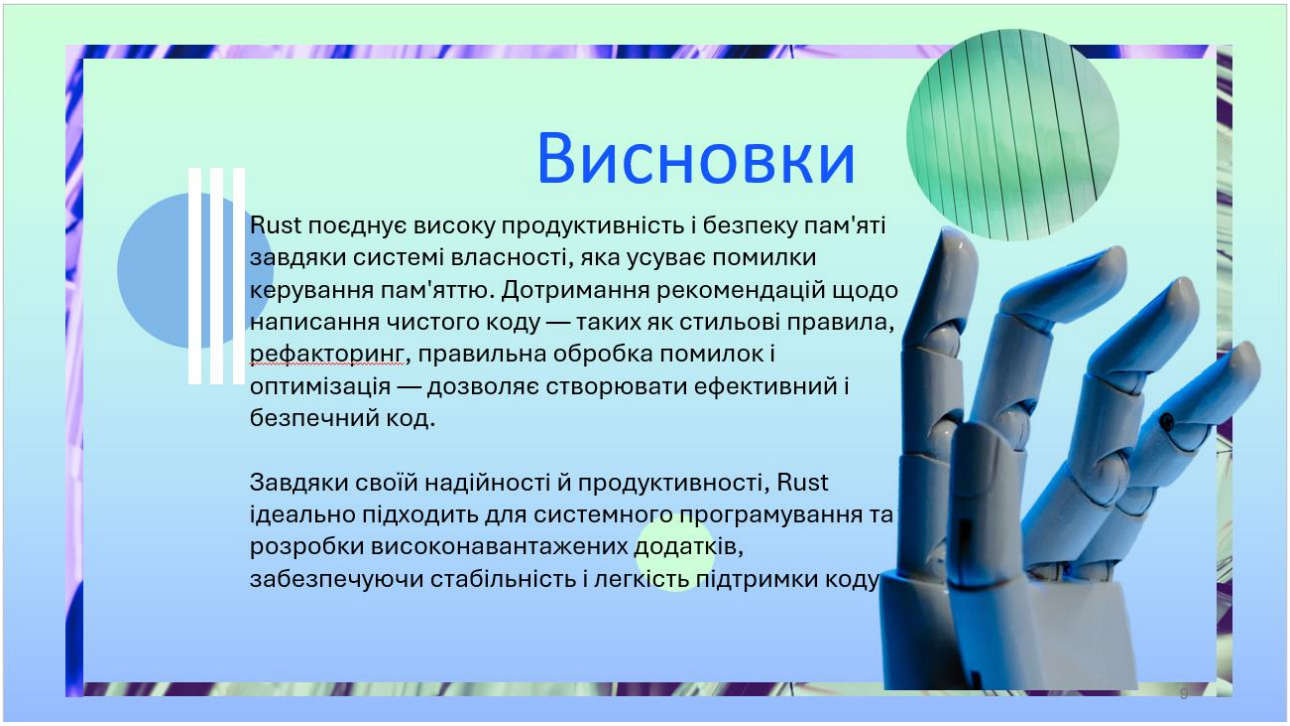

Рисунок 9 – Слайд із висновками

Рисунок 10 – Слайд із подякою за увагу