Informe del Trabajo Práctico 1

I102 – Paradigmas de Programación
Grupo: G1
Universidad de San Andrés
Fecha de entrega: 16 de abril de 2025
Nombre del estudiante: Gonzalez Tomas

Resumen Ejecutivo

Este informe presenta una solución integral al Trabajo Práctico 1 de la materia I102 – Paradigmas de Programación. El trabajo se dividió en tres ejercicios secuenciales: (1) el diseño e implementación de un sistema de jerarquías de personajes y armas, (2) la creación de una fábrica que genera instancias dinámicas de estos elementos, y (3) un simulador de batalla por consola que los pone en interacción (con condiciones iniciales seteadas por la consigna).

Introducción

El trabajo práctico fue planteado como un proyecto integral que permitiera aplicar los conceptos de programación orientada a objetos en un entorno simulado de combate entre personajes. A lo largo del desarrollo se hizo uso del lenguaje C++, con énfasis en el diseño de clases abstractas, interfaces, uso de punteros inteligentes, modularidad mediante archivos de cabecera y fuentes, y control de compilación a través de Makefiles.

Estructura del Proyecto

El proyecto se organizó en tres carpetas independientes: `EJ1`, `EJ2` y `EJ3`, cada una con su propio `Makefile`. `EJ1` contiene la implementación de personajes y armas, dividida en carpetas `Headers`, `Sources` y `Utilities`. `EJ2` define la fábrica de personajes (`PersonajeFactory`) y reutiliza el código del ejercicio anterior. `EJ3` contiene el simulador de batalla. Los Makefiles se diseñaron para compilar todos los archivos necesarios de cada ejercicio sin conflictos de dependencias.

Flags y Compilación

Los programas fueron compilados con los flags -Wall -Wextra sin haber ningún warning. Por otro lado, para compilar cada ejercicio, se deberá dirigir a la carpeta deseada y ejecutar el comando “make”, ya que todos los ejercicios cuentan con un makefile y un main.cpp . Esto genera un ejecutable que se elimina luego de finalizar el programa.

Aclaraciones

En repetidas partes del informe opté por abreviar la escritura de las interfaces, por ejemplo, la InterfazDePersonaje aparecerá en algunos casos como IPersonaje. El motivo de esto fue mantener las definiciones en un solo renglón, que a mi parecer, sumaban prolijidad.

Ejercicio 1: Jerarquía de personajes y armas

El primer ejercicio del Trabajo Práctico tuvo como objetivo central la implementación de una jerarquía de clases para un sistema de rol, considerando dos grandes conjuntos de entidades: personajes y armas. Se adoptó un enfoque orientado a objetos, haciendo uso extensivo de herencia, clases abstractas e interfaces para garantizar la escalabilidad, la reutilización de código y la separación clara de responsabilidades.

Tanto las armas como los personajes se estructuraron partiendo de interfaces puras (`InterfazArma` e `InterfazPersonajes`, respectivamente). Estas interfaces definen las operaciones básicas que deben implementar todas las clases concretas derivadas, como `usar()`, `aplicarEfecto()` o `mostrarInfo()`. A partir de estas interfaces, se diseñaron dos clases abstractas intermedias en cada jerarquía: `ArmaMagica` y `ArmaCombate` para las armas, y `Mago` y `Guerrero` para los personajes.

Este diseño permite encapsular comportamientos comunes para cada tipo (mágico o físico), y luego delegar la implementación específica a las clases concretas como `Baston`, `Espada`, `Hechicero`, `Caballero`, entre otras. Se buscó mantener alta cohesión en cada clase y bajo acoplamiento entre jerarquías, facilitando así futuras extensiones o modificaciones.

Interfaz de Arma (InterfazArma)

La interfaz "InterfazArma" define el comportamiento base que deben cumplir todas las armas del sistema, ya sean mágicas o físicas. A continuación se detallan los métodos definidos, todos ellos públicos y declarados como virtuales puros.

`virtual string getNombre() const = 0;`
Devuelve el nombre del arma (por ejemplo: *"Espada"*, *"Bastón"*, *"Amuleto"*).

`virtual string getTipo() const = 0;`
Informa el tipo de arma: *"mágica"* o *"física"*. Este dato es fundamental para evaluar la compatibilidad con el tipo de personaje que la utiliza.

`virtual Rareza getRareza() const = 0;`
Retorna la rareza del arma, definida mediante un `enum class` con valores como COMÚN, RARO, ÉPICO, etc.

`virtual int getDmgBase() const = 0;`
Devuelve el valor base de daño que realiza el arma.

`virtual int getTasaCritico() const = 0;`
Devuelve la probabilidad de realizar un golpe crítico (representado como porcentaje).

virtual void mostrarInfo() const = 0;
Imprime en consola la información completa del arma, incluyendo nombre, tipo, rareza, daño base y efectos activos.

virtual void usar(shared_ptr<IPersonaje> personaje, shared_ptr<IPersonaje> rival) = 0;
Método central para la ejecución del arma. Se pasa tanto al personaje que usa el arma como a su rival, permitiendo que el arma afecte a ambos en función de sus efectos (por ejemplo, una poción podría curar al usuario y dañar al enemigo).

virtual pair<Efecto, Efecto> getEfectos() const = 0;
Devuelve un par de efectos asociados al arma. Cada uno puede ser un efecto concreto (como Sangrado, Asustar, etc.) o el valor NINGUNO, en caso de no tener efecto. Este diseño permite que un arma tenga hasta dos efectos simultáneos.

virtual ~InterfazArma() = default;
Destructor virtual obligatorio en una interfaz para asegurar la correcta destrucción de objetos derivados a través de punteros a base.

Interfaz de Personaje (InterfazPersonaje)

La Interfaz "InterfazPersonaje" define el contrato que deben cumplir todos los personajes del sistema, tanto magos como guerreros. Al ser una interfaz pura, todos los métodos son públicos y virtuales, forzando a las clases derivadas a implementar su comportamiento específico. A continuación se detallan sus métodos:

virtual bool estaVivo() const = 0;
Retorna true si el personaje aún tiene puntos de vida (HP) mayores a cero, y false en caso contrario.

virtual string getNombre() const = 0;
Devuelve el nombre del personaje ("Hechicero", "Bárbaro", etc.).

virtual Rango getRango() const = 0;
Retorna el Rango del personaje, definido como un enum class con valores como NOVATO, INTERMEDIO, LEGENDARIO, entre otros. Este valor afecta directamente las estadísticas del personaje.

virtual bool Compatibilidad(IArma* arma) const = 0;
Verifica si el tipo de arma (física o mágica) coincide con la clase del personaje. Devuelve true si son compatibles.

virtual int getVida() const = 0;
Devuelve el valor actual de puntos de vida (HP) del personaje.

virtual int getMana() const = 0;
Devuelve el nivel actual de mana, utilizado para habilidades mágicas y armas mágicas.

virtual int getResistencia() const = 0;
Devuelve la resistencia física del personaje, usada como recurso para ataques físicos o armas de combate.

virtual void mostrarInfo() const = 0;
Muestra en consola todos los atributos actuales del personaje: nombre, tipo, vida, maná, resistencia, armas equipadas y efectos activos.

virtual vector<pair<Efecto, int>> getEstados() const = 0;
Retorna una lista de efectos actuales aplicados al personaje. Cada par incluye un Efecto (como Asustado o Sangrado) y una duración en turnos.

virtual void actualizarEfectos() = 0;
Reduce en 1 la duración de cada efecto activo. Además, se llama internamente a propagarEfectos(), que aplica los efectos según su tipo.

virtual void aplicarEfecto(Efecto efecto, int duracion) = 0;
Aplica un nuevo efecto al personaje, agregándolo a su lista de estados activos con la duración especificada.

virtual pair<unique_ptr<IArma>, unique_ptr<IArma>> getArmas() const = 0;
Devuelve las dos armas equipadas por el personaje. Si alguna está vacía, se retorna como nullptr.

virtual void recibirDmg(int dmg) = 0;
Resta el daño recibido (dmg) a los puntos de vida del personaje.

virtual void curarVida(int cant) = 0;
Suma la cant a los puntos de vida actuales del personaje.

virtual void consumirResistencia(int cant) = 0;
Resta cant a la resistencia física actual.

virtual void regenerarResistencia(int cant) = 0;
Suma cant a la resistencia física actual.

virtual void consumirMana(int cant) = 0;
Resta cant al mana actual.

virtual void regenerarMana(int cant) = 0;
Suma cant al mana actual.

virtual void setArmas(pair<unique_ptr<IArma>, unique_ptr<IArma>> armas) = 0;
Permite equipar al personaje con una o dos armas luego de su creación. Esto se utiliza en la fábrica de personajes (PersonajeFactory) del Ejercicio 2.

virtual ~InterfazPersonaje() = default;
Destructor virtual obligatorio para asegurar la correcta destrucción de las clases derivadas a través de punteros a interfaz.

Clases abstractas de armas: ArmaMagica y ArmaCombate

El diseño del sistema de combate distingue entre dos grandes grupos de armas: mágicas y físicas. Ambos tipos derivan de la interfaz común InterfazArma, pero implementan comportamientos específicos según su naturaleza. Para ello, se crearon dos clases abstractas: ArmaMagica y ArmaCombate.
Estas clases encapsulan atributos y métodos comunes a sus respectivas subclases, permitiendo organizar de manera coherente el comportamiento compartido entre armas similares, sin repetir código. Por ejemplo, todas las armas mágicas consumen maná y tienen un número limitado de usos, mientras que las de combate consumen resistencia física y se deterioran con el uso (durabilidad).

Clase ArmaMagica (abstracta)

Esta clase representa armas que requieren energía mágica. Todas las armas mágicas comparten los siguientes atributos y comportamientos:

Atributos protegidos:

string nombre;
Nombre específico del arma mágica (ej. "Baston").

string tipo = "mágica";
Fijado como string para indicar el tipo de arma. Se utiliza en los métodos de compatibilidad.

Rareza rareza;
Rareza del arma. Afecta al daño y los usos disponibles.

int dmg_base;
Daño base que inflige el arma.

int consumo_mana_base;
Cantidad de maná que consume cada vez que se usa el arma.

int usos;
Cantidad de veces que puede utilizarse antes de romperse.

int probabilidad_de_critico;
Probabilidad de realizar un golpe crítico (doble daño).

Métodos principales:
int getConsumoManaBase() const;
Devuelve el valor de maná consumido al usar el arma.

int getUsos() const;
Devuelve los usos restantes antes de que el arma quede inutilizable.

Además de implementar todos los métodos definidos en InterfazArma.

Clase ArmaCombate (abstracta)

Esta clase agrupa las armas de combate cuerpo a cuerpo. A diferencia de las mágicas, estas no consumen maná sino resistencia física y poseen durabilidad. A continuación solo menciono las diferencias con ArmaMagica.

Atributos protegidos:
string tipo = "combate";
Identificador textual fijo que señala el tipo de arma.

int durabilidad;
Valor que decrece con cada uso. Cuando llega a cero, el arma se rompe.

int consumo_resist_base;
Resistencia física que debe gastar el personaje al utilizar el arma.

Métodos principales:
int getDurabilidad() const;
Devuelve la cantidad restante de usos antes de romperse.

int getConsumoRestBase() const;
Retorna la cantidad de resistencia física que consume el arma.

Hereda e implementa los métodos de la interfaz InterfazArma.

Clases abstractas de personajes: Mago y Guerrero

Los personajes del sistema se dividen en dos grandes familias: Magos y Guerreros, cada una con especializaciones, atributos y mecánicas únicas. Ambas clases derivan de la interfaz común InterfazPersonaje, pero implementan sus propias variantes de comportamiento y lógica interna, lo que permite una experiencia diferenciada de juego.

Estas clases no son instanciables por sí mismas, ya que están diseñadas como clases abstractas que agrupan comportamientos compartidos por sus subclases concretas (Hechicero, Paladín, etc.). Su objetivo es facilitar la extensibilidad y modularidad del sistema, evitando la repetición de lógica.

Clase Mago (abstracta)

Representa a todos los personajes especializados en magia. Sirve como base para clases concretas como Conjurador, Hechicero, Brujo y Nigromante.

Atributos protegidos:

string nombre;
Nombre del mago (se pasa como string en el constructor).

string clase = "Mago";
Cadena fija que indica la categoría del personaje (se usa para compatibilidad y visualización).

Rango rango;
Define la rareza jerárquica del personaje (Novato, Avanzado, Dios, etc.). Influye en vida, maná y resistencia.

int vida;
Puntos de vida del personaje.

int mana;
Energía mágica usada para habilidades y armas mágicas.

int resistencia;
Resistencia física básica del personaje (en general menor que la de los guerreros).

pair<unique_ptr<IArma>, unique_ptr<IArma>> armas;
Armas equipadas (puede tener una o dos, o ninguna).

vector<pair<Efecto, int>> estados;
Efectos activos sobre el personaje, junto con su duración.

Método único del mago:

void usarHechizo(shared_ptr<IPersonaje> rival, int mana, int dmg);
Permite realizar un ataque mágico básico si no tiene armas. Consume maná y causa daño directo al rival.

Además, la clase implementa o hereda todos los métodos definidos en la InterfazDePersonajes

Clase Guerrero (abstracta)

Modelo para personajes con habilidades físicas, como Bárbaro, Caballero, Mercenario, Gladiador y Paladín. Poseen mayor resistencia y vida que los magos, y dependen de armas de combate. Tan solo mencionare las diferencias con la clase Mago

Atributos protegidos:

string clase = "Guerrero";
String fijo que representa su categoría.

Método único del guerrero:

void ataqueMelee(shared_ptr<IPersonaje> rival, int resistencia, int dmg);
Ataque físico directo usado sin armas. Consume resistencia y causa daño físico al oponente.

Igualmente, implementa todos los métodos definidos en la interfaz InterfazPersonaje.

Diseño e implementación de personajes

La idea de mis personajes es que cualquiera pueda utilizar cualquier arma, con la diferencia de que el consumo de recursos no favorecerá las armas no compatibles con los personajes. Esto sucede en dos instancias, primero porque un Mago no tiene suficiente resistencia como para atacar en repetidas ocasiones con un arma física, además de que al no ser compatibles, el daño que infligira será reducido del stat base. En cuanto a la creación, el propósito fue querer utilizar la menor cantidad de parámetros para crear un personaje, ya que más adelante en el TP iba a tener que aleatorizar la creación, es por eso que decidí que todos las personajes tengan sus estadísticas base ya definidas en su constructor de Mago/Guerrero, y que tan solo el Rango sea quien afecte estos atributos base.

Magos: “Brujo”, “Conjurador”, “Hechicero” y “Nigromante”

Los magos poseen mayor cantidad de maná y un perfil orientado a efectos mágicos. Cada subclase concreta tiene dos habilidades únicas, de las cuales al menos una aplica un efecto sobre el enemigo o sobre sí mismo. De esta forma, se simula un personaje con habilidades tanto ofensivas como de soporte.

Por ejemplo, un Conjurador se crea de la siguiente forma:

```
Conjurador::Conjurador(Rango rango) //rango decidido
  : Mago("Conjurador", rango, 110, 130, 70, {nullptr, nullptr}) {
  this->clase = "Mago";
}
```

Vida base: 110/Maná Base: 130 /ResistenciaBase: 70
(Los valores son luego modificados internamente por el multiplicador asociado al rango)

los nullptr representa la creación del personaje sin armas

Ej. Rango:Avanzado = 1.3

```
void conjuroEspectral(std::shared_ptr<InterfazDePersonaje> rival);
```

// Habilidad única de un Conjurador que aplica una SOBRECARGA (Aumenta atributos un 30%) sobre el usuario, consume 25 de maná e inflige 35 de daño sobre el rival.

```
void canalizacionAstral();
```

// Habilidad única que cura 25hp, regenera 20 de resistencia y consume 10 de maná.

Guerreros: “Bárbaro”, “Caballero”, “Gladiador”,” Mercenario” y “Paladín”

Los guerreros tienen mayor vida y resistencia que los magos. También poseen dos habilidades únicas, de las cuales al menos una debe infligir daño físico. Sus métodos consumen resistencia y están diseñados para funcionar tanto con armas como en combate sin armas.

Por ejemplo, un Barbaro se crea de la siguiente forma:

```
Barbaro::Barbaro(Rango rango)
  : Guerrero("Bárbaro", rango, 130, 40, 120, {nullptr, nullptr}) {
  this->clase = "Guerrero";
}
```

Vida base: 130/Maná Base: 40 /ResistenciaBase: 120
(Los valores son luego modificados internamente por el multiplicador asociado al rango)

Validaciones e implementación modular

Cada personaje valida internamente si puede ejecutar ciertas acciones, por ejemplo:

- Los efectos se actualizan turno a turno con actualizarEfectos() y se propagan mediante propagarEfectos().
- Las habilidades de cada personaje están diseñadas para ser autocontenidas, y pueden modificar al rival, al usuario o ambos, aprovechando el sistema de estados y efectos.

Diseño e implementación de armas

En cuanto al diseño de mis armas, decidí que todas puedan tener desde 0 a 2 efectos asociados, los cuales se utilizaran en el método más importante de las armas: usar(), debajo le dedico un apartado a este método. Todas las armas, tanto mágicas como de combate cuentan con todos los atributos ya detallados en la interfaz y subclase acorde. Además, a todas aquellas armas a las cuales pensé que les quedaría bien tener más opciones de ataques o support, se lo agregue a gusto. En cuanto a su creación, esta funciona prácticamente igual a la de personajes. Por comodidad para los siguientes ejercicios, decidí que las armas sean creadas tan solo pasandole como parámetro su Rareza, que es el equivalente al Rango de los personajes. Esto lo hice pasandole parametros fijos al constructor de arma que tengan asociado (ArmaCombate/ArmaMagica)

Implementacion del metodo: usar( )

void usar(shared_ptr<IPersonaje> personaje, shared_ptr<IPersonaje> rival);

Cuando un arma en concreto utiliza el método usar(), lo primero que hace es ejecutar el usar() de su clase abstracta, este método recibe tanto el personaje como el rival de parámetro ya que todas del arma actúan o consultan información de ambos. Los chequeos que realiza este método son los siguientes:

1- Verificar si la cantidad de usos/durabilidad es mayor o igual a cero

2- Verificar si el personaje cuenta con el mana/resistencia suficiente para poder utilizar el arma

3- Alterar el daño base del arma según la compatibilidad de esta con el personaje (compatible multiplica por 1,3 y no compatible por 0,7)

4- Verificar los estados del personaje que está utilizando el arma, para ver si el personaje se encuentra Asustado, si tiene Reducción de Daño o por si tiene Sobrecarga. En caso de estar asustado: skipea el ataque, si tiene reducción de daño: altera el daño del arma y si tiene sobrecarga: aplica un 30% de daño extra y se consume un 30% más de maná en el ataque.

5- Chequea si el rival tiene el estado de Inmunidad, para ver si es o no capaz de atacar.

6- Inicia un random para ver si el ataque será crítico, para ello utiliza la tasa de crítico definida en cada arma.

7- Consume el maná respectivo del arma

8- Actualiza sus usos/durabilidad

9- Aplica daño al rival

10- Simula una elección del Efecto que tiene el arma, con el fin de aplicar tan solo uno de los Efectos disponibles por el arma (esta mecánica la hice para balancear los tipos de ataque disponibles, sino nunca pensarias en usar otro método de ataque)

11- Por ultimo, actualiza los efectos del rival (descuenta uno a todos los turnos restantes de cada efecto y se propagan los efectos en cuestión)

Rarezas y rangos

Uno de los elementos clave para balancear el sistema de combate en este proyecto fue la incorporación de dos niveles de clasificación: rangos para los personajes y rareza para las armas. Ambos conceptos aportan diversidad y escalabilidad al juego, permitiendo representar desde personajes novatos con armas simples hasta unidades legendarias portando armas poderosas.

La decisión de usar enum class en lugar del enum normal se basa en los beneficios que ofrece:

- Mayor seguridad de tipo: evita errores de conversión implícita.
- Mejor organización: permite agrupar valores bajo un mismo espacio lógico (Rango::novato).
- Legibilidad y mantenibilidad: facilita futuras expansiones

Cada rango actúa como un multiplicador de estadísticas. En lugar de tener que definir manualmente los atributos de cada personaje al instanciarlo, decidi que el constructor de Mago o Guerrero reciba solo el Rango, y que dentro de la clase se apliquen los multiplicadores sobre los valores base definidos para cada tipo concreto. Idem con la creación de armas.

Ej. clase Rarezas:

```
class Rarezas {
public:
    static float getMultiplicadorRareza(Rareza rareza) {
        switch (rareza) {
            case Rareza::COMUN: return 1.0;
            case Rareza::POCO_COMUN: return 1.2;
            case Rareza::RARO: return 1.3;
            case Rareza::EPICO: return 1.5;
            case Rareza::LEGENDARIO: return 1.8;
            default: return 0;
        }
    }
```

Efectos y estados

Se implementa un sistema de efectos especiales que pueden ser aplicados durante la batalla por medio de armas, habilidades o condiciones específicas. Cada efecto deriva de la clase abstracta InterfazDeEfecto y redefine el método aplicar(...), que recibe dos punteros compartidos (shared_ptr) al personaje que lo ejecuta y al rival.

Efectos de daño (EfectosDmg.hpp)

Asustar

- Descripción: Tiene una probabilidad del 60% de aplicarse el efecto, en caso de tenerlo, el rival no podra atacar por X cantidad de turnos

Envenenar

- Descripción: Aplica un daño fijo de 15 puntos por X cantidad de turnos.

Sangrado

- Descripción: Resta un 15% de la vida total del rival por X cantidad de turnos.

Reducción de Daño

- Descripción: Reduce el daño que el objetivo inflige en un 30% por X cantidad de turnos.

Efectos de soporte (EfectosSupport.hpp)

Regeneración

- Descripción: Cura 25 puntos de vida al personaje afectado por X cantidad de turnos.

Agotamiento

- Descripción: Resta 10 de resistencia y 15 de maná al oponente por X cantidad de turnos

Inmunidad

- Descripción: Aplica un estado de inmunidad con una probabilidad del 70%.

Robo de Maná

- Descripción: Absorbe un 20% del maná del rival y lo transfiere al usuario.

Robo de HP

- Descripción: Absorbe un 20% de la vida del rival y lo transfiere al usuario.

Sobrecarga

- Descripción: Aumenta el daño infligido en un 30% con un 75% de probabilidad de éxito.

El sistema de efectos implementa tres métodos claves para administrar su ciclo de vida e impacto en los personajes:

1- aplicar(shared_ptr<InterfazDePersonaje> , shared_ptr<InterfazDePersonaje>)

Es el método principal que se ejecuta al activar un efecto. Se encarga de inicializar el impacto del efecto sobre el personaje rival (o sobre el mismo personaje, según el caso). Por ejemplo, en Envenenar, este método agrega el estado correspondiente (Envenenado).

2- actualizarEfectos()

Este método se llama en cada turno para mantener los efectos activos. Si el efecto persiste (por ejemplo, Regeneración o Sangrado), actualizar ejecuta el impacto correspondiente (curación o daño)mediante el método propagarEfecto() y reduce su duración si está limitada. Esto permite efectos prolongados en el tiempo.

3- propagarEfecto(Efecto efecto, shared_ptr<InterfazDePersonaje>)

Este método se utiliza para ejecutar el efecto en cuestión, accede a los parámetros definidos de cada efecto y realiza la acción.

Main.cpp

Creé un main simplemente para testear la creación de personajes, armas y efectos.

UML

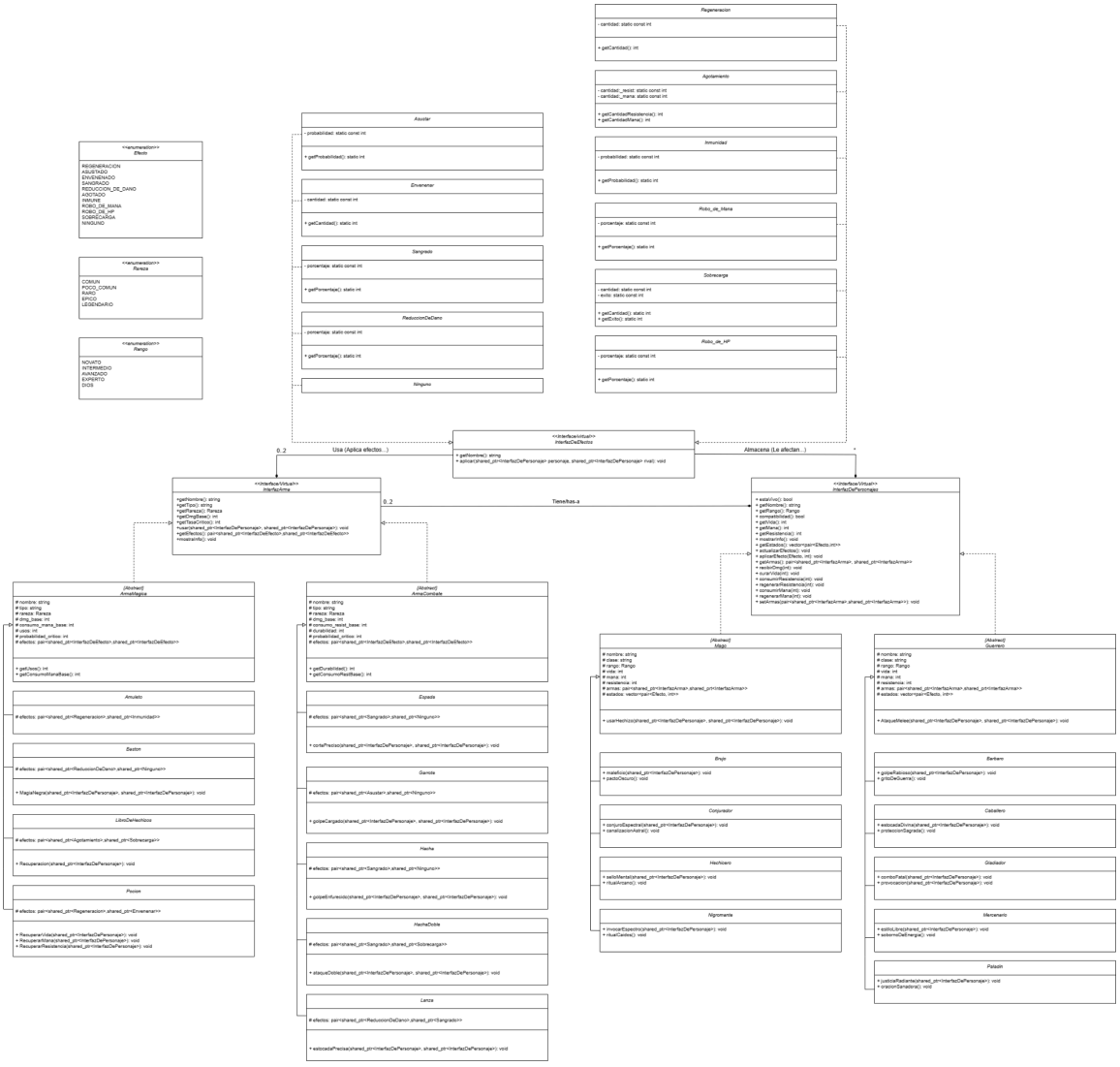

Imagen 1.0: Diagrama UML personajes, armas, efectos, rangos y rarezas

Nota: en el repo se encuentra el PNG en buena calidad para visualizar correctamente, aca solo se importó de manera esquemática.

EJ2: Fábrica de Personajes y Armas

El Ejercicio 2 está dedicado a la implementación de un sistema de fábricas para crear personajes y armas de forma controlada, modular y reutilizable, mediante funciones específicas conocidas como factory functions. Este enfoque facilita la generación de instancias tanto de manera manual (por nombre) como aleatoria, sirviendo de base para el simulador del Ejercicio 3.

Implementación de las funciones factory

Se definen tres funciones principales dentro de PersonajeFactory:

1- crearPersonaje(std::string nombre, Rango rango)

Devuelve un shared_ptr<InterfazDePersonaje>, instanciando la subclase correspondiente según el nombre recibido.

El shared_ptr se usa porque los personajes pueden ser referenciados desde múltiples lugares (por ejemplo, efectos aplicados por otros personajes), y este tipo de puntero garantiza una correcta gestión de memoria compartida.

2- crearArma(std::string nombre, Rareza rareza)

Retorna un unique_ptr<InterfazArma>, ya que cada arma pertenece exclusivamente a un único personaje, no necesita compartirse, y debe garantizarse su destrucción automática al liberar al personaje que la contiene.

3- crearPersonajeArmado(std::string nombrePersonaje, Rango rango, std::string nombreArma, Rareza rareza)

Retorna un shared_ptr<InterfazDePersonaje> que ya posee un arma instanciada y equipada. Esta función encapsula el proceso de creación de ambas entidades y establece su asociación de forma directa.

Uso de SmartPointers

shared_ptr para personajes:

Permite múltiples referencias a un mismo personaje, especialmente útil cuando distintos módulos (combate, efectos, registros) acceden a los mismos datos. También facilita la propagación de efectos y evita problemas de ownership compartido.

unique_ptr para armas:

Asegura que cada arma tenga un único propietario (el personaje que la porta). Es más eficiente en cuanto a gestión de recursos, y refuerza el diseño lógico de que un arma no puede estar equipada por más de un personaje a la vez.

Main.cpp

En primer lugar se generan dos cantidades random de guerreros y magos entre 3 y 7. Luego creo de manera aleatoria esa cantidad de personajes mediante el método crearPersonajeArmado(nombre, rango). Ambos parámetros son randoms, ya que el nombre es un rand entre los nombres de los guerreros/magos y el rango es un rand sobre los Rangos posibles. Dentro de crearPersonajeArmado() se encuentra la logica de creación y asignación de armas, que también cuenta con un rand para determinar la cantidad de armas que se le asignan a cada personaje. Por último, muestro la información de cada personaje generado con sus armas y si son compatibles entre sí.

EJ3: Simulación de batalla

Se implementó un simulador de combate entre personajes, donde un jugador puede elegir un personaje y un arma para enfrentarse a un rival generado aleatoriamente.

Definición de nombres y armas:

Se definen listas de nombres para personajes y armas (nombresPersonajes y nombresArmas) que se utilizarán para la selección del jugador y la generación del rival.

Funciones auxiliares:

entradaCorrecta(): Valida la entrada del usuario para evitar errores en el flujo del programa.

limpiarPantalla(): Limpia la consola según el sistema operativo.

menuPersonajes() y menuArmas(): Muestran los menús de selección de personajes y armas.

menuGolpes(): Muestra las opciones de ataque disponibles.

elegirGolpe(): Permite al jugador seleccionar un estilo de golpe (pesado, veloz o defensa reactiva).

golpeAleatorioCPU(): Genera un golpe aleatorio para el rival.

descripcionGolpe(): Devuelve una descripción textual del estilo de golpe seleccionado.

barraDeVida(): Genera una representación visual de la vida restante de un personaje.

mostrarEstado(): Muestra el estado actual del combate, incluyendo los nombres de los personajes, sus armas y sus barras de vida.

Selección del jugador:

El jugador elige un personaje y un arma de las listas disponibles. Se crea el personaje y se le asigna el arma seleccionada utilizando la clase PersonajeFactory.

Generación del rival:

El rival se genera aleatoriamente seleccionando un personaje y un arma de las listas. También se utiliza PersonajeFactory para crear el personaje y asignarle el arma.

Sistema de combate:

Se implementa un sistema de turnos donde el jugador y el rival eligen un estilo de golpe.

Se utiliza una matriz de reglas (reglas) para determinar el resultado del enfrentamiento entre los estilos de golpe. (Piedra, papel o tijeras)

Dependiendo del resultado, se reduce la vida del jugador o del rival en 10 puntos. El combate continúa hasta que uno de los personajes quede sin vida.

Resultado del combate:

Al finalizar el combate, se muestra el ganador en la consola.