Projecto para: LICENCIATURA EN INGENIERÍA EN INFORMÁTICA Y SISTEMAS

Projecto Final

Sebastian Echeverria Flores 1138122

Estudiante de Universidad Rafael Landivar

Noviembre 2024

Projecto para LICENCIATURA EN INGENIERÍA EN INFORMÁTICA Y SISTEMAS

Projecto Final

Ingeniero: JOSEPH ABRAHAM SOTO GUTIÉRREZ

Projecto para LICENCIATURA EN INGENIERÍA EN INFORMÁTICA Y SISTEMAS

Sebastian Echeverria Flores 1138122

Noviembre 2024

Department of INGENIERÍA EN INFORMÁTICA Y SISTEMAS
Estudiante de Universidad Rafael Landivar

Table of Contents

Abstract

Projecto 2

Sebastian Echeverria

Departamento de Ingenieria.

Estudiante en la universidad

Rafael Landivar

Chapter 1

Objetivos

Objetivo General: Realizar un proyecto que involucre la comprensión y aplicación de los pilares de la programación orientada a objetos (herencia, polimorfismo, abstracción y encapsulamiento), además de explorar conceptos adicionales relacionados con la POO.

Objetivos Específicos:

1. Investigar y describir al menos tres paradigmas de programación, incluyendo sus casos de uso, relevancia actual y los lenguajes de programación que los soportan.

2. Comprender y definir claramente los pilares de la POO (herencia, polimorfismo, abstracción y encapsulamiento) explicando cada uno y proporcionando un ejemplo de su aplicación.

3. Relacionar cada pilar con conceptos aprendidos en clase, como punteros y memoria, estructuras de datos, ordenamiento y métodos de búsqueda.

4. Investigar y explicar cómo se relacionan con la POO los conceptos de clases abstractas, interfaces, pruebas unitarias y modificadores de acceso.

5. Realizar una operación matemática para determinar el tema o escenario que corresponde al grupo.

6. Según el tema asignado, diseñar una estrategia que aplique herencia y polimorfismo mediante la creación de clases, incluyendo un diagrama de clases UML y pruebas unitarias para validar el código.

7. Establecer un mecanismo para generar valores únicos de ID para los objetos de las clases, considerando su utilización en una lista de objetos.

8. Definir estrategias para ordenar y buscar eficientemente los objetos en la lista.

9. Implementar el diagrama de clases en código utilizando C++ aplicando los pilares de la POO identificados en la estrategia.

Chapter 2

Investigacion

2.1 Paradigmas de programacion

Los paradigmas de programación son enfoques o estilos fundamentales para desarrollar software. Cada paradigma tiene su propia forma de abordar la resolución de problemas y organizar el código. Aquí están tres de los paradigmas más utilizados:

2.1.1 ***Paradigma Imperativo/Procedural:***

Casos de uso/aplicaciones: Este paradigma se centra en las instrucciones que cambian el estado de los datos mediante la ejecución de procedimientos o funciones. Se utiliza ampliamente en sistemas embebidos, desarrollo de sistemas operativos y aplicaciones donde se prioriza la eficiencia y el control de recursos.

Relevancia actual: Aunque ha sido superado por paradigmas más modernos en algunos contextos, sigue siendo relevante y se utiliza en muchos sistemas críticos y de bajo nivel debido a su eficiencia en el manejo directo de recursos.

Lenguajes de programación: C, Fortran y Pascal son ejemplos de lenguajes que siguen este paradigma.

2.1.2 ***Paradigma Orientado a Objetos:***

Casos de uso/aplicaciones: El paradigma orientado a objetos se centra en la representación de entidades del mundo real como objetos que tienen

atributos y comportamientos. Es ampliamente utilizado en el desarrollo de aplicaciones empresariales, videojuegos, sistemas distribuidos y aplicaciones que requieren reutilización de código.

Relevancia actual: Sigue siendo muy relevante y ampliamente utilizado en el desarrollo de software moderno debido a su capacidad para modelar sistemas complejos y favorecer la reutilización de código.

Lenguajes de programación: Java, C++, Python, y C son ejemplos de lenguajes que soportan la programación orientada a objetos.

2.1.3 ***Paradigma Funcional:***

Casos de uso/aplicaciones: Se centra en las funciones puras y en la evaluación de expresiones. Es utilizado en aplicaciones donde la concurrencia y la tolerancia a fallos son importantes, así como en el procesamiento de datos y la programación matemática.

Relevancia actual: Ha ganado relevancia en los últimos años debido a su capacidad para manejar mejor la concurrencia y la programación en paralelo, aspectos cada vez más importantes en la computación moderna.

Lenguajes de programación: Lisp, Haskell, Erlang, y F son ejemplos de lenguajes que admiten la programación funcional.

Es esencial notar que muchos lenguajes de programación son multiparadigmáticos, lo que significa que permiten la combinación de varios paradigmas en una misma aplicación. Por ejemplo, lenguajes como Python, JavaScript, y C++ admiten elementos de programación imperativa, orientada a objetos y funcional. Esto permite a los desarrolladores elegir el enfoque más adecuado para resolver un problema específico.

2.2 Programacion Orientada a Objetos (POO)

1. **Herencia:**

   Definición: La herencia es un mecanismo en el que una clase (llamada clase derivada o subclase) hereda propiedades y comportamientos de otra clase (llamada clase base o superclase). Esto permite la reutilización de código y la creación de jerarquías entre clases.

   Ejemplo: Un ejemplo podría ser la relación entre las clases `Animal` y `Perro`. La clase `Perro` puede heredar atributos y métodos de la clase `Animal` (como `comer()` o `dormir()`), además de tener sus propios métodos y atributos específicos (`ladrar()`).

   Relación con otros conceptos: En el contexto de punteros y memoria, la herencia puede afectar la disposición de memoria para objetos y subobjetos en lenguajes como C++. También puede influir en la organización de estructuras de datos y en cómo se implementan los algoritmos de ordenamiento y búsqueda para clases derivadas.

2. **Polimorfismo:**

   Definición: El polimorfismo permite que un objeto pueda tomar diferentes formas o comportarse de maneras diferentes según el contexto. Se puede lograr a través del uso de métodos con el mismo nombre pero con diferentes implementaciones en diferentes clases.

   Ejemplo: Supongamos una interfaz `Figura` con un método `calcularArea()`. Las clases `Cuadrado` y `Círculo` implementarán este método de manera diferente según sus propias fórmulas de área.

   Relación con otros conceptos: El polimorfismo está relacionado con

la manipulación de punteros a objetos, ya que permite tratar objetos de diferentes clases de manera uniforme a través de un puntero base. También influye en cómo se estructuran y acceden los datos en diversas estructuras de datos.

3. **Abstracción:**

   Definición: La abstracción es el proceso de enfocarse en los aspectos esenciales de un objeto y ocultar los detalles irrelevantes o complejos. Las clases abstractas y las interfaces son ejemplos de abstracción.

   Ejemplo: Una clase abstracta `Vehiculo` podría tener métodos como `acelerar()` y `frenar()`, pero no se define cómo se implementan en un vehículo específico. Las clases derivadas como `Coche` o `Bicicleta` implementarán esos métodos de acuerdo con sus propias características.

   Relación con otros conceptos: La abstracción puede influir en la manipulación de la memoria y los punteros al permitir la creación de estructuras de datos más abstractas y genéricas. También puede ser importante al definir algoritmos de búsqueda y ordenamiento que operan en objetos abstractos.

4. **Encapsulamiento:**

   Definición: El encapsulamiento es el principio de ocultar los detalles internos de un objeto y permitir el acceso controlado a través de interfaces bien definidas. Esto se logra utilizando modificadores de acceso (como público, privado, protegido) para los atributos y métodos de una clase.

   Ejemplo: Una clase `CuentaBancaria` puede tener atributos privados como `saldo` y métodos públicos como `depositar()` y `retirar()`, que

controlan el acceso a esos atributos.

Relación con otros conceptos: El encapsulamiento se relaciona con la gestión de la memoria al limitar el acceso a los datos internos de un objeto. También influye en la elección de estructuras de datos y cómo se implementan algoritmos de ordenamiento y búsqueda para mantener la integridad de los datos encapsulados.

2.2.1 Conceptos y relaciones

1. **Clases abstractas:**

   Relación con la programación orientada a objetos: Las clases abstractas son un componente fundamental en la programación orientada a objetos, ya que permiten definir comportamientos comunes y definir métodos que deben ser implementados por clases derivadas. Esto facilita la reutilización de código y la definición de una jerarquía de clases.

2. **Interfaces:**

   Definición: Las interfaces son un conjunto de métodos abstractos que definen un contrato para las clases que las implementan. No pueden contener implementaciones de métodos, solo la firma de los métodos. Las clases pueden implementar múltiples interfaces.

   Relación con la programación orientada a objetos: Las interfaces proporcionan un mecanismo para la implementación de polimorfismo en la programación orientada a objetos. Permiten definir un conjunto de métodos que las clases deben implementar, independientemente de su jerarquía de clases. Esto facilita la creación de código flexible y modular.

3. **Pruebas unitarias:**

   Definición: Las pruebas unitarias son un enfoque de prueba que evalúa unidades individuales de código (como métodos o funciones) para asegurar que funcionen correctamente de forma aislada. Se centran en probar cada componente por separado, simulando su entorno y condiciones.

   Relación con la programación orientada a objetos: En el contexto de la programación orientada a objetos, las pruebas unitarias se centran en probar métodos de clases individuales. La modularidad y el encapsulamiento permiten que los métodos de las clases sean probados independientemente, lo que facilita la escritura de pruebas unitarias efectivas y específicas.

4. **Modificadores de acceso:**

   Definición: Los modificadores de acceso controlan la visibilidad y accesibilidad de clases, atributos y métodos en la programación orientada a objetos. Los principales son: público (public), privado (private), protegido (protected), y por defecto (default o package-private, en algunos lenguajes).

   Relación con la programación orientada a objetos: Los modificadores de acceso son esenciales para el encapsulamiento en la programación orientada a objetos. Permiten controlar qué partes de una clase son accesibles desde otras clases y cómo se accede a ellas. Esto ayuda a mantener la integridad de los datos y a establecer interfaces bien definidas para interactuar con los objetos.

Chapter 3

Estrategia

Este es un proyecto bastante amplio que implica varios pasos, desde el diseño de clases hasta la implementación de pruebas unitarias y la codificación en C++. Aquí te ofrezco un esquema detallado de cómo abordaría cada parte de este proyecto:

1. Sistema de Clases con Herencia y Polimorfismo Clases Propuestas:

Vehículo (Clase Base) Propiedades: ID, marca Funciones: obtenerInfo(), generarID()

Automóvil (Hereda de Vehículo) Propiedades: tipoTransmision, numeroAsientos Funciones: cambiarTransmision(), mostrarAsientos()

Motocicleta (Hereda de Vehículo) Propiedades: cilindrada, tipoManillar Funciones: mostrarCilindrada(), ajustarManillar()

Camión (Hereda de Vehículo) Propiedades: capacidadCarga, longitud Funciones: cargar(), mostrarLongitud()

Bicicleta (Hereda de Vehículo) Propiedades: tipoBicicleta, tamañoRueda Funciones: cambiarTipo(), mostrarTamañoRueda()

Pruebas Unitarias Para cada clase, podríamos tener pruebas unitarias como:

Automóvil: Probar cambiarTransmision() y mostrarAsientos(). Motocicleta: Probar mostrarCilindrada() y ajustarManillar(). Y así sucesivamente para cada clase.

2. Propiedad ID Única Generación de ID Tipo de Dato: int o long (dependiendo del rango esperado). Generación: Utilizar un contador estático en la clase base que se incremente cada vez que se cree un nuevo objeto.

Momento y Accesibilidad de ID Generado en el constructor de la clase base (Vehículo). Accesible a través de una función miembro, por ejemplo, getID().

3. Ordenamiento y Búsqueda Eficiente Estrategia de Ordenamiento Implementar una función de comparación o sobrecargar el operador $<$ en la clase base. Utilizar algoritmos de ordenamiento eficientes como quicksort o mergesort en la lista de objetos.

4. Implementación en C++ Aplicación de POO Herencia: Clases derivadas de la clase base Vehículo. Polimorfismo: Métodos como obtenerInfo() podrían ser implementados de forma diferente en cada subclase. Encapsulamiento: Uso de propiedades privadas/protected y métodos públicos para acceder/modificar estas propiedades.

Generación de ID Implementado en el constructor de Vehículo. Creación de Objetos en Main Instanciar 5 objetos de cada tipo (Automóvil, Motocicleta, etc.) en main().

Chapter 4

Conclusiones

1. **Importancia de los Pilares de la POO:**

   Los pilares de la Programación Orientada a Objetos (POO) son fundamentales para el diseño y desarrollo de sistemas de software eficientes y mantenibles. La comprensión y aplicación de conceptos como herencia, polimorfismo, abstracción y encapsulamiento permiten crear código más modular, reutilizable y escalable.

2. **Relevancia de los Paradigmas de Programación:**

   Los paradigmas de programación, como el imperativo, orientado a objetos y funcional, ofrecen diferentes enfoques para abordar la resolución de problemas. La elección del paradigma adecuado depende del contexto y de los requisitos del proyecto, permitiendo utilizar herramientas y enfoques específicos según sea necesario.

3. **Relación entre Conceptos Clave:**

   Los conceptos vistos en clase, como punteros y memoria, estructuras de datos, ordenamiento y métodos de búsqueda, se relacionan directamente con los pilares de la POO. La comprensión de estos conceptos es crucial para aplicar correctamente la POO y optimizar la eficiencia y funcionalidad del código.

4. **Aplicación Práctica y Diseño de Clases:**

   La aplicación de herencia y polimorfismo en el diseño de clases permite crear sistemas flexibles y extensibles.

La estrategia propuesta para generar identificadores únicos y para ordenar y buscar objetos en una lista responde a la necesidad de eficiencia en el manejo de objetos en un sistema.

Chapter 5

Referencias

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

2. Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *Unified Modeling Language User Guide, The (2nd Edition)*. Addison-Wesley Professional.

3. Eckel, B. (2003). *Thinking in C++*. Prentice Hall PTR.

4. Martin, R. C. (2002). *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education.

5. Stroustrup, B. (2013). *The C++ Programming Language*. Addison-Wesley Professional.

6. McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press.

7. Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall.

8. Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

9. Sommerville, I. (2016). *Software Engineering*. Pearson Education Limited.