Resolución de ejercicios

Tp1 paradigmas de la programación.

Door Lorenzo.
Universidad De San Andrés .

Compilación:

Para la compilación del código cada carpeta de cada ejercicio 2, 3 y 4 (ahora 1, 2 y 3) tiene su respectivo Makefile por ende deberá tenerlo instalado al igual que g + +. Este Makefile compila cada archivo necesario para la funcionalidad del código y prueba con c + + 11 y 17 poniendo en terminal make. Además, utilizo -Wall -Wextra , que no generaron ningún error, ni ningún Warning. Por lo que, los códigos se compilan y están en funcionamiento. Cada ejercicio tiene su propio main.cpp donde se ejecuta lo pedido, o en el caso del ejercicio 2 se hacen pruebas (tests) para mostrar la funcionalidad del mismo ya que por consigna no se pide realizar ningún programa con funcionalidad. Para probarlo luego de hacer el make hay que poner en terminal ./programa que es mi ejecutable.

Ejercicio 2: (ahora 1)

2.1) para empezar con la implementación de las armas pide crear una interfaz:
Arma:
En esta interfaz declaré todas las funciones como virtuales puras, siguiendo la definición de una interfaz. Esto permite que cualquier clase que herede de Arma esté obligada a implementar todos estos métodos, garantizando así una estructura uniforme entre todas las armas, sean mágicas o de combate.

Métodos de la interfaz:

- obtenerTipo(): devuelve el tipo del arma, que puede ser Mágica o De Combate. Para esto uso un enum class ES, que me resulta muy práctico para este tipo de categorizaciones.
- obtenerDano(): devuelve el daño base que inflige el arma.
- obtenerVelocidadAtaque(): devuelve la velocidad de atacar del arma.
- obtenerCostoAtaque(): representa el "costo" del ataque.
- atacar(): este método realiza el ataque. Su implementación varía según el arma.
- obtenerTipoArma(): retorna el subtipo de arma específica, por ejemplo, espada, lanza, bastón, etc. Este tipo también está definido por un enum class TipoDeArma, que está implementado en el archivo de Personaje.

- obtenerPeso(): devuelve el peso del arma.

Una vez definida la interfaz Arma, paso a crear clases abstractas que representan categorías más específicas.

Mágica:
En este caso, la clase Mágica representa todas aquellas armas que pertenecen al tipo mágico, como por ejemplo un bastón o un libro de hechizos.
Esta clase hereda de Arma y a su vez tiene un método virtual por definición (abstracta), habilidadMagica(). Esto implica que no se pueden crear objetos del tipo Mágica directamente, pero sí se pueden crear clases concretas que la hereden (derivadas finales) como antes mencione al bastón o libro de hechizos que luego implementen esta habilidad.

Atributos:
- ES tipo: guarda si es mágica o de combate (en este caso siempre va a ser Mágica).
- int dano: el daño base del arma.
- int velocidadAtaque: la velocidad con la que puede atacar.
- int costoAtaque: recurso que consume al atacar
- int peso: peso del arma.
- TipoDeArma TipoArma: guarda el subtipo del arma (el nombre o que arma es)

Métodos sobrescritos de la interfaz Arma:
Cada uno de los métodos de la interfaz está implementado en esta clase, con comportamientos base comunes a todas las armas mágicas:
- obtenerTipo(): retorna ES::Mágica.
- obtenerDano(), obtenerVelocidadAtaque(), obtenerCostoAtaque(), obtenerPeso(), obtenerTipoArma(): devuelven los valores correspondientes que fueron inicializados en el constructor.
- atacar(): realiza un ataque básico, mostrando el daño por consola y devolviéndole el dano.

Método adicional:
habilidadMagica(): Este método es puro, así que obliga a las derivadas a definir una habilidad mágica específica para cada arma mágica.

Combate:

realiza exactamente lo mismo que la clase abstracta mágica, la unica diferencia que obtenerTipo() retornara ES::Combate y que el método virtual puro se llama ataqueEspecial() que es lo mismo que habilidadMagica() pero esta es para las armas de combate.

Una vez definidas las dos clases abstractas principales (Mágica y Combate), el siguiente paso es crear las clases derivadas finales, que representan armas concretas.
Estas clases heredan de forma directa de Mágica o Combate y completan la implementación de todos los métodos, especialmente aquellos que quedaron como virtuales puros en las clases abstractas (como habilidadMagica() o ataqueEspecial() en el caso de combate).
Cada clase final modela un arma específica con su propio comportamiento, daño, peso, velocidad y habilidades particulares.

Amuleto:
La clase Amuleto es una implementación concreta de un arma mágica. Hereda de la clase abstracta Mágica, por lo tanto debe implementar la función habilidadMagica()
Implementación de métodos:
Constructor: inicializa todos los valores llamando al constructor de Magica con:

- ES::Magica como tipo,
- 0 de daño,
- 1 de velocidad de ataque,
- 10 de costo de ataque,
- 0 de peso,
- TipoDeArma::bastón como subtipo.

habilidadMagica(): tiene un 20% de probabilidad de activarse. Cuando se activa, el amuleto genera un destello mágico que enseguece a los enemigos cercanos. En este caso, retorna el daño (que es 0 por defecto, pero se podría modificar si se quiere que inflija algún tipo de daño). Si no se activa, muestra un mensaje indicando el fallo.

escudoInduvidual(): es un método adicional propio del Amuleto que devuelve una descripción de un escudo mágico. Este escudo no está implementado funcionalmente, pero representa una capacidad adicional del arma.

El resto de derivadas de la clase abstracta Magica son: Baston, libroDeHechizos y Pocion

todas estas armas funcionan igual que el amuleto, solo que para diferenciarlas se inicializan con valores diferentes, además la nueva definición de habilidad (cada una la define diferente) hace que se diferencien entre sí y cada una tiene una habilidad única del arma y que no viene de arriba(de la abstracta y que todas las mágicas tienen). esto lo pensé para darle un toque de originalidad y para que no sean todas las armas iguales.

Espada:
La clase Espada representa un arma de combate concreta. Hereda directamente de la clase abstracta Combate y, como tal, debe implementar la función ataqueEspecial().

Implementación de métodos:
Constructor: Inicializa todos los atributos llamando al constructor de Combate con los siguientes valores:

- ES::Combate como tipo,
- 10 de daño,
- 3 de velocidad de ataque,
- 10 de costo de ataque,
- 20 de peso,
- TipoDeArma::espada como subtipo.

ataqueEspecial(): tiene un 30% de probabilidad de infligir un corte crítico, que duplica el daño del arma. Si no se activa, el ataque realiza el daño base. Esto se implementa utilizando una probabilidad aleatoria con rand().

filo(): es un método adicional propio de la espada, que representa un ataque con filo que aumenta el daño en un 50%. Este método no proviene de la clase base y fue pensado como una habilidad secundaria que distingue aún más esta arma de otras dentro del sistema.

El resto de derivadas de la clase abstracta Combate son: hacha, hachaDoble, garrote, lanza.
todas estas armas funcionan igual que la espada, solo que para diferenciarlas se inicializan con valores diferentes, además la nueva definición de habilidad, hace que se diferencien entre sí y cada una tiene una habilidad única del arma y que no viene de arriba(de la abstracta y que todas las de combate tienen). esto lo pensé para darle un toque de originalidad y para que no sean todas las armas iguales.

Como se ve, las derivadas finales de las mágicas y las de combate funcionan de la misma manera pero con algunos atributos y métodos simples que las diferencias para darle un toque de creatividad.

2.2) Se define una interfaz Personaje, Esta interfaz garantiza un comportamiento uniforme y permite que los personajes puedan ser manipulados de forma polimórfica, sin necesidad de conocer su tipo concreto.

Enumeraciones:

- TipoDeArma: están todas las armas disponibles (las derivadas finales del 2.1)
- TipoPersonaje: están todos los personajes disponibles (las derivadas finales de este inciso que vamos a ver más adelante)

Métodos puros virtuales:

- int obtenerVida() const: devuelve la cantidad actual de vida del personaje.
- TipoPersonaje obtenerTipo() const: devuelve el tipo de personaje (por ejemplo, conjurador o gladiador).
- void recibirDano(int dano): reduce la vida del personaje según el daño recibido.
- void curar(int vida): incrementa la vida del personaje en una cantidad determinada.
- pair<unique_ptr<Arma>, unique_ptr<Arma>>& obtenerArmas(): devuelve las dos armas que tiene equipadas el personaje.
- void equiparArma(unique_ptr<Arma> arma): permite equipar un arma al personaje.
- bool estaMuerto(): indica si la vida del personaje llegó a 0.

Este diseño cumple con las mismas características que la creación de armas, solo que acá se piensa que los personajes poseen esas armas.
Por lo tanto, ahora se definen las dos clases abstractas magos y guerreros.
Magos:
La clase Magos representa todas los personajes que son magos, como por ejemplo un nigromante o un hechicero. Esta clase hereda de Personaje y a su vez tiene un método virtual por definición (abstracta), habilidad(shared_ptr<Personaje> enemigo, unique_ptr<Arma> a). Que tiene un objetivo (el enemigo) y un arma con la que lo ataca. que cada derivada final de los magos la va a definir y va a tener su propia habilidad que la diferencia del resto.

Guerreros:

la clase guerreros funciona igual que los magos, define los atributos que van a estar en el constructor, es decir que van a tener las derivadas, y también, definen los métodos virtuales que vienen de la interfaz. similarmente a como es la creación de armas.
no escribo nuevamente como hice en el 2.1 los métodos y atributos porque es lo que explique arriba en unas oraciones y no creo que sea necesario. pero si decir que en general Cada personaje va a tener sus atributos básicos (definidos en el constructor) y va a poder tener un arma única que le pertenezcan, con esa arma va a poder usar su habilidad para atacar al enemigo.
Ahora veremos un ejemplo de derivada final de un mago y un guerrero para entender mejor esto y poder verlo.

Conjurador:
La clase Conjurador representa un personaje mago especializado en el uso de libros de hechizos. Hereda directamente de la clase abstracta Magos y, como tal, debe implementar la función habilidad().

Implementación de métodos:
Constructor: Inicializa los atributos llamando al constructor de Magos con los siguientes valores:

- TipoPersonaje::conjurador como tipo de personaje,
- 100 puntos de vida,
- 200 puntos de energía,
- false porque no está muerto, al crearse esta vivo,
- un pair de armas recibido por parámetros (arma principal y secundaria), usando unique_ptr. (porque son únicas para este personaje)

habilidad(): este método define la habilidad especial del Conjurador. Recibe como parámetros un enemigo y un arma, y actúa dependiendo del tipo de arma:

- Si el arma es mágica y de tipo libro de hechizos, el Conjurador duplica el daño del arma como bonificación por afinidad.

- Si el arma es mágica pero no es un libro de hechizos, inflige el daño base del arma.

- Si el arma es de combate, también puede utilizarla, pero solo inflige el daño base, sin bonificación alguna.

Paladín:
La clase Paladín representa un personaje de combate especializado en el uso de armas de lucha, en particular, en el uso de espadas. Hereda directamente de la clase abstracta Guerrero y, como tal, debe implementar la función habilidad().

Implementación de métodos:
Constructor: Inicializa los atributos llamando al constructor de Guerrero con los siguientes valores:

- TipoPersonaje::paladín como tipo de personaje,
- 100 puntos de vida,
- 150 puntos de energía,
- false para indicar que no está muerto,
- un pair de armas recibido como parámetros (arma principal y secundaria), utilizando unique_ptr.

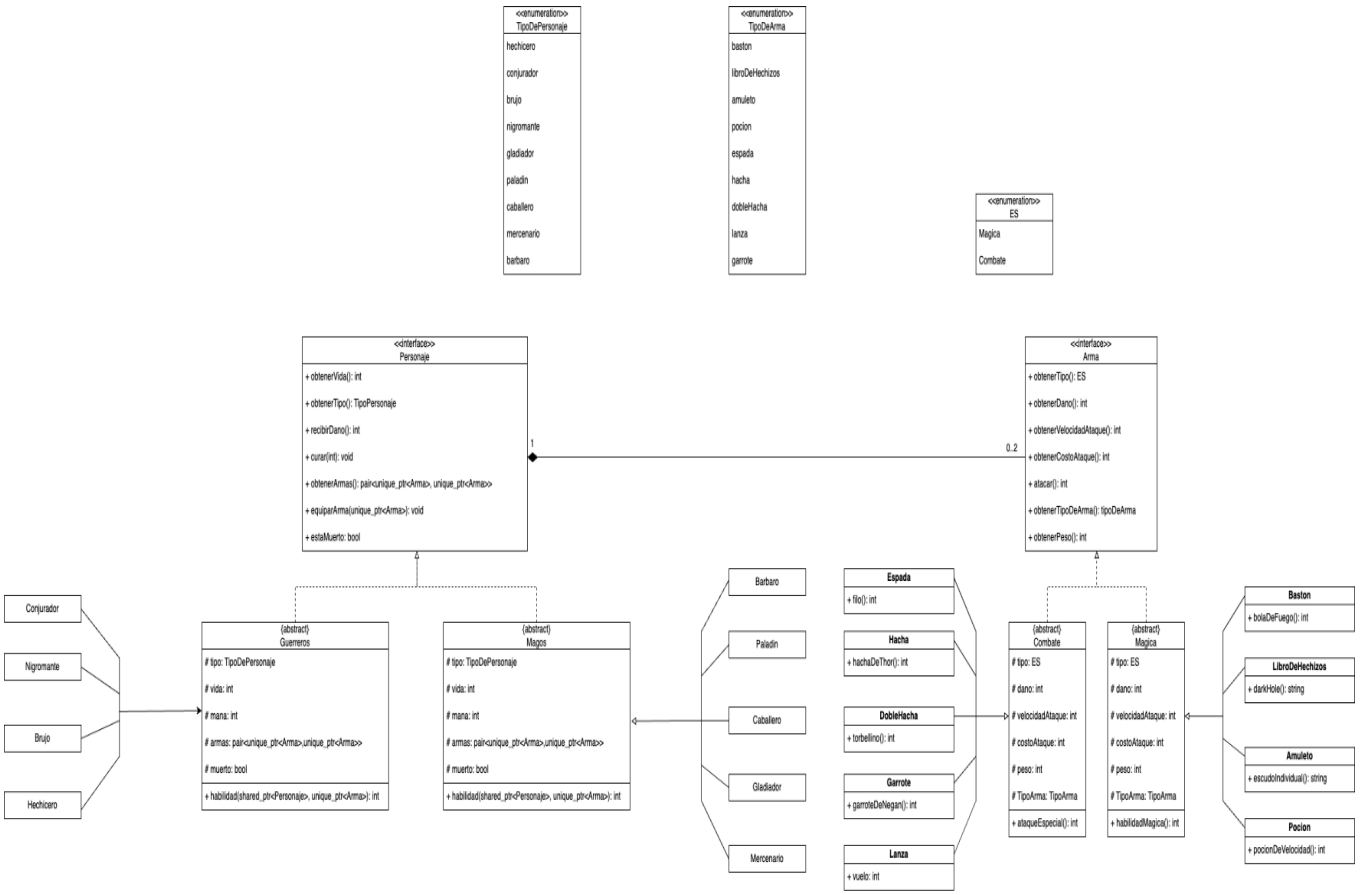
habilidad(): este método define la habilidad especial del Paladín. Recibe como parámetros un enemigo y un arma, y actúa dependiendo del tipo de arma:

- Si el arma es mágica, inflige el daño base del arma.

- Si el arma es de combate y es una espada, el Paladín inflige el doble de daño como bonificación por su afinidad con este tipo de arma.

- Si el arma de combate no es una espada, inflige el daño base del arma sin bonificación.

2.3) Ahora mostraré un diagrama UML mostrando como es la distribución del código para que se amas claro y se entienda mejor.



Ejercicio 3: (ahora 2)

3.1 y 3.2) Este punto lo pensé en conjunto ya que necesito del 3.1 para hacer el 3.2 y me pareció bastante sencillo.

Mi idea fue crear una única función que contenga dos vectores uno con la cantidad de magos y la cantidad de armas que van a poseer y otro igual pero con los guerreros.

¿Cómo se implementa esto?

Lo que hice fue, crear los dos vector y generar dos números random entre 3 y 7

este número va a indicar el largo del vector es decir la cantidad de magos y guerreros que se van a crearer

random1 = 3

random2 = 4

se crea el vector magos de largo 3 y el vector de guerreros de largo 4, ¿cómo se van a completar?

Hay otro generador de números random, que da un número de 0 a 2 y por largo del vector los asigna. Ejemplo:

vector magos: [0,1,2] esto significa que se van a crear 3 magos y el número en cada índice del vector va a significar la cantidad de armas que va a tener

vector guerreros; [1,0,2,1] lo mismo que arriba.

3.3) Para cumplir con la consigna de crear personajes y armas en tiempo de ejecución (run-time) de forma dinámica, diseñe una clase llamada PersonajeFactory que encapsula la lógica de creación de instancias mediante el uso de funciones estáticas y smart pointers (unique_ptr y shared_ptr), lo cual garantiza una gestión segura de memoria.
¿Por qué una Factory?
Este patrón permite desacoplar la lógica de creación de objetos del resto del código. De esta forma, no hace falta conocer los detalles de implementación de cada personaje o arma al momento de instanciar los; basta con pasar el tipo deseado y la fábrica se encarga del resto.

- Método crearArma(TipoDeArma): Este método devuelve un unique_ptr<Arma>, usando un switch para instanciar la subclase correspondiente según el enum TipoDeArma. El uso de unique_ptr garantiza que cada arma creada tenga un solo dueño, lo cual tiene sentido porque un arma solo puede estar equipada por un personaje a la vez.
- Método crearPersonaje(TipoPersonaje): Este método devuelve un shared_ptr<Personaje> con armas nulas (nullptr). La idea es poder crear personajes sin armas, en caso de que se quiera equiparlos más adelante.
- Método crearPersonajeArmado(TipoPersonaje, pair<unique_ptr<Arma>, unique_ptr<Arma>>): Este es el método que permite crear personajes ya equipados con armas en ambas manos. Se reciben dos armas como unique_ptr, y se hace std::move para transferir la propiedad al constructor del personaje, sin copiar ni duplicar el recurso.

Los tres métodos de fábrica son static, por lo que se pueden llamar directamente sin necesidad de instanciar la clase. Esto es consistente con la consigna que pide que no sea necesario crear objetos de la clase PersonajeFactory para usarla.
Además como utilice muchos enums porque me parecen muy prácticos para visualizar las cosas y para los switchs, necesite crear funciones que me pasen los tipos de armas y personajes a string para poder mostrarlo en pantalla.

Ejercicio 4: (ahora 3)
La lógica del juego es:
Primero, el jugador elige su personaje y un arma, ambos seleccionados a través de opciones numéricas que se traducen en tipos definidos por los enum class TipoPersonaje y TipoDeArma. Estos valores se utilizan para instanciar objetos mediante la clase

PersonajeFactory, que devuelven smart ptr a personajes con sus respectivas armas equipadas.

El oponente es generado de forma aleatoria, tanto en su tipo de personaje como en el arma que porta, utilizando funciones auxiliares como crearPersonajeRival. Esta aleatoriedad le aporta variabilidad y dinamismo a cada partida.

Durante el combate, los ataques disponibles son tres: Golpe Fuerte, Golpe Rápido y Defensa y Golpe, representados por el enum Ataque. El jugador elige su ataque manualmente, mientras que el ataque del oponente se genera de forma aleatoria.

La resolución de cada ronda se lleva a cabo en la función resolverRonda, la cual compara los ataques seleccionados por cada personaje. Siguiendo una lógica similar a un sistema de piedra-papel-tijera, uno de los personajes recibe daño si su ataque es superado por el del oponente. La función muestra además los puntos de vida actuales de ambos personajes luego de cada enfrentamiento.

Finalmente, el ciclo del juego continúa hasta que uno de los personajes pierde todos sus puntos de vida, concluyendo así la partida.

conclusiones finales del tp:
Durante el desarrollo del trabajo práctico, uno de los mayores desafíos que enfrenté fue el aspecto creativo. Al comenzar, interpreté que debía implementar un juego completo y funcional, más allá del sistema tipo piedra-papel-tijera, lo cual me llevó a intentar desarrollar funciones y estructuras complejas que luego se volvieron difíciles de sostener. Esta confusión inicial hizo que decidiera comenzar el TP nuevamente desde cero, con un enfoque más simple y concreto.
En este segundo intento, opté por diseñar los personajes y las armas de manera conjunta, lo que me permitió ir probando todo al mismo tiempo y asegurarse de que todo vaya funcionando. Esta estrategia me ayudó a mantener una base más estable y coherente en el desarrollo.
Más adelante, al trabajar con la clase Factory, me encontré con una nueva dificultad: no comprendía cómo debía pasar correctamente los parámetros para crear un personaje que ya viniera con su arma equipada. Esto me llevó a modificar nuevamente a los Personajes, cambiando su constructor para que aceptara directamente un arma como parámetro.

Finalmente, uno de los errores conceptuales más significativos que detecté fue en la relación de composición entre Personaje y Arma. Inicialmente, utilicé shared_ptr para las armas, sin considerar que la composición significaba que cada arma pertenece a un único personaje, por lo que debía usarse unique_ptr. Esta corrección implicó un cambio importante en todo mi código, desde el ejercicio 2 hasta el 4. Si bien el reemplazo de shared_ptr por unique_ptr en las armas no fue complejo a nivel técnico, sí requirió tiempo y una revisión minuciosa de todo el código. En el caso de los personajes, no fue necesario realizar este cambio, ya que no forman parte de una relación de composición (como sí lo sería, por ejemplo, en el caso de un equipo que posee varios personajes).

En resumen, este TP me enfrentó a decisiones de diseño y organización del código que me ayudaron a comprender mejor los conceptos necesarios para su desarrollo.