Trabajo Práctico 1

I102 - Paradigmas de Programación
Autor: Lautaro Valentín Caminoa

Universidad de San Andrés - 2025

Resumen

Este informe presenta la solución al **Trabajo Práctico 1** de Paradigmas de Programación, que consta de tres ejercicios interrelacionados, en los cuales se implementa un sistema de juego de rol en C++. El **Ejercicio 1** desarrolla una biblioteca de clases para modelar personajes (magos y guerreros) y armas (mágicas y de combate). El **Ejercicio 2** introduce una clase *PersonajeFactory* para generar personajes y armas dinámicamente con números aleatorios. El **Ejercicio 3** implementa un sistema de batalla interactivo basado en el modelo piedra-papel-tijera. El informe muestra la metodología empleada, analiza el cumplimiento de los requisitos, identifica problemas en la implementación con sus respectivas soluciones, detalla los warnings del compilador, y proporciona los comandos de compilación y ejecución.

1. Introducción

El **Trabajo Práctico 1** tiene como objetivo aplicar conceptos de programación orientada a objetos (herencia, polimorfismo, encapsulación, y composición) para diseñar e implementar un sistema de juego de rol. Los tres ejercicios se complementan de la siguiente forma:

- **Ejercicio 1**: Establece la base con clases para personajes y armas, utilizando interfaces y herencia.
- **Ejercicio 2**: Implementa una fábrica que genera personajes y armas de forma aleatoria.
- **Ejercicio 3**: Termina con una batalla interactiva que utiliza lo anterior.

Este trabajo utiliza *std::unique_ptr* para una gestión segura de la memoria, se encuentra modularizado (proyectos separados en directorios *Ej1/, Ej2/, Ej3/*), crea números aleatorios mediante el uso de *std::rand()* y genera ejecutables independientes para cada ejercicio.

2. Metodología

La solución se desarrolló siguiendo un enfoque estructurado:

1. **Diseño Orientado a Objetos**:

    - **Ejercicio 1**: Se diseñaron interfaces (*InterfazPersonaje, InterfazArma*) y clases abstractas (*Guerrero, Mago, ArmaDeCombate, ArmaMagica*) para modelar personajes y armas, asegurando extensibilidad mediante herencia y polimorfismo.
    - **Ejercicio 2**: Se implementó una clase *Factory* estática para crear instancias dinámicas, utilizando composición para asociar armas a personajes.
    - **Ejercicio 3**: Se desarrolló un sistema de batalla interactivo, integrando las clases previas y añadiendo lógica para manejar ataques y daños.
2. **Implementación**:
    - Código modular en directorios separados (*Ej1/, Ej2/, Ej3/*).
    - Uso de *std::unique_ptr* para gestionar memoria.
    - Validación de entradas y manejo de excepciones.
3. **Pruebas y Depuración**:
    - Compilación con *g++ -std=c++17 -Wall -Wextra* para detectar warnings.
    - Ejecución de cada programa (*main2* para Ejercicio 2, *main3* para Ejercicio 3) para verificar funcionalidad.
4. **Documentación**:
    - Inclusión de docstrings y comentarios en el código.
    - Elaboración de este informe para explicar la solución, problemas, y compilación.

3. Descripción de la Implementación

3.1 Ejercicio 1: Biblioteca de Personajes y Armas

3.1.1 Objetivo

Implementar un sistema de clases para modelar:

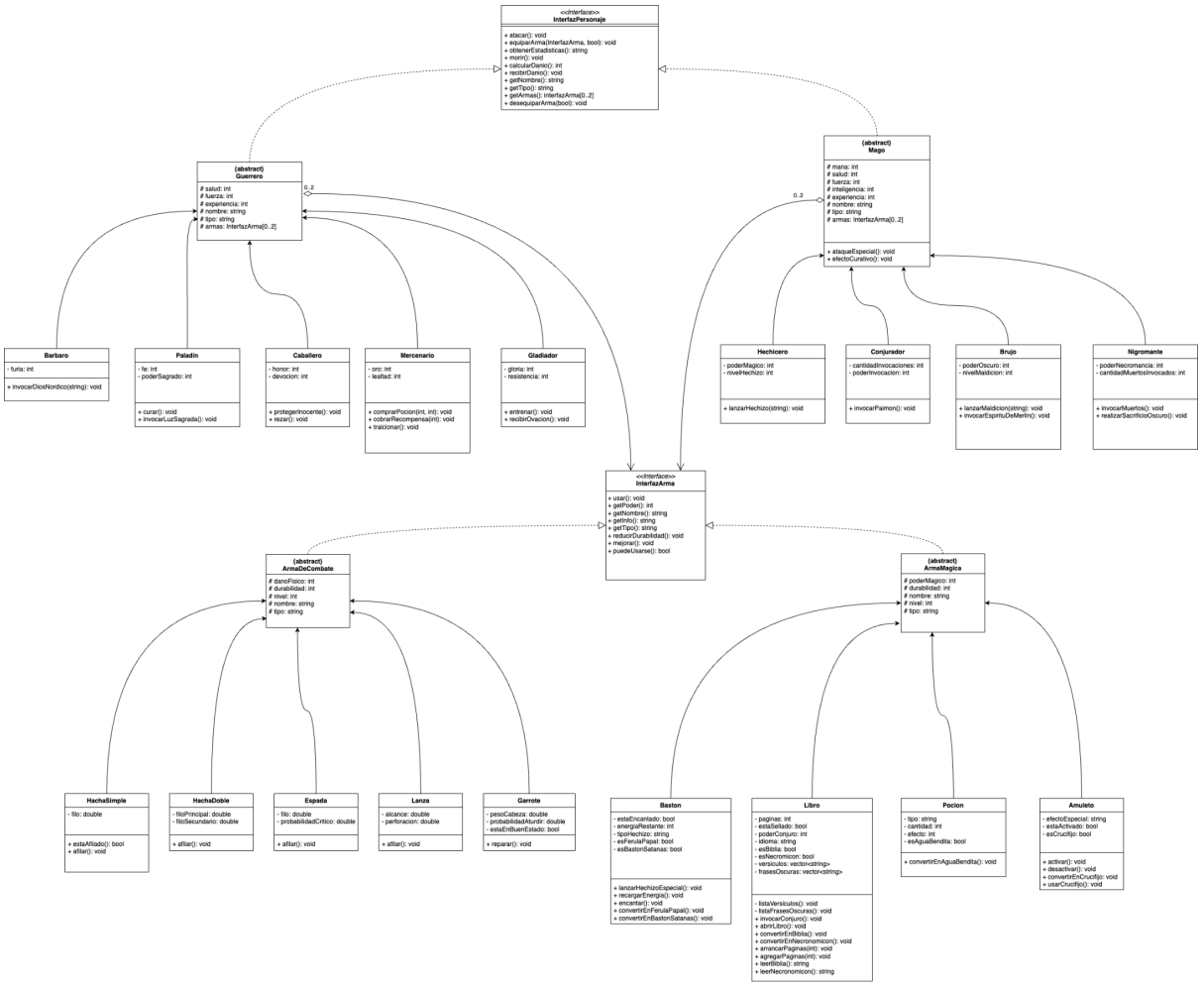
- **Personajes**: Dos grupos (*Guerreros*: Bárbaro, Paladín, Caballero, Mercenario, Gladiador; *Magos*: Hechicero, Conjurador, Brujo, Nigromante), derivados de una interfaz (*InterfazPersonaje*) y clases abstractas (*Guerrero, Mago*), con al menos 5 atributos y 5 métodos por clase derivada.
- **Armas**: Dos grupos (*Armas de Combate*: Hacha Simple, Hacha Doble, Espada, Lanza, Garrote; *Items Mágicos*: Bastón, Libro, Poción, Amuleto), derivados de una interfaz (*InterfazArma*) y clases abstractas *(ArmaDeCombate, ArmaMagica*), con al menos 5 atributos y 5 métodos por clase derivada.
- Relación: Cualquier personaje puede usar cualquier arma (composición).

3.1.2 Implementación

- **Archivos**:

    - *Ej1/Guerreros.hpp, Ej1/Guerreros.cpp*: Implementan *Guerrero* y sus derivadas.
    - *Ej1/Magos.hpp, Ej1/Magos.cpp*: Implementan *Mago* y sus derivadas.

    - *Ej1/ArmasCombate.hpp*, *Ej1/ArmasCombate.cpp*: Implementan *ArmaDeCombate* y sus derivadas.
    - *Ej1/ArmasMagicas.hpp*, *Ej1/ArmasMagicas.cpp*: Implementan *ArmaMagica* y sus derivadas.
    - *Ej1/IPersonajes.hpp*, *Ej1/IArmas.hpp*: Definen interfaces.
- **Características**:
    - **Personajes**:
        - Atributos comunes: *salud*, *fuerza*, *experiencia*, *nombre*, *tipo* (*Guerrero*); *mana*, *inteligencia* adicionales (*Mago*).
        - Atributos específicos: Ejemplo, *Barbaro::furia*, *Nigromante::poderNecromancia*.
        - Métodos: *atacar()*, *equiparArma()*, *calcularDano()*, *obtenerEstadisticas()*, *ataqueEspecial()*, más métodos únicos (ejemplo, *Paladin::invocarLuzSagrada*).
    - **Armas**:
        - Atributos comunes: *danoFisico* o *poderMagico*, *durabilidad*, `nivel`, *nombre*, *tipo*.
        - Atributos específicos: Ejemplo, *Espada::probabilidadCritico*, *Baston::tipoHechizo*.
        - Métodos: *usar(), getPoder(), puedeUsarse()*, *reducirDurabilidad()*, *getInfo()*, más métodos únicos (ejemplo, *Libro::leerBiblia*).
    - **Composición**: *Guerrero* y *Mago* almacenan un vector de *std::unique_ptr<InterfazArma>* (tamaño 2), permitiendo equipar cualquier arma mediante *equiparArma()*.
- **Cumplimiento**:
    - Cada clase derivada tiene ≥5 atributos y ≥5 métodos.
    - Se implementa herencia y polimorfismo.
    - Relación "has-a" modelada con *std::unique_ptr*.
- **UML:**



3.1.3 Compilación

Como el Ejercicio 1 no tiene un archivo main en el que hacer algo, no es necesario compilar y ejecutar nada. Este ejercicio servirá para los próximos.

3.2 Ejercicio 2: PersonajeFactory

3.2.1 Objetivo

Implementa una clase *PersonajeFactory* que:

- Genera dinámicamente personajes y armas usando *std::rand()* (rangos: *[3, 7]* para cantidad de personajes, *[0, 2]* para armas por personaje).
- Usa métodos estáticos y *std::unique_ptr*.
- Modela la relación "has-a" entre personajes y armas.
- Integra las clases del **Ejercicio 1**.

3.2.2 Implementación

- **Archivos**:

- *Ej2/PersonajeFactory.hpp*: Declara la clase *Factory* con métodos estáticos *crearPersonaje* y *crearArma*.
- *Ej2/PersonajeFactory.cpp*: Implementa la creación de personajes y armas.
- *Ej2/main.cpp*: Genera personajes y armas aleatoriamente.

- **Características**:
  - **Clase *Factory***:
    - *crearArma(int tipo)*: Crea armas según *tipo* (0-4: armas de combate; 5-8: armas mágicas).
    - *crearPersonaje(int personaje, vector<unique_ptr<InterfazArma>> armas)*: Crea personajes según *personaje* (0-4: guerreros; 5-8: magos) y equipa armas.
    - Usa *std::unique_ptr* para gestionar memoria.
  - **Aleatoriedad**:
    - Inicializa *std::rand()* con *srand(time(0))*.
    - Genera *[3, 7]* personajes por tipo:
      int cantidadPersonajes = numAleatorio(3, 7);
    - Asigna *[0, 2]* armas por personaje:
      int cantidadArmas = numAleatorio(0, 2);
  - **Integración**: Usa clases de *Ej1/* para instanciar personajes y armas.
- **Cumplimiento**:
  - Métodos estáticos implementados.
  - Usa *std::unique_ptr* y polimorfismo.
  - Genera personajes y armas en los rangos especificados.
  - Modela composición correctamente.

3.2.3 Compilación

Para compilar el ejercicio 2 utilizamos el comando:
g++ -std=c++17 -Wall -Wextra -o main2 Ej2/main.cpp Ej2/PersonajeFactory.cpp Ej1/ArmasCombate.cpp Ej1/ArmasMagicas.cpp Ej1/Guerreros.cpp Ej1/Magos.cpp -IEj1 -IEj2

Ejecución:
El comando de compilación generará un ejecutable llamado “main2”, el cual podrá ser ejecutado utilizando el comando:
./main2

3.3 Ejercicio 3: Batalla Estilo Piedra-Papel-Tijera

3.3.1 Objetivo

Implementar un sistema de batalla entre dos personajes, donde:

- Jugador 1 elige ataques por teclado (Golpe Fuerte, Golpe Rápido, Defensa y Golpe).
- Jugador 2 elige ataques aleatoriamente con *std::rand()*.
- Reglas de daño:
  - Golpe Fuerte > Golpe Rápido: 10 puntos de daño.
  - Golpe Rápido > Defensa y Golpe: 10 puntos de daño.

    - Defensa y Golpe > Golpe Fuerte: 10 puntos de daño.
    - Empate: Sin daño.
- Mostrar tipo de personaje, arma, y daño por ronda.
- Finalizar cuando un personaje llega a 0 HP.

3.3.2 Implementación

- **Archivos**:
    - *Ej3/main.hpp*: Declara *Ataque* (enum), *mostrarMenuPersonajes*, *mostrarMenuArmas*, *calcularDanio*, *ejecutarBatalla*.
    - *Ej3/main.cpp*: Implementa la lógica de la batalla.
- **Características**:
    - **Selección**:
        - Jugador 1 elige personaje y arma mediante menús:

```
mostrarMenuPersonajes();
cout << "Elija su personaje (0-8): ";
int tipoJugador1;
cin >> tipoJugador1;
```

        - Jugador 2 recibe personaje y arma aleatorios:

```
int tipoJugador2 = rand() % 9;
int armaJugador2 = rand() % 9;
```

    - **Ataques**:
        - Jugador 1 elige ataque (1-3), con validación:

```
while (opcionJugador1 < 1 || opcionJugador1 > 3) {
   cout << "Opción inválida. Intente de nuevo: ";
   cin >> opcionJugador1;
}
```

        - Jugador 2 elige aleatoriamente:

```
int opcionJugador2 = rand() % 3 + 1;
```

    - **Daño**:
        - Implementado en *calcularDanio*:

```
int calcularDanio(Ataque ataque1, Ataque ataque2) {
   if (ataque1 == GOLPE_FUERTE && ataque2 == GOLPE_RAPIDO) return 2;
   if (ataque1 == GOLPE_RAPIDO && ataque2 == DEFENSA_Y_GOLPE) return 2;
   if (ataque1 == DEFENSA_Y_GOLPE && ataque2 == GOLPE_FUERTE) return 2;
   if (ataque1 == ataque2) return 0;
   return 1;
}
```

        - Aplica 10 puntos de daño por victoria.
    - **Interacción**:

- Muestra estado, ataques, y resultados:

cout << jugador1->getNombre() << " ataca con " << jugador1->getArmas()[0]->getNombre() << " y hace 10 puntos de daño.\n";

- **Cumplimiento**:
  - Implementa las reglas de batalla correctamente.
  - Integra *PersonajeFactory* y clases de *Ej1/*.

3.3.3 Compilación

Para compilar el ejercicio 3 utilizamos el comando:
g++ -std=c++17 -Wall -Wextra -o main3 Ej3/main.cpp Ej2/PersonajeFactory.cpp Ej1/ArmasCombate.cpp Ej1/ArmasMagicas.cpp Ej1/Guerreros.cpp Ej1/Magos.cpp -IEj1 -IEj2 -IEj3

Ejecución:
El comando de compilación generará un ejecutable llamado “main3”, el cual podrá ser ejecutado utilizando el comando:
./main3

4. Warnings del Compilador

La compilación se realizó con *g++ -std=c++17 -Wall -Wextra*, esto para garantizar el uso de las características modernas de C++17 y activar los warnings detallados que detectan posibles errores, como variables no inicializadas o errores en el código, mejorando la calidad y robustez del código.

A la hora de compilar el trabajo, la totalidad de los warnings fueron corregidos, por lo que no debería haber problema alguna a la hora de compilar y ejecutar.

6. Conclusiones

El **Trabajo Práctico 1** se completó exitosamente, implementando un sistema de juego de rol que cumple con todos los requisitos especificados. El **Ejercicio 1** proporciona una biblioteca de personajes y armas, el **Ejercicio 2** le añade dinamismo con *PersonajeFactory*, y el **Ejercicio 3** genera una batalla interactiva y funcional. Los problemas pudieron ser abordados sin problema, y los warnings del compilador se corrigieron.

Este trabajo me permitió mejorar mi comprensión acerca de la relación entre el código y su representación mediante diagramas UML, fortaleciendo mis habilidades para manejar estructuras de clases, herencias e interfaces. Además, me aportó un mayor dominio en la identificación de detalles como visibilidad, métodos virtuales y relaciones de asociación, así como poder establecer una coherencia entre el código y su diagrama, lo que sin duda me servirá para proyectos futuros.