Informe de Programación

Trabajo Práctico 1

Franco Agustín Sandri

Paradigmas de la Programación

Introducción:

Como parte de mi trabajo práctico, desarrollé un programa en C++ que simula un sistema de combate por turnos inspirado en juegos de rol. El objetivo fue crear un entorno donde personajes con características únicas, como magos y guerreros, puedan enfrentarse usando armas y estrategias tácticas. El programa está dividido en tres ejercicios: Ejercicio_1 define las clases base y derivadas, Ejercicio_2 implementa un generador aleatorio de personajes y armas, y Ejercicio_3 permite al usuario participar en combates interactivos. A lo largo del desarrollo, enfrenté desafíos relacionados con la gestión de vida, daño efectivo y la lógica de combate, que resolví mediante refactorización y optimización del código. Este informe explica qué hace mi programa, cómo lo construí y las razones detrás de mis decisiones técnicas.

Descripción General del Programa

Mi programa simula un universo de fantasía donde personajes, representados por la interfaz IPersonaje, combaten usando armas (IArma). Los personajes se dividen en dos categorías principales: magos (Mago), con atributos como maná y sabiduría, y guerreros (Guerrero), con fuerza y defensa. Clases derivadas, como Barbaro, Hechicero o Nigromante, añaden habilidades especiales, como furiaSalvaje o vinculoElemental. Las armas, como Espada, Bastón o LibroHechizos, tienen daño base y otras propiedades. El programa permite inicializar personajes, generar combates aleatorios y ejecutar batallas donde el usuario elige movimientos (Fuerte, Rápido, Defensa) que determinan el resultado de cada turno.

Propósito:

- Crear un sistema flexible para simular combates tácticos.
- Aplicar conceptos de programación orientada a objetos, como herencia y polimorfismo.
- Desarrollar una interfaz de consola interactiva que sea clara y funcional.

Estructura:

- **Ejercicio_1**: Define las clases base (IPersonaje, IArma, Mago, Guerrero) y derivadas (Barbaro, Hechicero, Nigromante, Paladin, Caballero, Mercenario, Gladiador, Conjurador, Brujo), junto con armas (Espada, Baston, LibroHechizos, Lanza, HachaSimple, HachaDoble, Garrote, Amuleto, Pocion).
- **Ejercicio_2**: Implementa un generador aleatorio (PersonajeFactory) y un programa principal para visualizar personajes.
- **Ejercicio_3**: Introduce un sistema de combate por turnos con interacción del usuario.

Ejercicio 1: Construcción de las Clases Fundamentales

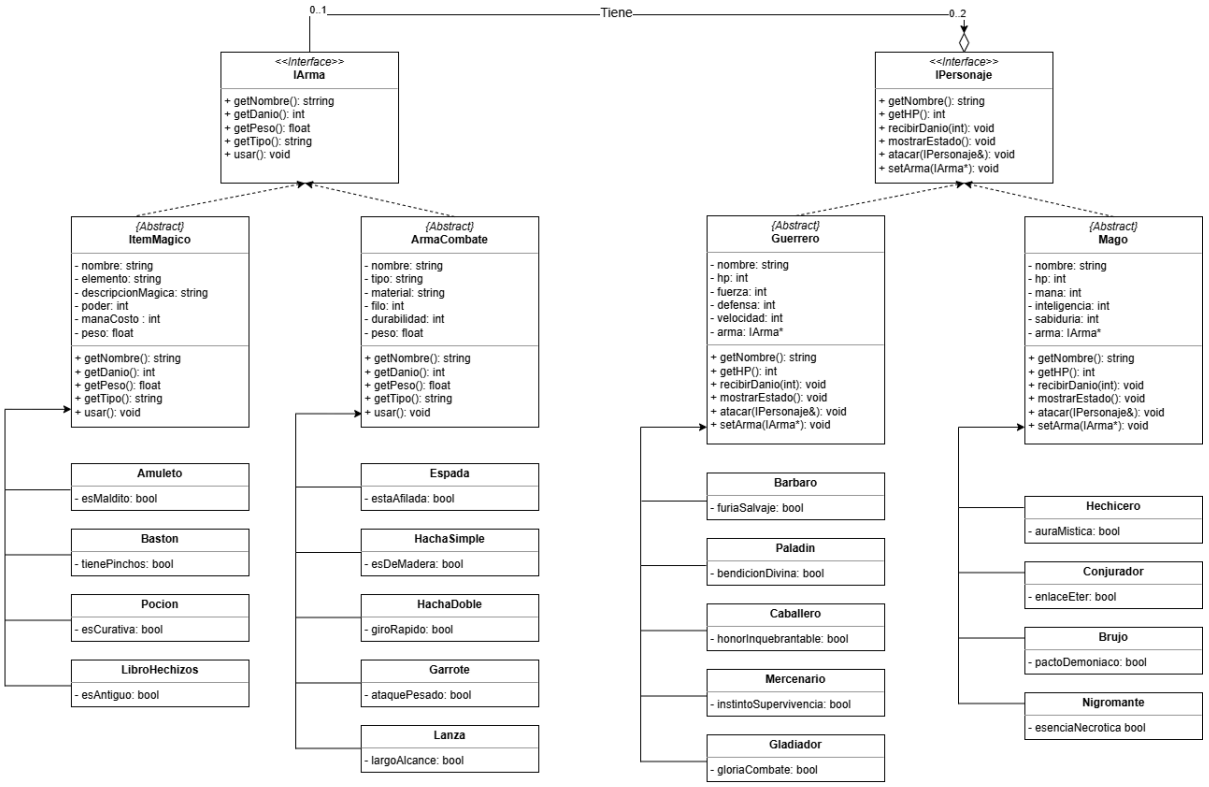
Qué hace:

En Ejercicio_1, construí el núcleo del programa al definir las clases que representan personajes y armas. Organicé el código en directorios headers/ (para interfaces y declaraciones) y src/ (para implementaciones). La interfaz IPersonaje establece métodos esenciales como getNombre(), getHP(), recibirDanio() y setArma(), mientras que IArma define getNombre() y getDanio(). Las clases base Mago y Guerrero proporcionan atributos comunes, y las derivadas añaden características únicas.

Qué hice:

- **Definición de jerarquías:** Implementé la clase base Mago con atributos como *maná*, *inteligencia* y *sabiduría*, y la clase base Guerrero con *resistencia*, *fuerza* y *defensa*. A partir de estas, derivé nueve clases concretas: cinco tipos de guerreros (Caballero, Paladín, Bárbaro, Mercenario y Gladiador) y cuatro tipos de magos (Conjurador, Hechicero, Nigromante y Brujo).
- **Inicialización de personajes:** Diseñé los constructores para asignar valores iniciales a los atributos, partiendo de una base de 100 puntos de vida como indica el punto 3 de la consigna. A estos puntos se les aplica un modificador según una habilidad especial aleatoria (un booleano generado con rand()), que puede sumar o restar 10 puntos de vida al personaje dependiendo de su valor.
- **Gestión del daño:** Implementé el método recibirDanio() para calcular y aplicar el daño que recibe un personaje durante un enfrentamiento.
- **Creación de armas:** Diseñé dos jerarquías de armas: ArmaCombate y ItemMagico. Cada arma se inicializa con un valor fijo de daño base (por ejemplo, una *espada* inflige 14 puntos de daño, mientras que un *bastón* inflige 17). Además, cada arma posee una habilidad especial determinada aleatoriamente (nuevamente, mediante un booleano), que puede incrementar el daño en 2 puntos o reducirlo en 1, con el fin de balancear mejor los combates.

Diagrama UML



Ejercicio 2: Generación Aleatoria de Personajes y Armas

Qué hace:

Ejercicio_2 extiende Ejercicio_1 al introducir PersonajeFactory, una clase que genera personajes y armas aleatorios. El programa principal (main.cpp) crea personajes y armas al azar y muestra su estado usando mostrarEstado().

Qué hice:

- **Implementé PersonajeFactory**: Diseñé métodos estáticos crearMagoRandom(), crearGuerreroRandom() y crearArmaRandom() para instanciar objetos con std::unique_ptr.
- **Creé un programa principal**: En main.cpp, generé personajes aleatorios, le asigné un arma y mostré su estado.

Por qué lo hice así:

- Usé std::unique_ptr para gestionar la memoria automáticamente, evitando fugas y asegurando la liberación de recursos.

- Diseñé PersonajeFactory como una clase estática para centralizar la lógica de creación, facilitando futuras expansiones.
- Mantuve rand() en Ejercicio_2 al igual que en el ejercicio anterior.

Ejercicio 3: Sistema de Combate Interactivo

Qué hace:

Ejercicio_3 implementa un sistema de combate por turnos donde el usuario elige un personaje y un arma, enfrenta a un oponente aleatorio y selecciona movimientos (Fuerte, Rápido, Defensa). La función resolverCombate() determina el ganador de cada turno según una lógica de piedra-papel-tijera, y el daño se aplica usando recibirDanio().

Qué hice:

- **Diseñé la lógica de combate**: En ejercicio3.cpp, creé resolverCombate() para comparar movimientos.
- **Implementé la interacción**: En mostrarMenu(), mostré el estado de los personajes, pedí al usuario un movimiento y simulé el movimiento del oponente.
- **Creé un programa principal**: En main.cpp, permití al usuario elegir un personaje y arma, generando un oponente aleatorio con PersonajeFactory.

Desafíos y soluciones:

- **Vida y daño inconsistente**: Inicialmente,no podía inicializar la vida según las habilidades especiales y no se modificaba la vida, lo que corregí moviendo la lógica al constructor. Lo mismo pasaba con el daño y esto lo corregí implementando en la función getDanio() un condicional que verifica el estado de la variable especial y le sumaba o le restaba puntos de daño.

Integración y Compilación

Qué hice:

- Organicé el proyecto en directorios (Ejercicio_1/, Ejercicio_2/, Ejercicio_3/) para separar responsabilidades.
- El programa compila sin Warnings.

Comandos para compilar:

- cd Ejercicio_1
    - make
    - ./programa
    - make clean

- cd Ejercicio_2
    - make
    - ./programa
    - make clean

- cd Ejercicio_3
    - make
    - ./programa
    - make clean

Resultados y Reflexiones

El programa cumple su propósito: simula combates tácticos con personajes y armas variadas. La salida típica muestra el estado de los personajes, las elecciones de movimientos y el daño efectivo, terminando cuando un personaje llega a hp <= 0.

Aprendizajes:

- La herencia y el polimorfismo me permitieron modelar personajes diversos sin duplicar código.
- Aprendí a gestionar errores de tipos y precedencia.
- La refactorización constante mejoró la claridad y eficiencia del código.

Posibles mejoras:

- Añadir más movimientos o habilidades especiales.

Conclusión

Cada ejercicio aportó una capa de funcionalidad en el desarrollo. Estoy satisfecho con el resultado y considero que por más que hay cosas por mejorar e implementar, se entiende que comprendí los conceptos de POO.

Link al repositorio de Github:

https://github.com/FrancoSandri/TP1_Sandri