Міністерство освіти і науки України
Харківський національний університет радіоелектроніки
Кафедра програмної інженерії

Практична робота № 1

з дисципліни «Аналіз та рефакторинг коду»

з теми: « Основні рекомендації написання коду на мові GoLang»

Виконав
Ст. гр. ПЗПІ-22-10
Єлізаренко Олександр Андрійович

Перевірив
Ст. викл.
Сокорчук І. П.

2024

1. Основні положення мови GoLang. Вступ.

Golang, також відомий як Go, є сучасною, відкритою та простою мовою програмування. Golang швидко набирає популярності серед розробників програмного забезпечення. Ця мова була створена для того, щоб вирішувати сучасні проблеми в сфері розподілених систем, веб-розробки, автоматизації та багатьох інших областях.

2. Рекомендація 1:

Використовуйте осмислені назви змінних і функцій

Опис: Правильне й осмислене іменування допомагає іншим розробникам швидко розуміти призначення коду. Уникайте використання абревіатур і коротких імен без ясного значення. Це особливо важливо в мовах, як Go, де простота і читабельність коду є пріоритетами.

Чому це важливо:

Читабельність: Код, написаний зі зрозумілими іменами, легше читати і розуміти, що зменшує час, необхідний для його вивчення.

Розуміння логіки: Зрозумілі імена допомагають зрозуміти логіку та намір коду без необхідності його аналізу.

Наслідки недотримання:

Складнощі при підтримці: Використання незрозумілих імен ускладнює підтримку коду і внесення до нього змін, що може призвести до помилок.

Збільшення часу на навчання: Нові розробники витрачатимуть більше часу на розуміння коду, що уповільнює процес розробки.

Приклад:

```
// Поганий приклад
func f(a int) int {
   return a * 2
}
```

```
// Добрий приклад
func multiplyByTwo(number int) int {
    return number * 2
}
```

3. Рекомендація 2: Завжди обробляйте помилки

   Опис: У Go помилки не викидаються як винятки, а повертаються як значення. Це дає змогу явно керувати обробкою помилок і робить код надійнішим. Неправильна обробка помилок може призвести до непередбачуваних результатів.

   Чому це важливо:

   Надійність: Обробка помилок гарантує, що ваш застосунок коректно реагуватиме на несподівані ситуації, як-от збої в з'єднанні з базою даних або недоступність файлів.

   Зручність налагодження: Явна обробка помилок допомагає швидко знаходити й усувати проблеми в коді.

   Наслідки недотримання:

   Уразливість до збоїв: Якщо помилки не обробляються, ваш застосунок може аварійно завершитися або працювати неправильно, що призведе до втрати даних або неправильних результатів.

   Складність налагодження: Пропуск опрацювання помилок ускладнює налагодження, оскільки помилки можуть залишитися непоміченими доти, доки не виникне серйозна проблема.

   Приклад:

```
// Гарний приклад, помилки оброблені
```

```
func connectDB() (*sql.DB, error) {
    db, err := sql.Open("sqlite3", "example.db")
    if err != nil {
        return nil, fmt.Errorf("не вдалося підключитися: %w", err)
    }
    return db, nil
}

func main() {
    db, err := connectDB()
    if err != nil {
        log.Fatalf("Ошибка: %v", err)
    }
    defer db.Close()
    // Подальша робота...
}
// Помилки ігноруються
func connectDB() (*sql.DB, error) {
    db= sql.Open("sqlite3", "example.db"
    return db
}

func main() {
    db := connectDB()
  defer db.Close()
    // Подальша робота...
}
```

4. Рекомендація 3: Пишіть тести для вашого коду

Опис: Тестування - це важлива частина процесу розробки, що дає змогу переконатися, що ваш код працює так, як задумано. У Go є вбудований пакет testing, який спрощує написання і виконання тестів. Це допомагає підтримувати високу якість коду.

Чому це важливо:

Впевненість у якості: Наявність тестів дає змогу вам упевнено вносити зміни до коду, оскільки ви можете перевірити, що нічого не зламалося.

Запобігання регресіям: Автоматичні тести допомагають виявити та запобігти регресіям, тобто помилкам, що виникають під час внесення змін до коду.

Наслідки недотримання:

Складність змін: Без тестів може бути важко зрозуміти, як зміни в коді вплинуть на його роботу.

Вищі витрати на підтримку: За відсутності тестів час на виявлення та виправлення помилок збільшується, що в кінцевому підсумку призводить до підвищення витрат.

Приклад:

```
package main

import «testing»

// Функція
func Add(a int, b int) int {
    return a + b
}

// Тест для функції Add
```

```
func TestAdd(t *testing.T) {
    result := Add(1, 2)
    expected := 3
    if result != expected {
        t.Errorf(«Очікувалося %d, отримано %d», expected, result)
    }
}
```

5. Рекомендація 4: Використовуйте golint і go vet

golint: Цей інструмент перевіряє ваш код на відповідність стильовим рекомендаціям Go. Він допомагає виявити погані практики і поліпшити читабельність коду, забезпечуючи однаковість у проекті. Наприклад, golint може повідомляти про те, що слід використовувати певний стиль іменування для функцій або змінних.

go vet: Цей інструмент виконує глибший аналіз вашого коду, виявляючи потенційні помилки, які можуть не бути виявлені компілятором. Він перевіряє на наявність проблем, таких як неправильне використання форматів рядків, відсутнє значення під час виклику функцій та інші помилки, які можуть призвести до неправильної поведінки програми.

Чому це важливо:

- Поліпшення якості коду: Обидва інструменти допомагають виявити й усунути потенційні помилки до того, як код буде запущено, що знижує ймовірність виникнення багів у виробничому середовищі.

- Дотримання стандартів кодування: golint допомагає підтримувати загальноприйняті стилі кодування в проєкті, що полегшує читання і

розуміння коду іншими розробниками. Це особливо важливо в командах, де кілька розробників працюють над одним проєктом.

- Спрощення налагодження: go vet може виявляти проблеми, які не виявляються компілятором, даючи змогу розробникам налагодити код ще до його виконання. Це економить час і ресурси, оскільки дає змогу уникнути виправлення помилок, що виникли в процесі виконання програми.

- Підтримка майбутнього коду: Чистий і зрозумілий код легше підтримувати та розширювати. Відповідність стандартам допомагає новим розробникам швидше зрозуміти структуру та логіку коду.

6. Рекомендація 5: Використовуйте defer для управління ресурсами
   defer в Go - це оператор, який дозволяє відкладати виконання функції до завершення навколишньої функції. Це особливо корисно для управління ресурсами, такими як відкриті файли, мережеві з'єднання і блокування, забезпечуючи їх автоматичне звільнення, навіть якщо в процесі виникає помилка.

   Чому це важливо:

   - Безпека ресурсів: Використання defer допомагає гарантувати, що ресурси будуть звільнені, навіть якщо виникла помилка або panic. Це запобігає витоку ресурсів, які можуть спричинити проблеми з продуктивністю та стабільністю програми.

- Читабельність коду: Код стає чистішим і легше читається. Замість того щоб додавати код для закриття ресурсів наприкінці функції, ви можете використовувати defer, щоб явно вказати, що ресурс має бути закритий.
- Спрощення налагодження: При використанні defer вам не потрібно турбуватися про те, де у функції потрібно закрити ресурс, що спрощує налагодження і зменшує ймовірність помилок.

Наслідки недотримання

Витоки ресурсів: Якщо ви забуваєте звільняти ресурси, напиклад файли або мережеві з'єднання, ваш застосунок може використовувати всі доступні ресурси, що призведе до збоїв або зависань.

Збільшення складності коду: Відсутність defer призводить до додавання багаторазового коду для звільнення ресурсів у кількох місцях, що ускладнює супровід і розуміння коду.

Неочевидні помилки: Якщо помилка станеться перед звільненням ресурсу, це може призвести до неочевидних проблем, які важко відстежити.

Приклади:

```
func main() {
file, err := os.Open("file.txt")
if err != nil {
fmt.Println("Помилка при відкритті:", err)
return
}
// не закривається файл при помилці
```

```
data := make([]byte, 100)
_, err = file.Read(data)
if err != nil {
fmt.Println("Помилка читання:", err)
// Файл не залишається не закритим
return
}
fmt.Println("Данные:", string(data))
// Не відбулося закриття
}
```

Висновки: під час доповіді було розглянуто основні рекомендації для написання чистого та зрозуміло коду. Це дозволяє зменшити кількість помилок під час виконання програми, дає змогу швидко їх виправити на місці.

Додаток А

Рекомендації написання коду на мові Go

Єлізаренко Олександр ПЗПІ22-10

Вступ до мови

основні положення

- **Простота і лаконічність**
- **Статична типізація**
- **Збирач сміття**
- **Горутини**
- **Інтерфейси**
- **Кроссплатформенність**
- **Команда і спільнота**
- **Продуктивність**

Рекомендація 1

Використовуйте осмислені назви змінних і функцій

Використовуйте осмислені назви змінних і функцій

Уникайте використання абревіатур і коротких імен без ясного значення

Код, написаний зі зрозумілими іменами, легше читати і розуміти

Приклад

```
//  Поганий приклад
func f(a int) int {
   return a * 2
}


//  Добрий приклад
func multiplyByTwo(number int) int {
   return number * 2
}
```

Рекомендація 2

Завжди обробляйте помилки

Неправильна обробка помилок може призвести до непередбачуваних результатів

Обробка помилок гарантує, що ваш додаток буде коректно реагувати на несподівані ситуації

Якщо помилки не обробляються, ваш додаток може аварійно завершитися або працювати неправильно

Приклад

```
// Гарний приклад помилки оброблені
func connectDB() (*sql.DB, error) {
  db, err := sql.Open("sqlite3", " example.db")
  if err != nil {
    return nil, fmt.Errorf ("не вдалося підключитися: %w", err)
  }
  return db, nil
}

func main() {
  db, err := connectDB()
  if err != nil {
    log.Fatalf("Ошибка: %v", err)
  }
  defer db.Close()
  // Подальша робота ...
}
```

```
// Помилки ігноруються
func connectDB() (*sql.DB, error) {
  db= sql.Open("sqlite3", " example.db"
  return db
}

func main() {
  db := connectDB()
    defer db.Close()
  // Подальша робота ...
}
```

Рекомендація 3

пишіть тести для коду

Наявність тестів дає вам змогу впевнено вносити зміни до коду, адже ви можете перевірити, що нічого не зламалося.

За відсутності тестів час на виявлення та виправлення помилок збільшується

Приклад

```
//  Функція
func Add(a int, b int) int {
   return a + b
}

//  Тест  дляфункції Add
func TestAdd(t *testing.T) {
   result := Add(1, 2)
   expected := 3
   if result != expected {
      t.Errorf («Очікувалося%d, отримано  %d», expected, result)
   }
}
```

Рекомендація 4

Використовуйте golint і go vet

це інструменти статичного аналізу коду в Go, які допомагають розробникам поліпшити якість і читабельність їхнього коду. Вони аналізують вихідний код і виявляють потенційні проблеми, які можуть призвести до помилок під час виконання

Приклад

Встановлення:
go get -u golang.org/x/lint/golint

Вискористання:
golint ./...
go vet ./...

Рекомендація 5

Використовуйте defer для управління ресурсами

Використання defer допомагає гарантувати, що ресурси будуть звільнені, навіть якщо виникла помилка.
При використанні defer вам не потрібно турбуватися про те, де у функції потрібно закрити ресурс, що спрощує налагодження і зменшує ймовірність помилок.
Якщо ви забуваєте звільняти ресурси, наприклад файли або мережеві з'єднання, ваш застосунок може використовувати всі доступні ресурси

Приклад

```
func main() {
    file, err := os.Open("file.txt")
    if err != nil {
        fmt.Println("Помилкапри відкритті :", err)
        return
    }

    // не закривається файл при помилці
    data := make([]byte, 100)
    _, err = file.Read(data)
    if err != nil {
        fmt.Println("Помилкачитання :", err)
        // Файл не залишається не закритим
        return
    }

    fmt.Println("Дані:", string(data))
    // Не відбулося закриття
}
```

```
func main() {
    file, err := os.Open("file.txt")
    if err != nil {
        fmt.Println("Помилкапри відкритті :", err)
        return
    }
    defer file.Close() // Навіть при помилці файл закриється

    data := make([]byte, 100)
    _, err = file.Read(data)
    if err != nil {
        fmt.Println("Помилкачитання :", err)
        // все одно закриється завдяки defer
        return
    }

    fmt.Println("Дані:", string(data))
}
```