UNIVERSIDAD DE LAS FUERZAS ARMADAS-ESPE

SEDE SANTO DOMINGO DE LOS TSÁCHILAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN - DCCO-SS

CARRERA DE INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN

PERIODO :

ASIGNATURA : Programación Orientada a Objetos

TEMA : Actividad sobre el Capítulo 2: “Nombres Significativos” y Capitulo 3: “Funciones”--Libro: Código Limpio de Robert C. Martin

ESTUDIANTE : Angel Steven Rodriguez Chavez

NIVEL-PARALELO - NRC: Segundo B NRC: 23069

DOCENTE : Ing. Jhon Cruz

FECHA DE ENTREGA : 30/04/2025

SANTO DOMINGO – ECUADOR

I. Introducción

El libro *Código Limpio: Un manual de artesanía del software* de Robert C. Martin es igualmente aplicable a C++. Este informe adapta los principios de los capítulos 2 y 3 al lenguaje C++, enfatizando:

- **Nombres significativos**: Uso de identificadores claros en variables, funciones y clases.
- **Funciones bien diseñadas**: Funciones cortas con un único propósito, usando características de C++ como parámetros por referencia y tipos fuertes.

II. 2. Objetivos:

2.1. Objetivo General:

Mejorar la calidad del código C++ aplicando principios de código limpio.

2.2. Objetivos Específicos:

- Renombrar correctamente los identificadores.
- Dividir funciones grandes en funciones pequeñas y con una sola responsabilidad.

III. 3. Desarrollo / Marco Teórico/ Práctica

Codigo inicial:

```cpp
#include <iostream>

class B {                       // ¿B de qué?
    int a[10];                  // ¿qué es "a"?
    int i = 0;                  // índice… ¿"i" de qué?
public:
    void x(int v) {             // "x" agrega… ¿qué?  "v" tampoco dice mucho
        a[i++] = v;
    }

    int y() {                   // "y" calcula la suma
        int s = 0;              // "s"… ¿suma? ¿salario?
        for (int j = 0; j < i; ++j) { // "j" → contador, pero poco claro
            s += a[j];
        }
        return s;
    }

    double z() {                // "z" devuelve el promedio
        return i == 0 ? 0.0 : static_cast<double>(y()) / i;
    }
};
```

```cpp
int main() {
    B p;                        // Objeto "p"… ¿de qué?
    int t;                      // "t" → ¿total? no queda claro

    std::cout << "¿Cuántos números ingresará? ";
    std::cin  >> t;

    for (int k = 0; k < t; ++k) { // "k" → contador genérico
        std::cout << "Número: ";
        int d;                  // "d" → ¿dato? poco informativo
        std::cin >> d;
        p.x(d);
    }

    std::cout << "Suma = "      << p.y() << '\n';
    std::cout << "Promedio = "  << p.z() << '\n';
    return 0;
}
```

El Codigo se analizo correctamente y se aplico cada regla del capitulo 2 “Nombres Significativos”

Nº	Identificador problemático	Regla del libro que viola	Propuesta de nombre claro y justificación
1	BadNames	Nombre de clase no comunica propósito	NumberStatistics – Indica que es una clase para estadísticas numéricas.
2	n	Nombre críptico	numbers – Clarifica que es un arreglo de números.
3	i	Nombre vago, ambigüo	count o currentIndex – Aclara que representa la cantidad actual de elementos.
4	a	Nombre de método sin intención	addNumber(int number) – Muestra claramente que agrega un número.
5	v	Parámetro sin sentido	number – Explica qué es lo que se recibe.
6	b	Nombre de método sin contexto	getSum() – Informa que devuelve la suma de los números.
7	s	Variable sin claridad	sum – Clarifica que almacena la suma acumulada.
8	c	Nombre de método opaco	getAverage() – Indica que calcula el promedio.

Nº	Identificador problemático	Regla del libro que viola	Propuesta de nombre claro y justificación
9	x	Nombre de objeto poco informativo	stats o calculator – Da sentido al objeto.
10	t	Variable poco significativa	totalNumbers – Explica que representa el total de números a ingresar.
11	k	Contador sin contexto	index – Es más claro en un ciclo de ingreso de datos.

```cpp
#include <iostream>

class NumberStats {                  // Clase con propósito claro
    int numbers[10];                 // Arreglo donde guardaremos hasta 10 números
    int currentIndex = 0;            // Índice actual para insertar

public:
    void addNumber(int number) {     // Método que agrega un número al arreglo
        numbers[currentIndex++] = number;  // Agrega el número y avanza el índice
    }

    int getSum() {                   // Método que calcula la suma
        int sum = 0;                 // Acumulador de la suma
        for (int index = 0; index < currentIndex; ++index) { // Recorre los números ingresados
            sum += numbers[index];
        }
        return sum;
    }

    double getAverage() {            // Método que devuelve el promedio
        return currentIndex == 0 ? 0.0 : static_cast<double>(getSum()) / currentIndex;
    }
};

int main() {
    NumberStats stats;               // Objeto que almacena los números y operaciones
    int totalNumbers;                // Total de números que se ingresarán

    std::cout << "¿Cuántos números ingresara? ";
    std::cin  >> totalNumbers;

    for (int index = 0; index < totalNumbers; ++index) { // Ciclo de ingreso
        std::cout << "Número: ";
        int inputNumber;             // Número ingresado por el usuario
        std::cin >> inputNumber;
        stats.addNumber(inputNumber);
    }

    std::cout << "Suma = "     << stats.getSum()     << '\n';
    std::cout << "Promedio = " << stats.getAverage() << '\n';
    return 0;
}
```

Para el Capitulo 3: “Funciones”, se reutilizo el código anterior y aplicamos las reglas respectivas de acuerdo al capitulo actual:

Nº	Fragmento original	Problema detectado según el Cap. 3	Propuesta corregida y justificación
1	x(int v)	Nombre no dice qué hace. Parámetro sin intención.	addNumber(int number) – nombre claro y descriptivo.
2	y()	Nombre no indica el resultado.	calculateSum() – Expresa qué se calcula y evita ambigüedad.
3	z()	No revela que calcula un promedio.	calculateAverage() – Se entiende sin leer el cuerpo de la función.
4	int s = 0; for(...)	Cuerpo largo que hace demasiadas cosas.	Extraer a una función auxiliar separada para claridad.
5	return i == 0 ? 0.0 : ...	Lógica importante en una sola línea: poco clara.	Usar if y separar el cálculo para mejorar expresividad.
6	B p; en main()	Nombre de clase y objeto no revelan propósito.	Usar NumberAggregator aggregator; para mayor claridad.
7	Entrada de datos dentro del for	Mezcla lógica de entrada y procesamiento.	Separar la entrada en una función readNumberFromInput() para claridad.
8	main() muy cargado	Hace demasiadas cosas (entrada, lógica y salida).	Dividir en funciones pequeñas y expresivas (askTotalNumbers(), showResults()).

```cpp
#include <iostream>
using namespace std;

// Clase que almacena y procesa números
class NumberAggregator {
    int numbers[10];          // Arreglo de hasta 10 números
    int currentIndex = 0;     // Cuántos números se han guardado

public:
    // Agrega un número al arreglo
    void addNumber(int number) {
        if (currentIndex < 10) {
            numbers[currentIndex++] = number;
        }
    }

    // Calcula y retorna la suma de los números
    int calculateSum() {
        int sum = 0;
        for (int index = 0; index < currentIndex; ++index) {
            sum += numbers[index];
        }
        return sum;
    }

    // Calcula y retorna el promedio
    double calculateAverage() {
        if (currentIndex == 0) {
            return 0.0;
        }
        int total = calculateSum();
        return static_cast<double>(total) / currentIndex;
    }
};

```

```cpp
// Pregunta cuántos números se van a ingresar
int askTotalNumbers() {
    int total;
    cout << "¿Cuántos números ingresará? ";
    cin >> total;
    return total;
}

// Lee un número desde la consola
int readNumberFromInput() {
    int number;
    cout << "Número: ";
    cin >> number;
    return number;
}

// Muestra la suma y el promedio
void showResults(NumberAggregator aggregator) {
    cout << "Suma = " << aggregator.calculateSum() << '\n';
    cout << "Promedio = " << aggregator.calculateAverage() << '\n';
}

// Función principal
int main() {
    NumberAggregator aggregator;
    int totalNumbers = askTotalNumbers();

    for (int i = 0; i < totalNumbers; ++i) {
        int number = readNumberFromInput();
        aggregator.addNumber(number);
    }

    showResults(aggregator);
    return 0;
}
```

IV. 4. Conclusiones

- Se ha demostrado la importancia de aplicar los principios de "Nombres Significativos" y "Funciones" del libro "Código Limpio" para mejorar la claridad y mantenibilidad del código C++.
- La elección de nombres descriptivos para clases, variables y funciones facilita la comprensión del código y reduce la ambigüedad.
- La refactorización de funciones para que sean más pequeñas y realicen una única tarea mejora la organización y legibilidad del código, haciéndolo más fácil de mantener y probar.
- La aplicación de estos principios resulta en un código más limpio, profesional y eficiente, lo cual es crucial para el desarrollo de software de calidad.

V. 5. Recomendaciones

- Se recomienda a los desarrolladores de C++ estudiar y aplicar continuamente los principios de código limpio presentados en el libro de Robert C. Martin.

- Es aconsejable realizar revisiones de código periódicas para asegurar la adherencia a las buenas prácticas de nomenclatura y diseño de funciones.
- Se sugiere utilizar herramientas de análisis estático de código que ayuden a identificar posibles problemas de legibilidad y mantenibilidad.
- Se propone la creación de guías de estilo internas en los equipos de desarrollo para homogeneizar el estilo de codificación y facilitar la colaboración.

VI. 6. Bibliografía/ Referencias

Martin, Robert C. (2008). *Código Limpio: Un manual de artesanía del software*.

VII. 7. Anexos:

LINK GITHUB: angeldev7/Actividad2Resuelto

VIII. 8. Legalización de documento

Nombres y Apellidos: Angel Steven Rodriguez Chavez

CI: 2300817083