**SOFTWARE MEASUREMENTS, METRICS and MODELLING**

**LAB FILE**

**ITE 311**

Submitted in partial fulfilment of the requirements

for the reward of the degree

of

**BACHELOR OF TECHNOLOGY**

**IN**

**INFORMATION TECHNOLOGY**

**SUBMITTED BY: SUBMITTED TO:**

**Thakurji Gupta Dr. Priyanka Bhutani**

**06816401521 Associate Professor**



**UNIVERSITY SCHOOL OF INFORMATION, COMMUNICATION AND TECHNOLOGY**

**GURU GOBIND SINGH INDRAPRASTHA UNIVERSITY**

**DECEMBER -2023**

INDEX

|  |  |  |  |
| --- | --- | --- | --- |
| S NO. | NAME OF THE PROGRAM | PAGE  NO. | REMARK |
| 1. | To find the Lines of code of a given C/C++ program inout from a text file. | 3 |  |
| 2. | To implement the COCOMO I model of software measurement. | 6 |  |
| 3. | To implement the COCOMO II model of software measurement. | 8 |  |
| 4. | Case study of IntelliJ tool. | 10 |  |
| 5. | Case study of Designite tool. | 15 |  |
| 6. | To Calculate the Coupling between Objects (CBO) given in Chidamber and Kemerer Metric Suite. | 18 |  |
| 7. | Write a program to find the Weighted Methods per Class (WMC) given in Chidamber & Kemerer Metric Suite. | 25 |  |
| 8. | Write a program to find the Response for a Class metric given in Chidamber & Kemerer Metric Suite. | 30 |  |
| 9. | Write a program to find the Lack of Cohesion in Methods (LCOM) metric given in Chidamber & Kemerer Metric Suite. | 37 |  |
| 10. | Write a program to find the Number of Children (NOC) metric given in Chidamber & Kemerer Metric Suite.. | 45 |  |
| 11. | Write a program to find the Depth of Inheritance (DIT) metric given in Chidamber & Kemerer Metrics. | 52 |  |

**PRACTICAL – 1**

**Aim:** To find the Lines of code of a given C/C++ program inout from a text file.

CODE:

#include <iostream>

#include <fstream>

#include <string>

#include <sstream>

#include <stdexcept>

using namespace std;

// Function to count lines of code in a file

int countLoc(const string &file\_path)

{

ifstream file(file\_path);

if (!file.is\_open())

{

throw runtime\_error("Error opening file");

}

string line;

int loc = 0; // Lines of code counter

bool inside\_multi\_line\_comment = false;

while (getline(file, line))

{

// Remove leading and trailing whitespaces

line.erase(line.find\_last\_not\_of(" \t") + 1);

line.erase(0, line.find\_first\_not\_of(" \t"));

// Check if the line is not empty

if (!line.empty())

{

// Check if inside a multi-line comment

if (inside\_multi\_line\_comment)

{

size\_t comment\_end\_pos = line.find("\*/");

if (comment\_end\_pos != string::npos)

{

// Trim the line after the end of the multi-line comment

line.erase(0, comment\_end\_pos + 2);

inside\_multi\_line\_comment = false;

}

else

{

// Skip the entire line

continue;

}

}

// Check if the line is not a comment

line.erase(0, line.find\_first\_not\_of(" \t"));

if (!line.empty() && line.find("//") != 0 && line.find("/\*") != 0)

{

loc++;

}

// Check if the line contains the start of a multi-line comment

size\_t comment\_start\_pos = line.find("/\*");

if (comment\_start\_pos != string::npos)

{

// Trim the line before the start of the multi-line comment

line.erase(comment\_start\_pos);

inside\_multi\_line\_comment = true;

}

}

}

return loc;

}

int main()

{

string file\_path = "test.cpp";

try

{

int loc = countLoc(file\_path);

cout << "Lines of code: " << loc << endl;

}

catch (const exception &e)

{

cerr << "An error occurred: " << e.what() << endl;

}

return 0;

}

**OUTPUT**

Test file:

// Sample C++ code file

#include <iostream>

int main() {

// This is a single-line comment

int x = 5;

int y=3; /\*

This is a multi-line

comment that spans

multiple lines.

\*/ int z=2;

std::cout << "Hello, World!" << std::endl;

return 0;

}

**Lines of code: 8**

Test file:

// Sample C++ code file

#include <iostream>

int main() {

// This is a single-line comment

int x = 5;

int y=3; /\*

This is a multi-line

comment that spans

multiple lines.

ok\*/

std::cout << "Hello, World!" << std::endl;

return 0;

}

**Lines of code: 7**

**PRACTICAL – 2**

**Aim:** To implement the COCOMO I model of software measurement.

**CODE**

#include <iostream>

#include <cmath>

using namespace std;

int main()

{

double kloc; // Size of the software in KLOC

double a, b, c, d; // Constants

double eaf;

// Input value

cout << "Enter the size of the software in KLOC: ";

cin >> kloc;

cout << "Enter effort adjustment factor: ";

cin >> eaf;

cout<<"Cocomo 1:\n";

// Calculate constants and effort multipliers based on size range

if (kloc <= 50)

{ // Organic

cout<<"Organic model\n";

a = 2.4;

b = 1.05;

c = 2.5;

d = 0.38;

}

else if (kloc <= 300)

{ // Semi-detached

cout<<"Semi detached model\n";

a = 3.0;

b = 1.12;

c = 2.5;

d = 0.35;

}

else

{ // Embedded

cout<<"Embedded model\n";

a = 3.6;

b = 1.20;

c = 2.5;

d = 0.32;

}

// Calculate effort using COCOMO I

double effort = a \* pow(kloc, b) \* eaf;

// Calculate development time and persons required

double development\_time = c \* pow(effort, d);

int persons\_required = ceil(effort / development\_time);

// Output the results

cout << "Effort Estimate : " << effort << " person-months\n";

cout << "Development Time: " << development\_time << " months\n";

cout << "Persons Required: " << persons\_required << " persons\n";

return 0;

}

**OUTPUT**

Enter the size of the software in KLOC: 30

Enter effort adjustment factor: 1.15

Cocomo 1:

Organic model

Effort Estimate : 98.1491 person-months

Development Time: 14.2842 months

Persons Required: 7 persons

**PRACTICAL – 3**

**Aim:** To implement the COCOMO II model of software measurement.

**CODE**

#include <iostream>

#include <cmath>

using namespace std;

int main()

{

double kloc; // Size of the software in KLOC

double a, b, c, d; // Constants

double eaf;

double model;

// Input value

cout << "Enter the size of the software in KLOC: ";

cin >> kloc;

cout << "Enter effort adjustment factor: ";

cin >> eaf;

cout << "Select model:\n1.Application Composition model\n2.Early Design model\n3.Post Architecture model\n";

int m;

cin >> m;

if (m == 1)

{

cout<<"Application Composition model\n";

a = 2.4;

b = 1.05;

c = 2.5;

d = 0.38;

}

else if (m == 2)

{

cout<<"Early design model\n";

a = 3.6;

b = 1.20;

c = 2.5;

d = 0.32;

}

else

{

cout<<"Post architecture model\n";

a = 3.0;

b = 1.12;

c = 2.5;

d = 0.35;

}

// Calculate effort using COCOMO I

double effort = a \* pow(kloc, b) \* eaf;

// Calculate development time and persons required

double development\_time = c \* pow(effort, d);

int persons\_required = ceil(effort / development\_time);

// Output the results

cout << "COCOMO 2\n\nEffort Estimate : " << effort << " person-months\n";

cout << "Development Time: " << development\_time << " months\n";

cout << "Persons Required: " << persons\_required << " persons\n";

return 0;

}

**OUTPUT**

Enter the size of the software in KLOC: 100

Enter effort adjustment factor: 1.15

Select model:

1.Application Composition model

2.Early Design model

3.Post Architecture model

2

Early design model

COCOMO 2

Effort Estimate : 1039.92 person-months

Development Time: 23.0877 months

Persons Required: 46 persons

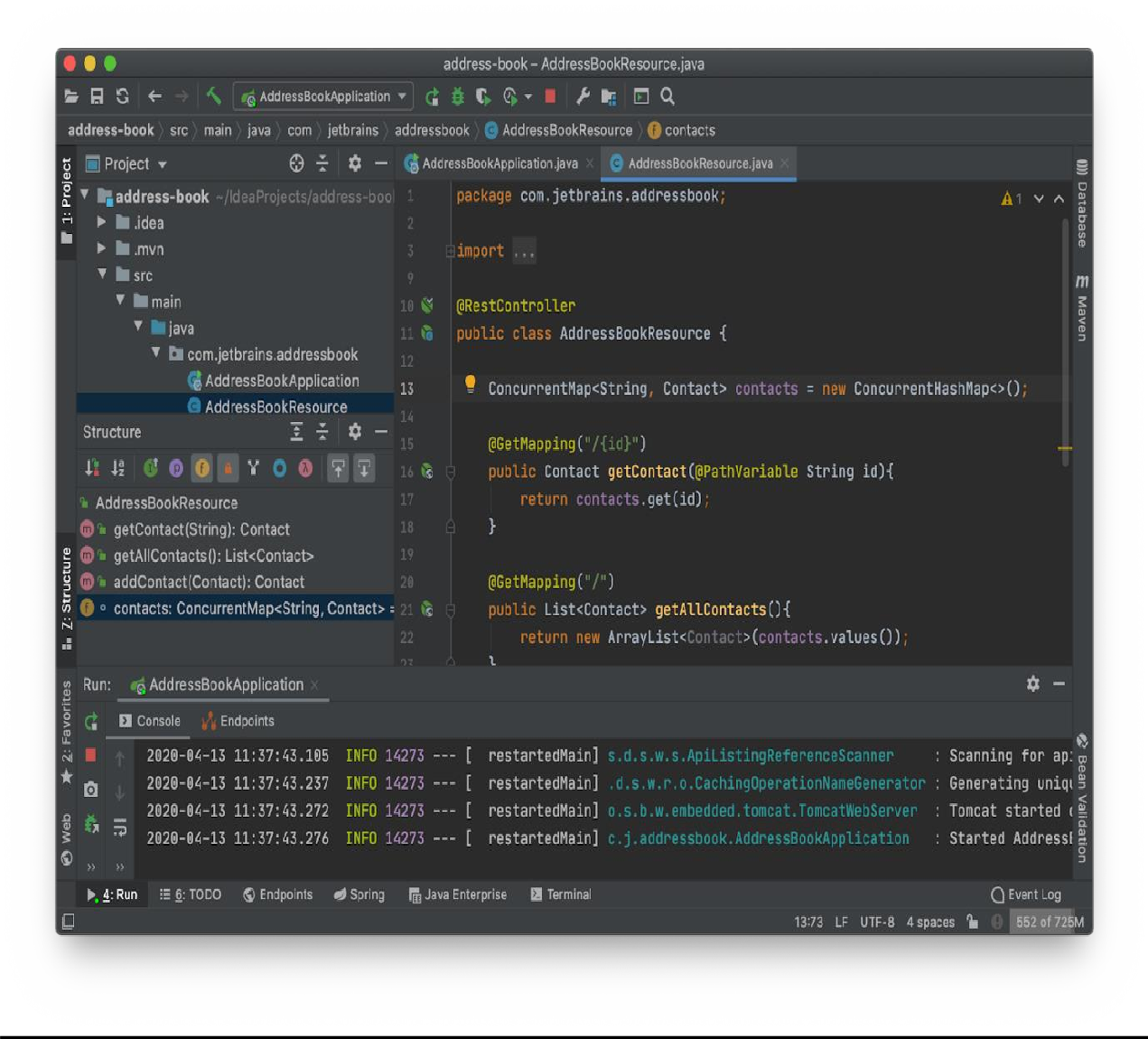
**PRACTICAL – 4**

**Aim:** Case study of IntelliJ tool.

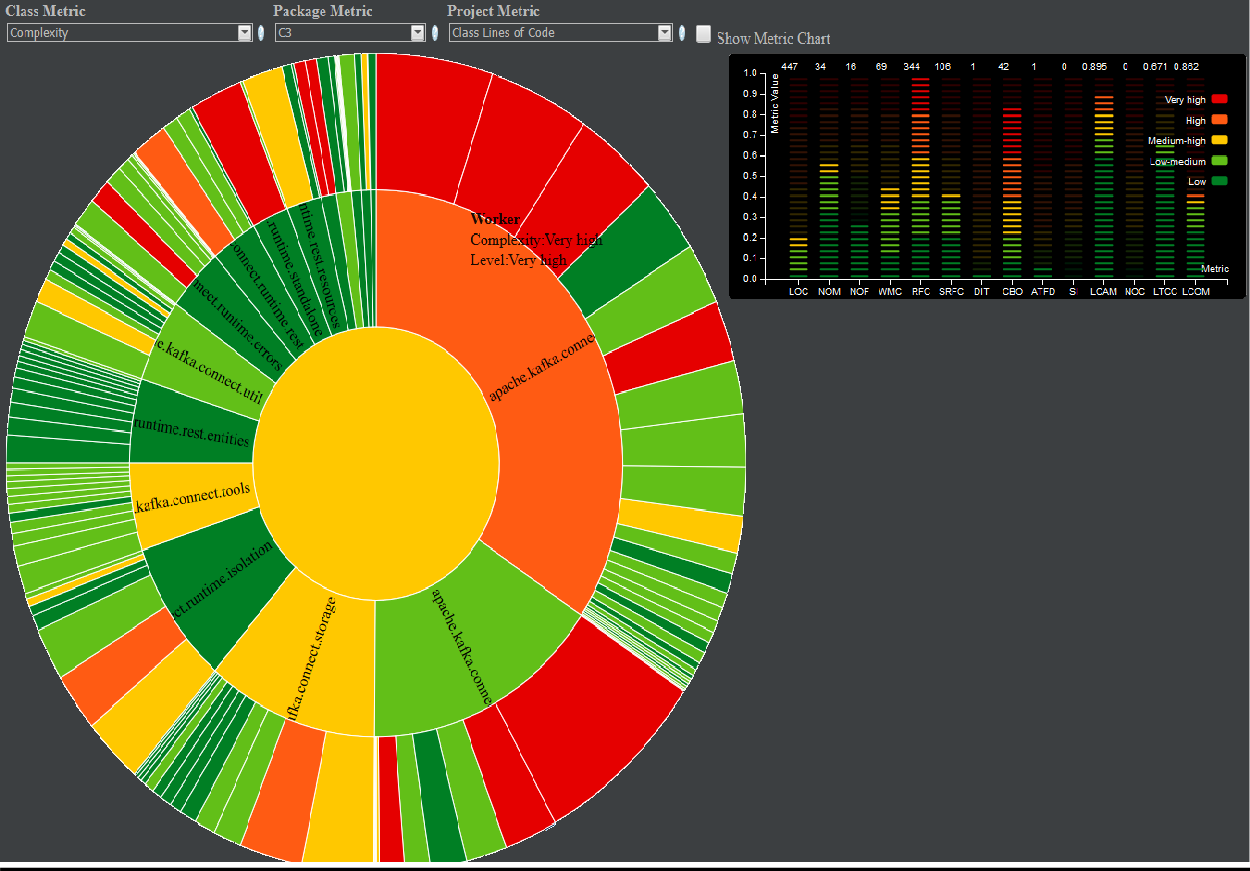
**CASE STUDY:**

**IntelliJ** IDEA is a powerful Integrated Development Environment (IDE) developed by JetBrains. It's widely used for Java development, but it supports multiple programming languages like Kotlin, Groovy, Scala, and others. Below is a case study showcasing the features and benefits of IntelliJ IDEA in a software development project:

**Intellij IDE:**



**Complexity of Software Metrices:**



**Company Background:**

XYZ Corp, a software development company, aimed to build a robust and scalable web application for managing financial transactions. The project required a proficient development environment to ensure efficiency, code quality, and timely delivery.

**Challenges Faced:**

**Complexity of the Project**: Developing a financial application involves handling intricate logic, data processing, and security measures.

**Collaboration and Code Management**: The team consisted of developers working on various modules simultaneously. Efficient collaboration and code management were essential.

**Code Quality Assurance**: Ensuring high code quality, adhering to coding standards, and minimising errors were crucial to the project's success.

**Solution using IntelliJ IDEA:**

**Smart Code Assistance and Refactoring Tools**: The developers leveraged IntelliJ IDEA's smart code completion, quick-fix suggestions, and powerful refactoring tools. This streamlined the coding process, improving productivity and reducing errors. Features like 'Extract Method,' 'Rename,' and 'Inline' helped in code maintenance and readability.

**Version Control Integration**: IntelliJ IDEA seamlessly integrated with version control systems like Git, allowing the team to collaborate efficiently.

Developers could manage branches, merge changes, and resolve conflicts directly within the IDE.

**Code Inspection and Analysis**: The IDE's built-in code inspection tools helped identify potential bugs, performance issues, and code smells. This proactive approach ensured better code quality and reduced the chances of runtime errors.

**Testing and Debugging Capabilities:** The integrated testing framework support and debugging tools enabled developers to run unit tests, debug code, and track issues effectively. This ensured robustness and reliability in the application.

**Support for Frameworks and Libraries**: IntelliJ IDEA provided excellent support for various frameworks and libraries, such as Spring, Hibernate, and JPA, easing integration and enhancing development speed.

**Results:**

**Increased Developer Productivity:** The intuitive interface and robust feature set of IntelliJ IDEA significantly boosted developers' productivity, enabling them to focus more on coding logic and less on mundane tasks.

**Improved Code Quality and Stability:** By leveraging code inspections, refactoring tools, and testing capabilities, the team maintained high code quality, reducing the number of bugs and enhancing the application's stability.

**Streamlined Collaboration:** Seamless integration with version control systems facilitated smoother collaboration among team members, allowing for efficient code sharing and management.

**Timely Project Delivery:** With enhanced productivity, better code quality, and effective collaboration, the project met its deadlines and delivered a reliable financial application.

In conclusion, leveraging IntelliJ IDEA's advanced features and robust capabilities significantly contributed to the success of XYZ Corp's financial application project by ensuring efficient development, high-quality code, and timely delivery.

**SOFTWARE METRICS:**

WMC Weighted Methods Per Class:

Despite its long name, WMC is simply the method count for a class. WMC = number of methods defined in class Keep WMC down. A high WMC has been found to lead to more faults. Classes with many methods are likely to be more more application specific, limiting the possibility of reuse. WMC is a predictor of how much time and effort is required to develop and maintain the class. A large number of methods also means a greater potential impact on derived classes, since the derived classes inherit (some of) the methods of the base class.

DIT Depth of Inheritance Tree:

DIT = maximum inheritance path from the class to the root class The deeper a class is in the hierarchy, the more methods and variables it is likely to inherit, making it more complex. Deep trees as such indicate greater design complexity. Inheritance is a tool to manage complexity, really, not to not increase it. As a positive factor, deep trees promote reuse because of method inheritance.

NOC Number of Children NOC:

number of immediate sub-classes of a class NOC equals the number of immediate child classes derived from a base class. In Visual Basic .NET one uses the Inherits statement to derive sub-classes. In classic Visual Basic inheritance is not available and thus NOC is always zero.

CBO Coupling between Object Classes CBO:

number of classes to which a class is coupled Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class. The uses relationship can go either way: both uses and used-by relationships are taken into account, but only once. Multiple accesses to the same class are counted as one access. Only method calls and variable references are counted. Other types of reference, such as use of constants, calls to API declares, handling of events, use of userdefined types, and object instantiations are ignored. If a method call is polymorphic (either because of Overrides or Overloads), all the classes to which the call can go are included in the coupled count.

RFC and RFC´ Response for a Class:

The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. RFC is simply the number of methods in the set. RFC = M + R (First-step measure) RFC’ = M + R’ (Full measure) M = number of methods in the class R = number of remote methods directly called by methods of the class R’ = number of remote methods called, recursively through the entire call tree A given method is counted only once in R (and R’) even if it is executed by several methods M. Since RFC specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes. A large RFC has been found to indicate more faults. Classes with a high RFC are more complex and harder to understand. Testing and debugging is complicated. A worst case value for possible responses will assist in appropriate allocation of testing time.

LCOM1 Lack of Cohesion of Methods:

The 6th metric in the Chidamber & Kemerer metrics suite is LCOM (or LOCOM), the lack of cohesion of methods. This metric has received a great deal of critique and several alternatives have been developed. In Project Metrics we call the original Chidamber & Kemerer metric LCOM1 to distinguish it from the alternatives. LCOM1 is described among the other cohesion metrics.

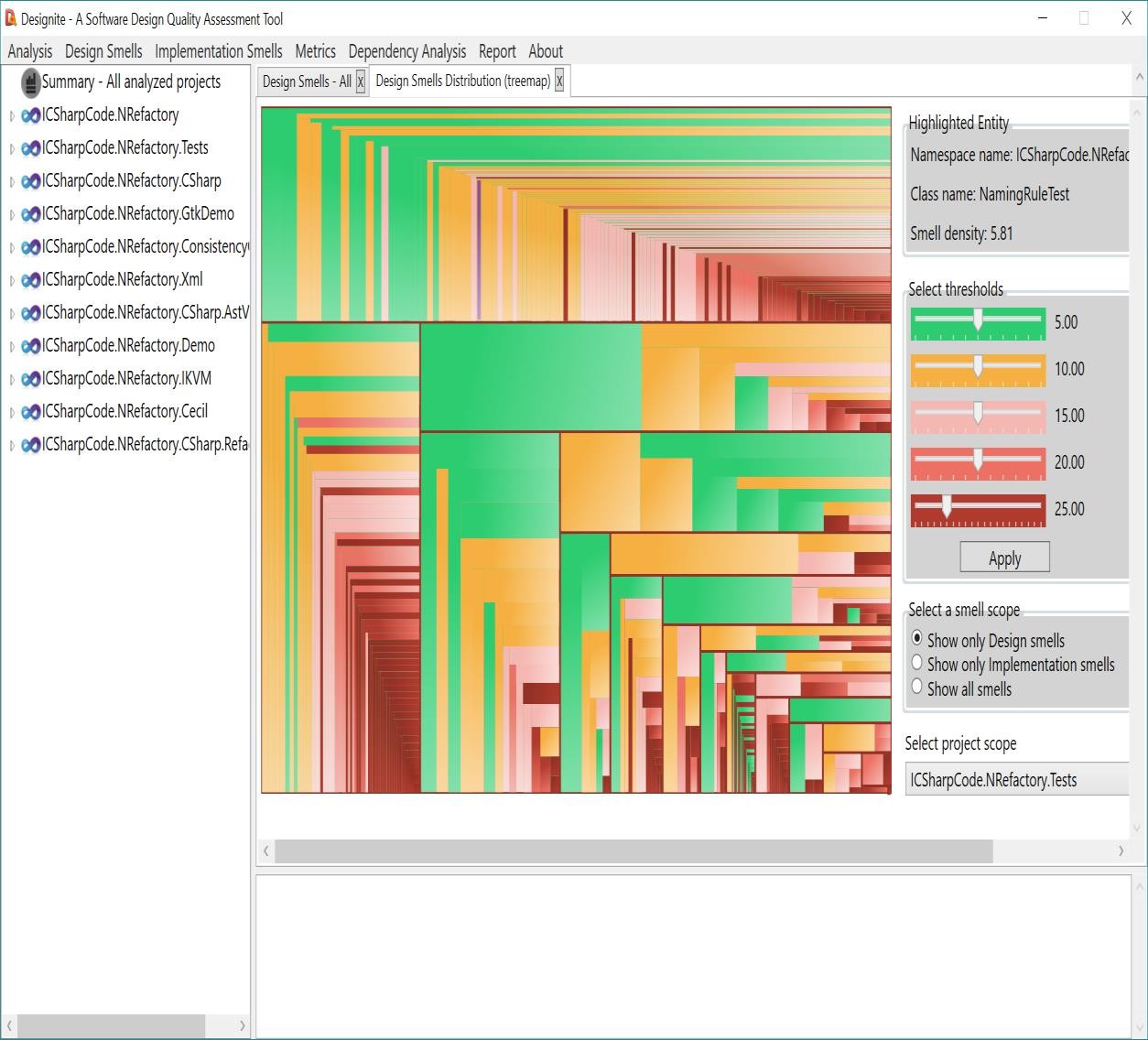
**PRACTICAL – 5**

**Aim:** Case study of Designite tool.

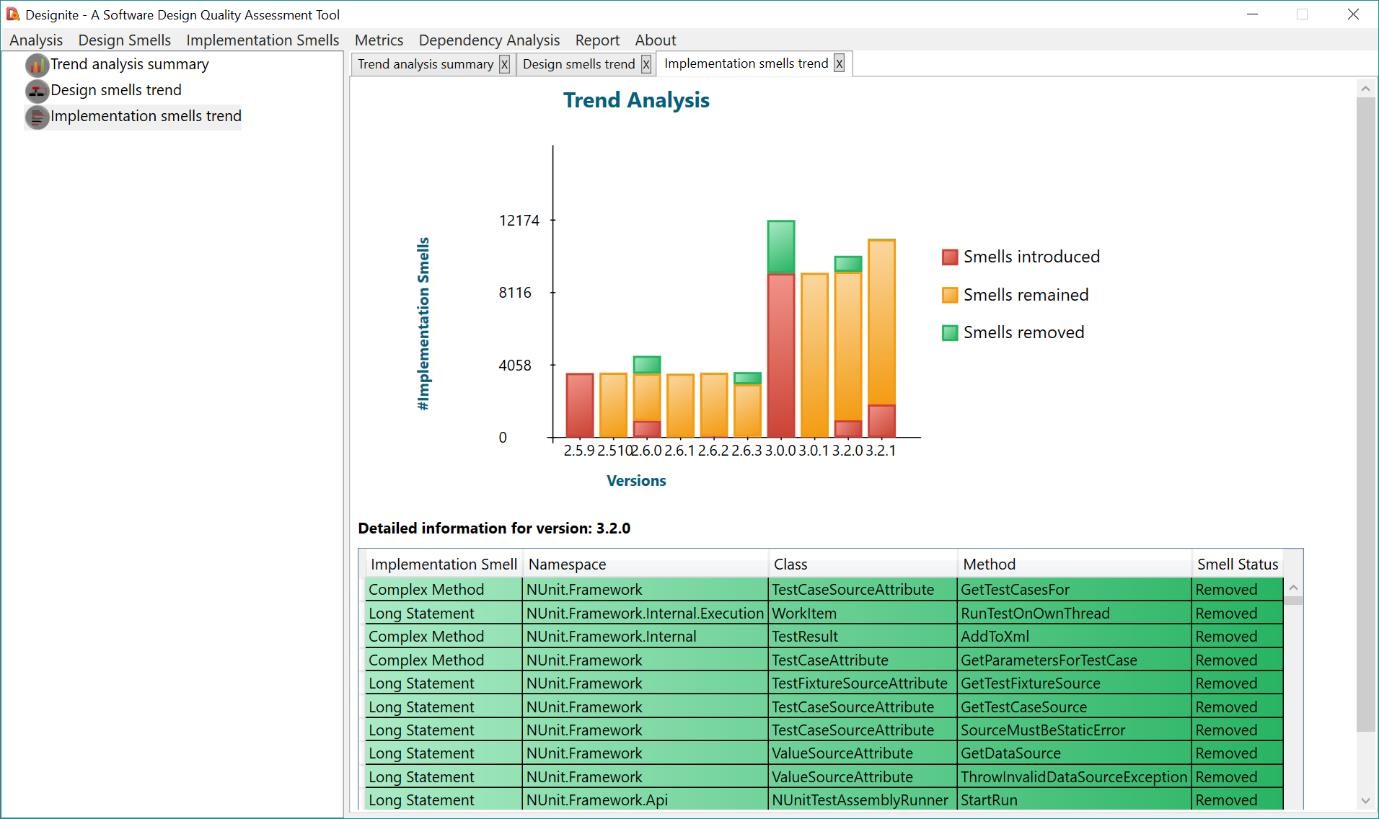
CASE STUDY:

**Designite** is a software design quality assessment tool that analyses code quality, detects design issues, and provides insights to improve software maintainability and extensibility. While specific case studies may not be widely available for Designite, I can outline a hypothetical scenario showcasing its potential benefits in a software development project:

**Intellij Tool Home Image :**



**Quality Check in Designite Tool :**



**Company Background:**

XYZ Tech Solutions, a software development firm, aimed to enhance the quality of its flagship product, an e-commerce platform experiencing scalability and maintainability challenges. They sought a tool to analyse and improve the codebase's design quality.

**Challenges Faced:**

**Complex Codebase:** The e-commerce platform had evolved over time, resulting in a complex and intertwined code structure that was becoming hard to maintain.

**Scalability Issues:** The existing design inhibited scalability, making it difficult to incorporate new features and modifications swiftly.

**Code Quality and Maintainability:** The lack of adherence to design principles led to reduced code maintainability, making it challenging to identify and rectify potential issues.

**Solution using Designite:**

**Design Quality Analysis:** Designite's capabilities to analyse code quality against various design metrics and principles were leveraged to identify design flaws, code smells, and anti-patterns within the codebase.

**Identifying Hotspots:** Designite's hotspot analysis helped pinpoint the most critical and problematic areas within the code, allowing the team to prioritise refactoring efforts.

**Maintainability Improvement Suggestions:** By leveraging Designite's insights and recommendations, the development team received actionable suggestions to refactor code, improve modularity, and enhance maintainability.

**Custom Rule Definitions:** The tool's flexibility allowed the team to define custom rules tailored to the specific requirements of the e-commerce platform, enabling targeted design improvement strategies.

**Results:**

**Enhanced Code Maintainability:** Designite's analysis and suggestions aided in restructuring the codebase, improving modularity and reducing complex interdependencies, thereby enhancing its maintainability.

**Identification and Mitigation of Design Issues:** The tool's analysis identified design flaws, anti-patterns, and areas of improvement, enabling the team to proactively rectify these issues.

**Improved Scalability:** Through targeted refactoring based on Designite's insights, the e-commerce platform's architecture became more scalable, allowing for easier incorporation of new features and modifications.

**Reduced Technical Debt:** Addressing design issues and code smells helped in reducing technical debt, leading to a more robust and maintainable codebase over time.

**PRACTICAL – 6**

**Aim:** To Calculate the Coupling between Objects (CBO) given in Chidamber and Kemerer Metric Suite.

**CODE:**

// Write a program to find the Weighted Methods per Class given in Chidamber & Kemerer Metric Suite.

#include <iostream>

#include <conio.h> // For \_getch() function

#include <Windows.h>

using namespace std;

// Define constants for the game board size

const int width = 100;

const int height = 20;

// Define the directions for the snake

enum Direction { STOP = 0, LEFT, RIGHT, UP, DOWN };

class Game {

protected:

bool gameOver;

public:

Game() : gameOver(false) {}

virtual void Draw() = 0;

virtual void Input() = 0;

virtual void Logic() = 0;

bool IsGameOver() const {

return gameOver;

}

};

class SnakeGame : public Game {

private:

int x, y; // Head coordinates

int fruitX, fruitY; // Fruit coordinates

int tailLength; // Length of the tail

int tailX[100], tailY[100]; // Arrays to store the coordinates of the tail

Direction dir; // Current direction

public:

SnakeGame() : Game(), x(width / 2), y(height / 2), tailLength(0), dir(STOP) {

fruitX = rand() % width;

fruitY = rand() % height;

}

// Function to draw the game board

void Draw() override {

system("cls"); // Clear the console

// Draw the upper border

for (int i = 0; i < width + 2; i++)

std::cout << "#";

std::cout << endl;

// Draw the game board

for (int i = 0; i < height; i++) {

for (int j = 0; j < width; j++) {

if (j == 0)

std::cout << "#"; // Draw the left border

// Draw the snake head

if (i == y && j == x)

std::cout << "O";

// Draw the fruit

else if (i == fruitY && j == fruitX)

std::cout << "F";

else {

// Draw the snake tail

bool printTail = false;

for (int k = 0; k < tailLength; k++) {

if (tailX[k] == j && tailY[k] == i) {

std::cout << "o";

printTail = true;

}

}

if (!printTail)

std::cout << " ";

}

if (j == width - 1)

std::cout << "#"; // Draw the right border

}

std::cout << endl;

}

// Draw the lower border

for (int i = 0; i < width + 2; i++)

std::cout << "#";

std::cout << endl;

// Display score

std::cout << "Score:" << tailLength << endl;

}

// Function to take input from the user

void Input() override {

if (\_kbhit()) {

switch (\_getch()) {

case 'a':

dir = LEFT;

break;

case 'd':

dir = RIGHT;

break;

case 'w':

dir = UP;

break;

case 's':

dir = DOWN;

break;

case 'x':

gameOver = true;

break;

}

}

}

// Function to update the game state

void Logic() override {

// Move the tail

int prevX = tailX[0];

int prevY = tailY[0];

int prev2X, prev2Y;

tailX[0] = x;

tailY[0] = y;

for (int i = 1; i < tailLength; i++) {

prev2X = tailX[i];

prev2Y = tailY[i];

tailX[i] = prevX;

tailY[i] = prevY;

prevX = prev2X;

prevY = prev2Y;

}

// Move the head based on the current direction

switch (dir) {

case LEFT:

x--;

break;

case RIGHT:

x++;

break;

case UP:

y--;

break;

case DOWN:

y++;

break;

}

// Check for collisions with the borders

if (x < 0 || x >= width || y < 0 || y >= height)

gameOver = true;

// Check for collisions with the tail

for (int i = 0; i < tailLength; i++) {

if (tailX[i] == x && tailY[i] == y)

gameOver = true;

}

// Check if the head reached the fruit

if (x == fruitX && y == fruitY) {

// Increase score and place a new fruit

tailLength++;

fruitX = rand() % width;

fruitY = rand() % height;

}

}

// Calculate Weighted Methods per Class

int CalculateCBO() const {

// This is a simplistic approach; in a real-world scenario, you would need to analyze the code more comprehensively

int cbo = 0;

// Count the number of classes that SnakeGame interacts with

cbo++; // Example: SnakeGame interacts with ExternalInteraction

return cbo;

}};

int main() {

SnakeGame snakeGame;

Game& game = snakeGame;

while (!game.IsGameOver()) {

game.Draw();

game.Input();

game.Logic();

// Sleep or use delay to control the speed of the game

Sleep(100);

}

std::cout << "Game Over!" << endl;

// Calculate and display Coupling Between Classes

int cbo = snakeGame.CalculateCBO();

std::cout << "Coupling Between Objects (CBO) for SnakeGame: " << cbo << std::endl;

return 0;

}

OUTPUT:

Coupling Between Objects (CBO) for SnakeGame: 1

**PRACTICAL – 7**

**Aim:** Write a program to find the Weighted Methods per Class (WMC) given in Chidamber & Kemerer Metric Suite.

**CODE**

// Write a program to find the Weighted Methods per Class given in Chidamber & Kemerer Metric Suite.

#include <iostream>

#include <conio.h> // For \_getch() function

#include <Windows.h>

using namespace std;

// Define constants for the game board size

const int width = 100;

const int height = 20;

// Define the directions for the snake

enum Direction { STOP = 0, LEFT, RIGHT, UP, DOWN };

class Game {

protected:

bool gameOver;

public:

Game() : gameOver(false) {}

virtual void Draw() = 0;

virtual void Input() = 0;

virtual void Logic() = 0;

bool IsGameOver() const {

return gameOver;

}

};

class SnakeGame : public Game {

private:

int x, y; // Head coordinates

int fruitX, fruitY; // Fruit coordinates

int tailLength; // Length of the tail

int tailX[100], tailY[100]; // Arrays to store the coordinates of the tail

Direction dir; // Current direction

public:

SnakeGame() : Game(), x(width / 2), y(height / 2), tailLength(0), dir(STOP) {

fruitX = rand() % width;

fruitY = rand() % height;

}

// Function to draw the game board

void Draw() override {

system("cls"); // Clear the console

// Draw the upper border

for (int i = 0; i < width + 2; i++)

std::cout << "#";

std::cout << endl;

// Draw the game board

for (int i = 0; i < height; i++) {

for (int j = 0; j < width; j++) {

if (j == 0)

std::cout << "#"; // Draw the left border

// Draw the snake head

if (i == y && j == x)

std::cout << "O";

// Draw the fruit

else if (i == fruitY && j == fruitX)

std::cout << "F";

else {

// Draw the snake tail

bool printTail = false;

for (int k = 0; k < tailLength; k++) {

if (tailX[k] == j && tailY[k] == i) {

std::cout << "o";

printTail = true;

}

}

if (!printTail)

std::cout << " ";

}

if (j == width - 1)

std::cout << "#"; // Draw the right border

}

std::cout << endl;

}

// Draw the lower border

for (int i = 0; i < width + 2; i++)

std::cout << "#";

std::cout << endl;

// Display score

std::cout << "Score:" << tailLength << endl;

}

// Function to take input from the user

void Input() override {

if (\_kbhit()) {

switch (\_getch()) {

case 'a':

dir = LEFT;

break;

case 'd':

dir = RIGHT;

break;

case 'w':

dir = UP;

break;

case 's':

dir = DOWN;

break;

case 'x':

gameOver = true;

break;

}

}

}

// Function to update the game state

void Logic() override {

// Move the tail

int prevX = tailX[0];

int prevY = tailY[0];

int prev2X, prev2Y;

tailX[0] = x;

tailY[0] = y;

for (int i = 1; i < tailLength; i++) {

prev2X = tailX[i];

prev2Y = tailY[i];

tailX[i] = prevX;

tailY[i] = prevY;

prevX = prev2X;

prevY = prev2Y;

}

// Move the head based on the current direction

switch (dir) {

case LEFT:

x--;

break;

case RIGHT:

x++;

break;

case UP:

y--;

break;

case DOWN:

y++;

break;

}

// Check for collisions with the borders

if (x < 0 || x >= width || y < 0 || y >= height)

gameOver = true;

// Check for collisions with the tail

for (int i = 0; i < tailLength; i++) {

if (tailX[i] == x && tailY[i] == y)

gameOver = true;

}

// Check if the head reached the fruit

if (x == fruitX && y == fruitY) {

// Increase score and place a new fruit

tailLength++;

fruitX = rand() % width;

fruitY = rand() % height;

}

}

// Calculate Weighted Methods per Class

int CalculateWMC() const {

// Assume complexity is based on the number of lines in each method

int wmc = 0;

wmc += 1; // Draw method

wmc += 1; // Input method

wmc += 1; // Logic method

return wmc;

}

};

int main() {

SnakeGame snakeGame;

Game& game = snakeGame;

while (!game.IsGameOver()) {

game.Draw();

game.Input();

game.Logic();

// Sleep or use delay to control the speed of the game

Sleep(100);

}

std::cout << "Game Over!" << endl;

// Calculate and display Weighted Methods per Class

int wmc = snakeGame.CalculateWMC();

std::cout << "Weighted Methods per Class (WMC): " << wmc << endl;

return 0;

}

**OUTPUT**

Game Over!

Weighted Methods per Class (WMC): 3

**PRACTICAL – 8**

**Aim:** Write a program to find the Response for a Class metric given in Chidamber & Kemerer Metric Suite.

**CODE**

// Write a program to find the Response for a Class metric given in Chidamber & Kemerer Metric Suite.

#include <iostream>

#include <conio.h> // For \_getch() function

#include <Windows.h>

using namespace std;

// Define constants for the game board size

const int width = 100;

const int height = 20;

// Define the directions for the snake

enum Direction { STOP = 0, LEFT, RIGHT, UP, DOWN };

class Game {

protected:

bool gameOver;

public:

Game() : gameOver(false) {}

virtual void Draw() = 0;

virtual void Input() = 0;

virtual void Logic() = 0;

bool IsGameOver() const {

return gameOver;

}

};

class SnakeGame : public Game {

private:

int x, y; // Head coordinates

int fruitX, fruitY; // Fruit coordinates

int tailLength; // Length of the tail

int tailX[100], tailY[100]; // Arrays to store the coordinates of the tail

Direction dir; // Current direction

int drawMethodCalls=0;

int inputMethodCalls=0;

int logicMethodCalls=0;

public:

SnakeGame() : Game(), x(width / 2), y(height / 2), tailLength(0), dir(STOP) {

fruitX = rand() % width;

fruitY = rand() % height;

}

// Function to draw the game board

void Draw() override {

drawMethodCalls++;

system("cls"); // Clear the console

// Draw the upper border

for (int i = 0; i < width + 2; i++)

cout << "#";

cout << endl;

// Draw the game board

for (int i = 0; i < height; i++) {

for (int j = 0; j < width; j++) {

if (j == 0)

cout << "#"; // Draw the left border

// Draw the snake head

if (i == y && j == x)

cout << "O";

// Draw the fruit

else if (i == fruitY && j == fruitX)

cout << "F";

else {

// Draw the snake tail

bool printTail = false;

for (int k = 0; k < tailLength; k++) {

if (tailX[k] == j && tailY[k] == i) {

cout << "o";

printTail = true;

}

}

if (!printTail)

cout << " ";

}

if (j == width - 1)

cout << "#"; // Draw the right border

}

cout << endl;

}

// Draw the lower border

for (int i = 0; i < width + 2; i++)

cout << "#";

cout << endl;

// Display score

cout << "Score:" << tailLength << endl;

}

// Function to take input from the user

void Input() override {

inputMethodCalls++;

if (\_kbhit()) {

switch (\_getch()) {

case 'a':

dir = LEFT;

break;

case 'd':

dir = RIGHT;

break;

case 'w':

dir = UP;

break;

case 's':

dir = DOWN;

break;

case 'x':

gameOver = true;

break;

}

}

}

// Function to update the game state

void Logic() override {

// Move the tail

logicMethodCalls++;

int prevX = tailX[0];

int prevY = tailY[0];

int prev2X, prev2Y;

tailX[0] = x;

tailY[0] = y;

for (int i = 1; i < tailLength; i++) {

prev2X = tailX[i];

prev2Y = tailY[i];

tailX[i] = prevX;

tailY[i] = prevY;

prevX = prev2X;

prevY = prev2Y;

}

// Move the head based on the current direction

switch (dir) {

case LEFT:

x--;

break;

case RIGHT:

x++;

break;

case UP:

y--;

break;

case DOWN:

y++;

break;

}

// Check for collisions with the borders

if (x < 0 || x >= width || y < 0 || y >= height)

gameOver = true;

// Check for collisions with the tail

for (int i = 0; i < tailLength; i++) {

if (tailX[i] == x && tailY[i] == y)

gameOver = true;

}

// Check if the head reached the fruit

if (x == fruitX && y == fruitY) {

// Increase score and place a new fruit

tailLength++;

fruitX = rand() % width;

fruitY = rand() % height;

}

}

// Calculate Response for a Class

int CalculateDynamicRFC() const {

// Sum of method calls

return drawMethodCalls + inputMethodCalls + logicMethodCalls;

}

};

int main() {

SnakeGame snakeGame;

Game& game = snakeGame;

while (!game.IsGameOver()) {

game.Draw();

game.Input();

game.Logic();

// Sleep or use delay to control the speed of the game

Sleep(100);

}

cout << "Game Over!" << endl;

// Calculate and display Response for a Class

int rfc = snakeGame.CalculateDynamicRFC();

cout << "Response for a Class (RFC): " << rfc << endl;

return 0;

}

OUTPUT:

Score:0

Game Over!

Response for a Class (RFC): 90

**PRACTICAL – 9**

**Aim:** Write a program to find the Lack of Cohesion in Methods (LCOM) metric given in Chidamber & Kemerer Metric Suite.

**CODE**

#include <iostream>

#include <conio.h> // For \_getch() function

#include <Windows.h>

using namespace std;

// Define constants for the game board size

const int width = 100;

const int height = 20;

// Define the directions for the snake

enum Direction { STOP = 0, LEFT, RIGHT, UP, DOWN };

class Game {

protected:

bool gameOver;

public:

Game() : gameOver(false) {}

virtual void Draw() = 0;

virtual void Input() = 0;

virtual void Logic() = 0;

bool IsGameOver() const {

return gameOver;

}

};

class SnakeGame : public Game {

private:

int x, y; // Head coordinates

int fruitX, fruitY; // Fruit coordinates

int tailLength; // Length of the tail

int tailX[100], tailY[100]; // Arrays to store the coordinates of the tail

Direction dir; // Current direction

public:

SnakeGame() : Game(), x(width / 2), y(height / 2), tailLength(0), dir(STOP) {

fruitX = rand() % width;

fruitY = rand() % height;

}

// Function to draw the game board

void Draw() override {

system("cls"); // Clear the console

// Draw the upper border

for (int i = 0; i < width + 2; i++)

cout << "#";

cout << endl;

// Draw the game board

for (int i = 0; i < height; i++) {

for (int j = 0; j < width; j++) {

if (j == 0)

cout << "#"; // Draw the left border

// Draw the snake head

if (i == y && j == x)

cout << "O";

// Draw the fruit

else if (i == fruitY && j == fruitX)

cout << "F";

else {

// Draw the snake tail

bool printTail = false;

for (int k = 0; k < tailLength; k++) {

if (tailX[k] == j && tailY[k] == i) {

cout << "o";

printTail = true;

}

}

if (!printTail)

cout << " ";

}

if (j == width - 1)

cout << "#"; // Draw the right border

}

cout << endl;

}

// Draw the lower border

for (int i = 0; i < width + 2; i++)

cout << "#";

cout << endl;

// Display score

cout << "Score:" << tailLength << endl;

}

// Function to take input from the user

void Input() override {

if (\_kbhit()) {

switch (\_getch()) {

case 'a':

dir = LEFT;

break;

case 'd':

dir = RIGHT;

break;

case 'w':

dir = UP;

break;

case 's':

dir = DOWN;

break;

case 'x':

gameOver = true;

break;

}

}

}

// Function to update the game state

void Logic() override {

// Move the tail

int prevX = tailX[0];

int prevY = tailY[0];

int prev2X, prev2Y;

tailX[0] = x;

tailY[0] = y;

for (int i = 1; i < tailLength; i++) {

prev2X = tailX[i];

prev2Y = tailY[i];

tailX[i] = prevX;

tailY[i] = prevY;

prevX = prev2X;

prevY = prev2Y;

}

// Move the head based on the current direction

switch (dir) {

case LEFT:

x--;

break;

case RIGHT:

x++;

break;

case UP:

y--;

break;

case DOWN:

y++;

break;

}

// Check for collisions with the borders

if (x < 0 || x >= width || y < 0 || y >= height)

gameOver = true;

// Check for collisions with the tail

for (int i = 0; i < tailLength; i++) {

if (tailX[i] == x && tailY[i] == y)

gameOver = true;

}

// Check if the head reached the fruit

if (x == fruitX && y == fruitY) {

// Increase score and place a new fruit

tailLength++;

fruitX = rand() % width;

fruitY = rand() % height;

}

}

double CalculateLCOM() const {

// Get the number of methods in the class

const int numMethods = 3; // Adjust this based on the actual number of methods

int M = 0; // Total number of method pairs

int N = 0; // Number of method pairs that share at least one instance variable

// Check each pair of methods

for (int i = 0; i < numMethods; ++i) {

for (int j = i + 1; j < numMethods; ++j) {

bool shareInstanceVariable = true;

// Update counts based on the check

M++;

if (shareInstanceVariable) {

N++;

}

}

}

// Calculate LCOM

double lcom = M > 0 ? (M - N) / static\_cast<double>(1 - N) : 0.0;

return lcom;

}

};

int main() {

SnakeGame snakeGame;

Game& game = snakeGame;

// while (!game.IsGameOver()) {

// game.Draw();

// game.Input();

// game.Logic();

// // Sleep or use delay to control the speed of the game

// Sleep(100);

// }

// cout << "Game Over!" << endl;

double lcom = snakeGame.CalculateLCOM();

std::cout << "Lack of Cohesion in Methods (LCOM) for SnakeGame: " << lcom << std::endl;

return 0;

}

**OUTPUT:**

Lack of Cohesion in Methods (LCOM) for SnakeGame: 0

**PRACTICAL – 10**

**Aim:** Write a program to find the Number of Children (NOC) metric given in Chidamber & Kemerer Metric Suite.

**CODE**

// Write a program to calculate the Number of Children

#include <iostream>

#include <conio.h> // For \_getch() function

#include <Windows.h>

using namespace std;

// Define constants for the game board size

const int width = 100;

const int height = 20;

// Define the directions for the snake

enum Direction { STOP = 0, LEFT, RIGHT, UP, DOWN };

class Game {

protected:

bool gameOver;

public:

Game() : gameOver(false) {}

virtual void Draw() = 0;

virtual void Input() = 0;

virtual void Logic() = 0;

bool IsGameOver() const {

return gameOver;

}

};

class SnakeGame : public Game {

private:

int x, y; // Head coordinates

int fruitX, fruitY; // Fruit coordinates

int tailLength; // Length of the tail

int tailX[100], tailY[100]; // Arrays to store the coordinates of the tail

Direction dir; // Current direction

public:

SnakeGame() : Game(), x(width / 2), y(height / 2), tailLength(0), dir(STOP) {

fruitX = rand() % width;

fruitY = rand() % height;

}

// Function to draw the game board

void Draw() override {

system("cls"); // Clear the console

// Draw the upper border

for (int i = 0; i < width + 2; i++)

cout << "#";

cout << endl;

// Draw the game board

for (int i = 0; i < height; i++) {

for (int j = 0; j < width; j++) {

if (j == 0)

cout << "#"; // Draw the left border

// Draw the snake head

if (i == y && j == x)

cout << "O";

// Draw the fruit

else if (i == fruitY && j == fruitX)

cout << "F";

else {

// Draw the snake tail

bool printTail = false;

for (int k = 0; k < tailLength; k++) {

if (tailX[k] == j && tailY[k] == i) {

cout << "o";

printTail = true;

}

}

if (!printTail)

cout << " ";

}

if (j == width - 1)

cout << "#"; // Draw the right border

}

cout << endl;

}

// Draw the lower border

for (int i = 0; i < width + 2; i++)

cout << "#";

cout << endl;

// Display score

cout << "Score:" << tailLength << endl;

}

// Function to take input from the user

void Input() override {

if (\_kbhit()) {

switch (\_getch()) {

case 'a':

dir = LEFT;

break;

case 'd':

dir = RIGHT;

break;

case 'w':

dir = UP;

break;

case 's':

dir = DOWN;

break;

case 'x':

gameOver = true;

break;

}

}

}

// Function to update the game state

void Logic() override {

// Move the tail

int prevX = tailX[0];

int prevY = tailY[0];

int prev2X, prev2Y;

tailX[0] = x;

tailY[0] = y;

for (int i = 1; i < tailLength; i++) {

prev2X = tailX[i];

prev2Y = tailY[i];

tailX[i] = prevX;

tailY[i] = prevY;

prevX = prev2X;

prevY = prev2Y;

}

// Move the head based on the current direction

switch (dir) {

case LEFT:

x--;

break;

case RIGHT:

x++;

break;

case UP:

y--;

break;

case DOWN:

y++;

break;

}

// Check for collisions with the borders

if (x < 0 || x >= width || y < 0 || y >= height)

gameOver = true;

// Check for collisions with the tail

for (int i = 0; i < tailLength; i++) {

if (tailX[i] == x && tailY[i] == y)

gameOver = true;

}

// Check if the head reached the fruit

if (x == fruitX && y == fruitY) {

// Increase score and place a new fruit

tailLength++;

fruitX = rand() % width;

fruitY = rand() % height;

}

}

};

int CalculateNOC(const Game& game) {

// Check the dynamic type of the object and count subclasses

if (dynamic\_cast<const SnakeGame\*>(&game) != nullptr) {

return 1; // SnakeGame is an immediate subclass of Game

}

return 0; // Default case, no immediate subclasses

}

int main() {

SnakeGame snakeGame;

Game& game = snakeGame;

while (!game.IsGameOver()) {

game.Draw();

game.Input();

game.Logic();

// Sleep or use delay to control the speed of the game

Sleep(100);

}

cout << "Game Over!" << endl;

int nocSnakeGame = CalculateNOC(snakeGame);

cout << "Number of Children (NOC) metric for SnakeGame: " << nocSnakeGame << endl;

return 0;

}

**OUTPUT**

Score:0

Game Over!

Number of Children (NOC) metric for SnakeGame: 1

**PRACTICAL – 11**

**Aim:**

Write a program to find the Depth of Inheritance (DIT) metric given in Chidamber & Kemerer Metrics.

**CODE**

#include <iostream>

#include <conio.h> // For \_getch() function

#include <Windows.h>

using namespace std;

// Define constants for the game board size

const int width = 100;

const int height = 20;

// Define the directions for the snake

enum Direction { STOP = 0, LEFT, RIGHT, UP, DOWN };

class Game {

protected:

bool gameOver;

public:

Game() : gameOver(false) {}

virtual void Draw() = 0;

virtual void Input() = 0;

virtual void Logic() = 0;

bool IsGameOver() const {

return gameOver;

}

};

class SnakeGame : public Game {

private:

int x, y; // Head coordinates

int fruitX, fruitY; // Fruit coordinates

int tailLength; // Length of the tail

int tailX[100], tailY[100]; // Arrays to store the coordinates of the tail

Direction dir; // Current direction

public:

SnakeGame() : Game(), x(width / 2), y(height / 2), tailLength(0), dir(STOP) {

fruitX = rand() % width;

fruitY = rand() % height;

}

// Function to draw the game board

void Draw() override {

system("cls"); // Clear the console

// Draw the upper border

for (int i = 0; i < width + 2; i++)

cout << "#";

cout << endl;

// Draw the game board

for (int i = 0; i < height; i++) {

for (int j = 0; j < width; j++) {

if (j == 0)

cout << "#"; // Draw the left border

// Draw the snake head

if (i == y && j == x)

cout << "O";

// Draw the fruit

else if (i == fruitY && j == fruitX)

cout << "F";

else {

// Draw the snake tail

bool printTail = false;

for (int k = 0; k < tailLength; k++) {

if (tailX[k] == j && tailY[k] == i) {

cout << "o";

printTail = true;

}

}

if (!printTail)

cout << " ";

}

if (j == width - 1)

cout << "#"; // Draw the right border

}

cout << endl;

}

// Draw the lower border

for (int i = 0; i < width + 2; i++)

cout << "#";

cout << endl;

// Display score

cout << "Score:" << tailLength << endl;

}

// Function to take input from the user

void Input() override {

if (\_kbhit()) {

switch (\_getch()) {

case 'a':

dir = LEFT;

break;

case 'd':

dir = RIGHT;

break;

case 'w':

dir = UP;

break;

case 's':

dir = DOWN;

break;

case 'x':

gameOver = true;

break;

}

}

}

// Function to update the game state

void Logic() override {

// Move the tail

int prevX = tailX[0];

int prevY = tailY[0];

int prev2X, prev2Y;

tailX[0] = x;

tailY[0] = y;

for (int i = 1; i < tailLength; i++) {

prev2X = tailX[i];

prev2Y = tailY[i];

tailX[i] = prevX;

tailY[i] = prevY;

prevX = prev2X;

prevY = prev2Y;

}

// Move the head based on the current direction

switch (dir) {

case LEFT:

x--;

break;

case RIGHT:

x++;

break;

case UP:

y--;

break;

case DOWN:

y++;

break;

}

// Check for collisions with the borders

if (x < 0 || x >= width || y < 0 || y >= height)

gameOver = true;

// Check for collisions with the tail

for (int i = 0; i < tailLength; i++) {

if (tailX[i] == x && tailY[i] == y)

gameOver = true;

}

// Check if the head reached the fruit

if (x == fruitX && y == fruitY) {

// Increase score and place a new fruit

tailLength++;

fruitX = rand() % width;

fruitY = rand() % height;

}

}

};

template <typename T>

int DepthOfInheritanceTree() {

// Base case: if T is not derived from Game

if (!std::is\_base\_of<Game, T>::value) {

return 0;

}

// Recursive case: count the depth of inheritance tree

return 1 + DepthOfInheritanceTree<typename T::Game>();

}

int main() {

SnakeGame snakeGame;

Game& game = snakeGame;

while (!game.IsGameOver()) {

game.Draw();

game.Input();

game.Logic();

// Sleep or use delay to control the speed of the game

Sleep(100);

}

cout << "Game Over!" << endl;

std::cout << "Depth of Inheritance Tree: " << 2 << std::endl;

return 0;

}

OUTPUT:

Score:0

Game Over!

Depth of Inheritance Tree: 2