# Informe – Trabajo Práctico 1
# Paradigmas de Programación

Pizarro, Federico

16 de abril de 2025

## Índice

# 1. Introducción

El objetivo de este trabajo práctico, es trabajar con interfaces, clases abstractas y sus respectivas clases derivadas. También, esto nos ayudará a entender e internalizar los conceptos de las relaciones que existen entre estas clases. Se pide saber analizar un diagrama UML, y también poder crear uno para representar un proyecto. Además debemos investigar acerca de cómo generar y utilizar números aleatorios en C++.

# 2. Consigna y Requisitos

## 2.1. Descripción del Problema

Se pide implementar un juego de rol. Específicamnete, se debe crear una interfaz única, que sirva como base para crear armas. De la interfaz se derivan dos grupos distintos de armas, las cuales se deben implementar como clases abstractas. Cada una de estas clases abstractas tiene distintas derivadas finales. Muy parecido se pide para crear personajes, donde hay una interfaz, luego dos clases abstractas y por ultimo las derivadas finales. Estas derivdadas deben contar con al menos 5 atributos y 5 métodos. Por otro lado, se debe implementar un diagrama UML que respresente estas clases y sus relaciones.

Además, hay que generar números aleatorios en los rangos 0-2 y 3-7. Utilizaremos esto para generar entre tres y siete personajes de cada tipo de ellos. Luego, cada uno de estos personajes puede portar cero, una o dos armas, para ello haremos uso de la implementación de la generación de números aleatorios entre 0-2. Debemos crear una clase fábrica que genere en forma dinámica los personajes y armas, también debe retornar estos objetos sin la necesidad de ser instanciada.

Por último se debe implementar una batalla al estilo piedra, papel, tijera, donde el atacante debe elegir la opción por terminal, y el enemigo lo elegirá de forma aleatoria. Donde un personaje gana, cuando el otro personaje pierde toda su vida.

## 2.2. Requerimientos Técnicos

### Librerías

- `iostream`: Permite el uso de flujos de entrada (`cin`)/salida (`cout`).
- `string`: Manipulación de cadenas de caracteres.
- `memory`: Uso de smart pointers (`std::shared_ptr`, `std::weak_ptr`).
- `vector`: Administrar dinámicamente colecciones de objetos.
- `random`: Generación de números aleatorios.
- `limits`: Obtener las propiedades y límites numéricos de los tipos básicos.
- `stdexcpet`: Permite lanzar excepciones estándar.

### Compilación

- Para compilar este proyecto se utilizó el estándar C++17.
- Archivo Makefile para compilar el proyecto.
- Flags utilizados: `-std=c++17`, `-Wall`, `-g`.
- Ejecución con valgrind para descartar pérdidas de memoria.
- Compilado y ejecutado en WSL.

### Documentación

- Para la generación del informe utilice el formateador de texto LaTeX.
- Los diagramas UML son generados con *draw.io*

# 3. Diseño, Arquitectura e Implementación

Esta sección integra el análisis del diseño y la arquitectura del proyecto junto con los detalles relevantes de la implementación en código, permitiendo ver cómo se tradujeron las decisiones de diseño en funcionalidades concretas.

## 3.1. Diseño y Diagramas UML

#### Interfaces

Las clases más generales del programa son las interfaces `Character` y `Weapon`, en las cuales todos sus métodos son virtuales puros, algunos de ellos se sobrescriben en las clases abstractas y otras directamente en las clases derivadas finales. Se presentan a continuación los diagramas UML de sus estructuras. Todos los métodos implementados en las interfaces son necesarias para cualquier clase derivada, y deben implementarlas. Hay algunos métodos `protected` y la mayoría son `public`. Dado que las interfaces únicamente se utilizan como plantillas para luego implementar clases finales, estas no cuentan con ningún constructor.

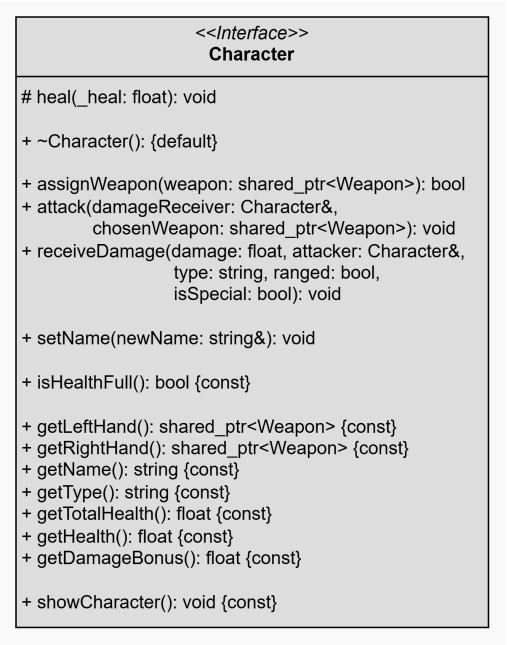


Figura 1: Diag. UML de Personajes

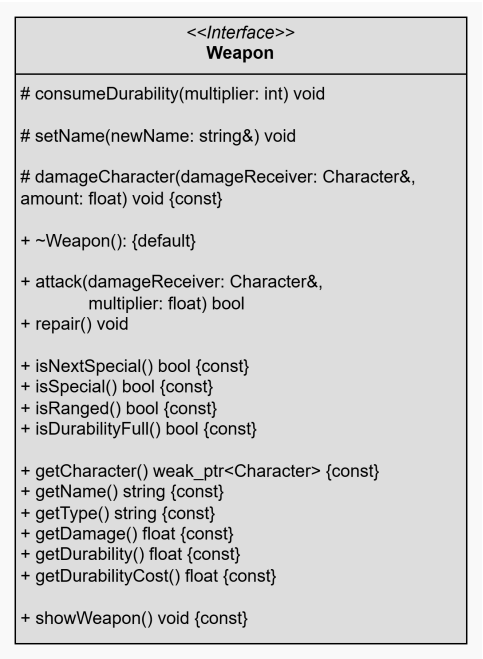


Figura 2: Diag. UML de Armas

### Clases Abstractas

En cada una de estas clases se implementan todos aquellos métodos de las interfaces que funcionan de igual manera para todas las clases derivadas finales. Además se implementa un método virtual puro, lo cuál la convierte en una clase abstracta. Este método virtual no es tan general como para ser declarado en la interfaz general, es decir, solo lo tienen sus propias derivadas, además de que cada una las implementa de manera particular.

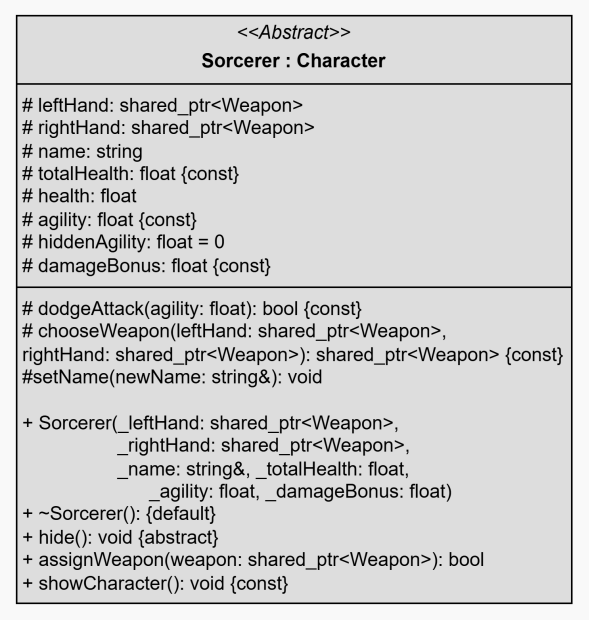


Figura 3: Diag. UML de Magos

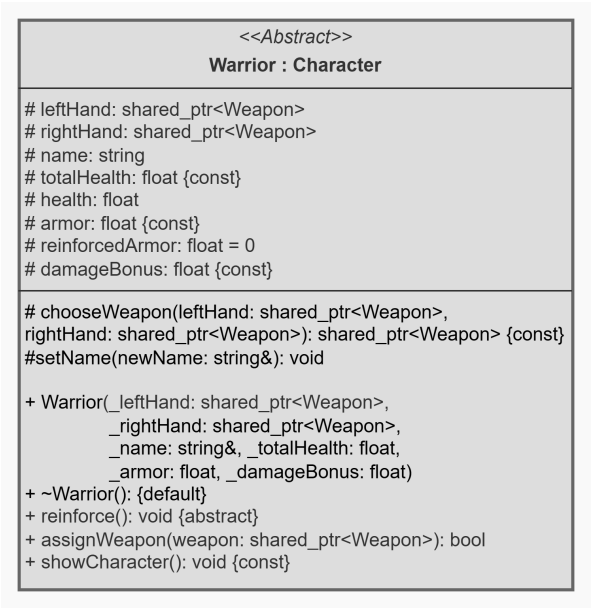


Figura 4: Diag. UML de Guerreros

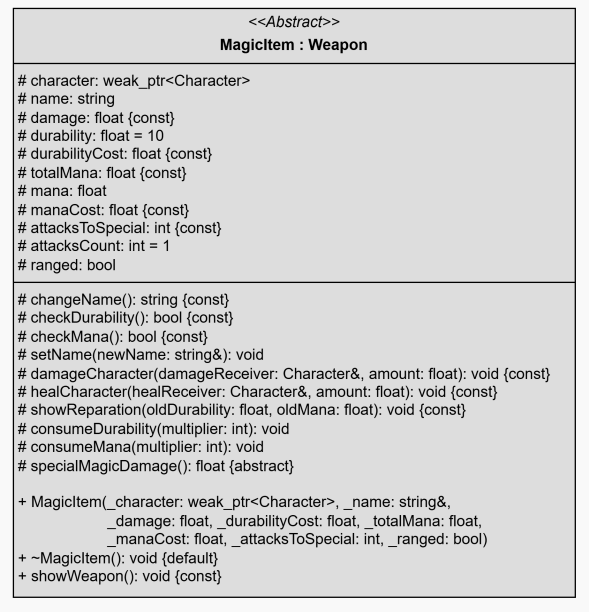


Figura 5: Diag. UML de Items Mágicos

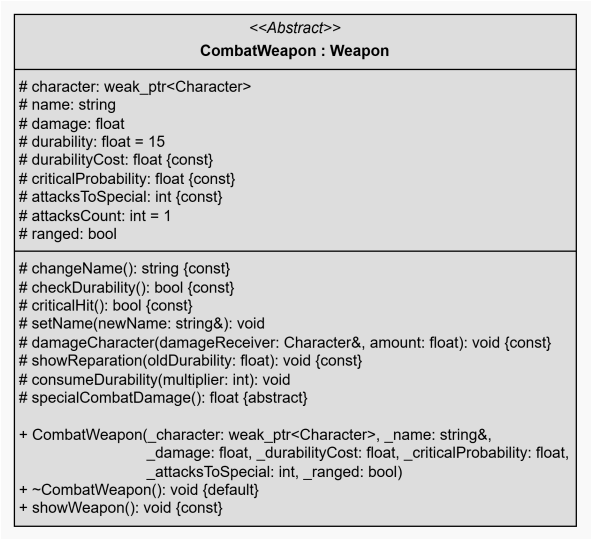


Figura 6: Diag. UML de Armas de Combate

### Clases Derivadas Finales

Estas clases implementan los métodos que pide la interfaz, pero deben ser implementados de manera particular en cada una de ellas, por ejemplo, las lógicas de `attack()`, todos los personajes y armas cuentan con el método para atacar, pero cada derivada final lo hace de distintas manera.

El constructor de cada una de las clases derivadas utiliza el constructor de su clase abstracta. El cual es llamado con algunos parámetros, pero la mayoría son inicializados con valor predeterminados, los cuales son las estadísticas iniciales del objeto, que luego

variar durante el transcurso del juego.

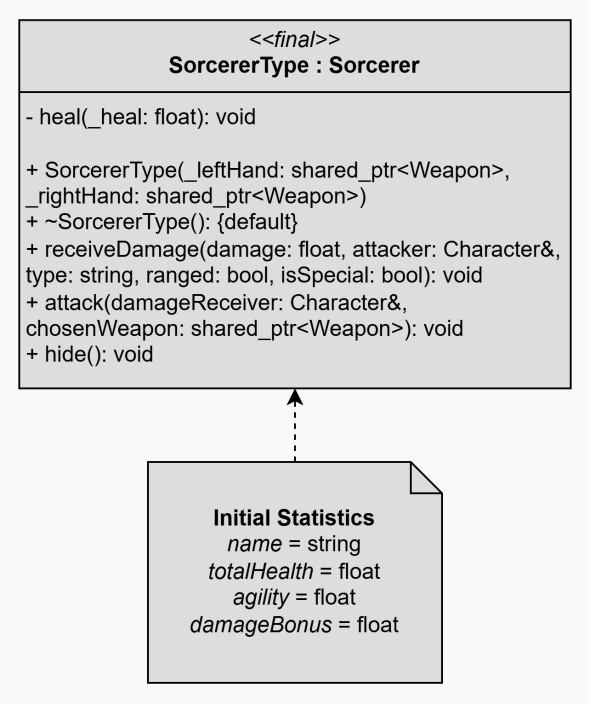


Figura 7: Diag. UML de Derivada Final de Magos

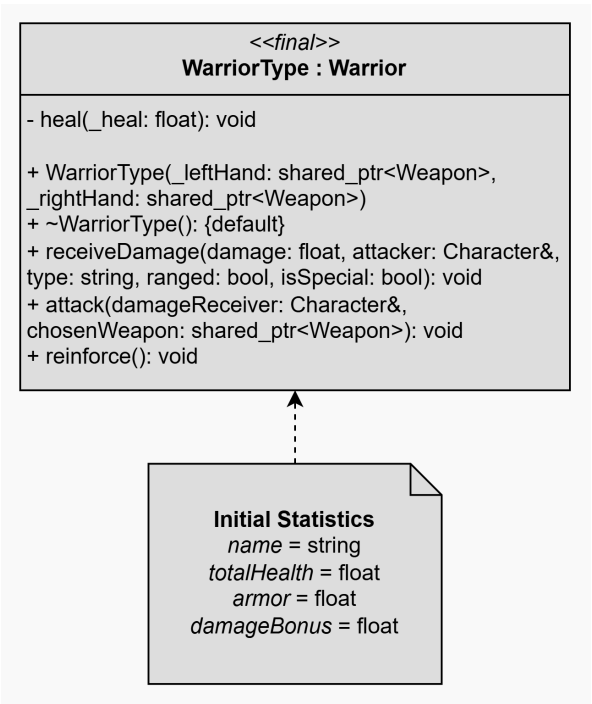


Figura 8: Diag. UML de Derivada Final de Guerreros

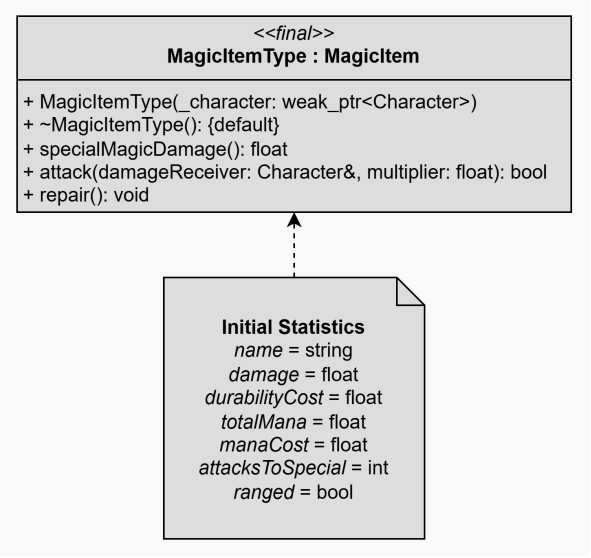


Figura 9: Diag. UML de Derivada Final de Items Mágicos

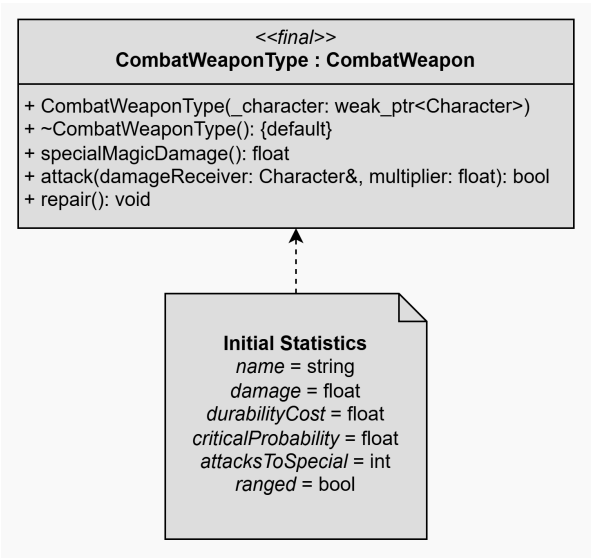


Figura 10: Diag. UML de Derivada Final de Armas de Combate

## 3.2. Decisiones de Diseño y Arquitectura

El trabajo práctico contaba con algunas restricciones con las que cumplir, como por ejemplo que las clases bases sean interfaces, y contar con clases abstractas intermedias de las cuales se terminan implementando clases derivadas finales. Esto al comienzo me

ayudo mucho como guía para comenzar a trabajar, sin embargo durante el desarrollo del proyecto me surgieron ideas que tal vez salían de estos requisitos. También, cualquier personaje debe poder portar y utilizar cualquier tipo de arma. Muchas decisiones e ideas son sacadas de juegos que alguna vez probé.

- Utilicé smart pointers (`shared_ptr` y `weak_ptr`) para garantizar una gestión eficiente de la memoria y evitar fugas.
- Se decidió implementar métodos de la interfaz en las clases abstractas, ya que son distintos para cada clase intermedia, pero luego trabajan de igual manera en las clases derivadas finales.
- Al implementar la clase `CharacterFactory`, para cumplir que la creación sea dinámica sin ser instanciada, declare sus métodos como `static`.
- Para darle sentido a las clases abstractas intermedias decidí darle rasgos de `agility` a los Magos, y de `armor` a los Guerreros. En el caso de las armas, los Items Mágicos cuentan con `mana`, y a las Armas de Combate `criticalProbability`.
- Las clases intermedias debían tener al menos una clase virtual pura para que sea abstracta, por esto cada tipo de arma tiene una función de daño especial (`specialMagicDamage()` y `specialCombatDamage()`), que debe ser implementado de manera particular en cada derivada. De forma similar para los personajes con `hide()` y `reinforce()`.
- Cada personaje y cada arma tiene lógicas de defensa, ataque y manejo de recursos únicos, lo cuál genera un juego dinámico y distinto para cada partida. Existen combos que se potencian, y elecciones que contrarestan al rival.
- Dado que las lógicas de ataque y defensa utilizan métodos y atributos tanto de los personajes como de las armas, asigne punteros compartidos en los personajes que permiten acceder a las armas que porta, mientras que por otro lado utilice un puntero débil (para evitar relaciones circulares) que indica el personaje que la porta.

## 3.3. Implementación y Detalles del Código

Aquí se detalla la implementación del sistema:

- **Personajes:** Se implementaron clases derivadas de `Character` (por ejemplo, `Sorcerer` y `Warrior` y sus especializaciones) utilizando métodos virtuales puros y sobreescritura.
- **Armas:** La interfaz `Weapon` y sus ramas, `CombatWeapon` y `MagicItem`, contienen la lógica de ataque, reparación y consumo de recursos (durabilidad y maná). Se han integrado salidas coloreadas para facilitar la lectura de resultados.
- **Generación de Números Aleatorios:** Se emplea la biblioteca `<random>` para definir rangos en la generación de personajes y armas.
- **Interfaz de Usuario:** El archivo `main.cpp` implementa el flujo del juego, con menús interactivos que permiten iniciar partidas, ver reglas, y alternar turnos de combate.

## 3.4. Integración entre Diseño e Implementación

La integración de la arquitectura y el código se evidencia en:

- La correspondencia directa entre los diagramas UML y la implementación en C++.
- La utilización de smart pointers para mantener la coherencia de las relaciones de composición definidas en el diseño.
- La implementación de la clase fábrica que permite recrear en tiempo de ejecución la estructura del sistema tal como se mostró en los diagramas.
- El uso de colores en las salidas por consola, lo cual fue concebido en el diseño para mejorar la experiencia de usuario.

Este esquema integrado permite entender de forma global cómo las decisiones de diseño se tradujeron en una implementación funcional y modular, facilitando la ampliación o modificación futura del sistema.

---

# 4. Diseño y Arquitectura

## 4.1. Diagrama UML de Personajes

## 4.2. Diagrama UML de Armas

Se explica el diagrama UML correspondiente a la interfaz `Weapon` y sus derivados (`CombatWeapon` y `MagicItem`). Se deben destacar las relaciones de herencia, los métodos clave (por ejemplo, `attack`, `repair`, `showWeapon`) y la importancia de emplear smart pointers para la administración de memoria.

# 5. Implementación

## 5.1. Implementación de Personajes

Se detalla la implementación de las clases de personajes, explicando el uso de métodos virtuales puros, la sobreescritura en clases derivadas y la integración de capacidades especiales como `hide()` o `reinforce()`.

## 5.2. Implementación de Armas

Aquí se describe cómo se implementó la interfaz `Weapon` y las dos ramas principales: armas de combate y ítems mágicos. Se explica la lógica de ataque, consumo de durabilidad y maná, y la incorporación de colores en los outputs para mejorar la presentación.

## 5.3. Fábrica de Personajes y Armas

Se explica la función de la clase `CharacterFactory` para la creación dinámica de personajes y armas. Se hace énfasis en el uso de smart pointers y en cómo se implementa la lógica de armado (compromiso de composición "has-a").

## 5.4. Interfaz de Usuario y Flujo del Juego

Esta sección detalla el funcionamiento de `main.cpp`:

- El menú principal permite iniciar una nueva partida, ver las reglas o salir.
- Durante el combate, se alternan los turnos entre jugador y rival. Se implementan opciones para atacar, usar habilidades especiales y rendirse.
- Se utiliza el cambio de colores en los mensajes (por ejemplo, para daño, salud y consumos) para mejorar la legibilidad.

# 6. Resultados y Discusión

Se describen los resultados obtenidos al compilar y ejecutar el juego. En esta sección se puede incluir:

- Comentarios sobre la estabilidad y rendimiento del sistema.

- Dificultades encontradas durante la implementación.
- Posibles mejoras o extensiones futuras.

# 7. Conclusiones

El trabajo práctico me parecio interesante y divertido, ya sea por la temática, que es la implementación de un juego, o por la libertad que esto te permite, ya que le puedes dar tus propias ideas a cada personaje y armas. Le dediqué bastante tiempo, pero por que me atrapo.

Creo que me ayudo a internalizar un poco más las nociones de clases, en particular cómo funcionan y para qué sirven las interfaces y clases abstractas. También continué trabajando con *smart pointers*.

Le dí importancia a declarar como `const` a todas las variables, atributos y métodos necesarios, para asegurarme de que en ninguna parte del código modificarlos sin darme cuenta.

Conocí lo que es un diagrama UML, aprender a leer y hacer uno básico me pareció interesante e importante. Conocí *draw.io* que me permitió generar los diagramas UML.

Por otro lado, este es el segundo informe que hago con LaTeX, ya que me parece importante saber utilizarlo. Además creo que se nota la diferencia de un trabajo hecho en LaTeX, tanto para presentarlo, como las facilidades que presenta para generarlo.

# 8. Referencias

Se enlistan las fuentes bibliográficas, documentos o recursos utilizados durante el desarrollo del proyecto.


- *cppreference.com* (Documentación de C++)
- *draw.io* (Software para generar Diagramas UML)
- Material y guías de la materia de Paradigmas de Programación.
- Recursos sobre UML.


# A. Código Fuente

En este anexo se puede incluir el código fuente completo o fragmentos relevantes, comentando brevemente cada parte importante o justificando las decisiones de diseño.