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

Практична робота № 2

з дисципліни «Аналіз та рефакторинг коду»

з теми: «Методи рефакторингу коду програмного забезпечення»

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

Перевірив
Ст. викл.
Сокорчук І. П.

2024

1. Вступ

   Рефакторинг коду є важливою складовою частиною процесу розробки програмного забезпечення, що сприяє покращенню його якості та зручності для подальшої підтримки та розширення. З метою підвищення читаємості, зниження складності та покращення продуктивності, розробники застосовують різноманітні методи рефакторингу.

   Серед популярних методів рефакторингу можна виокремити такі, як **Preserve Whole Object**, **Push Down Field** та **Tease Apart Inheritance**. Кожен з них дозволяє розв'язувати конкретні проблеми, що виникають в процесі розробки та розширення програмного коду

   У цій роботі буде розглянуто застосування зазначених методів рефакторингу в контексті мови Go

2. **Preserve Whole Object**

   Preserve Whole Object - це рефакторинговий підхід, він передбачає передачу цілого об'єкта у функцію або метод замість передачі окремих його полів аргументами. Це допомагає зробити код більш читабельним, стійким до змін, і зменшує кількість параметрів у методах.

   Ідея:

   Якщо у вас є метод, який використовує кілька атрибутів об'єкта, замість того щоб передавати ці поля окремо, краще передати цілий об'єкт. Це полегшує розширення функціональності, оскільки додавання нових полів до об'єкта не вимагає зміни інтерфейсу методу. Код стає простішим і зрозумілішим, а кількість переданих параметрів скорочується.

Приклад

```
package main

import (
"fmt"
)

type Rectangle struct {
Width  float64
Height float64
}

func CalculateArea(width, height float64) float64 {
return width * height
}

func CalculatePerimeter(width, height float64) float64 {
return 2 * (width + height)
}

func main() {
rect := Rectangle{Width: 10, Height: 5}

// передаємо окремі поля кожен раз
area := CalculateArea(rect.Width, rect.Height)
perimeter := CalculatePerimeter(rect.Width, rect.Height)

fmt.Printf("Area: %.2f, Perimeter: %.2f\n", area, perimeter)
```

```
}
```

Приклад після рефакторингу:

```
package main

import (
"fmt"
)

type Rectangle struct {
Width  float64
Height float64
}

// Тут вже передаємо весь обєкт
func CalculateArea(rect Rectangle) float64 {
return rect.Width * rect.Height
}

func CalculatePerimeter(rect Rectangle) float64 {
return 2 * (rect.Width + rect.Height)
}

func main() {
rect := Rectangle{Width: 10, Height: 5}

// вже передаємо весь об’єкт
area := CalculateArea(rect)
```

```
perimeter := CalculatePerimeter(rect)

fmt.Printf("Area: %.2f, Perimeter: %.2f\n", area, perimeter)
}
```

Переваги Preserve Whole Object

Спрощення коду: Замість передачі кількох параметрів передається один об'єкт.

Краща читабельність: З коду функції відразу зрозуміло, що вона працює з об'єктом.

Стійкість до змін: Якщо об'єкт розширюється, не потрібно змінювати сигнатури всіх функцій, що працюють із цим об'єктом.

Коли варто застосовувати:

Коли функція або метод використовує 2 і більше поля з одного об'єкта.

Коли потрібно зробити код більш гнучким і готовим до змін.

Коли ви хочете скоротити кількість аргументів у методах.

Цей метод особливо корисний у великих проєктах, де зміни структури даних можуть швидко ускладнити супровід коду.

3. **Push Down Field**

Push Down Field - це рефакторинг, під час якого поле, визначене в батьківському класі, переміщується в підклас, в якому воно використовується.

Ідея:

Якщо поле в суперкласі використовується тільки деякими підкласами, його слід перемістити вниз, у ці конкретні підкласи.

Це допомагає зробити код чистішим, зменшує навантаження суперкласу і покращує інкапсуляцію, оскільки поле існує тільки там, де воно дійсно потрібне.

Приклад до рефакторингу:

```
package main

import "fmt"

// Батьківський клас
type Employee struct {
    Name       string
    Department string // Це поле використовується не у всіх підкласах
}

// Підклас менеджер
type Manager struct {
    Employee
}

// Підклас інженер
type Engineer struct {
    Employee
}

func main() {
    manager := Manager{Employee{Name: "Alice", Department: "HR"}}
    engineer := Engineer{Employee{Name: "Bob", Department: ""}}
```

```
    fmt.Printf("Manager: %s, Department: %s\n", manager.Name,
manager.Department)
    fmt.Printf("Engineer: %s, Department: %s\n", engineer.Name,
engineer.Department)
}
```

Тут видно що Department використовується не всюди, тому він забруднює батьківський клас.

Приклад після рефакторингу:

```
package main

import "fmt"

type Employee struct {
    Name string
}

type Manager struct {
    Employee
    Department string // Поле переміщено в підклас
}

type Engineer struct {
    Employee
}
```

```
func main() {
    manager := Manager{Employee: Employee{Name: "Alice"}, Department:
"HR"}
    engineer := Engineer{Employee: Employee{Name: "Bob"}}

    fmt.Printf("Manager: %s, Department: %s\n", manager.Name,
manager.Department)
    fmt.Printf("Engineer: %s\n", engineer.Name)
}
```

Переваги Push Down Field

Чистота коду: Видаляються непотрібні поля з суперкласу, залишаючи в ньому тільки те, що відноситься до всіх підкласів.

Інкапсуляція: Поля знаходяться ближче до тих класів, які їх дійсно використовують.

Зменшення ризику помилок: Поля не заважають в коді, в місцях де вони не потрібні.

Коли застосовувати:

Коли поле суперкласу використовується тільки частиною його підкласів.

Коли суперклас починає розширятися непотрібними полями, які не належать до всіх підкласів.

Коли потрібно поліпшити читабельність та інкапсуляцію.

Іноді після застосування Push Down Field має сенс розглянути можливість видалення суперкласу, якщо після перенесення полів він стає марним або містить занадто мало загальної функціональності.

4. **Tease Apart Inheritance**

Tease Apart Inheritance - це рефакторинг, який розділяє заплутану і перевантажену ієрархію успадкування на кілька простіших, незалежних структур. Мета полягає в усуненні небажаних зв'язків і змішання відповідальності між класами.

Ідея:

Якщо клас виконує кілька ролей (відповідальностей), він порушує принцип SRP Це робить код складним для розуміння і супроводу.

Замість того щоб використовувати одну ієрархію для представлення кількох аспектів, створюються незалежні класи або інтерфейси для розділення ролей. Це допомагає зменшити зв'язаність і поліпшити читабельність.

Приклад до рефакторингу:

```
package main

import "fmt"

// Базовий клас
type Employee struct {
Name   string
Salary float64
}

// Менеджер, який має спільні методи з  Employee
type Manager struct {
Employee
TeamSize int
```

```
}

// Инженер, теж наслідує Employee, але також додає свою логіку
type Engineer struct {
Employee
Skillset []string
}

func main() {
manager := Manager{Employee: Employee{Name: "Alice", Salary: 80000},
TeamSize: 5}
engineer := Engineer{Employee: Employee{Name: "Bob", Salary: 70000}, Skillset:
[]string{"Go", "Python"}}

fmt.Printf("Manager: %s, Team Size: %d\n", manager.Name, manager.TeamSize)
fmt.Printf("Engineer: %s, Skills: %v\n", engineer.Name, engineer.Skillset)
}
```

Проблеми коду:

- Клас Employee містить загальні властивості, але Manager і Engineer додають специфічну логіку, що робить успадкування надлишковим.
- Додавання нових ролей може призвести до зростання зв'язності в ієрархії.
- Логіка управління співробітниками та їхньою функціональністю перемішана.

Приклад після рефакторингу

```
package main

import "fmt"

// Базовий клас
type Employee struct {
Name   string
Salary float64
}

// Інтерфейс ролі менеджера
type ManagerRole struct {
TeamSize int
}

// Інтерфейс ролі інженера
type EngineerRole struct {
Skillset []string
}

func main() {
// Менеджер
manager := struct {
        Employee
        ManagerRole
}{
        Employee:    Employee{Name: "Alice", Salary: 80000},
```

```
        ManagerRole: ManagerRole{TeamSize: 5},
}

// Інженер
engineer := struct {
        Employee
        EngineerRole
}{
        Employee:     Employee{Name: "Bob", Salary: 70000},
        EngineerRole: EngineerRole{Skillset: []string{"Go", "Python"}},
}

fmt.Printf("Manager: %s, Team Size: %d\n", manager.Name, manager.TeamSize)
fmt.Printf("Engineer: %s, Skills: %v\n", engineer.Name, engineer.Skillset)
}
```

Переваги Tease Apart Inheritance

Чистота ієрархії: Прибираються надлишкові зв'язки, які могли заважати розумінню.

Гнучкість: Ролі можна комбінувати або змінювати без зміни основної структури.

Розширюваність: Легше додавати нові ролі або властивості, не зачіпаючи інших класів.

Слабка пов'язаність: Класи стають незалежними і простіше тестуються.

Коли застосовувати

Коли клас або ієрархія виконують кілька різних ролей

Коли додавання нових класів до наявної ієрархії призводить до надмірної складності.

Коли частину функціональності класу краще виділити в окремі компоненти.

Додаток А

Методи рефакторингу коду програмного забезпечення

Єлізаренко Олександр ПЗПІ-22-10

Вступ до мови

Обрані методи:

- Preserve Whole Object
- Push Down Field
- Tease Apart Inheritance

Preserve Whole Object

- Спрощення коду
- Краща читабельність
- Стійкість до змін

Приклад

```
type Rectangle struct {
 Width float64
 Height float64
}
func CalculateArea(width, height float64) float64 {
 return width * height
}
func CalculatePerimeter(width, height float64) float64 {
 return 2 * (width + height)
}
func main() {
 rect := Rectangle{Width: 10, Height: 5}
 // передаємо окремі поля кожен раз
 area := CalculateArea(rect.Width, rect.Height)
 perimeter := CalculatePerimeter(rect.Width, rect.Height)

 fmt.Printf("Area: %.2f, Perimeter: %.2f\n", area, perimeter)
}
```

```
type Rectangle struct {
 Width float64
 Height float64
}
// Тут вже передаємо весь обєкт
func CalculateArea(rect Rectangle) float64 {
 return rect.Width * rect.Height
}
func CalculatePerimeter(rect Rectangle) float64 {
 return 2 * (rect.Width + rect.Height)
}
func main() {
 rect := Rectangle{Width: 10, Height: 5}
 // вже передаємо весь об'єкт
 area := CalculateArea(rect)
 perimeter := CalculatePerimeter(rect)
 fmt.Printf("Area: %.2f, Perimeter: %.2f\n", area, perimeter)
}
```

Push Down Field

ІДЕЯ

Якщо поле в суперкласі використовується тільки деякими підкласами, його слід перемістити вниз, у ці конкретні підкласи.
Це допомагає зробити код чистішим, зменшує навантаження суперкласу і покращує інкапсуляцію, оскільки поле існує тільки там, де воно дійсно потрібне.

Приклад

```
// Батьківський клас
type Employee struct {
    Name      string
    Department string // Це поле використовується не у всіх підкласах
}
// Підклас менеджер
type Manager struct {
    Employee
}
// Підклас інженер
type Engineer struct {
    Employee
func main() {
    manager := Manager{Employee{Name: "Alice", Department: "HR"}}
    engineer := Engineer{Employee{Name: "Bob", Department: ""}}
```

```
type Employee struct {
    Name string
}
type Manager struct {
    Employee
    Department string // Поле переміщено в підклас
}
type Engineer struct {
    Employee
}
func main() {
    manager := Manager{Employee: Employee{Name: "Alice"}, Department: "HR"}
    engineer := Engineer{Employee: Employee{Name: "Bob"}}

    fmt.Printf("Manager: %s, Department: %s\n", manager.Name, manager.Department)
    fmt.Printf("Engineer: %s\n", engineer.Name)
}
```

Push Down Field

ПЕРЕВАГИ

- Чистота коду: Видаляються непотрібні поля з суперкласу, залишаючи в ньому тільки те, що відноситься до всіх підкласів.
- Інкапсуляція: Поля знаходяться ближче до тих класів, які їх дійсно використовують.
- Зменшення ризику помилок: Поля не заважають в коді, в місцях де вони не потрібні.

Tease Apart Inheritance

Якщо клас виконує кілька ролей, він порушує принцип SRP Це робить код складним для розуміння і супроводу.
Замість того щоб використовувати одну ієрархію для представлення кількох аспектів, створюються незалежні класи або інтерфейси для розділення ролей.
Це допомагає зменшити зв'язаність і поліпшити читабельність

Приклад

```
// Базовий клас
type Employee struct {
 Name  string
 Salary float64
}
// Менеджер, який має спільні методи з  Employee
type Manager struct {
 Employee
 TeamSize int
}
// Инженер, теж наслідує Employee, але також додає свою логіку
type Engineer struct {
 Employee
 Skillset []string
}
func main() {
 manager := Manager{Employee: Employee{Name: "Alice", Salary: 80000}, TeamSize: 5}
 engineer := Engineer{Employee: Employee{Name: "Bob", Salary: 70000}, Skillset: []string{"Go", "Python"}}
```

```
type Employee struct {
 Name  string
 Salary float64
}
// Інтерфейс ролі менеджера
type ManagerRole struct {
 TeamSize int
}
// Інтерфейс ролі інженера
type EngineerRole struct {
 Skillset []string
}
func main() {
 // Менеджер
 manager := struct {
        Employee
        ManagerRole
 }{
        Employee:   Employee{Name: "Alice", Salary: 80000},
        ManagerRole: ManagerRole{TeamSize: 5},
 }
 // Інженер
 engineer := struct {
        Employee
        EngineerRole
 }{
        Employee:    Employee{Name: "Bob", Salary: 70000},
        EngineerRole: EngineerRole{Skillset: []string{"Go", "Python"}},
 }
```

Tease Apart Inheritance

КОЛИ НЕ ТРЕБА ВИКОРИСТОВУВАТИ

- Якщо успадкування використовується правильно і класи вже мають чітке розмежування обов'язків.
- Якщо поділ ролей призводить до занадто складної структури, яку складно підтримувати.

Використані джерела

Martin Fowler. Refactoring. Improving the Design of Existing Code– Addison-Wesley Professional, 1999. – 464 p.