# Trabajo Práctico 1

II02 – Paradigmas de Programación

**Autor:** Luca Fassio

**Universidad:** Universidad de San Andrés (UdeSA)

**Año:** 2025

# Introducción

Este informe tiene como objetivo implementar una arquitectura de clases que simule un juego de rol aplicando los fundamentos de la OOP, haciendo un uso de jerarquías de clases, herencia, polimorfismo, encapsulamiento y composición. Se trata de una implementación que simula un entorno de combate entre personajes diferenciados por su rol, habilidades, armas y efectos.

El núcleo del sistema está compuesto por dos jerarquías principales: la de personajes (`Character`) y la de armas (`Weapon`). A partir de estas interfaces, se desarrollan clases especializadas que representan distintos tipos de personajes (como guerreros o magos), luego de estos se crean subtipos concretos con comportamientos específicos, tales como `Barbarian`, `Paladin`, `Necromancer` o `Warlock`, cada uno con habilidades distintivas. Lo mismo ocurre con las armas, que se dividen en mágicas y de combate, derivando en objetos específicos como `Sword`, `Axe`, `Staff` o `Potion`.

Para enriquecer el sistema se implementan aspectos más avanzados como la gestión de efectos mágicos y físicos (por ejemplo, burning, poison, freezing), la composición de equipos (`class Team`), y una clase fábrica (`Factory`) que permite la creación controlada y centralizada de instancias de manera estática. Asimismo, se hace uso de punteros inteligentes (`shared_ptr` y `unique_ptr`) para garantizar una gestión eficiente y segura de la memoria, minimizando riesgos de fugas o errores de *ownership*.

A continuación haré un recorrido de manera ordenada la descripción funcional de cada clase, sus responsabilidades, atributos y métodos relevantes, estableciendo claramente sus relaciones e interacciones dentro del sistema.

# Compilación

Para compilar y ejecutar el código correctamente, es necesario contar con el compilador `g++` y la herramienta `make` instalados en el sistema. Se recomienda ejecutar los ejercicios 2, 3 y 4 desde el directorio principal del Trabajo Práctico, ya que el `Makefile` está diseñado para gestionar eficientemente la estructura de carpetas del proyecto.

El uso del `Makefile` centralizado no solo simplifica la compilación, sino que también es imprescindible para el correcto funcionamiento del sistema de generación de nombres aleatorios. Dicho sistema depende de la lectura de archivos de texto ubicados en rutas relativas al directorio de ejecución. Compilar desde un subdirectorio o desde una ubicación distinta puede provocar errores de compilación, o impedir que se localicen correctamente los archivos `.txt` necesarios para la generación de nombres.

Por lo tanto, se sugiere encarecidamente seguir la estructura propuesta y ejecutar los comandos de compilación desde la raíz del proyecto para evitar inconvenientes.

## Comandos disponibles

A continuación se presenta una lista de los comandos disponibles para compilar y ejecutar el trabajo, junto con una breve descripción de su funcionalidad:

- `make all` Compila todo el código necesario del proyecto. Es normal que este proceso tome un tiempo considerable la primera vez.
- `make runX` Vuelve a compilar **únicamente** los archivos que han sido modificados desde la última compilación y ejecuta automáticamente el ejercicio `X`.
  *Ejemplo:* `make run3` ejecuta el ejercicio 3.
- `bin/./EjX` Ejecuta la última versión compilada del ejecutable correspondiente al ejercicio `X`.
  *Ejemplo:* `bin/./Ej2` ejecuta el ejercicio 2.
- `make clean` Elimina todos los ejecutables y archivos objeto generados durante el proceso de compilación. Es útil para limpiar el entorno antes de una nueva compilación completa.

# Aclaraciones

El trabajo fue testeado a lo largo de todo su desarrollo con las flags `-Wall` y `-Wextra`. En su versión final, no se presenta ningún warning ni problemas de compilación.

Además, las carpetas y archivos están estructuradas siguiendo el formato inicial del trabajo, previo a la eliminación del ejercicio original 1. Por lo tanto, se considera que los ejercicios 2, 3 y 4 son en realidad los actuales 1, 2 y 3 respectivamente.

De todas formas, como en el momento de eliminar el ejercicio 1 este ya estaba terminado (aunque posiblemente sin estar completamente pulido), se decidió conservarlo, ya que funcionaba correctamente y no requirió posteriores modificaciones. Para ejecutar este ejercicio es necesario ubicarse en la carpeta `Ej1` (ejecutando `cd Ej1`) y luego utilizar el comando `make`. Para regresar al directorio principal del trabajo, se puede usar `cd ..`.

## Diablo 5

Por último, cabe aclarar que el archivo `main.cpp` general ubicado fuera de las carpetas de los ejercicios no se encuentra operativo actualmente (se encuentra completamente comentado). Esto no constituye un error, sino que corresponde a un proyecto en desarrollo que fue suspendido por falta de tiempo. No obstante, se incluye el avance realizado hasta el momento, ya que presenta un enfoque interesante y un nivel de desarrollo suficiente para ser considerado un aporte.

En caso de querer probar esta versión preliminar del juego, se deben descomentar tanto el archivo `main.cpp` ubicado en la raíz del proyecto como todos los archivos contenidos en `utils/HUD`. Luego, basta con ejecutar el comando `make runjuego` para iniciar la prueba.

Este proyecto consiste en un combate `5v5` en el cual se pretendía implementar todas las funcionalidades desarrolladas en el Ejercicio 2. La motivación fue aprovechar al máximo el potencial del sistema de clases y métodos creados, integrándolos en una experiencia más completa.

# Ejercicio 1 - Implementación de armas y personajes

Este ejercicio introduce la base funcional del sistema de combate. Aca es donde se implementaron las clases esenciales relacionadas con la jerarquía de armas y personajes, junto con sus funcionalidades principales como ataques, efectos, habilidades y relación entre objetos.

## Interfaces

Las interfaces constituyen la base abstracta del sistema, estableciendo el conjunto de métodos que deben implementar todas las clases derivadas.

- **Character**: es una interfaz abstracta que define el comportamiento básico que deben tener todos los personajes. Entre los métodos que declara están los getters principales de cada personaje (`getHealth()`, `getType()`, `getName()`), funcionalidades cruciales como `useWeapon()`, `heal()` o `receiveDamage()`, el manejo del inventario y además se declaran la mayoría de los métodos que llevan a cabo la implementación del sistema de efectos.
- **Weapon**: es una interfaz que representa una abstracción general de un arma. Define comportamientos esenciales como `getName()`, `attack()` y `use()`, que permiten a las clases derivadas distinguir entre armas de combate físico y armas mágicas.

Estas interfaces son fundamentales para permitir el polimorfismo, facilitando que diferentes tipos de personajes y armas puedan interactuar bajo un mismo contrato de comportamiento. Además, garantizan que las clases derivadas cumplan con una estructura funcional coherente dentro del sistema.

## Clases abstractas

El sistema cuenta con cuatro clases abstractas principales que sirven como base para derivar las clases concretas con funcionalidades específicas. Estas definen atributos y comportamientos comunes, y su objetivo mas que nada es facilitar la reutilización de código y la extensibilidad del sistema:

- **Warrior**: representa a los personajes centrados en el combate directo. Estos se diferencian de los magos porque resisten más por su atributo armor. Sin embargo, no saben usar armas mágicas como el Spellbook o el Staff.
- **Mage**: es la base para todos los personajes mágicos, establece los elementos comunes como el manejo de maná y la interacción con efectos especiales, además de habilidades únicas y la posibilidad de usar armas mágicas.
- **Combat**: clase abstracta para armas físicas. Utiliza atributos como el daño, durabilidad y material, permitiendo una base común para armas como espadas o hachas.
- **Magic**: la parte divertida de `Weapon`, agrupa todas las armas mágicas, estableciendo propiedades comunes como duración de efectos y el nombre del ítem.

## Personajes concretos

A partir de las clases abstractas `Warrior` y `Mage`, se desarrollaron diez clases concretas que representan personajes jugables con habilidades y comportamientos específicos. Estas clases enriquecen el sistema de combate aportando diversidad táctica y de efectos.

**Guerreros:**

- `Barbarian`
  Cada ataque del bárbaro trae consigo un 20% de probabilidades de activar el efecto de `RAGE` por 2 turnos, con el que su daño se verá aumentado x1.7. (inspirado de: Clash of Clans)

- `Gladiator`
  Junto al `Barbarian`, es uno de los únicos personajes que tienen un 20% de probabilidades de contraatacar al rival que los está atacando. Además, posee una habilidad especial que se activa automáticamente cuando es el único personaje en pie de su equipo. Esta habilidad no se desactiva (ya que no puede revivir nadie si solo hay un gladiador vivo) y le otorga aumento de daño, mayor resistencia y chances de reducir o incluso anular completamente el daño recibido.

- `Knight`
  Posee una habilidad que le permite mejorar considerablemente sus estadísticas. Esta mejora no tiene una duración determinada: se mantiene activa hasta que el caballero muere o la partida finaliza. Además, la habilidad es acumulable y puede volver a usarse cada vez que se cumple el turno de *cooldown* que trae.

- `Mercenary`
  El `Mercenary` tiene la capacidad de cambiar de equipo durante la partida, una estrategia pensada principalmente para situaciones donde su equipo original va perdiendo. Además, puede hacerse invisible, pero esto conlleva un 30% de probabilidades de que abandone la batalla por completo. También existe una posibilidad de que escape si su vida cae por debajo del 30%, en cuyo caso la probabilidad baja a 20% para que huya.

- `Paladin`
  El `Paladin` cumple un rol de soporte dentro del equipo. Puede curar a dos compañeros por turno (puede repetir el mismo objetivo) o, alternativamente, activar un escudo especial que lo protege de los próximos 5 golpes que reciba, anulando completamente su daño.

**Magos:**

- `Conjurer`
  El `Conjurer` es un mago versátil que puede aplicar efectos mágicos a los rivales, invocar un escudo protector que lo defiende durante un turno y potenciar su poder ofensivo mediante un hechizo que aumenta su daño, aunque esto le cuesta una porción de su vida.

- `Necromancer`
  El `Necromancer` se especializa en el control de la muerte. Tiene la capacidad de revivir a un aliado caído, invocar un esqueleto para que luche de su lado (este está pensado para que ataque aleatoriamente en un futuro) y drenar vida directamente de un enemigo para curarse a sí mismo.

- `Larry`
  Este es el esqueleto que invoca el `Necromancer`. Por fines prácticos se declara con `type = NECRO`, pero esto no afecta su funcionamiento ya que se sobreescribe el método correspondiente para que devuelva `"Skelly"`.

- `Sorcerer`
  El `Sorcerer` es un mago ofensivo que puede cambiar el tipo de magia que utiliza, alternando entre distintos elementos como fuego, agua, aire y tierra. Dependiendo del tipo seleccionado, sus ataques aplican distintos efectos elementales al enemigo.

- `Warlock`
  El `Warlock` es un mago de soporte avanzado. Su habilidad principal, `Soul Link`, conecta al equipo compartiendo el daño recibido entre todos los aliados. Además, cuenta con `Born Again`, una habilidad que puede usarse una sola vez por partida para revivir a todos los aliados caídos, restaurando un porcentaje de vida que se reparte equitativamente: 100 dividido entre la cantidad total de resurgidos. (inspirado de: Adam Warlock - Marvel Rivals)

## Armas concretas

**Armas de combate:**

- `Axe`
  El `Axe` inflige daño moderado. Su peso está definido por el material con el que fue creada, lo cual también afecta su daño total.
- `Basto`
  El `Basto` es un arma de bajo daño base que puede ser reforzada con distintos materiales para mejorar tanto su durabilidad como el daño que inflige. Es ideal para estrategias de desgaste y con un refuerzo puede llegar a hacer daño comparable a armas de combate mejores.
- `DoubleAxe`
  El `DoubleAxe` es una versión más pesada y poderosa del hacha común. Tiene una probabilidad adicional de realizar un golpe crítico dependiendo del material del que está hecha la cual representa un "segundo golpe".
- `Spear`
  El `Spear` tiene una longitud que deberia influir en su versatilidad, sin embargo no me permitía hacer mucho que el juego este en una terminal. Siendo sincero, no se me ocurrio nada original para esta.
- `Sword`
  El `Sword` es el arma mas común de las de combate, buen daño y la posibilidad de aumentarlo afilandola.

**Armas mágicas:**

- `Amulet`
  El `Amulet` aplica efectos mágicos pasivos o instantáneos sobre aliados o enemigos, dependiendo del tipo de propiedad con la que fue creado por una cierta cantidad de turnos.
- `Potion`
  La `Potion` puede ser utilizada sobre uno mismo o arrojada a distancia. Al lanzarla, se activa un efecto específico dependiendo de su tipo.
- `Spellbook`
  El `Spellbook` permite lanzar distintos hechizos ofensivos, ademas, permite invocar una copia del portador (un clon de toda la vida) para que pelee a su lado.
- `Staff`
  El `Staff` es el arma mágica más compleja. Permite realizar una variedad de hechizos mas complejos que el resto:
  - `Storm Chain`: encadena daño entre tres enemigos distintos con una reducción progresiva (100%, 50%, 25%).
  - `Mystic Echo`: pensado para el juego grande, guarda el movimiento del portador y lo repite en el siguiente turno dando la posibilidad de hacer dos movimientos en un mismo turno.
  - `Staff of Echoes`: cura al aliado con menos vida, le otorga regeneración y un buff dependiendo si es guerrero, aumenta el daño, o mago, regenera mana.
  - `Chaotic Flare`: lanza un ataque caótico que de manera aleatoria elige 5 objetivos del equipo contrario (vivos o muertos) y con una probabilidad de 50/50 aplica daño directo o agrega un efecto BURNING.

## Relaciones entre clases

Para la implementacion del sistema de combate, se plantearon algunas relaciones entre las clases, algunos ejemplos a continuacion:

- La clase `Character` se relaciona con `Weapon` mediante una composición, pudiendo tener entre 0 y 2 armas simultáneamente. Estas armas se administran desde la propia clase mediante el pair¡uniqueptr¡Weapon¿, uniqueptr¡Weapon¿¿. Se utilizaron uniques ya que estos describen la exclusividad que tiene Character sobre el ownership de cada Weapon (pueden creer que en Latex no se pueden poner guiones bajos, por eso "uniqueptr").
- Las clases derivadas `Warrior` y `Mage` tienen relaciones de amistad (`friend class`) con ciertas armas mágicas que necesitan acceder a sus atributos internos para ejecutar efectos especiales:
  - `Warrior` es `friend` de `Potion` y `Staff`.
  - `Mage` también es `friend` de `Potion` y `Staff`.
  - `Mage`, además, es `friend` de `Spellbook`.

  Estas relaciones permiten que las armas accedan directamente a los atributos protegidos de los personajes para modificar atributos de `Warrior` y `Mage` desde el arma. Para una mejor visualizacion de las relaciones entre las clases ver Figura 1, UML de clases que representa las relaciones entre cada una.

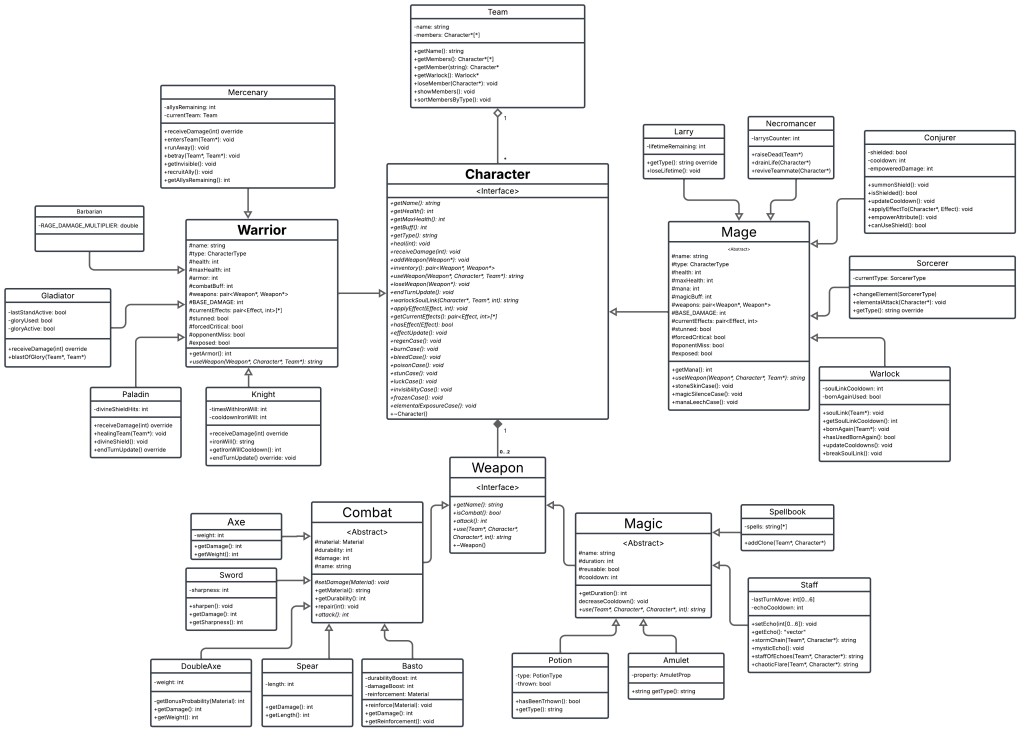


Figure 1: Diagrama general del sistema de clases

# Ejercicio 2 - Creación de equipos

En esta etapa se amplió la funcionalidad del sistema introduciendo una nueva clase llamada `Team`, que permite agrupar personajes en estructuras lógicas y funcionales. Esto permite crear escenarios de combate mas realistas entre equipos enfrentados.

## Clase Team

La clase `Team` representa un conjunto de personajes que comparten un mismo bando. Para este ejercicio se eligieron 2 numeros entre 3 y 7 que representan la cantidad de warrios y Mages. Cada personaje puede tener entre 0 y 2 armas asignadas. La clase ofrece métodos para acceder a los personajes por nombre, perder miembros, y organizar la estructura interna del equipo.

## Clase Factory

Para facilitar la creación de personajes y equipos, se implementó la clase `Factory` como una clase completamente estática. Su propósito es centralizar y automatizar la construcción de objetos del sistema, mejorando la organización general del código.

Entre sus funciones principales se encuentran:

- Crear personajes a partir de un tipo y un nombre.
- Crear armas a partir de un tipo y sus parámetros (material, efecto o propiedad) ademas de asignarsela a alguien.
- Asignar personajes a un equipo.

Gracias a esta arquitectura, es posible construir de forma sencilla un equipo con entre 3 y 7 personajes, en el main se pone a prueba creando 2 equipos, uno a mano y el otro random (tranquilo que los nombres se ponen solos de manera aleatoria y no vas a tener que pensar en ninguno).

# Ejercicio 3 - Simulación de combate entre dos personajes

En este ejercicio se desarrolla una versión simplificada de combate utilizando una lógica inspirada en un piedra-papel-tijera. La implementación pone en juego la interacción entre personajes y armas en un ciclo de turnos hasta que uno de los participantes es derrotado.

## Sistema de combate

Cada ronda de combate consiste en que ambos jugadores eligen un ataque entre tres posibles:

- `Golpe Fuerte`
- `Golpe Rápido`
- `Defensa y Golpe`

La resolución de los ataques sigue una relación cíclica tipo piedra-papel-tijera:

- `Golpe Fuerte` vence a `Golpe Rápido`
- `Golpe Rápido` vence a `Defensa y Golpe`
- `Defensa y Golpe` vence a `Golpe Fuerte`

En caso de empate (misma acción), no se realiza daño y se pasa al siguiente turno.

Cada ataque exitoso inflige 10 puntos de daño. El juego finaliza cuando uno de los personajes alcanza 0 puntos de vida.

## Interacción con el usuario

El jugador 1 elige:

- El nombre de su personaje.
- Un personaje jugable de la lista disponible.

El jugador 2 es generado aleatoriamente.

## Implementación

El combate se gestiona a través de una función principal `fight()`, que realiza los siguientes pasos en bucle:

1. Muestra la salud actual de ambos jugadores.
2. Solicita al jugador 1 que elija un tipo de ataque.
3. Genera aleatoriamente la acción del jugador 2.
4. Evalúa el resultado del enfrentamiento según las reglas del juego.
5. Aplica el daño correspondiente al personaje perdedor.
6. Repite el ciclo hasta que uno de los dos pierda todos sus puntos de vida.