Nota: Aunque el trabajo práctico incluye una carpeta correspondiente al Ejercicio 1, este fue eliminado del enunciado final luego de varias modificaciones por parte del profesor. Se decidió conservarlo en la entrega para mantener la trazabilidad del desarrollo realizado y justificar la estructura del proyecto.

El profesor indicó via Gmail:

“Si lo hubiera dejado tendrías que hacer los ejercicios 2, 3 y 4 de todas formas. Si querés entregalo y si veo que te falta puntaje para aprobar veo de tomarlo de ese ejercicio, pero los otros ejercicios de una forma u otra los tenías que hacer.”

Por lo tanto, lo incluyo en mi entrega del TP1 como material adicional.

Para compilar cualquier ejercicio, por favor consulte el archivo README.md, donde se detalla el comando necesario para compilar y ejecutar cada uno.

Ejercicio 1:

Este ejercicio tenía como objetivo modelar una estructura organizativa a partir de un diagrama UML ambiguo. El sistema debía representar entidades jerárquicas como **centrales regionales**, **compañías**, **departamentos** y **empleados**, manteniendo relaciones consistentes y restricciones de negocio entre ellas.

Para abordar el problema, se establecieron tres pilares base en el diseño de clases:

1. Organizational_Entity: interfaz abstracta que sirve como base para Company y Central_Regional.
2. Department: clase concreta que encapsula empleados y funciones de gestión interna.
3. Employee: clase que representa a cada trabajador, incluyendo atributos de puesto, salario y antigüedad.

Relaciones entre clases

- **Central_Regional** hereda de **Organizational_Entity** y representa la entidad organizativa de más alto nivel. Contiene:
    - Un conjunto de países (set<string>).
    - Un máximo de 20 Middle Managers y 5 Senior Managers, almacenados como vector<shared_ptr<Middle_Manager>> y vector<shared_ptr<Senior_Manager>> respectivamente.
    - Un conjunto de compañías (set<shared_ptr<Company>>), añadidas mediante el método add_entity(), definido en la interfaz base.
    - Un contador de managers (manager_amount) que facilita el cálculo de personal jerárquico.
- **Company** también hereda de **Organizational_Entity** y representa una empresa individual. Contiene:
    - Una lista de departamentos (vector<shared_ptr<Department>>).
    - Una dirección modificable (address).

    - Métodos para añadir, quitar o recuperar departamentos por nombre, y para obtener los nombres de todos los departamentos existentes.
- **Department** es una clase independiente que contiene:
    - Un vector<shared_ptr<Employee>> con todos los empleados del departamento.
    - Una variable estática employee_amount, compartida por todas las instancias, que contabiliza la cantidad total de empleados en el sistema.
    - Métodos para contratar (hire()) y despedir (fire()) empleados, así como acceder a todos ellos y consultarlos individualmente.
- **Employee** es la clase base de cada trabajador. Incluye:
    - Nombre, puesto (job), salario y antigüedad.
    - Métodos para modificar estos atributos, como update_salary(), update_seniority() y change_job().
    - La antigüedad (seniority) puede actualizarse anualmente (comentado como una función a automatizar en el futuro).

Una de las ambigüedades del UML original era cómo calcular la cantidad de empleados que tenía una central regional. La consigna pedía un int directo, pero eso no contemplaba adecuadamente empleados distribuidos en múltiples compañías y departamentos. Para resolver esto, se optó por una estrategia flexible, que incluye tres formas distintas de conteo:

1. **Conteo global del sistema:**
    a. Sumando el total de managers (manager_amount) más el valor estático global de empleados (Department::employee_count()).
2. **Conteo local de una central regional:**
    a. Se recorren todas las compañías asociadas a la central. En cada compañía, se recorre cada departamento y se suman los empleados locales (local_employee_count()), generando un conteo real de los empleados pertenecientes únicamente a esa rama organizativa.
3. **Conteo exclusivo de managers:**
    a. Se devuelve simplemente la cantidad de middle y senior managers almacenados en la central.

Este enfoque permitió además extender el sistema para que **existan múltiples centrales regionales** sin superposición de datos. Si se hubiera utilizado solamente el contador estático global, habría sido imposible distinguir cuántos empleados pertenecen a una central específica.

Algunos punto de la decisión del diseño:

- Se utilizó shared_ptr en casi todas las relaciones jerárquicas (centrales con compañías, compañías con departamentos, departamentos con empleados) no solo para facilitar la gestión automática de memoria, sino también porque se asumió que ciertas entidades podrían ser compartidas entre diferentes ramas organizativas. Por ejemplo, se consideró razonable que una misma compañía pudiera compartir departamentos o empleados con otras compañías dentro de la misma región, y que un manager, especialmente en niveles altos, pudiera operar simultáneamente en

más de una central regional. Este diseño favorece la reutilización de nodos en la estructura sin restricciones innecesarias, y permite reflejar dinámicas realistas del mundo corporativo donde ciertas figuras o áreas funcionales pueden intervenir en múltiples unidades.
- Se implementaron funciones como get_company_names(), get_department_names() y get_employees() para permitir inspección y debugging de forma cómoda.
- Cada clase mantiene encapsulado su estado, con métodos para modificar datos sin exposición directa de sus miembros privados.

Este ejercicio requirió numerosas decisiones de interpretación, ya que el UML original no especificaba ni el tipo de punteros a usar, ni los detalles de propagación de empleados, ni cómo evitar conflictos entre múltiples centrales. A pesar de haber sido eliminado del práctico por el profesor, se conserva en esta entrega para documentar el desarrollo realizado y demostrar la capacidad de adaptar modelos incompletos a implementaciones funcionales y escalables.

Ejercicio 2 (ahora 1):

Para las armas:

Se definió una interfaz Weapon con métodos virtuales puros que representan el comportamiento básico esperado de cualquier arma, como get_name(), get_type(), get_weight() y display(). De esta interfaz se derivan dos clases abstractas: Combat_Weapon y Magic_Item.

Cada arma tiene estadísticas iniciales fijas, elegidas por mí para mantener un entorno balanceado entre los distintos tipos de armas y facilitar su comparación. Se implementaron constructores vacíos en todas las clases concretas para simplificar la creación de instancias dentro de la fábrica y permitir generación aleatoria sin necesidad de pasar múltiples parámetros.

Las armas de combate (Axe, Sword, Spear, Double_Axe, Club, etc.) definen atributos como damage, weight, crit_chance, crit_multiplier y level. También pueden estar marcadas como **“enchanted”**, lo que modifica sus estadísticas base, por ejemplo aumentando el daño o reduciendo el peso.

Los ítems mágicos (Potion, Amulet, Spellbook, Staff) definen atributos como effect_strength, rarity (que puede ser Common, Rare, Epic o Legendary), weight y uses. Estas rarezas afectan la intensidad del efecto aplicado por el objeto. A su vez, el atributo uses fue agregado para evitar un uso ilimitado de los ítems, lo cual podría llevar a abusos o buffs automáticos constantes. Como medida adicional, se estableció también un atributo max_hp en los personajes, para asegurar que no se pueda superar su salud máxima al usar, por ejemplo, una poción. Esta max_hp aumenta dinámicamente al subir de nivel.

Para los personajes:

Se definió una interfaz Character, de la cual derivan las clases abstractas Wizard y Warrior. Estas últimas se especializan luego en clases concretas como Paladin, Knight, Barbarian, Gladiator, Sorcerer, Conjurer, Warlock, Necromancer, entre otras.

Cada personaje mantiene un vector<unique_ptr<Weapon>>, representando la composición con sus armas. Se utilizó unique_ptr para reflejar que el personaje es dueño exclusivo de las armas que posee. Además, los ítems mágicos (Magic_Item) mantienen un shared_ptr<Character> hacia su dueño, lo cual permite aplicar efectos directamente sobre él desde el objeto.

Todos los personajes tienen stats iniciales fijos, como hp, intelligence, strength, defence, definidos directamente en sus constructores. Esto permite mantener un sistema equilibrado y evita tener que ajustar dinámicamente los valores al momento de la creación. Adicionalmente, un personaje puede ser instanciado como **“Master”** (si es mago) o **“Lord”** (si es guerrero), lo cual incrementa sus estadísticas base desde el inicio.

A nivel funcional, todos los personajes implementan métodos como attack(), receive_damage(), get_hp(), get_type(), display() y get_weapon_by_name(). El método receive_damage() aplica el daño recibido reduciéndolo según la defensa (defence) del personaje, y luego descuenta el daño neto sobre sus puntos de vida (hp). La variable max_hp, como se mencionó antes, actúa como límite para evitar curaciones que excedan la salud máxima del personaje.

Cada personaje también posee una habilidad especial según su clase:

- Los magos utilizan el atributo **intelligence**, que determina la probabilidad de éxito del ataque especial cast_spell().
- Los guerreros dependen de **strength** para potenciar su ataque especial ram_attack().

Además, se implementó un sistema de experiencia y progresión por niveles. A través del método add_xp(float xp), los personajes acumulan experiencia al atacar o usar habilidades. Cuando la experiencia acumulada supera un cierto umbral (threshold), el personaje sube de nivel, lo cual incrementa su salud máxima (max_hp) y regenera parcialmente su vida actual (hp). El threshold también crece dinámicamente para hacer que cada nuevo nivel sea más difícil de alcanzar. Este sistema también se aplica a las Combat_Weapon, aumentando su daño.

Este sistema garantiza una progresión fluida y balanceada durante el juego, con una curva de dificultad creciente pero también con mejoras al subir de nivel.

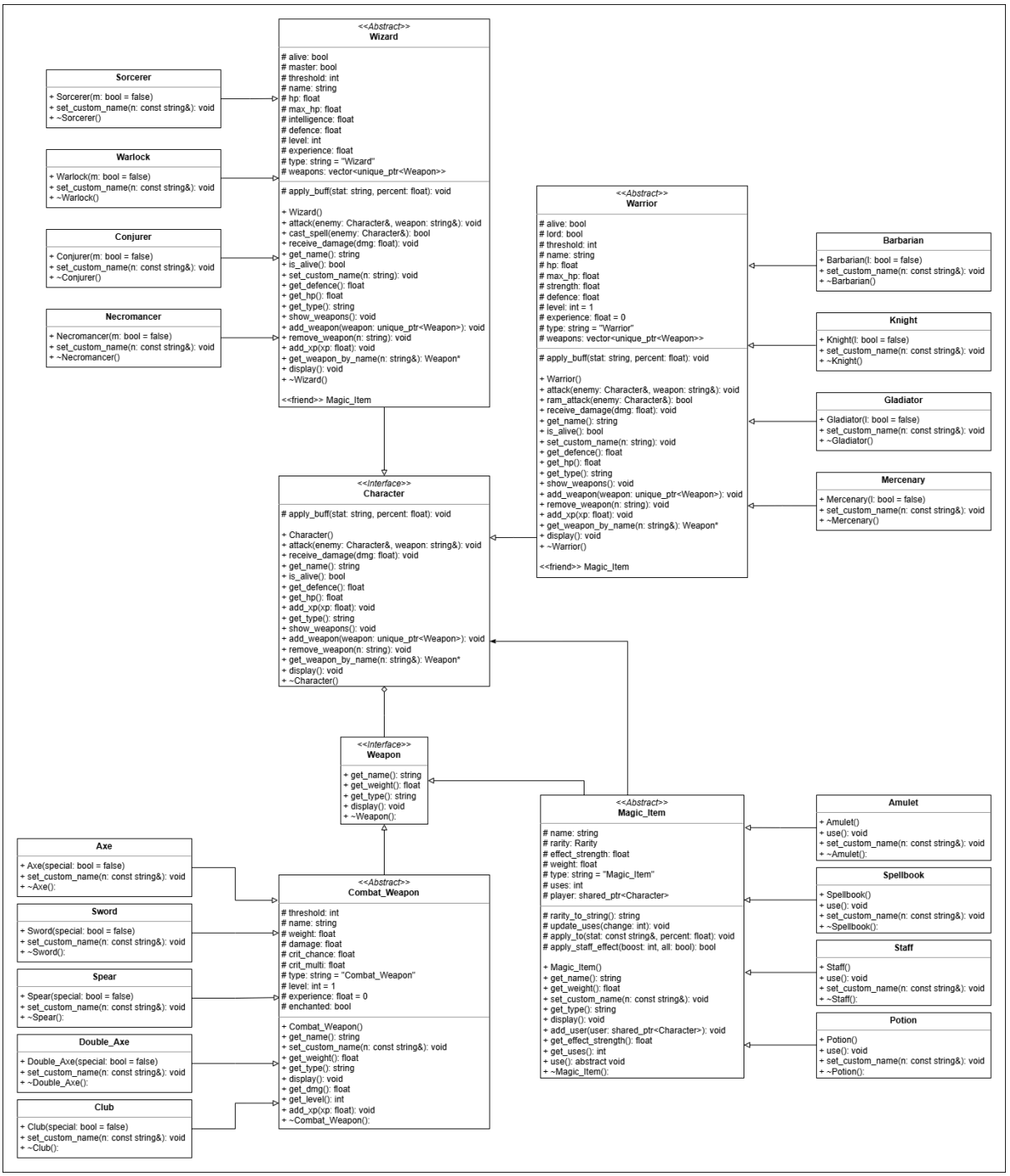
Por otro lado, cada **Magic_Item** implementa su efecto de forma única:

- **Potion**: regenera puntos de vida (hp).
- **Amulet**: mejora la defensa (defence).
- **Spellbook**: potencia la habilidad especial (intelligence o strength, según la clase).
- **Staff**: incrementa los usos disponibles de los demás ítems mágicos en posesión.

Para que un Magic_Item pueda modificar atributos específicos del personaje, se implementó un método apply_to(string stat, float percent), que llama internamente al método protegido apply_buff(stat, percent) del personaje. Este último se encuentra definido como virtual puro en la clase Character. Para permitir esta comunicación sin violar encapsulamiento, la clase Magic_Item fue declarada friend tanto de Wizard como de Warrior. Esto permite modificar

atributos internos sin exponerlos directamente y sin romper la abstracción general del sistema.

Aqui esta el UML de estas relaciones:



Ejercicio 3 (ahora 2):

Se creó la clase Character_Factory para crear de forma dinámica personajes y armas. Esta clase agrupa métodos estáticos que retornan smart pointers (unique_ptr) a objetos del tipo deseado, con posibilidad de instanciación aleatoria o específica según el método. Las funciones principales implementadas fueron create_character_by_type(string type, string weapon), create_random_character() y create_random_weapon().

Para los casos aleatorios:

- Se utilizó rand() junto con srand(time(nullptr)), inicializado una única vez en main(), para asegurar que las instancias generadas fueran distintas entre ejecuciones.
- Se seleccionó un personaje aleatorio de entre los magos y guerreros disponibles, y un arma correspondiente mediante un número aleatorio acotado al rango de opciones posibles (rand() % size de las opciones).
- Esta estrategia permite realizar pruebas rápidas con combinaciones variadas.

Desde el punto de vista del diseño de la interfaz, se optó por solicitar al usuario que indique, por consola, qué tipo de personaje y arma desea crear. Si se deja la respuesta en blanco, se interpreta que se desea una instancia aleatoria. Esta lógica fue pensada para facilitar la prueba. Si se quiere, se puede forzar un personaje puntual, o simplemente generar muchos casos aleatorios de manera rápida.

Tanto los personajes como las armas son creados usando unique_ptr, lo cual refleja una relación de propiedad exclusiva y facilita la gestión automática de memoria sin necesidad de liberación manual. En particular, para que un personaje obtenga un arma, se utiliza std::move(weapon) al momento de pasar el puntero único al método add_weapon(). Esto asegura que el ownership del arma pase correctamente al personaje y evita duplicaciones o pérdidas de memoria.

Los métodos de Character_Factory fueron definidos como estáticos ya que no requieren mantener ningún estado interno ni instancia propia de la clase. Su única responsabilidad es construir objetos y retornarlos según los parámetros recibidos, por lo que no tenía sentido obligar al usuario a instanciar una fábrica. Esta decisión también mejora la claridad de uso y reduce el acoplamiento en el código cliente.

Una vez instanciado el personaje, ya sea de forma aleatoria o manual, se utiliza el método display() para imprimir en consola su información básica: nombre, clase, tipo, HP actual, y los datos del arma equipada.

Ejercicio 4 (ahora 3):

Se desarrolló un juego de combate siguiendo las características indicadas por la consigna, implementando una batalla entre dos personajes en base a un sistema de decisión tipo “piedra, papel o tijera”. El jugador elige una de tres acciones posibles: Golpe Fuerte, Golpe Rápido o Defensa y Golpe. El enemigo elige aleatoriamente una opción utilizando rand(). Cada acción gana, pierde o empata frente a las otras según la lógica pedida. Si una acción gana, se inflige un daño fijo de 10 puntos de vida, mientras que los empates no tienen consecuencias. Los personajes están seteados a 100 de vida.

Para simplificar y enfocar el código en la dinámica pedida, se eliminaron completamente los ítems mágicos (Magic_Item), ya que no contribuían a la lógica del juego ni tenían lugar dentro del flujo de combate definido. También se recortaron significativamente los atributos y métodos de los personajes, conservando solo aquellos estrictamente necesarios para manejar el HP, el ataque, y la impresión por pantalla. Esto implicó borrar más de la mitad de las variables y funciones que sí estaban presentes en ejercicios anteriores, dejando una versión liviana de las clases.

Se fijaron valores constantes para el HP (100) de los personajes y el daño (10) de las armas para mantener el sistema simple y según lo pedido, evitando la influencia de estadísticas secundarias como strength o intelligence, que no eran relevantes para este ejercicio ya que no existía otra forma de combate que no sea con la lógica del “piedra, papel o tijera”.

A nivel de interacción, se diseñó una interfaz por consola, donde se informa el HP actual de ambos personajes, se solicita al jugador su elección, se imprime qué acción eligió cada uno y luego se muestra el resultado del turno. El método attack() de cada personaje ya se encarga de imprimir mensajes como el nombre del atacante, el arma utilizada y el daño infligido, por lo que el bucle de combate se mantuvo conciso y sin duplicación de lógica.

Esta versión acotada del sistema sirvió como demostración de la adaptabilidad de mis clases y funciones.

Conclusión

Este trabajo práctico me dejó varias herramientas importantes tanto en lo técnico como en lo conceptual. Pude aplicar de forma concreta los principios de la programación orientada a objetos, especialmente en lo que respecta al uso de herencia, clases abstractas, punteros inteligentes, y estructuras jerárquicas. También me ayudó a entender cómo organizar y modular un proyecto complejo, con múltiples carpetas, relaciones entre clases y distintos niveles de abstracción.

Al mismo tiempo, el proceso fue mucho más caótico de lo esperado. Hubo muchas inconsistencias, correcciones tardías y ambigüedades en el enunciado, especialmente en el primer ejercicio, que terminó siendo eliminado pero igual demandó tiempo y trabajo para poder entenderlo. Esto me obligó a tomar decisiones por mi cuenta, interpretar libremente diagramas poco claros y reestructurar partes del sistema que no estaban del todo especificadas.

Más allá de eso, creo que logré construir una solución sólida, bien pensada y funcional. A pesar del contexto, me llevo una experiencia de desarrollo bastante cercana a la realidad: tener que avanzar sin que todo esté perfectamente definido, adaptarse a cambios y resolver problemas con criterio propio.