Battler for Olympus

Course	Programación
Complete	☐
Person	M Macarena

Documentación del Juego de Rol

1. Introducción

Este documento describe la implementación del juego de rol basado en la mitología griega, centrándose en el cumplimiento de los requisitos mínimos y además del uso de patrones de diseño, estructuras de datos avanzadas y la cobertura de pruebas unitarias.

2. Requisitos Implementados

1. **Estructura del Modelo**
   - Se ha implementado una jerarquía de clases para representar los personajes del juego.
   - Cada personaje pertenece a una categoría (dios, héroe, monstruo, etc.).
   - Se han definido atributos esenciales como nombre, tipo, salud, ataque, defensa y suerte
2. **Relación entre `Character` y `Potion` (Agregación)**

   En este caso, un personaje puede tener varias pociones, pero las pociones no dependen directamente de un personaje específico. El `Character` usa `Potion` como una relación de **agregación**. Es decir, las pociones pueden existir independientemente de los personajes, pero un personaje puede contener y usar varias.

   ```
   private final List<Potion> potions;
   ```

3. **Relación entre `Character` y `Weapon` (Agregación)**

   En este caso, un personaje puede tener un arma, pero las armas no dependen directamente de un personaje específico. El `Character` se equipa de un `Weapon` como una relación de **agregación**. Es decir, las pociones pueden existir independientemente de los personajes, pero un personaje puede contener y usar varias.

   ```
   private Weapon weapon;
   ```

4. **Herencia**

   La herencia está implementada con la clase `Character` como clase base abstracta y las subclases `God` , `Hero` , `Human` , `MythologicalAnimal` y `Titan` que extienden `Character` .

   ```
   public class God extends Character {
       private God(GodBuilder builder) {
           super(builder);
       }
   }
   ```

   Este patrón permite que `God` , `Hero` , `Human` , `MythologicalAnimal` y `Titan` compartan el comportamiento común de `Character` (como el método `attack()` , `receiveDamage()` , etc.), pero también pueden tener comportamientos adicionales si es necesario.

5. **Polimorfismo**

   El polimorfismo se manifiesta cuando se invoca el mismo método (`attack()` , por ejemplo) en diferentes tipos de objetos (`God` , `Hero` , `Human` , `MythologicalAnimal` y `Titan`). Cada tipo de personaje puede tener su propia implementación del método, y el comportamiento se resuelve en tiempo de ejecución.

6. **Array de Armas (`Weapon[]`)**

   - El array `weapons` de la clase `WeaponController` almacena las instancias de armas (`Weapon`).

- Se define un límite máximo de armas con la constante `MAX_WEAPONS`, asegurando que el array no crezca más allá de este límite.
- **Agregación de elementos al Array**:
  - El método `addWeapon` agrega nuevas armas al array, incrementando el contador (`counter`) que rastrea cuántas armas han sido agregadas. El array se llena mediante este método.
- **Acceso a elementos del Array**:
  - El método `getWeaponByIndex` permite acceder a las armas almacenadas en el array mediante su índice, proporcionando una forma de recuperar una arma específica.
  - Además, el método `getRandomWeapon` selecciona una arma aleatoria del array, proporcionando diversidad en el combate.
- **Gestión de límites**:
  - Se asegura que el número de armas no exceda el límite predefinido (`MAX_WEAPONS`) para evitar que el array crezca más allá de su capacidad. Si se intenta agregar más armas de las permitidas, se lanza una excepción `IllegalStateException`.
- **Cumplimiento de los requisitos de combate**:
  - Al usar este array para almacenar las armas, se hace posible la asignación de armas a los personajes, como se ve en el método `equipWeapon`, que le asigna un arma a un personaje.

7. **Sistema de Combate**
   - **Inicio del Combate:**
     - Se presenta la opción de jugar contra otro jugador o contra la máquina (modo aleatorio).
     - Se muestran los personajes que participarán en el enfrentamiento.
   - **Turnos de los Personajes:**
     - Cada personaje, en su turno, puede realizar una de las siguientes acciones:

1. **Atacar:** Se calcula el daño basado en el ataque del personaje y un posible golpe crítico.
2. **Usar una Poción:** Si el personaje tiene pociones disponibles, puede utilizarlas para recuperar salud.
3. **Equipar un Arma:** Puede cambiar su arma actual por otra disponible para modificar sus estadísticas de ataque.

- **Ejecución de Acciones:**
  - Si el personaje es manejado por la máquina (modo aleatorio), el sistema elige una acción aleatoriamente.
  - Si el personaje es manejado por un jugador, este selecciona su acción manualmente.
- **Cálculo del Daño y Condiciones Especiales:**
  - Un ataque básico causa daño equivalente al atributo de ataque del personaje atacante.
  - Existe una probabilidad de golpe crítico que triplica el daño causado.
  - El daño se resta a la salud del personaje defensor, y se muestra el estado actual del combate.
- **Finalización del Combate:**
  - El combate continúa hasta que uno de los personajes pierde toda su salud.
  - Se muestra un mensaje indicando el ganador de la batalla.

8. **Patrones de Diseño Aplicados**

- **Builder:** Se usa en la clase `Character` y sus subclases (como `God`, `Hero`, `Human`, `MythologicalAnimal`, `Titan`).
  - Permite construir objetos complejos paso a paso sin necesidad de múltiples constructores con diferentes combinaciones de parámetros.
  - Ventajas:
    - Facilita la creación de personajes con diferentes atributos sin necesidad de múltiples constructores.

        - Mejora la legibilidad y mantenimiento del código.
        - Permite extender fácilmente la creación de personajes con nuevos atributos en el futuro.
    - **Factory Method:** Se usa en la clase `CharacterFactory` para centralizar la creación de personajes según su tipo.
        - Ventajas:
            - Desacopla la creación de los objetos del código que los usa.
            - Simplifica la lógica en otras partes del código, ya que no necesitan conocer los detalles de la creación de cada personaje.
            - Permite añadir nuevos tipos de personajes sin modificar el código existente.
9. **Estructuras de Datos Complejas**
    - **Streams:** Se han utilizado `Stream()` para filtrar y manipular colecciones de personajes, armas y pociones.
    - **HashMap:** Se ha empleado para almacenar y recuperar rápidamente los personajes y armas mediante identificadores.

3. Implementación de las Pruebas Unitarias

Se han desarrollado pruebas unitarias en JUnit para verificar la correcta funcionalidad de los controladores:

3.1. Pruebas de `CharacterController`

- `addCharacter`: Verifica que un personaje se ha agregado correctamente con sus atributos definidos.
- `findCharacterByName`: Comprueba la búsqueda de personajes por nombre.
- `findCharacterByName_NotFound`: Prueba la excepción cuando el personaje no existe.
- `getRandomCharacter`: Confirma que se devuelve un personaje aleatorio válido.
- `getAllCharacters`: Valida que la lista de personajes no esté vacía.

3.2. Pruebas de `CombatController`

- `testAttack`: Asegura que un ataque reduce la salud del oponente.

- `testEquipManualWeapon`: Prueba la asignación manual de un arma a un personaje.

3.3. Pruebas de `PotionController`

- `assignPotionToCharacter`: Comprueba que una poción se asigna correctamente.
- `usePotion`: Verifica que el uso de una poción incrementa la salud del personaje.
- `removePotionFromCharacter`: Confirma que una poción puede ser eliminada correctamente del inventario.

3.4. Pruebas de `WeaponController`

- `createWeapon`: Comprueba la creación de un arma con valores correctos.
- `equipWeapon`: Asegura que el arma se asigna correctamente a un personaje.
- `getWeaponByIndex`: Verifica la recuperación de un arma por índice en la lista de armas.

4. Conclusión

El juego de rol basado en la mitología griega ha sido implementado siguiendo principios de diseño de software, utilizando patrones de diseño y estructuras de datos eficientes. Las pruebas unitarias garantizan su correcto funcionamiento y permiten futuras ampliaciones con robustez.