Міністерство освіти і науки України
Львівський національний університет імені Івана Франка
Факультет електроніки та комп'ютерних технологій

ЗВІТ

про виконання лабораторної роботи № 1
“Аналіз коду: виявлення проблемних місць”
з курсу “Аналіз та рефакторинг коду, моделювання та проектування ПЗ”

Виконав:
Студент. гр. ФеП-31 Фіняк Олег
Перевірив:
Притула М.М.

Львів – 2025

Призначення
Дублювання коду: випадки, коли ідентичний код з'являється в кількох місцях, що ускладнює обслуговування.
Надмірна складність: Занадто складний код, що ускладнює його розуміння та зміну.
Магічні числа: Жорстко закодовані значення, які не мають контексту, знижуючи читабельність коду та зручність обслуговування.
Великі класи: Класи, які беруть на себе занадто багато обов'язків, порушуючи принцип єдиної відповідальності.
Завдання

Крок 1: Ознайомтеся з вихідним кодом
Оберіть проект: Виберіть невеликий проект, наданий інструктором, або створіть свій. Якщо ви вирішите створити власний, переконайтеся, що він достатньо малий, щоб забезпечити всебічний аналіз у межах часових обмежень завдання.
Якщо ви використовуєте наданий проект, завантажте та налаштуйте його на своєму локальному комп'ютері, переконавшись, що він працює безперебійно.
Розуміти код: Знайдіть час, щоб прочитати код, щоб зрозуміти його структуру та функціональність. Робіть нотатки про ключові компоненти, такі як класи, функції та їх взаємодія.
Розгляньте можливість створення блок-схеми або діаграми для візуалізації взаємозв'язків між різними частинами коду. Це може допомогти ефективніше виявляти потенційні проблемні зони.

Крок 2: Виконайте статичний аналіз коду
Оберіть свій інструмент: Виберіть інструмент статичного аналізу коду, який підходить для мови програмування вашого проекту. Наприклад, використовуйте SonarQube для Java або ESLint для JavaScript.
Якщо ви не знайомі з цими інструментами, витратьте деякий час, щоб переглянути їхню документацію, щоб зрозуміти, як їх налаштувати та ефективно використовувати їхні функції.
Проведіть аналіз: Дотримуйтесь інструкцій для обраного вами інструменту, щоб запустити аналіз вашого проекту. Зазвичай це включає налаштування інструменту та виконання команди або натискання кнопки в інтерфейсі користувача. Приділіть пильну увагу виведенню, згенерованому інструментом, який висвітлить різні проблеми в коді.
Визначте основні проблеми: Уважно ознайомтеся з результатами аналізу. Складіть список основних проблем, виявлених інструментом, класифікувавши їх на основі типів запахів коду (наприклад, дублювання, складність).
Для кожної виявленої проблеми відзначайте конкретні рядки коду або компоненти, які є проблемними.

Крок 3: Поясніть шкідливі наслідки виявлених проблем
Досліджуйте та розмірковуйте: Щодо кожної виявленої проблеми досліджуйте, чому вона вважається шкідливою. Шукайте ресурси, які пояснюють вплив

запахів коду на розробку, обслуговування та продуктивність програмного забезпечення.
Поміркуйте над власним розумінням та досвідом, пов'язаним із цими питаннями кодування.
Напишіть пояснення: Для кожної проблеми складіть чітке і лаконічне пояснення її шкідливого впливу. Розгляньте можливість включення:
Потенційний вплив на читабельність і ремонтопридатність коду.
Як це може вплинути на майбутні зусилля з розробки або співпрацю з іншими розробниками.
Будь-які наслідки для продуктивності, які можуть виникнути внаслідок проблеми.

Крок 4: Впроваджуємо модульні тести для основного функціоналу програми
Напишіть мінімум 10 тестів, які оцінюють ключові техніки та сценарії.
Використовуйте фреймворк для тестування (наприклад, JUnit, pytest або подібний інструмент) для реалізації ваших тестів.

Виконання:
Обраний проект – це лабораторна робота з курсу “Дискретна математика” про булеві операції. До нього в ході аналізу були додані коментарі, що пояснюють код, функції та інше.

Щодо оптимізації та виправлення коду:
Було:
```
bool NOT(bool a)
{
	if (a == true ) return false;
	else return true;
}
bool AND(bool a, bool b)
{
	if (a == true && b == true) return true;
	else return false;
}

bool OR(bool a, bool b)
{
	if (a == true || b == true) return true;
	else return false;
}
bool IMP(bool a, bool b)
{
	if (a == true && b == false) return false;
	else return true;
}
bool EQU(bool a, bool b)
{
	if (a == b) return true;
	else return false;
}
bool XOR(bool a, bool b)
{
	if (a == b) return false;
	else return true;
}
bool x1(bool a, bool b)
```

```
{
	return XOR(NOT(a), NOT(b));
}
bool x2(bool a, bool b, bool c)
{
	return IMP(NOT(c), (XOR(NOT(a), NOT(b))));
}
bool F11(bool a, bool b, bool c) {
	return IMP(a, b) && (IMP(NOT(c), (XOR(NOT(a), NOT(b)))));
}
```

Стало:

```
// Функція заперечення (NOT)
// Повертає протилежне значення змінної a
bool NOT(bool a)
{
    if (a == true) return false;
    else return true;
}

// Функція кон'юнкції (AND)
// Повертає true, якщо обидва значення істинні (a && b)
bool AND(bool a, bool b) {
    return a && b;
}

// Функція диз'юнкції (OR)
// Повертає true, якщо хоча б одне значення істинне (a || b)
bool OR(bool a, bool b)
{
    return a || b;
}

// Функція імплікації (IMP)
// Повертає true, якщо з a випливає b (логічне слідування)
bool IMP(bool a, bool b)
{
    return !a || b;
}

// Функція еквівалентності (EQU)
// Повертає true, якщо a і b рівні (a == b)
bool EQU(bool a, bool b)
{
    return a == b;
}

// Функція виключного АБО (XOR)
// Повертає true, якщо a і b різні (a != b)
bool XOR(bool a, bool b)
{
    return a != b;
}

// Функція x1 (спеціальна комбінація логічних операцій)
// Використовує заперечення NOT та операцію XOR
bool x1(bool a, bool b)
{
    bool na = NOT(a); // Заперечення a
    bool nb = NOT(b); // Заперечення b
    return XOR(na, nb); // Виключне АБО між NOT(a) та NOT(b)
}

// Функція x2 (складніша логічна операція)
// Використовує імплікацію та виключне АБО з запереченнями
bool x2(bool a, bool b, bool c)
{
```

```
    return IMP(NOT(c), XOR(NOT(a), NOT(b)));
}

// Функція F11 (складна комбінація логічних операцій)
// Використовує імплікацію, виключне АБО та заперечення
bool F11(bool a, bool b, bool c) {
    bool part1 = IMP(a, b); // Імплікація a → b
    bool part2 = XOR(NOT(a), NOT(b)); // Виключне АБО між NOT(a) та NOT(b)
    bool part3 = IMP(NOT(c), part2); // Імплікація NOT(c) → part2
    return part1 && part3; // Кон'юнкція (AND) між part1 та part3
}
```

Тестування:

Код для тестування:

```
#include "LogCon.h"
#include <gtest/gtest.h>

// Тест для NOT
TEST(LogConTest, NotOperation) {
    EXPECT_EQ(NOT(true), false);
    EXPECT_EQ(NOT(false), true);
}

// Тест для AND
TEST(LogConTest, AndOperation) {
    EXPECT_EQ(AND(true, true), true);
    EXPECT_EQ(AND(true, false), false);
    EXPECT_EQ(AND(false, true), false);
    EXPECT_EQ(AND(false, false), false);
}

// Тест для OR
TEST(LogConTest, OrOperation) {
    EXPECT_EQ(OR(true, true), true);
    EXPECT_EQ(OR(true, false), true);
    EXPECT_EQ(OR(false, true), true);
    EXPECT_EQ(OR(false, false), false);
}

// Тест для IMP (імплікація)
TEST(LogConTest, ImplicationOperation) {
    EXPECT_EQ(IMP(true, true), true);
    EXPECT_EQ(IMP(true, false), false);
    EXPECT_EQ(IMP(false, true), true);
    EXPECT_EQ(IMP(false, false), true);
}

// Тест для EQU (еквівалентність)
TEST(LogConTest, EquivalenceOperation) {
    EXPECT_EQ(EQU(true, true), true);
    EXPECT_EQ(EQU(true, false), false);
    EXPECT_EQ(EQU(false, true), false);
    EXPECT_EQ(EQU(false, false), true);
}

// Тест для XOR (виключне АБО)
TEST(LogConTest, XorOperation) {
    EXPECT_EQ(XOR(true, true), false);
    EXPECT_EQ(XOR(true, false), true);
    EXPECT_EQ(XOR(false, true), true);
    EXPECT_EQ(XOR(false, false), false);
}

// Тест для x1 (XOR з NOT)
TEST(LogConTest, X1Operation) {
    EXPECT_EQ(x1(true, true), false);
    EXPECT_EQ(x1(true, false), true);
```

```
    EXPECT_EQ(x1(false, true), true);
    EXPECT_EQ(x1(false, false), false);
}

// Тест для x2 (комбінація XOR і NOT)
TEST(LogConTest, X2Operation) {
    EXPECT_EQ(x2(true, true, true), true);
    EXPECT_EQ(x2(true, true, false), false);
    EXPECT_EQ(x2(false, false, false), false);
    EXPECT_EQ(x2(false, false, true), true);
}

// Тест для F11 (функція трьох змінних)
TEST(LogConTest, F11Operation) {
    EXPECT_EQ(F11(true, true, true), true);
    EXPECT_EQ(F11(true, true, false), false);
    EXPECT_EQ(F11(false, false, true), false);
    EXPECT_EQ(F11(false, false, false), false);
}

// Граничний тест (повторення функцій)
TEST(LogConTest, BoundaryTest) {
    EXPECT_EQ(AND(true, OR(false, NOT(false))), true);
    EXPECT_EQ(IMP(false, AND(true, XOR(false, true))), true);
}
```

CMakeLists.txt

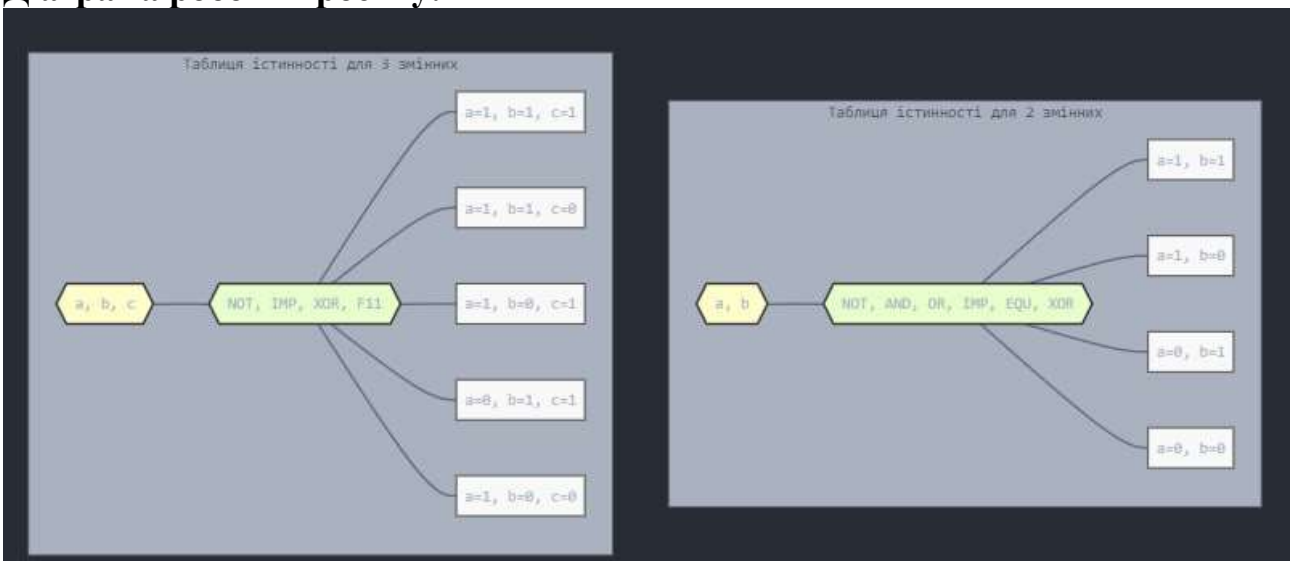
```
cmake_minimum_required(VERSION 3.10)
project(LogConTests)
# Використання C++17
set(CMAKE_CXX_STANDARD 17)
# Додаємо Google Test
enable_testing()
add_subdirectory(googletest)
# Основний вихідний код
add_library(LogCon LogCon.cpp)
# Файл з тестами
add_executable(runTests test_logcon.cpp)
target_link_libraries(runTests PRIVATE LogCon gtest gtest_main)
include(GoogleTest)
gtest_discover_tests(runTests)
```

Результати

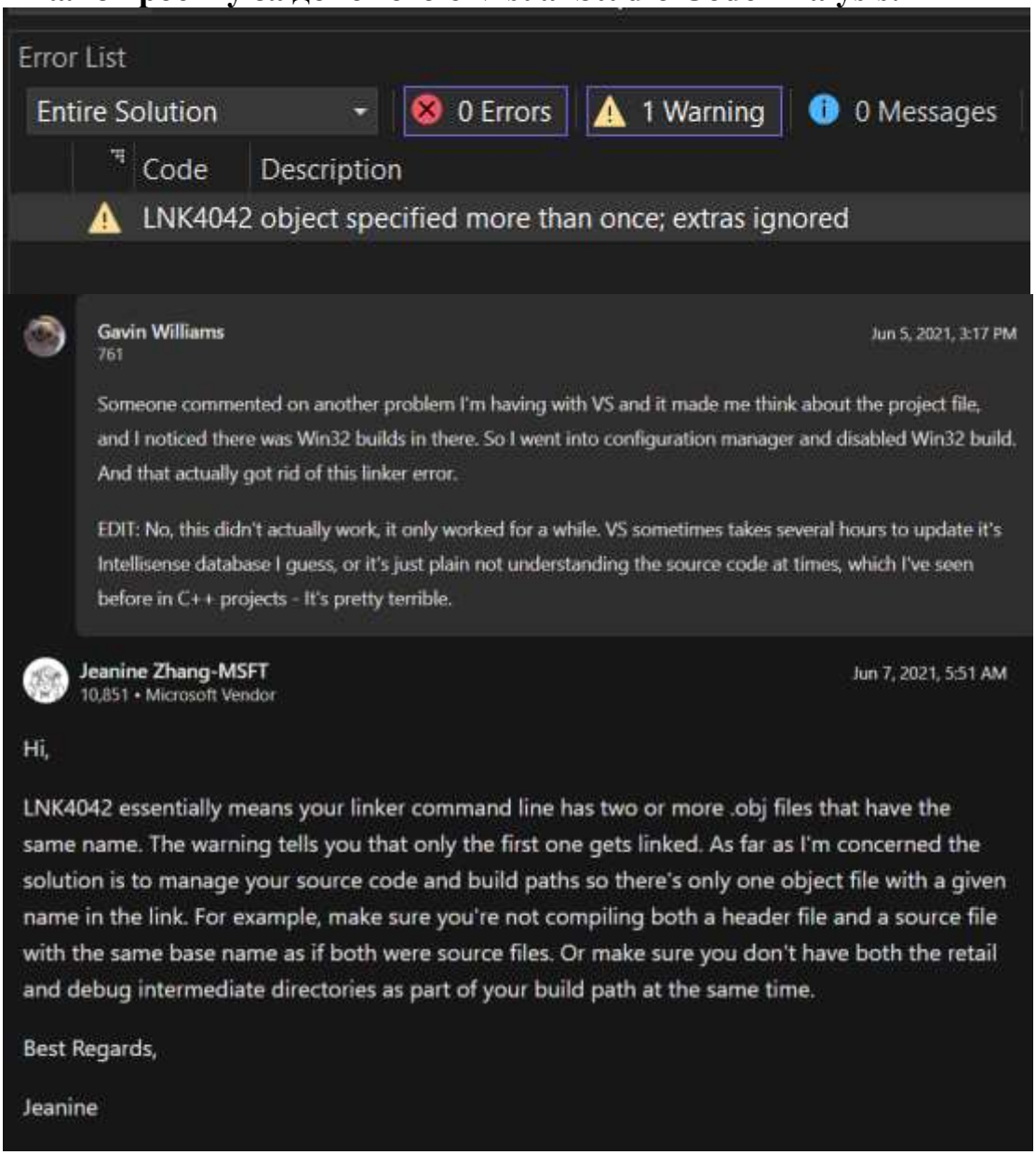
```
[==========] Running 10 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 10 tests from LogConTest
[ RUN      ] LogConTest.NotOperation
[       OK ] LogConTest.NotOperation (0 ms)
[ RUN      ] LogConTest.AndOperation
[       OK ] LogConTest.AndOperation (0 ms)
...
[ RUN      ] LogConTest.BoundaryTest
[       OK ] LogConTest.BoundaryTest (0 ms)
[----------] 10 tests from LogConTest (0 ms total)

[==========] 10 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 10 tests.
```

Діаграма роботи проекту:



Аналіз проекту за допомогою Visual Studio Code Analysis:



Висновок: в ході роботи я змінив всі функції, що використовувались у минулому коді, скоротивши їх, використавши логічні оператори, а не конструкції if else, також забрав дублювання коду у функціях, проаналізував код за допомогою Visual Studio Code Analysis та написав тести для цієї програми, інтегрувавши CMake та googletest.