Sistema de Empleados 1.0v

Este proyecto implementa un sistema de gestión de empleados con varias clases derivadas de `Empleado` y un `Departamento` que maneja a los empleados utilizando plantillas de clase.

Clases

Empleado

```
class Empleado{
protected:
    string nombre;
    double salario;
    int fechaContratacion;

public:
    Empleado(string nombre, double salario, int fechaContratacion){
        this -> nombre = std::move(nombre);
        this -> salario = salario;
        this -> fechaContratacion = fechaContratacion;
    }

    virtual double calcularSalario() = 0;
    virtual void mostrarDatos() = 0;
    virtual void ingresarDatos() = 0;

    const string &getNombre() const {
        return nombre;
    }

    void setNombre(const string &nombre) {
        Empleado::nombre = nombre;
    }

    double getSalario() const {
        return salario;
    }

    void setSalario(double salario) {
        Empleado::salario = salario;
    }

    const int &getFechaContratacion() const {
        return fechaContratacion;
    }
```

```
    void setFechaContratacion(const int &fechaContratacion) {
        Empleado::fechaContratacion = fechaContratacion;
    }
};
```

La clase `Empleado` es una clase abstracta que define la interfaz para los empleados. Tiene los siguientes métodos:

- `Empleado(const string& nombre, double salario, time_t fechaContratacion)`: Constructor que inicializa los atributos del empleado.
- `virtual ~Empleado() {}`: Destructor virtual.
- `virtual double calcularSalario() const = 0`: Método virtual puro para calcular el salario del empleado.
- `const string& getNombre() const { return nombre; }`: Devuelve el nombre del empleado.
- `double getSalario() const { return salario; }`: Devuelve el salario del empleado.
- `time_t getFechaContratacion() const { return fechaContratacion; }`: Devuelve la fecha de contratación del empleado.

Gerente

```
class Gerente : public Empleado{
private:
    string departamento;
    double bono;

public:
    explicit Gerente(string nombre = "", double salario = 0, int fechaContratacion =
0, string departamento = "", double bono = 0) :
            Empleado(std::move(nombre), salario, fechaContratacion){
        this->departamento = std::move(departamento);
        this->bono = bono;
    }

    double calcularSalario() override{
        return salario + bono;
    }

    void mostrarDatos() override {
        cout << "Nombre: " << nombre << endl;
        cout << "Salario: " << salario << endl;
        cout << "Fecha de contratacion: " << fechaContratacion << endl;
        cout << "Departamento: " << departamento << endl;
        cout << "Bono: " << bono << endl;
    }

    void ingresarDatos() override {
```

```
        cout << "Ingreso de datos de Gerente" << endl;
        cout << "   Nombre: "; cin >> nombre;
        cout << "   Salario: "; cin >> salario;
        cout << "   Fecha de contratacion: "; cin >> fechaContratacion;
        cout << "   Departamento: "; cin >> departamento;
        cout << "   Bono: "; cin >> bono;
    }

    const string &getDepartamento() const {
        return departamento;
    }

    void setDepartamento(const string &departamento) {
        Gerente::departamento = departamento;
    }

    double getBono() const {
        return bono;
    }

    void setBono(double bono) {
        Gerente::bono = bono;
    }
};
```

La clase `Gerente` es una clase derivada de `Empleado` que representa a un gerente. Tiene un atributo adicional `bono` y sobrescribe el método `calcularSalario`.

Desarrollador

```
class Desarrollador : public Empleado{

private:
    string lenguaje;
    int horasExtra;

public:
    explicit Desarrollador(string nombre = "", double salario = 0, int
fechaContratacion = 0, string lenguaje = "", int horasExtra = 0) :
            Empleado(std::move(nombre), salario, fechaContratacion){
        this->lenguaje = std::move(lenguaje);
        this->horasExtra = horasExtra;
    }

    double calcularSalario() override {
        return salario + (horasExtra * 50);
    }

    void mostrarDatos() override {
```

```
        cout << "Nombre: " << nombre << endl;
        cout << "Salario: " << salario << endl;
        cout << "Fecha de contratacion: " << fechaContratacion << endl;
        cout << "Lenguaje: " << lenguaje << endl;
        cout << "Horas extra: " << horasExtra << endl;
    }
    void ingresarDatos() override {
        cout << "Ingreso de datos de Desarrollador" << endl;
        cout << "   Nombre: ";
        cin >> nombre;
        cout << "   Salario: ";
        cin >> salario;
        cout << "   Fecha de contratacion: ";
        cin >> fechaContratacion;
        cout << "   Lenguaje:  ";
        cin >> lenguaje;
        cout << "   horasExtra:  ";
        cin >> horasExtra;
    }

    const string &getLenguaje() const {
        return lenguaje;
    }

    void setLenguaje(const string &lenguaje) {
        Desarrollador::lenguaje = lenguaje;
    }

    int getHorasExtra() const {
        return horasExtra;
    }

    void setHorasExtra(int extra) {
        Desarrollador::horasExtra = extra;
    }
};
```

La clase `Desarrollador` es una clase derivada de `Empleado` que representa a un desarrollador. Sobrescribe el método `calcularSalario`.

Diseñador

```
class Disenador : public Empleado{

private:
    string tipo;
    int proyectos;

public:
```

```
    explicit Disenador(string nombre = "", double salario = 0, int fechaContratacion =
0, string tipo = "", int proyectos = 0)
            : Empleado(nombre, salario, fechaContratacion){
        this->tipo = tipo;
        this->proyectos = proyectos;
    }

    double calcularSalario() override {
        return salario + (proyectos * 100);
    }

    void mostrarDatos() override {
        cout << "Nombre: " << nombre << endl;
        cout << "Salario: " << salario << endl;
        cout << "Fecha de contratacion: " << fechaContratacion << endl;
        cout << "Tipo: " << tipo << endl;
        cout << "Proyectos: " << proyectos << endl;
    }

    void ingresarDatos() override {
        cout << "Ingreso de datos de nuebo Disenador" << endl;
        cout << "   Nombre: ";
        cin >> nombre;
        cout << "   Salario: ";
        cin >> salario;
        cout << "   Fecha de contratacion: ";
        cin >> fechaContratacion;
        cout << "   Tipo de diseñador: ";
        cin >> tipo;
    }

    const string &getTipo() const {
        return tipo;
    }

    void setTipo(const string &tipo) {
        Disenador::tipo = tipo;
    }

    int getProyectos() const {
        return proyectos;
    }

    void setProyectos(int proyectos) {
        Disenador::proyectos = proyectos;
    }
};
```

La clase `Diseñador` es una clase derivada de `Empleado` que representa a un diseñador. Sobrescribe el método `calcularSalario`.

Tester

```
class Tester : public Empleado{

private:
    int bugs;

public:
    explicit Tester(string nombre = "", double salario = 0, int fechaContratacion = 0,
string tipo = "", int bugs = 0)
            : Empleado(std::move(nombre), salario, fechaContratacion){
        this->bugs = bugs;
    }

    double calcularSalario() override {
        return salario + (bugs * 10);
    }

    void mostrarDatos() override {
        cout << "Nombre: " << nombre << endl;
        cout << "Salario: " << salario << endl;
        cout << "Fecha de contratacion: " << fechaContratacion << endl;
        cout << "Bugs: " << bugs << endl;
    }

    void ingresarDatos() override {
        cout << "Ingreso de datos de Tester" << endl;
        cout << "   Nombre: "; cin >> nombre;
        cout << "   Salario: "; cin >> salario;
        cout << "   Fecha de contratacion: "; cin >> fechaContratacion;
        cout << "   Total de bugs: "; cin >> bugs;

    }

    int getBugs() const {
        return bugs;
    }

    void setBugs(int bugs) {
        Tester::bugs = bugs;
    }
};
```

La clase `Tester` es una clase derivada de `Empleado` que representa a un tester. Sobrescribe el método `calcularSalario`.

GestorArchivos

```
#include <fstream>
#include <stdexcept>
#include <string>
#include <vector>
#include "Empleado.h"
#include "Gerente.h"
#include "Desarrollador.h"
#include "Diseñador.h"

using namespace std;

class GestorArchivos {
public:
    template <typename T>
    static void guardar(const string& archivo, const vector<T*>& empleados) {
        ofstream ofs(archivo);
        if (!ofs) {
            throw runtime_error("Error al abrir el archivo para guardar");
        }
        for (const auto& empleado : empleados) {
            ofs << typeid(*empleado).name() << " "
                << empleado->getNombre() << " "
                << empleado->getSalario() << " "
                << empleado->getFechaContratacion() << endl;
        }
    }

    static Empleado* crearEmpleado(const string& tipo, const string& nombre, double
salario, time_t fechaContratacion) {
        if (tipo == typeid(Gerente).name()) {
            return new Gerente(nombre, salario, fechaContratacion, 1000); // Se podría
ajustar el bono aquí
        } else if (tipo == typeid(Desarrollador).name()) {
            return new Desarrollador(nombre, salario, fechaContratacion);
        } else if (tipo == typeid(Diseñador).name()) {
            return new Diseñador(nombre, salario, fechaContratacion);
        }
        return nullptr;
    }

    template <typename T>
    static void cargar(const string& archivo, vector<T*>& empleados) {
        ifstream ifs(archivo);
        if (!ifs) {
            throw runtime_error("Error al abrir el archivo para cargar");
        }
        string tipo, nombre;
        double salario;
```

```
        time_t fechaContratacion;
        while (ifs >> tipo >> nombre >> salario >> fechaContratacion) {
            Empleado* empleado = crearEmpleado(tipo, nombre, salario,
fechaContratacion);
            if (empleado) {
                empleados.push_back(static_cast<T*>(empleado));
            }
        }
    }
};
```

La clase `GestorArchivos` es responsable de guardar y cargar empleados desde y hacia un archivo. Utiliza plantillas para manejar diferentes tipos de empleados.

- `template <typename T> static void guardar(const string& archivo, const vector<T*>& empleados)`: Guarda una lista de empleados en un archivo.
- `static Empleado* crearEmpleado(const string& tipo, const string& nombre, double salario, time_t fechaContratacion)`: Crea un objeto `Empleado` basado en el tipo.
- `template <typename T> static void cargar(const string& archivo, vector<T*>& empleados)`: Carga una lista de empleados desde un archivo.

Departamento

```
#include <vector>
#include <algorithm>
#include <stdexcept>
#include <functional>
#include "GestorArchivo.h"
#include "Empleado.h"

using namespace std;

template <typename T>
class Departamento {
    vector<T*> empleados;

public:
    ~Departamento() {
        for (auto empleado : empleados) {
            delete empleado;
        }
    }

    void agregarEmpleado(T* empleado) {
        empleados.push_back(empleado);
    }

    void eliminarEmpleado(const string& nombre) {
```

```
        empleados.erase(remove_if(empleados.begin(), empleados.end(),
                                  [&nombre](T* empleado) {
                                      return empleado->getNombre() == nombre;
                                  }), empleados.end());
    }

    T* buscarEmpleado(const string& nombre) const {
        auto it = find_if(empleados.begin(), empleados.end(),
                          [&nombre](T* empleado) {
                              return empleado->getNombre() == nombre;
                          });
        if (it != empleados.end()) {
            return *it;
        } else {
            throw runtime_error("Empleado no encontrado");
        }
    }

    void ordenarEmpleados(function<bool(T*, T*)> comparador) {
        sort(empleados.begin(), empleados.end(), comparador);
    }


    void listarEmpleados() const {
        for (const auto& empleado : empleados) {
            cout << "Nombre: " << empleado->getNombre() << ", Salario: " << empleado-
>calcularSalario() << endl;
        }
    }

    void guardar(const string& archivo) const {
        GestorArchivos::guardar(archivo, empleados);
    }

    void cargar(const string& archivo) {
        GestorArchivos::cargar(archivo, empleados);
    }
};
```

La clase `Departamento` maneja una colección de empleados utilizando plantillas de clase. Permite agregar, eliminar, buscar, listar, guardar y cargar empleados.

- `~Departamento()`: Destructor que elimina todos los empleados.
- `void agregarEmpleado(T* empleado)`: Agrega un empleado al departamento.
- `void eliminarEmpleado(const string& nombre)`: Elimina un empleado del departamento por su nombre.
- `T* buscarEmpleado(const string& nombre) const`: Busca un empleado por su nombre.
- `void listarEmpleados() const`: Lista todos los empleados.

- `void guardar(const string& archivo) const`: Guarda la lista de empleados en un archivo.
- `void cargar(const string& archivo)`: Carga la lista de empleados desde un archivo.

main.cpp

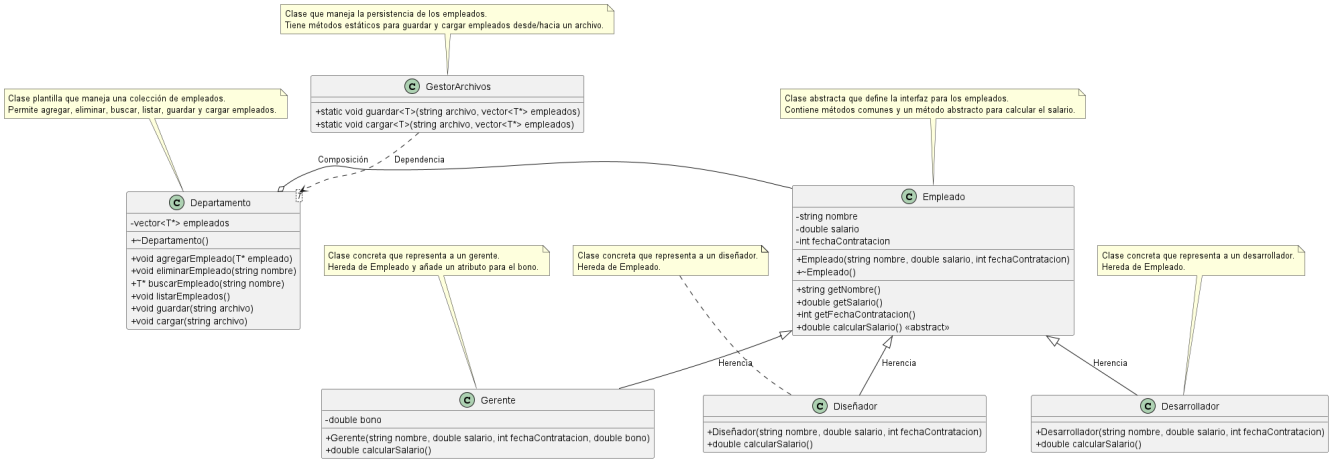
```
#include <iostream>
#include <vector>
#include <algorithm>
#include "clases/Departamento.h"
#include "clases/Empleado.h"


using namespace std;


int main() {

    Departamento<Empleado> sony;

    auto gerente = new Gerente("Alice", 5000,2, "departemente");
    auto desarrollador = new Desarrollador("Bob", 3000, 3);
    auto disenador = new Disenador("Charlie", 3500, 2);

    sony.agregarEmpleado(gerente);
    sony.agregarEmpleado(desarrollador);
    sony.agregarEmpleado(disenador);

    cout << "Lista de empleados antes de ordenar:" << endl;
    sony.listarEmpleados();

    // Ordenar empleados por nombre
    sony.ordenarEmpleados([](Empleado* a, Empleado* b) {
        return a->getNombre() < b->getNombre();
    });

    cout << endl;

    cout << "Lista de empleados despues de ordenar:" << endl;
    sony.listarEmpleados();

    sony.guardar("empleados.txt");


    Departamento<Empleado> nuevoDepartamento;
    nuevoDepartamento.cargar("empleados.txt");

    cout << "Lista de empleados cargados:" << endl;
    nuevoDepartamento.listarEmpleados();
```

```
    return 0;
}
```

El archivo `main.cpp` contiene el punto de entrada del programa. En este archivo se realiza lo siguiente:

- Crear un departamento y agregar empleados de diferentes tipos.
- Listar los empleados antes y después de ordenarlos por nombre.
- Guardar los empleados en un archivo y luego cargarlo en un nuevo departamento.

Diagrama UML



Uso

Para compilar y ejecutar el programa, sigue estos pasos:

1. Guarda todos los archivos fuente en una carpeta llamada SistemaEmpleados.
2. Abre una terminal y navega hasta la carpeta SistemaEmpleados.
3. Compila el programa utilizando un compilador de C++: