**Trabajo Práctico 1: Paradigmas de la Programación**

**Introducción**

En este informe se encuentra la resolución y el desarrollo de mi Trabajo Práctico 1. Cualquier información acerca de cómo implementé los ejercicios y cómo compilar el código se encontrarán en el mismo.

**Desarrollo**

Ejercicio 1: Estructura

Para resolver el ejercicio, implementé, en total, 23 clases: 2 interfaces, 4 clases abstractas y 17 clases derivadas. Esta organización dio lugar a una estructura que aprovecha los principios de polimorfismo y herencia, que como vimos en clase, son muy importantes en la programación orientada a objetos.

Clases interfaz

* *Personaje*: base de todos los personajes
* *Armas*: base de todas las armas

Clases Abstractas

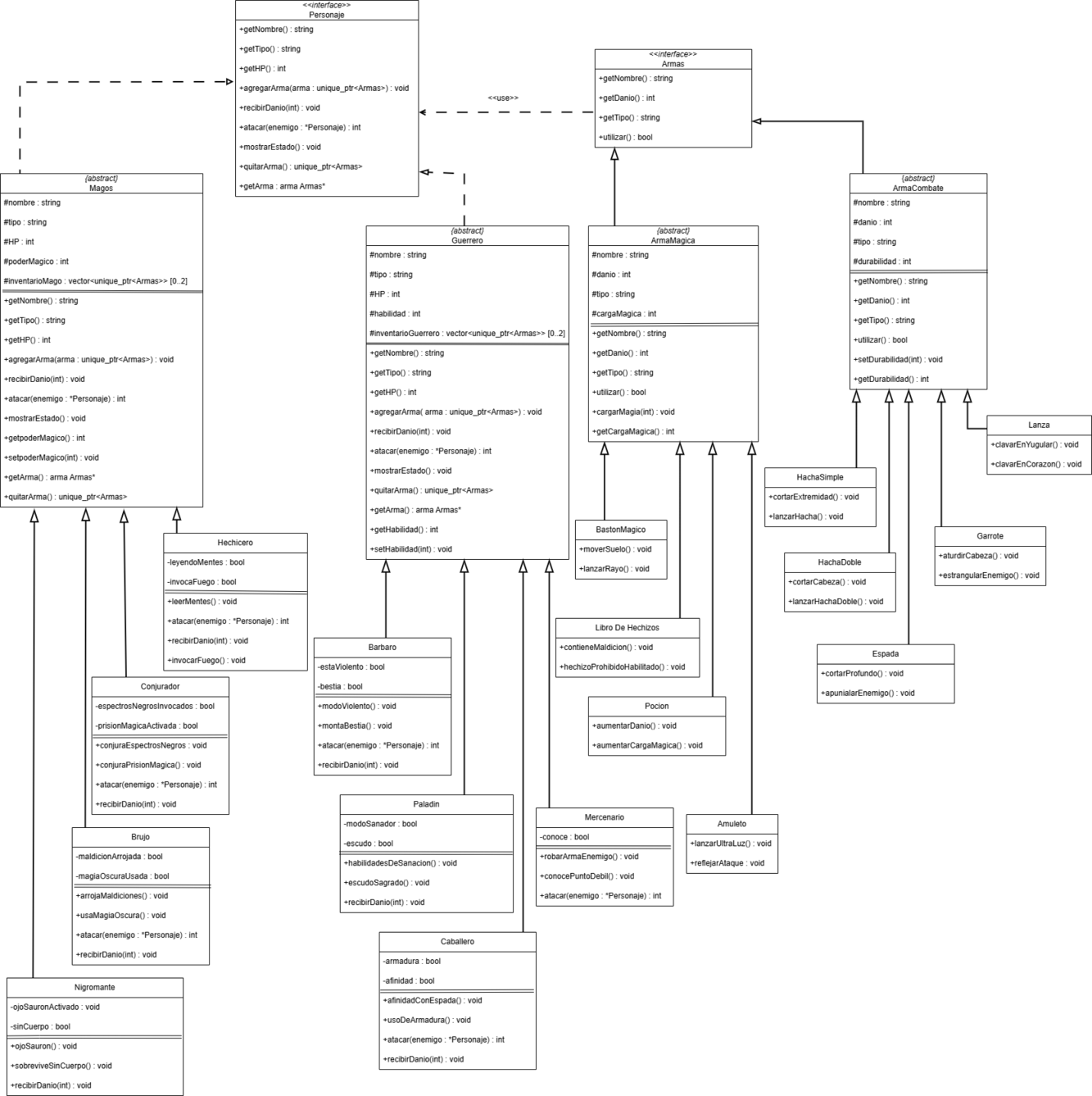
* *Magos*: derivada de Personaje
* *Guerrero:* derivada de Personaje
* *ArmaMagica:* derivada de armas
* *ArmaCombate*: derivada de armas

Clases derivadas

* *Magos: Hechicero, Conjurador, Brujo, Nigromante*
* *Guerrero: Barbaro, Paladin, Caballero, Mercenario*
* *ArmaMagica: BastonMagico, LibroDeHechizos, Pocion, Amuleto*
* *ArmaCombate: HachaSimple, HachaDoble, Espada, Lanza, Garrote*

Diseño de las clases: Métodos y Atributos

Podemos visualizar el diseño de la estructura en este diagrama UML:



Como podemos notar, la relación entre las clases *Personaje* y *Armas* es de composición, lo que significa que un personaje puede tener ningún o más de un arma, controlando explícitamente la existencia de estas, es decir, si el personaje es eliminado entonces las armas también lo serán. Esto cumple con lo pedido en el ejercicio ya que, la relación pedida era “has a” y en este caso la clase contenedora (Personaje), posee al objeto (Armas). Esta relación la podemos ver representada por el uso de *unique\_ptr<Armas>.* La relación de composición se representa con una flecha con punta de diamante pintada, pero al ser interfaces, para indicar quién implementa la interfaz se utiliza una línea discontinua con una flecha vacía desde *Armas* hasta *Personaje* en este caso. Esta flecha se utiliza para identificar la relación de dependencia y se agrega la leyenda *<<use>>.*

*Magos* y *Guerrero* representan tipos generales de *Personaje* y ambas definen atributos comunes como nombre, tipo, puntos de vida y métodos que heredan de *Personaje*.

Magos además implementa:

* Un atributo *poderMagico* que indica el poder del Mago
* *getpoderMagico(), setpoderMagico()*

Guerrero además implementa:

* Un atributo *habilidad* que indica la habilidad del Guerrero
* *getHabilidad(), setHabilidad()*

Las clases derivadas tanto de *Magos* como *Guerrero* heredan los métodos de sus respectivas clases base y además según su *poderMagico* o *habilidad*, pueden tener las capacidades especiales para atacar o defenderse de su enemigo de manera más efectiva. Estas se ven implementadas en el ejercicio 3.

Ejercicio 2: Personaje Factory

Para resolver este ejercicio implementé una clase llamada *PersonajeFactory* la cual contiene tres métodos de tipo *static*: *crearPersonaje*, *crearArma* y *PersonajeArmado*. El uso de *static* evita que tenga que instanciar *PersonajeFactory*, es decir, me permite llamar a las funciones sin la necesidad de crear un objeto. Además, utilizo shared\_ptr para manejar la memoria de manera inteligente, sin tener que preocuparme por liberarla.

Para comprobar el funcionamiento de *Personajefactory* implementé un archivo main en el cual:

* genero una cantidad de personajes *Magos* entre [3 ; 7] de forma aleatoria
* genero una cantidad de personajes *Guerrero* entre [3 ; 7] de forma aleatoria
* le asigno entre [0 ; 2] armas a cada personaje, de forma aleatoria. Si es Mago, tipo *armaMagica* y a cada Guerrero tipo *ArmaCombate*. Durante todo mi trabajo, los Magos sólo pueden tener armas de tipo mágicas y los Guerreros solo pueden tener armas de tipo combate.
* Con el método *mostarEstado()* muestro por consola cada personaje con sus características y armas asignadas

Ejercicio 3: Simula batalla entre dos personajes

Ambos personajes se crean a partir de la clase *PersonajeFactory* y las funciones implementadas en el ejercicio anterior. El jugador1 es elegido por el usuario donde el tipo de *Personaje* y de *Arma* se le es solicitado por consola, mientras que el jugador2 es generado de forma aleatoria.

Como indica el enunciado las reglas de la pelea siguen un modelo de piedra-papel-tijera, en este caso “Golpe Fuerte”, “Golpe Rápido” y “Defensa y Golpe” donde:

* El “Golpe Fuerte” le gana al “Golpe Rápido” y hace 10 puntos de daño a quien lanzó el “Golpe Rápido”.
* El “Golpe Rápido” le gana a la “Defensa y Golpe” y hace 10 puntos de daño a quien lanzó “Defensa y Golpe”.
* Si el personaje usa “Defensa y Golpe” bloquea el “Golpe Fuerte” haciendo 10 puntos de daño a quien lanzó el “Golpe Fuerte”.
* En caso de que los dos personajes realicen la misma acción, ningún personaje recibirá daño.

La batalla se desarrolla por consola, mostrando en cada turno quién ataca, quién recibe el golpe, cuánto daño se produce y cuánta vida (HP) le queda a cada jugador. Ambos comienzan con 100 puntos de HP. Cada arma inflige 10 puntos de daño; si el jugador no posee un arma, el daño base es de 10 puntos también. En caso de contar con una habilidad especial, el daño puede incrementarse o reducirse, dependiendo del tipo de habilidad.

Este enfrentamiento permite poner a prueba los métodos implementados para las clases *Mago* y *Guerrero* del ejercicio 1. Según se trate del Jugador 1 o el Jugador 2, y de acuerdo con su habilidad o poder mágico, pueden recibir un aumento en el ataque o una mejora en la defensa frente a su oponente.

Durante la batalla, si un personaje tiene disponible su capacidad especial, el daño causado se debe calcular en base a dicha habilidad y no al arma. Aunque por consola se muestra tanto el daño que genera el arma como el daño causado por la habilidad, el daño total se refleja en este último. Al mismo tiempo, puede ser confuso que a veces aparezca que el personaje ya no tiene la habilidad o poder mágico para utilizar el arma y aun así aparezca que atacó con más fuerza o se defendió del ataque enemigo. Esto no significa que esté funcionando mal, sino que como los métodos se activan mediante atributos de tipo bool, estos quedan activados de alguna ronda anterior en la batalla aclarando que se utilizarán en la siguiente posibilidad de ataque o defensa, dependiendo que tipo de método sea.

Luego, solo aquellos jugadores que posean armas van a ser capaces de utilizar sus capacidades especiales en los ataques, sino golpearán sin arma. Por último, al inicio de la batalla sólo se imprime por consola la primera arma asignada de cada jugador que es con la que empezarán batallando. Como el jugador2 se genera aleatorio y el jugador1 tiene la posibilidad de elegir sus armas de forma aleatoria, a estos se le pueden asignar entre 0 y 2 armas. Por lo tanto, si en algún momento de la batalla se acaba la durabilidad o carga mágica del arma y esta ya no puede utilizarse, el jugador va a pasar a usar las siguientes si es que se le fueron asignadas.

**Compilación**

Para poder compilar los ejercicios se debe instalar *make* y *g++.* Los ejercicios por compilar son el ejercicio 2 y el ejercicio 3, ambos contienen en sus carpetas un archivo *makefile*, con flags -*Wall* y -*Wextra,* y un archivo main.cpp.

Para ejecutar debe pararse sobre la carpeta indicada haciendo cd nombre\_de\_la\_carpeta y escribir el comando *make*.