МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Кафедра «Програмна інженерія»

ЗВІТ
до практичного заняття №2 з дисципліни
«Аналіз та рефакторинг коду»
На тему: «Методи рефакторингу коду програмного забезпечення»

Виконав:
ПЗПІ-22-10
Петриков Олександр Дмитрович

Перевірив:ст. гр.
ст. викладач кафедри ПІ
Сокорчук Ігор Петрович

Харків 2024

Мета:

Ознайомитися з різними методами рефакторингу програмного коду та навчитися використовувати їх на реальних прикладах.

Завдання:

Обрати три методи рефакторингу коду із книги Мартіна Фаулера «Refactoring. Improving the Design of Existing Code». Створити презентацію де продемонструвати використання обраних методів на своєму коді.

Remove Middle Man
Substitute Algorithm
Replace Magic Number with Symbolic Constant

https://youtu.be/1-4AW7TqAXg

Вступ

Що таке рефакторинг?

- Рефакторинг — це процес внесення змін до внутрішньої структури коду без зміни його зовнішньої поведінки.
- Основна мета: покращення читабельності, підтримуваності та масштабованості коду.

Навіщо потрібен рефакторинг?

- Полегшення розуміння та підтримки коду.
- Зменшення ризиків внесення помилок у майбутньому.

1. Remove Middle Man

Метод використовується для видалення посередницької логіки в коді, яка додає зайву складність і не приносить додаткової цінності.
Замість того щоб викликати метод через "посередника", доступ до необхідного елемента здійснюється напряму.

Коли застосовувати:

- Якщо метод посередника просто делегує виклик іншому методу або об'єкту без додавання власної логіки.
- Коли це робить код важким для розуміння та обслуговування.

Переваги:

- Зменшення рівня абстракції.
- Простий і читабельний код.

Приклад:

До рефакторингу:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char name[50];
    int age;
} Employee;
typedef struct {
    Employee *manager;
} Department;

// Посередник для отримання менеджера
Employee* get_manager(Department *dept) {
    return dept->manager;
}
```

```
int main() {
    Employee *manager = (Employee *)malloc(sizeof(Employee));
    Department dept = {manager};
    // Доступ до менеджера через посередника
    Employee *dept_manager = get_manager(&dept);
    printf("Manager age: %d\n", dept_manager->age);
    free(manager);
    return 0;
}
```

Після рефакторингу:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char name[50];
    int age;
} Employee;

typedef struct {
    Employee *manager;
} Department;
```

```
int main() {
    Employee *manager = (Employee *)malloc(sizeof(Employee));
    Department dept = {manager};

    // Прямий доступ до менеджера
    printf("Manager age: %d\n", dept.manager->age);

    free(manager);
    return 0;
}
```

Пояснення:

1. До рефакторингу код використовував функцію get_manager, яка виступала посередником.
    - Це додає зайву абстракцію і робить код складнішим для розуміння.
2. Після рефакторингу ми прибрали метод-посередник, отримуючи менеджера напряму через dept.manager.

2. Substitute Algorithm

Цей метод передбачає заміну існуючого алгоритму більш простим, ефективним або зрозумілим, зберігаючи ту саму функціональність.

Коли застосовувати:

- Коли існуючий алгоритм працює коректно, але є надто складним або важко зрозумілим.

- Якщо новий алгоритм пропонує кращу продуктивність або є більш узагальненим для різних випадків використання.

Переваги:

- Покращення продуктивності та зрозумілості коду.
- Полегшення підтримки та розширення функціональності.

Приклад:

```
#include <stdio.h>

#include <string.h>

// Пошук найкоротшого слова у рядку (неоптимальний алгоритм)

void find_shortest_word(char *sentence, char *shortest_word) {

    int min_length = 100;

    char *word = strtok(sentence, " ");

    while (word != NULL) {

        int len = strlen(word);

        if (len < min_length) {

            min_length = len;

            strcpy(shortest_word, word);

        }

        word = strtok(NULL, " ");

    }

}


int main() {
```

```
    char sentence[] = "C programming is versatile";
    char shortest_word[50];
    find_shortest_word(sentence, shortest_word);
    printf("Shortest word: %s\n", shortest_word);
    return 0;
}
```

Після рефакторингу:

```
#include <stdio.h>
#include <string.h>

// Оптимізований алгоритм для пошуку найкоротшого слова
void find_shortest_word(const char *sentence, char *shortest_word) {
    const char *start = sentence;
    const char *current = sentence;
    int min_length = 100;

    while (*current) {
        if (*current == ' ' || *(current + 1) == '\0') {
            int word_length = current - start + (*(current + 1) == '\0');
            if (word_length < min_length) {
                min_length = word_length;
                strncpy(shortest_word, start, word_length);
```

```
                shortest_word[word_length] = '\0';
            }
            start = current + 1;
        }
        current++;
    }
}

int main() {
    const char sentence[] = "C programming is versatile";
    char shortest_word[50];
    find_shortest_word(sentence, shortest_word);
    printf("Shortest word: %s\n", shortest_word);
    return 0;
}
```

Пояснення:

1. До рефакторингу використовувався strtok, який змінює оригінальний рядок, що робить алгоритм небезпечним для повторного використання.
2. Після рефакторингу алгоритм працює без зміни вихідного рядка, використовуючи індекси для визначення слів.

3. Replace Magic Number with Symbolic Constant

Цей метод полягає у видаленні "магічних чисел" (чисел,

значення яких незрозуміле з контексту) і заміні їх на константи з осмисленими іменами.

Коли застосовувати:

- Коли в коді використовуються числа, призначення яких неочевидне.
- Якщо число використовується багаторазово і його може знадобитися змінити в майбутньому.

Переваги:

- Збільшення зрозумілості коду.
- Зменшення помилок, пов’язаних із внесенням змін у кількох місцях.

Приклад:

До рефакторингу:

```
#include <stdio.h>


// Перевірка віку для доступу до послуги
void check_access(int age) {
    if (age < 18) {
        printf("Access denied. Age below 18.\n");
    } else {
        printf("Access granted.\n");
    }
}


int main() {
```

```
    check_access(16);
    check_access(20);
    return 0;
}
```

Після рефакторингу:

```
#include <stdio.h>

#define MINIMUM_AGE 18 // Символічна константа

// Перевірка віку для доступу до послуги
void check_access(int age) {
    if (age < MINIMUM_AGE) {
        printf("Access denied. Age below %d.\n", MINIMUM_AGE);
    } else {
        printf("Access granted.\n");
    }
}

int main() {
    check_access(16);
    check_access(20);
    return 0;
```

```
}
```

Пояснення:

1. До рефакторингу значення 18 було вбудоване в код. Це ускладнює зміни, якщо правила зміняться.
2. Після рефакторингу значення 18 замінено на символічну константу MINIMUM_AGE.

Інструменти для рефакторингу

1. JetBrains IDE: Інструмент для автозаміни, перейменування та аналізу.
2. Visual Studio: Містить засоби для рефакторингу та перевірки якості.
3. Eclipse: Пропонує інструменти для роботи з великими проектами.

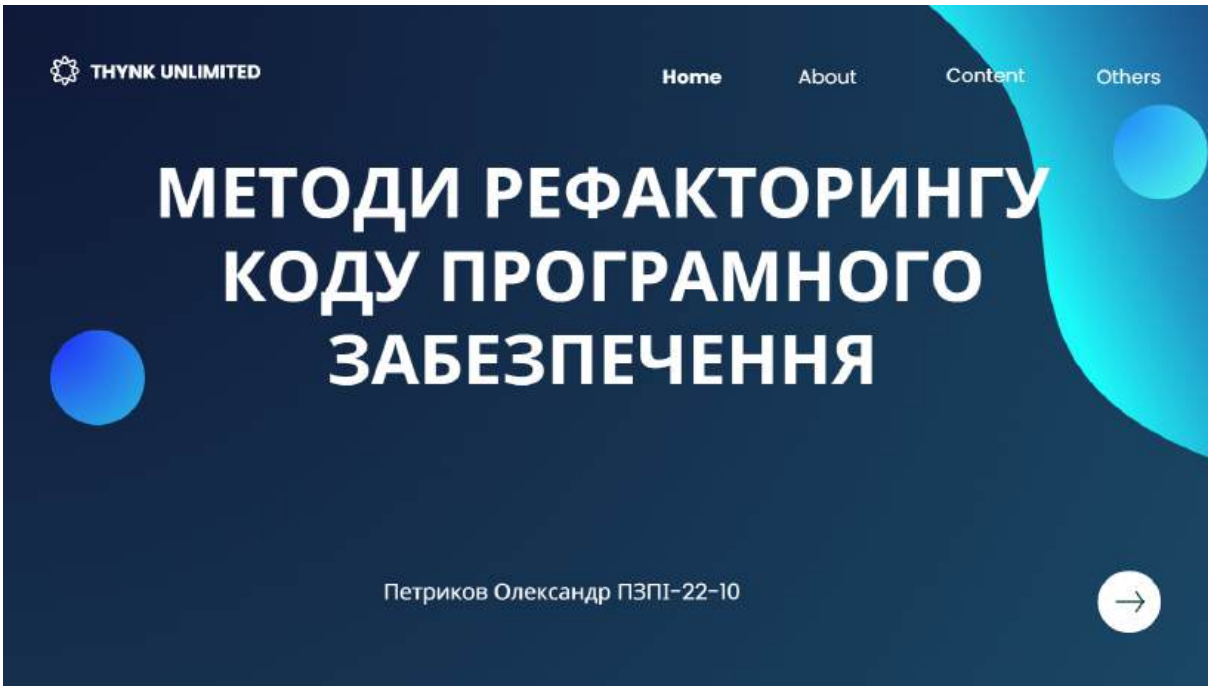
Висновки

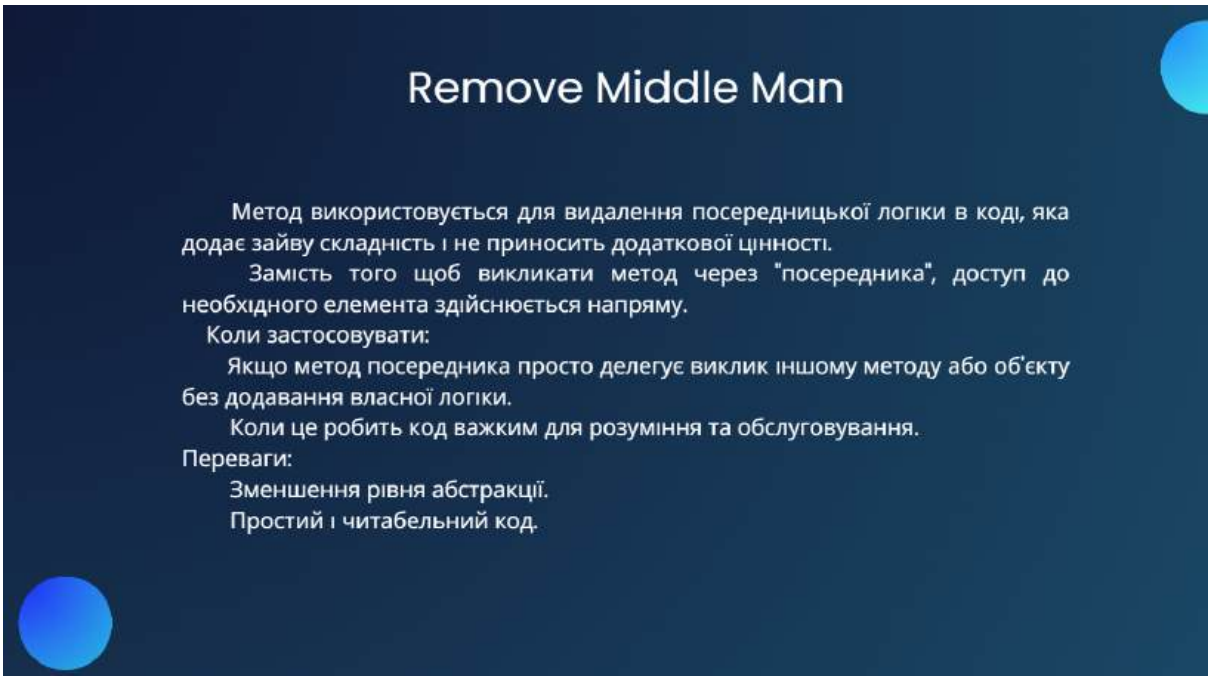
- Рефакторинг дозволяє підтримувати якість коду на високому рівні.
- Застосування рефакторингу важливе для зменшення технічного боргу.
- Методи, як-от Remove Middle Man, Substitute Algorithm, та Replace Magic Number with Symbolic Constant, роблять код більш читабельним і ефективним

Список використаних джерел

1. Фаулер Мартін. "Refactoring: Improving the Design of Existing Code".
2. Мартін Роберт. "Clean Code: A Handbook of Agile Software Craftsmanship".

3. Refactoring.Guru: https://refactoring.guru.
4. Документація JetBrains IDE: https://www.jetbrains.com/help/idea/refactoring-source-code.html.





Remove Middle Man

До рефакторингу:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   char name[50];
   int age;
} Employee;
typedef struct {
   Employee *manager;
} Department;

// Посередник для отримання менеджера
Employee* get_manager(Department *dept) {
   return dept->manager;
}
int main() {
   Employee *manager = (Employee *)malloc(sizeof(Employee));
   Department dept = {manager};
   // Доступ до менеджера через посередника
   Employee *dept_manager = get_manager(&dept);
   printf("Manager age: %d\n", dept_manager->age);
   free(manager);
   return 0;
}
```

Після рефакторингу:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   char name[50];
   int age;
} Employee;

typedef struct {
   Employee *manager;
} Department;

int main() {
   Employee *manager = (Employee *)malloc(sizeof(Employee));
   Department dept = {manager};

   // Прямий доступ до менеджера
   printf("Manager age: %d\n", dept.manager->age);

   free(manager);
   return 0;
}
```

Substitute Algorithm

Цей метод передбачає заміну існуючого алгоритму більш простим, ефективним або зрозумілим, зберігаючи ту саму функціональність.

Коли застосовувати:

Коли існуючий алгоритм працює коректно, але є надто складним або важко зрозумілим.

Якщо новий алгоритм пропонує кращу продуктивність або є більш узагальненим для різних випадків використання.

Переваги:

Покращення продуктивності та зрозумілості коду.

Полегшення підтримки та розширення функціональності.

Remove Middle Man

```c
#include <stdio.h>
#include <string.h>
// Пошук найкоротшого слова у рядку (неоптимальний алгоритм)
void find_shortest_word(char *sentence, char *shortest_word) {
    int min_length = 100;
    char *word = strtok(sentence, " ");
    while (word != NULL) {
        int len = strlen(word);
        if (len < min_length) {
            min_length = len;
            strcpy(shortest_word, word);
        }
        word = strtok(NULL, " ");
    }
}

int main() {
    char sentence[] = "C programming is versatile";
    char shortest_word[50];
    find_shortest_word(sentence, shortest_word);
    printf("Shortest word: %s\n", shortest_word);
    return 0;
}
```

Після рефакторингу:

```c
#include <stdio.h>
#include <string.h>

// Оптимізований алгоритм для пошуку найкоротшого слова
void find_shortest_word(const char *sentence, char *shortest_word) {
    const char *start = sentence;
    const char *current = sentence;
    int min_length = 100;

    while (*current) {
        if (*current == ' ' || *(current + 1) == '\0') {
            int word_length = current - start + (*(current + 1) == '\0');
            if (word_length < min_length) {
                min_length = word_length;
                strncpy(shortest_word, start, word_length);
                shortest_word[word_length] = '\0';
            }
            start = current + 1;
        }
        current++;
    }
}

int main() {
    const char sentence[] = "C programming is versatile";
    char shortest_word[50];
    find_shortest_word(sentence, shortest_word);
    printf("Shortest word: %s\n", shortest_word);
    return 0;
}
```

Replace Magic Number with Symbolic Constant

Цей метод полягає у видаленні "магічних чисел" (чисел, значення яких незрозуміле з контексту) і заміні їх на константи з осмисленими іменами.

Коли застосовувати:

- Коли в коді використовуються числа, призначення яких неочевидне.
- Якщо число використовується багаторазово і його може знадобитися змінити в майбутньому.

Переваги:

- Збільшення зрозумілості коду.
- Зменшення помилок, пов'язаних із внесенням змін у кількох місцях.

Remove Middle Man

До рефакторингу:

```
#include <stdio.h>

// Перевірка віку для доступу до послуги
void check_access(int age) {
    if (age < 18) {
        printf("Access denied. Age below 18.\n");
    } else {
        printf("Access granted.\n");
    }
}

int main() {
    check_access(16);
    check_access(20);
    return 0;
}
```

Після рефакторингу:

```
#include <stdio.h>

#define MINIMUM_AGE 18 // Символічна константа

// Перевірка віку для доступу до послуги
void check_access(int age) {
    if (age < MINIMUM_AGE) {
        printf("Access denied. Age below %d.\n", MINIMUM_AGE);
    } else {
        printf("Access granted.\n");
    }
}

int main() {
    check_access(16);
    check_access(20);
    return 0;
}
```

Інструменти для рефакторингу

1. JetBrains IDE: Інструмент для автозаміни, перейменування та аналізу.
2. Visual Studio: Містить засоби для рефакторингу та перевірки якості.
3. Eclipse: Пропонує інструменти для роботи з великими проектами.

Висновки

1. Рефакторинг дозволяє підтримувати якість коду на високому рівні.
2. Застосування рефакторингу важливе для зменшення технічного боргу.
3. Методи, як-от Remove Middle Man, Substitute Algorithm, та Replace Magic Number with Symbolic Constant, роблять код більш читабельним і ефективним

Список використаних джерел

1. Фаулер Мартін. "Refactoring: Improving the Design of Existing Code".
2. Мартін Роберт. "Clean Code: A Handbook of Agile Software Craftsmanship".
3. Refactoring.Guru: https://refactoring.guru.
4. Документація JetBrains IDE: (https://www.jetbrains.com/help/idea/refactoring-source-code.html).