Informe – Trabajo Práctico 1
Paradigmas de Programación

Pizarro, Federico

17 de abril de 2025

Índice

1. Introducción

El objetivo de este trabajo práctico fue trabajar con interfaces, clases abstractas y sus respectivas clases derivadas. También, esto me ayudó 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 tuve que investigar acerca de cómo generar y utilizar números aleatorios en C++.

El paradigma que se utilizó en este proyecto es programación orientada a objetos (POO), el cual permite encapsular comportamientos y características en estructuras modulares y escalables. En particular, el uso de interfaces y clases abstractas favorece un diseño más flexible y extensible, lo que facilita incorporar nuevas funcionalidades sin modificar significativamente el código existente, ya que respeta el principio abierto/cerrado, el cual permite que la estructura esté abierta para añadir elementos y funcionalidades, sin la necesidad de modificar las existentes.

Asimismo, la utilización de punteros inteligentes (smart pointers) ayuda a prevenir problemas comunes de gestión de memoria en C++, tales como pérdidas de memoria y accesos inválidos.

Por último, la representación gráfica mediante diagramas UML proporciona claridad visual al diseño del software, permitiendo una mejor comprensión de la estructura y relaciones entre las clases.

2. Consigna y Requisitos

2.1. Descripción del Problema

Se pide implementar un juego de rol. Específicamente, 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 clases concretas 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.

Incialmente la consigna pedía que hacía daño quien ganaba el piedra papel o tijera, siendo los daños iguales para todos. Sin embargo, implementado la primer parte de la consigna cada personaje y arma, tenían lógicas de ataque, control de recursos y recibo de daño diferenciados. Por lo tanto, el piedra, papel o tijera solo decide quién jugará el turno, luego cada personaje decide que decisión tomar en base a sus armas y habilidades.

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.

Documentación

- Para la generación del informe utilice el compositor de texto LaTeX.
- Los diagramas UML son generados con *draw.io*
- Se incluye un README en el repositorio describiendo el proyecto.

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.

Para compilar el proyecto utilizo un archivo `Makefile`, que automatiza la construcción del ejecutable, esto facilitó la compilación sin errores al momento de incluir todos los archivos necesarios, ya que están configurados de forma sencilla en el archivo `Makefile`. El comando básico utilizado es `make`, que ejecuta un comando por defecto, donde incluye el compilador de C++ (`CXX`), todos los flags necesarios (`CXXFLAGS`), archivos que se compilan (`SRC`).

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 tomadas en funcionalidades concretas del proyecto.

3.1. Diseño y Diagramas UML

Para este proyecto se llevo a cabo un diseño en capas (Interfaz → Abstracta → Final) el cual facilita la extensibilidad del sistema y permite añadir nuevos personajes o armas sin alterar el código existente cumpliendo con el principio abierto/cerrado del diseño orientado a objetos.

3.1.1. 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.

Todos los métodos declarados en las interfaces son necesarias para cualquier clase derivada, y deben ser implementados en alguna jerarquía. Hay algunos métodos `protected`, mientras que la mayoría son `public`. Dado que las interfaces únicamente se utilizan como plantillas para luego implementar clases finales, estas no cuentan con ningún constructor.

Se presentan a continuación los diagramas UML de estas dos interfaces, donde se detallan los métodos requeridos para cualquier tipo de personaje o arma:

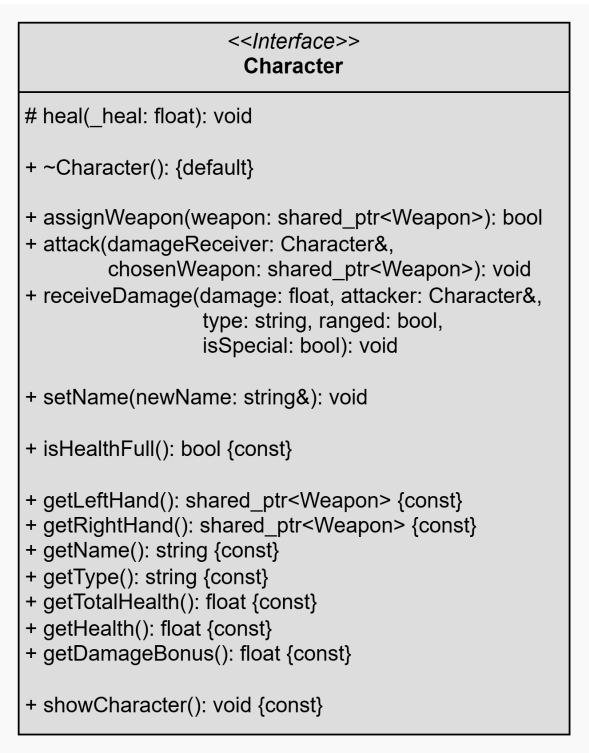

Figura 1: Diag. UML de Personajes

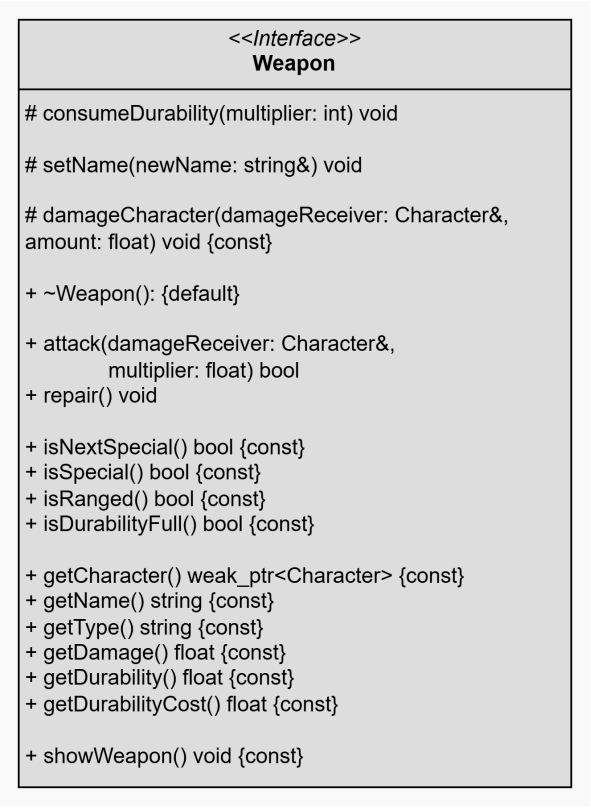

Figura 2: Diag. UML de Armas

3.1.2. Clases Abstractas

A partir de las interfaces, se definieron clases abstractas intermedias para agrupar comportamientos comunes. Estas son `Sorcerer` y `Warrior` para personajes, y `MagicItem` y `CombatWeapon` para armas.

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 cual 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.

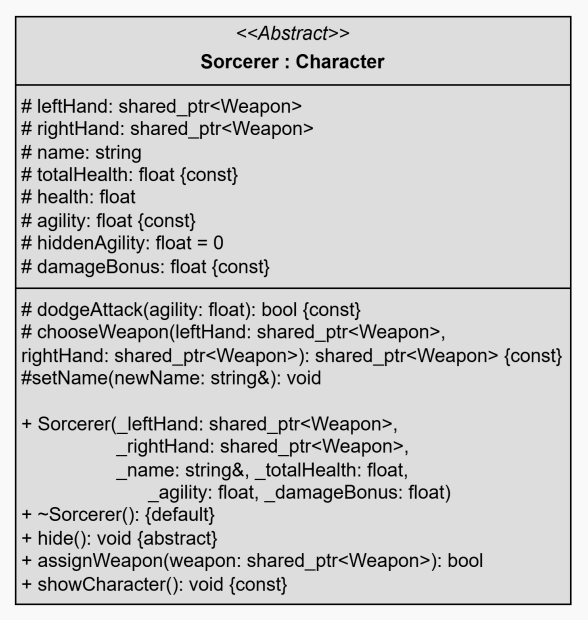

Figura 3: Diag. UML de Magos

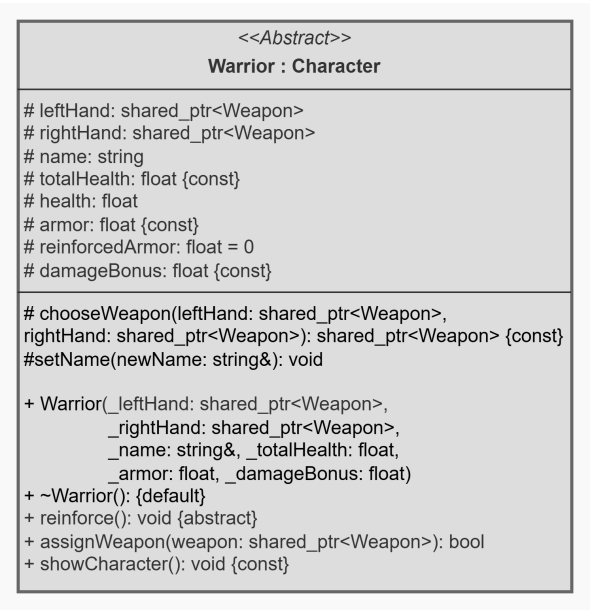

Figura 4: Diag. UML de Guerreros

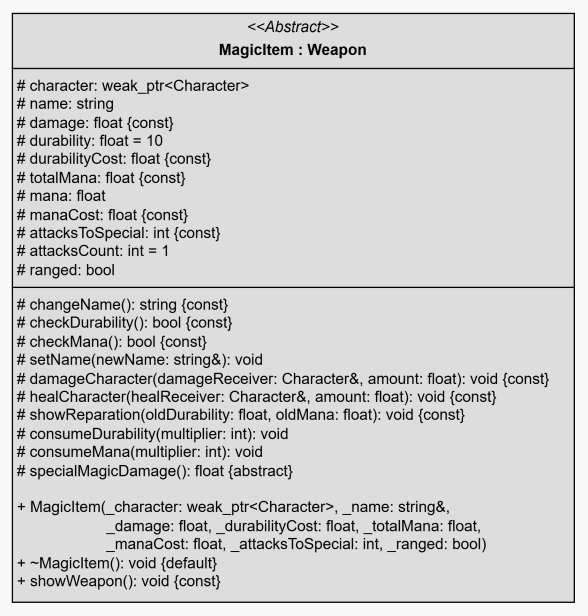

Figura 5: Diag. UML de Items Mágicos

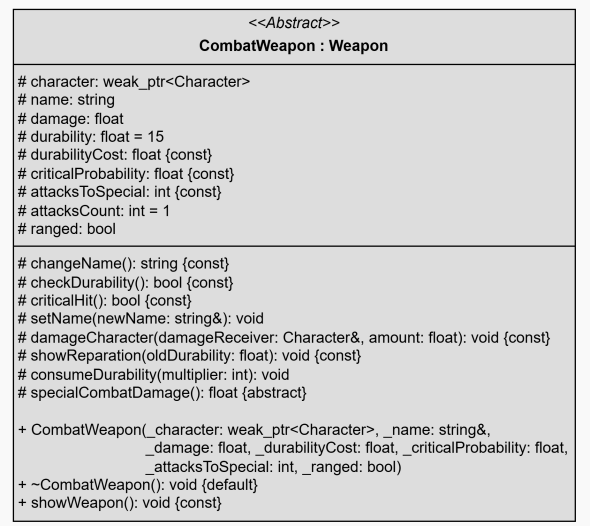

Figura 6: Diag. UML de Armas de Combate

3.1.3. 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 `repair()`, todos las armas cuentan con este método, pero cada derivada final lo hace de distinta manera, lo mismo con `receiveDamage()` en el caso de los personajes.

El constructor de cada una de las clases derivadas utiliza el constructor de su clase abstracta. Este constructor es invocado con algunos parámetros, pero la mayoría son inicializados con valor predeterminados, los cuales son las estadísticas iniciales del objeto, que luego pueden modificarse durante el transcurso del juego.

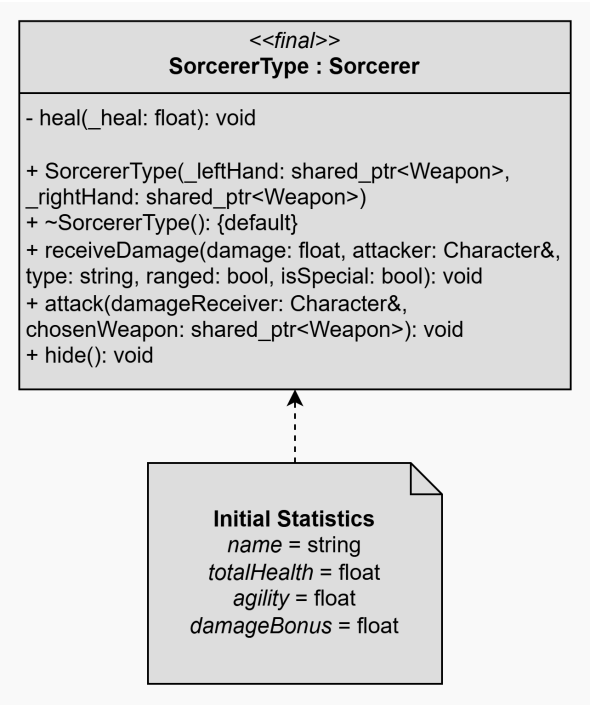

Figura 7: Diag. UML de Derivada Final de Magos

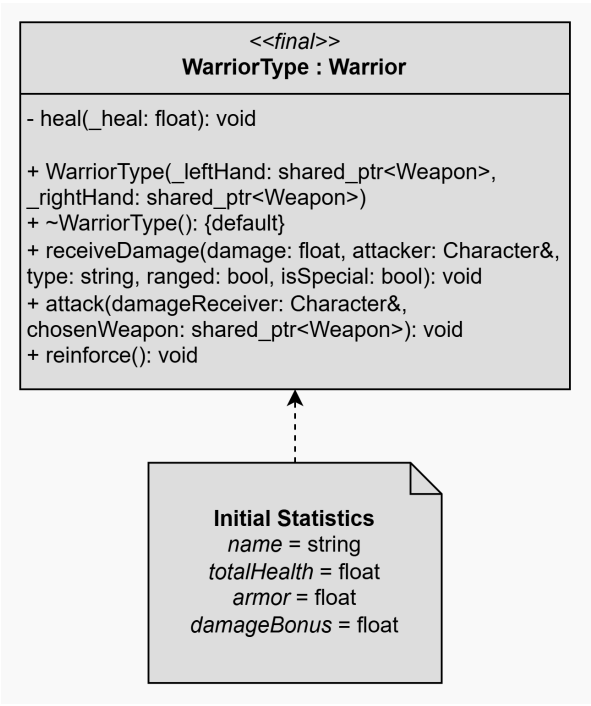

Figura 8: Diag. UML de Derivada Final de Guerreros

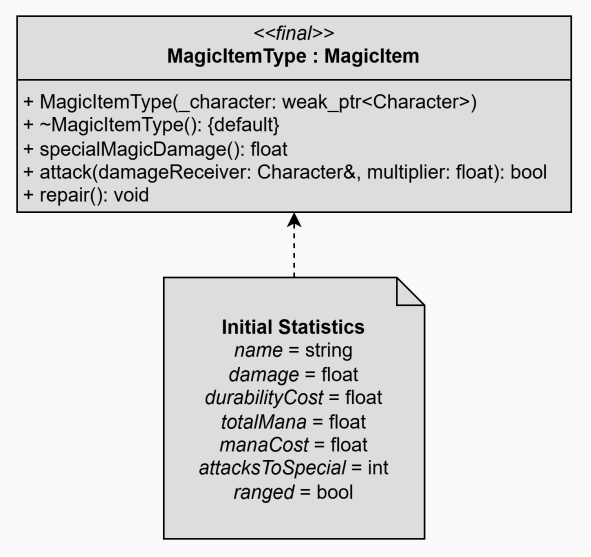

Figura 9: Diag. UML de Derivada Final de Items Mágicos

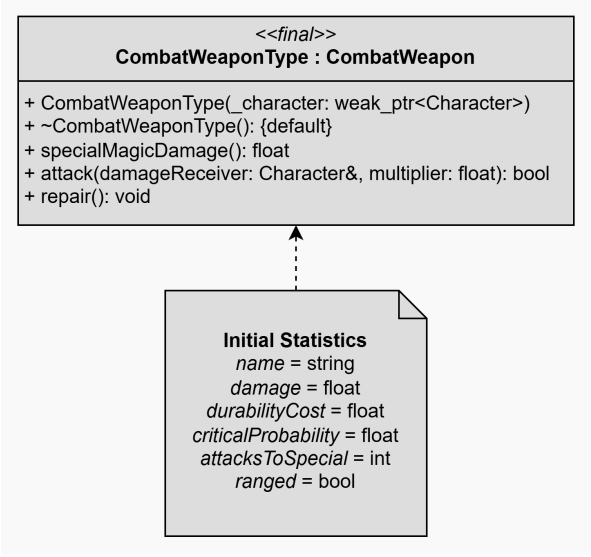

Figura 10: Diag. UML de Derivada Final de Armas de Combate

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. Al comienzo me ayudó 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. Se pide implementar una batalla a través de un mecanismo de piedra, papel o tijera.

Muchas decisiones e ideas son sacadas de juegos que alguna vez probé.

Algunas de las decisiones que tomé durante el diseño y creación de la arquitectura del programa:

- 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 cual 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, asigné 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

3.3.1. Implementación de Personajes

Los personajes de tipo `Sorcerer` tiene atributos y lógicas de `agility`, lo que les da la capacidad de esquivar ataques, para esto se implementa su método virtual puro `hide()` lo que les permite no atacar en su turno, a cambio de recuperar un porcentaje de la vida faltatante y añadir un porcentaje de agilidad para el próximo ataque recibido.

Por otro lado, `Warrior` implementa `armor`, lo que les permite reducir en un cierto porcentaje el daño recibido, de manera similar, cuentan con el méotdo `reinforce()`, el cual también cura al guerrero en cuestión, además de proporcionarle armadura adicional para la siguiente defensa. Estos atributos e interacciones con los daños son los que principalmente los diferencian entre si.

Hay métodos declarados en la interfaz general que son implementados en la clase abstracta, como por ejemplo `chooseWeapon()` o `showCharacter()`, ya que todas sus derivadas lo hacen de la misma manera.

```
// Ejemplo simplificado de implementación de showCharacter()

void Sorcerer::showCharacter() const {
    cout << "Nombre: " << getName() << endl;
    cout << "Mano Izquierda: " << getLeftHand()->getName() << endl;
    cout << "Mano Derecha: " << getRightHand()->getName() << endl;
    cout << "Vida: " << getHealth() << "/" << getTotalHealth() << endl;
    cout << "Agilidad: " << getAgility() << endl;
    cout << "Bonificación de daño: " << getDamageBonus() << endl;
}
```

Todos los personajes implementan en su jerarquía final las funciones más particulares, como puede ser `attack()`, ya que cada personaje implementa diferentes lógicas que utilizan de distinta manera los atributos tanto propios, como aquellos que son de los personajes que las usan.

```
// Ejemplo simplificado de implementación de attack()

void Summoner::attack(Character& damageReceiver,
                      shared_ptr<Weapon> chosenWeapon) {
    float multiplier = 1 + damageBonus;
    if (!chosenWeapon) {
        chosenWeapon = chooseWeapon(leftHand, rightHand);
    }
    if (auto weapon = chosenWeapon) {
        if (weapon->isNextSpecial()) {
            multiplier += 0.35;
            cout << getName() << " intensifica el especial.\n" << endl;
            cout << "¡Daño " << multiplier << "x!\n" << endl;
        }
        weapon->attack(damageReceiver, multiplier);
    } else {
        cout << "¡" << getName() << " no puede atacar!\n" << endl;
    }
}
```

En este último ejemplo se deja ver como `Summoner` implementa el método `attack()`, pero su lógica se diferencia ya que provoca más daño si el ataque es especial.

3.3.2. Implementación de Armas

De forma parecida a lo ya expuesto con respecto a los personajes sucede con las armas, ya que las armas de tipo `MagicItem` cuentan con un sistema de maná lo que restringue el uso de ellas, o en algunos casos potencia ataques especiales. El maná puede ser recuperado al reparar el arma con el método `repair()`, el cual se implementa de manera distinta a las Armas de Combate.

Las armas de tipo `CombatWeapon` tienen el atributo `criticalProbability`, por lo que usar un arma de estas puede generar grandes cantidades de daño. En este caso, la reparación únicamente afecta a `durability`, el cual es un atributo que tienen tambien los Items Mágicos.

```
// Ejemplo simplificado de implementación de specialDamage() y attack()

float Staff::specialMagicDamage() {return mana / 5;}

bool Staff::attack(Character& damageReceiver, const float multiplier) {
    if (!(checkDurability() && checkMana())) return false;
    float toDamage = damage;
    if (isNextSpecial()) {
        float specialDamage = specialMagicDamage();
        toDamage += specialDamage;
        attacksCount = 0;
        cout << "¡" << character.lock()->getName() << " realiza un
            ataque especial cargando todo el maná con " << getName() <<
            " provocando " << RED_FORMAT << specialDamage <<
            RESET_FORMAT << " de daño adicional!" << endl;
    }
    damageCharacter(damageReceiver, toDamage * multiplier);
    consumeResources();
    attacksCount++;
    return true;
}
```

```
// Ejemplo simplificado de implementación de repair() en MagicItem

void Staff::repair() {
    durability += (10 - durability) / 2;
    mana += (totalMana - mana) / 2;
}
```

```
// Ejemplo simplificado de implementación de repair() en CombatWeapon

void SingleAxe::repair() {durability += (15 - durability) / 2;}
```

En los últimos dos ejemplos se ve como todas las armas implementan el método `repair()`, por esto está declarado en la interfaz `Weapon`, sin embargo lo hacen de manera distinta.

3.3.3. Implementación de Fábrica

La clase `CharacterFactory` permite crear personajes y armas de manera dinámica pero sin la necesidad de ser instanciada, lo que da cierta facilidad para su objetivo. Cuenta con tres métodos `createCharacter`, `createWeapon` y `createArmedCharacter`, los cuales son definidos como `static`, ya que permite que estos métodos sean utilizados sin la necesidad de crear un objeto de la clase `CharacterFactory`.

Que la creación de personajes y armas sea de forma dinámica, quiere decir que podemos recibir parámetros, utilizar condiciones y switches, e instanciar y devolver objetos concretos derivados de una clase base en tiempo de ejecución.

```
// Ejemplo simplificado de implementación métodos de CharacterFactory

class CharacterFactory {
    public:
        static shared_ptr<Character> createCharacter(const string
            characterType);
        static shared_ptr<Weapon> createWeapon(const string weaponType,
             const shared_ptr<Character> character = nullptr);
        static shared_ptr<Character> createArmedCharacter(const string
            characterType, const string leftWeaponType, const string
            rightWeaponType = "");
};

shared_ptr<Character> CharacterFactory::createCharacter(const string
    characterType) {
    if (characterType == "Hechicero") {
        return make_shared<Wizard>(nullptr, nullptr);
    } else if (characterType == "Bárbaro") {
        return make_shared<Barbarian>(nullptr, nullptr);
    }
    throw runtime_error("Tipo de personaje no conocido: " +
        characterType);
}

shared_ptr<Weapon> CharacterFactory::createWeapon(const string
    weaponType, const shared_ptr<Character> character) {
    if (weaponType == "Bastón Mágico") {
        return make_shared<Staff>(character);
    } else if (weaponType == "Hacha Simple") {
        return make_shared<SingleAxe>(character);
    }
    throw runtime_error("Tipo de arma no conocido: " + weaponType);
}

shared_ptr<Character> CharacterFactory::createArmedCharacter(const
    string characterType, const string leftWeaponType, const string
    rightWeaponType) {
    shared_ptr<Character> character = createCharacter(characterType);
    character->assignWeapon(createWeapon(leftWeaponType, character));
    if (!rightWeaponType.empty()) {
        character->assignWeapon(createWeapon(rightWeaponType, character
            ));
    }
    return character;
}
```

3.3.4. Implementación de Interfaz de Usuario y Flujo del Juego

El archivo /`Ejercicio_3/main.cpp` contiene la interacción directa con el jugador mediante una interfaz simple y clara basada en texto. Esta interfaz está diseñada para ofrecer una experiencia amigable y sencilla de seguir. Cuenta con una estructura de colores, para representar claramente las distintas estadísticas y estados del juego.

Al ejecutar el juego, el jugador es recibido por un menú principal que ofrece tres opciones básicas:

- Iniciar una nueva partida
- Ver reglas del juego
- Salir del juego

Esta estructura básica permite al jugador comenzar rápidamente o consultar información adicional fácilmente.

Al comenzar una partida, se genera dinámicamente un enemigo usando números aleatorios. Esto asegura que cada partida tenga elementos nuevos y aleatorios, haciendo más dinámico y variado el juego. Luego se le presenta al usuario su rival, para conocer qué clase de personaje y armas utilizar para generar una estrategia adecuada.

Luego, una vez conocido el rival, al usuario se le presenta la posibilidad de crear su personaje:

1. Selección del tipo de personaje (Hechicero, Bárbaro, Paladín, etc.), cada uno con características y bonificaciones únicas.
2. Elección de armas para cada mano, permitiendo elegir hasta dos armas, lo cual influye directamente en la estrategia de combate del personaje.
3. Personalización del nombre del personaje creado.

A continuación se muestra un fragmento simplificado del código que implementa esta interacción inicial con el usuario:

```
==== CREACIÓN DE PERSONAJE ====

Elige el tipo de personaje:
1. Hechicero (Daño mágico aumentado)
2. Bárbaro (Daño cuerpo a cuerpo aumentado)
>> _

Elige un arma para la mano izquierda:

1. Bastón Mágico (Daño especial basado en maná)
2. Hacha Simple (Aumenta daño con el tiempo)
>> _

Ingresa el nombre de tu personaje:
>> _
```

Tanto para la generación aleatoria del enemigo y dentro de la función `createPlayerCharacter()` se utilizan los métodos de `CharacterFactory` para crear los personajes.

A partir de acá el proyecto se divide en dos, ya que había implementado todo el juego de manera que funcione con un sistema por turnos, pero la consigna pide un sistema de piedra, papel o tijera. Por lo tanto están implementados ambos sistemas, para jugar con la metodología piedra, papel o tijera o por turnos.

Una vez iniciado el combate, primero hay una ronda de piedra, papel o tijera, donde se define quién llevará a cabo una acción, y el perdedor pierde su turno. Una vez definida la ronda de piedra, papel o tijera, durante su turno el usuario tiene varias opciones:

- Atacar al enemigo seleccionando un arma.
- Utilizar la habilidad especial del personaje (esconderse para magos o reforzarse para guerreros).
- Mostrar información detallada sobre el personaje y sus armas.
- Rendirse y finalizar la partida inmediatamente.

El fragmento siguiente ilustra cómo se gestiona este menú interactivo durante el turno del jugador:

```
==== TU TURNO ====

¿Qué acción quieres realizar?

1. Atacar
2. Esconderse y reparar armas (aumenta agilidad y cura)
3. Mostrar información detallada de personaje y armas
4. Rendirse
>> _
```

El enemigo, mediante números aleatorios elige las decisiones que toma. Sin embargo, se ve obligado a reforzarse o esconderse si el arma elegida no tiene durabilidad o mana suficiente para llevar a cabo un ataque, lo cual regenerará dichos recursos. También cuenta con un mecanismo de decisiones aleatorias para atacar o usar habilidades especiales, haciendo que las partidas sean variadas e impredecibles.

Para hacer la experiencia de juego más atractiva y fácil de seguir, se utilizan códigos ANSI para colorear la salida en la terminal. Por ejemplo, los mensajes críticos se resaltan en rojo, la vida del personaje en verde, las opciones de menú en azul, etc.

```
// Ejemplo de uso de colores en la terminal

cout << RED_FORMAT << "¡Daño crítico!" << RESET_FORMAT << endl;
cout << GREEN_FORMAT << "Vida restante: " << player->getHealth() <<
   RESET_FORMAT << endl;
```

Estos colores ofrecen al jugador información visual clara e inmediata sobre los eventos importantes que ocurren en el juego.

El combate continúa alternando turnos hasta que alguno de los personajes pierde toda su vida o el usuario se rinde. Al finalizar, el programa muestra un mensaje de victoria o derrota, y el jugador puede optar por regresar al menú principal o cerrar el juego.

Esto brinda al usuario la posibilidad de jugar múltiples partidas consecutivas sin necesidad de cerrar y reabrir el programa, lo que favorece la rejugabilidad.

Este flujo es simple, pero efectivo asegurando una experiencia intuitiva y satisfactoria para el jugador.

4. Resultados y Discusión

Luego de implementar todo el código, el diagrama UML y hacer este informe resalto algunos aspectos que me resultaron interesantes en el proceso.

- La estructura final del proyecto busca permitir la escalabilidad del proyecto añadiendo más tipos de personajes o armas, incluso distintos tipos de clases abstractas, e incluso nuevas interfaces que se basen en `Weapon` y `Character`, como podría ser `Vehicles`.
- Durante el desarrollo del juego tuve dudas sobre donde declarar e implementar los métodos de manera que queden bien definidos, sin embargo creo que todas cumplen con el encapsulamiento correspondiente sin afectar su funcionalidad.
- Creo que un proyecto como esto podría ser exportado a algún motor de videojuegos como Unity, añadiendole figuras que respresenten a los personajes y armas, y una interfaz para mostrar los estados del juego, con la posibilidad de distribuirlo en distintas plataformas mediante un sistema gráfico y adaptado.

5. Conclusiones

El trabajo práctico me pareció 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 ya que me atrapo y me gustó poder plasmar mis ideas en cada personaje y arma.

Creo que me ayudó 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* y descubriendo como utilizarlos.

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, ya que se convirtió en un proyecto grande, difícil de mantener controlado.

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.

6. Referencias

Se listan las fuentes bibliográficas, documentos y recursos utilizados durante el desarrollo del proyecto.

- *cppreference.com* (Documentación de C++)
- *draw.io* (Software para generar Diagramas UML)
- *Manual de LaTeX: Símbolos* (Referencia para símbolos en LaTeX)
- Material y guías de la materia de Paradigmas de Programación.
- Recursos sobre UML.

7. Repositorio del Proyecto

Todo el código fuente desarrollado para este trabajo práctico, junto con los archivos adicionales como diagramas UML e informe en LaTeX, se encuentra disponible en el siguiente repositorio público de GitHub:

`https://github.com/FedePizarro15/PdP_Trabajo_Practico_1`

A. Diagrama UML General del Proyecto

El siguiente diagrama UML representa de forma completa la arquitectura del proyecto, incluyendo interfaces, clases abstractas y sus respectivas clases derivadas. Debido a sus dimensiones, se presenta un link directo:

Enlace al diagrama: Diagrama UML - Trabajo Práctico 1 - Pizarro, Federico

Este diagrama fue diseñado para facilitar la comprensión visual de la estructura del sistema, sus relaciones de herencia y composición, y cómo se integran las entidades principales del proyecto.