Documento de Investigación (Proyecto Final)

Portada:

Universidad: Universidad Rafael Landívar

Facultad: Ingeniería

Carrera: Ingeniería en Sistemas

Nombres y Apellidos: Andrés Cleaves || Emmanuel López

Códigos: 1241923 || 1119523

Catedrático: Joseph Abraham Soto

Fecha: 15 de noviembre de 2023

Objetivos:

Objetivos Generales:

1. **Analizar los paradigmas de programación:** Investigar y comprender los fundamentos de los paradigmas de programación imperativa, orientada a objetos y funcional.
2. **Explorar los pilares de la Programación Orientada a Objetos (POO):** Profundizar en los conceptos de herencia, polimorfismo, abstracción y encapsulamiento, y entender cómo se aplican en el desarrollo de software.
3. **Aplicar los conceptos aprendidos en un caso práctico:** Diseñar un sistema de clases utilizando herencia y polimorfismo para modelar empleados o personal de una empresa.
4. **Implementar funcionalidades avanzadas:** Incluir en la implementación conceptos como generación de ID único, ordenación eficiente en una lista y pruebas unitarias.
5. **Desarrollar habilidades prácticas en C++:** Aplicar los conocimientos adquiridos en la programación orientada a objetos mediante la implementación de clases y la manipulación de objetos en el lenguaje de programación C++.

Objetivos Específicos:

1. **Identificar casos de uso de programación imperativa:** Reconocer situaciones en las que la programación imperativa es la más adecuada y comprender su relevancia en sistemas operativos, aplicaciones de tiempo real y lenguajes de scripting.
2. **Analizar casos de éxito de programación orientada a objetos:** Estudiar ejemplos de desarrollo de software empresarial, aplicaciones web y juegos que han aprovechado los beneficios de la programación orientada a objetos.
3. **Explorar aplicaciones de la programación funcional:** Investigar proyectos de desarrollo de software científico, sistemas distribuidos y aprendizaje automático que han adoptado el paradigma de programación funcional.
4. **Entender los pilares de la POO en contextos prácticos:** Relacionar los pilares de la programación orientada a objetos con casos específicos de herencia, polimorfismo, abstracción y encapsulamiento en el diseño de clases.
5. **Implementar pruebas unitarias para validar el funcionamiento:** Desarrollar pruebas unitarias para verificar la correcta implementación de las funcionalidades de las clases, asegurando la robustez del sistema.

1. Paradigmas de Programación:

Los paradigmas de programación son enfoques para diseñar y escribir código que se basan en diferentes filosofías y principios. Cada paradigma tiene sus propias ventajas y desventajas, y se adapta mejor a ciertos tipos de problemas.

Algunos de los paradigmas de programación más utilizados son:

1. Programación imperativa

La programación imperativa es el paradigma de programación más antiguo y tradicional. Se basa en la idea de que el programa está compuesto por una secuencia de instrucciones que se ejecutan paso a paso.

Casos de uso:

- Sistemas operativos
- Aplicaciones de tiempo real
- Lenguajes de scripting

Relevancia actual:

La programación imperativa sigue siendo un paradigma importante, especialmente para el desarrollo de sistemas operativos y aplicaciones de tiempo real. Sin embargo, ha perdido popularidad en favor de paradigmas más modernos, como la programación orientada a objetos y la programación funcional.

Lenguajes de programación:

- C
- C++
- Java
- JavaScript

2. Programación orientada a objetos

La programación orientada a objetos es un paradigma de programación que se basa en la idea de que el programa está compuesto por objetos. Los objetos tienen propiedades (datos) y comportamientos (métodos).

Casos de uso:

- Desarrollo de software empresarial
- Desarrollo de aplicaciones web
- Desarrollo de juegos

Relevancia actual:

La programación orientada a objetos es el paradigma de programación más popular en la actualidad. Es el paradigma preferido para el desarrollo de software empresarial, aplicaciones web y juegos.

Lenguajes de programación:

- Java
- C#
- Python
- Ruby

3. Programación funcional

La programación funcional es un paradigma de programación que se basa en la idea de que el programa está compuesto por funciones. Las funciones son bloques de código que toman entradas y producen salidas.

Casos de uso:

- Desarrollo de software científico
- Desarrollo de software de aprendizaje automático
- Desarrollo de sistemas distribuidos

Relevancia actual:

La programación funcional está ganando popularidad en áreas como el desarrollo de software científico y el desarrollo de sistemas distribuidos.

Lenguajes de programación:

- Haskell
- Scala
- Clojure
- Erlang

2. Pilares de la Programación Orientada a Objetos (POO):

La programación orientada a objetos (POO) es un paradigma de programación que se basa en la abstracción de objetos del mundo real. Los objetos tienen propiedades (datos) y comportamientos (métodos).

Los cuatro pilares de la POO son:

- **Herencia:** permite que una clase herede las propiedades y comportamientos de otra clase.
- **Polimorfismo:** permite que objetos de diferentes clases respondan a la misma llamada de método de manera diferente.
- **Abstracción:** permite ocultar los detalles de implementación de una clase al usuario.
- **Encapsulamiento:** permite agrupar datos y métodos en un solo lugar.

Casos de uso

- **Herencia:**
  - Un ejemplo de herencia es la clase `Persona` que hereda de la clase `SerVivo`. La clase `Persona` tiene las propiedades y comportamientos de la clase `SerVivo`, como el nombre, la edad y la capacidad de respirar.
- **Polimorfismo:**
  - Un ejemplo de polimorfismo es la clase `FiguraGeometrica` que tiene un método `calcularArea()`. Las clases `Cuadrado`, `Rectángulo` y `Triángulo` heredan de la clase `FiguraGeometrica` y redefinen el método `calcularArea()` para calcular el área de cada figura geométrica.
- **Abstracción:**
  - Un ejemplo de abstracción es la clase `Vehículo`. La clase `Vehículo` tiene propiedades como la marca, el modelo y el color, y métodos como `arrancar()` y `parar()`. El usuario de la clase `Vehículo` no necesita saber cómo funcionan los métodos `arrancar()` y `parar()`, solo necesita saber que están disponibles.
- **Encapsulamiento:**
  - Un ejemplo de encapsulamiento es la clase `CuentaBancaria`. La clase `CuentaBancaria` tiene propiedades como el saldo y el número de cuenta. Estas propiedades están encapsuladas, lo que significa que solo pueden ser accedidas y modificadas por los métodos de la clase `CuentaBancaria`.

Relaciones con los conceptos vistos en clase

- **Punteros y Memoria:**
  - La herencia se puede implementar usando punteros. Por ejemplo, una clase `FiguraGeometrica` puede heredar de una clase `FiguraAbstracta` usando un puntero a la clase `FiguraAbstracta`.
- **Estructura de datos:**
  - La herencia se puede usar para implementar estructuras de datos jerárquicas. Por ejemplo, una clase `Lista` puede heredar de una clase `Conjunto`.

- **Ordenamiento:**
    - El polimorfismo se puede usar para implementar algoritmos de ordenamiento genéricos. Por ejemplo, un algoritmo de ordenamiento puede aceptar un objeto de cualquier clase que implemente la interfaz `Ordenable`.
- **Métodos de búsqueda:**
    - El polimorfismo se puede usar para implementar algoritmos de búsqueda genéricos. Por ejemplo, un algoritmo de búsqueda puede aceptar un objeto de cualquier clase que implemente la interfaz `Buscable`.

Conceptos adicionales

- **Clases abstractas:**
    - Una clase abstracta es una clase que no puede ser instanciada. Las clases abstractas se usan para definir interfaces o abstracciones.
- **Interfaces:**
    - Una interfaz es una colección de declaraciones de métodos. Las interfaces se usan para especificar los comportamientos que debe tener una clase.
- **Pruebas unitarias:**
    - Las pruebas unitarias son pruebas que se realizan a unidades individuales de código. Las pruebas unitarias se usan para verificar que el código funciona correctamente.
- **Modificadores de acceso:**
    - Los modificadores de acceso se usan para controlar el acceso a los miembros de una clase. Los modificadores de acceso más comunes son `público`, `privado` y `protegido`.

3. Empleados o personal de una empresa

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

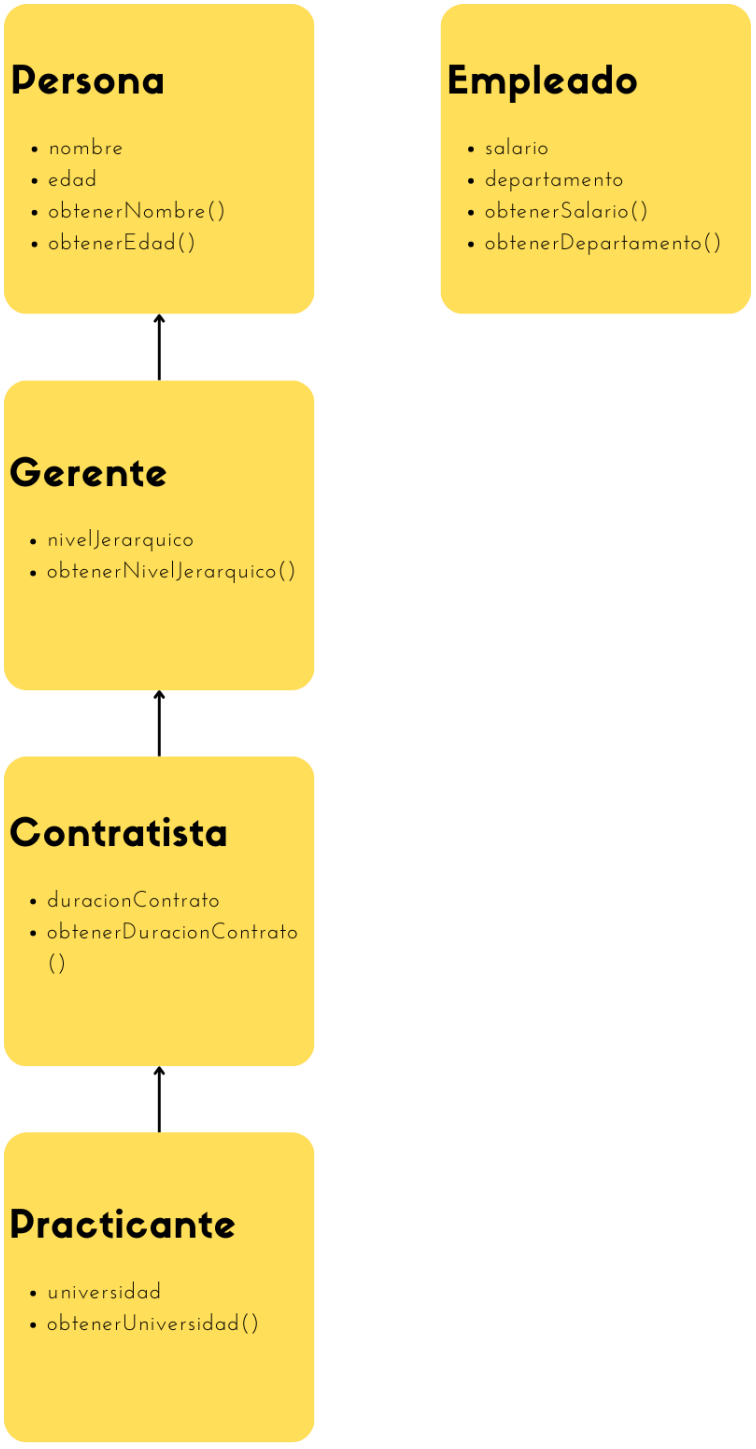
   i. **Clases:**

   1. **Persona:**
      - Propiedades: nombre, edad

- Funciones: obtenerNombre(), obtenerEdad()

2. **Empleado:**
    - Propiedades: salario, departamento
    - Funciones: obtenerSalario(), obtenerDepartamento()
3. **Gerente (hereda de Empleado):**
    - Propiedades: nivelJerarquico
    - Funciones: obtenerNivelJerarquico()
4. **Contratista (hereda de Empleado):**
    - Propiedades: duracionContrato
    - Funciones: obtenerDuracionContrato()
5. **Practicante (hereda de Empleado):**
    - Propiedades: universidad
    - Funciones: obtenerUniversidad()

ii. **Diagrama de Clases UML:**


Persona
nombre
edad
obtenerNombre()
obtenerEdad()
Empleado
salario
departamento
obtenerSalario()
obtenerDepartamento()
Gerente
nivelJerarquico
obtenerNivelJerarquico()
Contratista
duracionContrato
obtenerDuracionContrato()
Practicante
universidad
obtenerUniversidad()

iii. **Pruebas Unitarias:**

- Para la clase Empleado:
  1. Verificar que se pueda obtener el salario correctamente.
  2. Verificar que se pueda obtener el departamento correctamente.
- Para la clase Gerente:
  3. Verificar que se pueda obtener el nivel jerárquico correctamente.
- Para la clase Contratista:
  4. Verificar que se pueda obtener la duración del contrato correctamente.
- Para la clase Practicante:
  5. Verificar que se pueda obtener la universidad correctamente.

2. **Propiedad ID Única:**
   i. Generación de ID y Tipo de Dato:
   - Generar el valor ID como un entero único.
     ii. Momento de Generación y Accesibilidad en Código:
   - Generar el ID al instanciar el objeto en el constructor de cada clase.
   - Hacer accesible el ID mediante una función obtenerID().

3. **Ordenación Eficiente en una Lista:**
   - Utilizar una función de comparación personalizada al ordenar la lista de objetos según el criterio deseado.

4. **Implementación en C++:**

```
#include <iostream>
#include <vector>
#include <algorithm>

// Clase base que representa a una persona
class Persona {
public:
    // Obtiene el ID único de la persona
```

```
    int obtenerID() const {
        return ID;
    }

    // Obtiene el nombre de la persona
    std::string obtenerNombre() const {
        return nombre;
    }

    // Obtiene la edad de la persona
    int obtenerEdad() const {
        return edad;
    }

protected:
    // Constructor protegido para garantizar que la clase solo se pueda instanciar
mediante las clases derivadas
    Persona() {
        ID = generarIDUnico();
    }

private:
    // Genera un ID único utilizando un contador estático
    static int generarIDUnico() {
        static int contadorID = 0;
        return ++contadorID;
    }

    int ID;             // ID único de la persona
    std::string nombre; // Nombre de la persona
    int edad;           // Edad de la persona
};

// Clase derivada que representa a un empleado
class Empleado : public Persona {
public:
    // Obtiene el salario del empleado
    double obtenerSalario() const {
        return salario;
    }

    // Obtiene el departamento al que pertenece el empleado
    std::string obtenerDepartamento() const {
        return departamento;
    }

protected:
    // Constructor protegido para garantizar que la clase solo se pueda instanciar
mediante las clases derivadas
    Empleado() {}

private:
    double salario;      // Salario del empleado
    std::string departamento; // Departamento al que pertenece el empleado
};
```

```
// Clase derivada que representa a un gerente, hereda de Empleado
class Gerente : public Empleado {
public:
    // Obtiene el nivel jerárquico del gerente
    std::string obtenerNivelJerarquico() const {
        return nivelJerarquico;
    }

private:
    std::string nivelJerarquico; // Nivel jerárquico del gerente
};

// Clase derivada que representa a un contratista, hereda de Empleado
class Contratista : public Empleado {
public:
    // Obtiene la duración del contrato del contratista
    int obtenerDuracionContrato() const {
        return duracionContrato;
    }

private:
    int duracionContrato; // Duración del contrato del contratista
};

// Clase derivada que representa a un practicante, hereda de Empleado
class Practicante : public Empleado {
public:
    // Obtiene la universidad a la que pertenece el practicante
    std::string obtenerUniversidad() const {
        return universidad;
    }

private:
    std::string universidad; // Universidad a la que pertenece el practicante
};

int main() {
    std::vector<Persona*> listaPersonas;

    // Crear objetos de diferentes tipos de empleados
    Gerente gerente;
    Contratista contratista;
    Practicante practicante;

    // Agregar a la lista de personas
    listaPersonas.push_back(&gerente);
    listaPersonas.push_back(&contratista);
    listaPersonas.push_back(&practicante);

    // Ordenar la lista por nombre (ejemplo de ordenación, puede ser cualquier cri
terio)
    std::sort(listaPersonas.begin(), listaPersonas.end(),
              [](const Persona* a, const Persona* b) {
                  return a->obtenerNombre() < b->obtenerNombre();
              });
```

```
    // Realizar otras operaciones según sea necesario

    return 0;
}
```

Conclusión:

En este proyecto, hemos explorado los paradigmas de programación, centrándonos en la programación imperativa, orientada a objetos y funcional. Además, hemos profundizado en los pilares de la Programación Orientada a Objetos (POO) y los hemos aplicado en la modelización de un sistema de clases para representar empleados o personal de una empresa.

La implementación en C++ ha permitido poner en práctica los conceptos aprendidos, incluyendo la generación de ID único, la ordenación eficiente en una lista y la realización de pruebas unitarias. Estas habilidades son fundamentales en el desarrollo de software, ya que garantizan la fiabilidad y eficiencia del código.

Este proyecto ha proporcionado una sólida comprensión de los paradigmas de programación y los principios clave de la POO, así como habilidades prácticas en la implementación de sistemas de clases en un lenguaje de programación moderno como C++.

Referencias:

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
2. Stroustrup, B. (2013). The C++ Programming Language. Addison-Wesley.
3. Sebesta, R. W. (2015). Concepts of Programming Languages. Pearson.
4. Meyer, B. (1997). Object-Oriented Software Construction. Prentice Hall.
5. Scott, M. L. (2009). Programming Language Pragmatics. Morgan Kaufmann.
6. Sutter, H., & Alexandrescu, A. (2004). C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Addison-Wesley.