PROYECTO FINAL

MAR IBORRA

ALINA ROJAS

LAURA LLORENTE

LAIA DELGADO

DIEGO GARCÍA-ROMERO

LUCAS GARCÍA-ROMERO

FERNANDO CONTRERAS

DISEÑO DE SOFTWARE

2ºA MAIS U-TAD

Índice

1 Descripción general de la solución

El objetivo de este proyecto es desarrollar un juego de ordenador que se juega por consola, en el que el protagonista va combatiendo contra los enemigos que se va encontrando, utilizando para ello diversas armas, distintas acciones, etc.
La dinámica del juego se desarrolla por turnos, donde el jugador y la máquina se van turnando en cada combate. El jugador podrá elegir entre una serie de acciones disponibles, atacar o defenderse, y, tras cada ataque, el programa evaluará los daños causados al oponente, pierde el combate aquel contrincante que primero se quede sin vida. Con cada combate ganado el jugador consigue pasar hacia el siguiente mundo/escenario y el juego continúa combate tras combate, contra distintos tipos de enemigos, hasta que el jugador pierde o se llega al escenario final. Todos los personajes, tanto el jugador como los enemigos, tendrán una serie de atributos que afectarán a los resultados de los combates.
Habrá distintas categorías de enemigos, cada uno con sus características propias. Las mismas categorías aplican en todos los mundos/escenarios, pero en cada uno de dichos mundos cada categoría de enemigo presenta características diferentes. Cada enemigo se definirá también por una determinada estrategia, de entre varias disponibles, defensiva, ofensiva o una por defecto.
Para la realización de este proyecto se han utilizado diferentes patrones de diseño:

(1) **Patrón Strategy:** se ha utilizado para gestionar las estrategias de los enemigos en cada combate, permitiendo al sistema seleccionar la estrategia más adecuada para cada enemigo.

(2) **Patrón Decorator**: se ha utilizado para gestionar las distintas armas que tienen los personajes, permitiendo decorar su inventario básico con modificadores que dependerán de las características del personaje.

(3) **Patrón State:** se ha utilizado para controlar el estado del jugador durante el combate, permitiendo que pase de un estado a otro según las condiciones en las que se encuentre el jugador y dependiendo de las incidencias del combate.

(4) **Patrón Abstract Factory:** se ha utilizado para crear los diferentes tipos de enemigos, adaptados a cada mundo/escenario.

(5) **Patrón Singleton**: se ha utilizado en la clase encargada de hacer los cálculos sobre el resultado de los ataques, permitiendo que solo haya una instancia de dicho "calculador" en el sistema. También se implementa este patrón para la clase GameController, de la cual solo existirá una instancia.

(6) **Patrón Template Method**: se ha utilizado para implementar el algoritmo de cada clase Enemigo, permitiendo que decida la siguiente acción a realizar.

(7) **Patrón Facade**: se ha utilizado para la clase **GameController**, permitiendo simplificar la interfaz para que los clientes puedan interactuar con el sistema de manera más sencilla.

2 Componentes de la solución

Se dará una descripción general del programa, organizando esta por los distintos patrones implementados para aumentar su claridad:

2.1 Strategy & template

Interfaz ServiceStrategy: Es una interfaz de servicio que define un único método, applyStrategy(), que se utilizará para definir diferentes estrategias que serán implementadas en las clases de las estrategias concretas.
Para el patrón Template, este método nos servirá para posteriormente definir el orden de la secuencia en la que se ejecutarán los métodos de la interfaz EnemyStrategy.

Interfaz EnemyStrategy: Es una interfaz que define el patrón de servicio y extiende de la interfaz ServiceStrategy.
Define tres métodos:

- **DamagePlayer:** se usará para notificar de que el enemigo está atacando, imprimiendo un mensaje por consola.
- **ApplyDefense**: se usará para notificar de que el enemigo se está defendiendo, imprimiendo un mensaje por consola.
- **EnemyHasDied**: se usará para notificar de que el enemigo se ha muerto, imprimiendo un mensaje por consola.

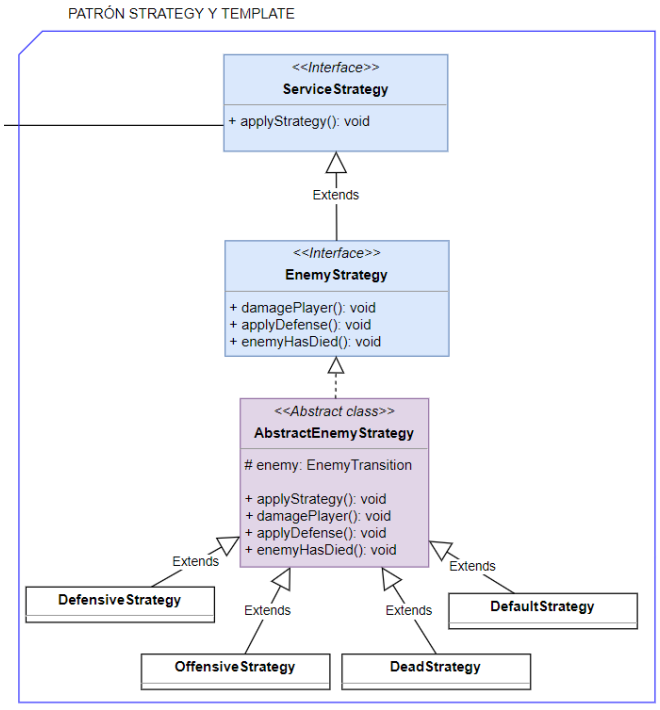
Clase abstracta AbstractEnemyStrategy: Es una clase abstracta que define el orden de la ejecución e implementa la interfaz EnemyStrategy.
Tiene un atributo EnemyTransition que nos servirá para poder decir qué enemigo (this.enemy.getName()) está atacando, defendiendo o muerto en los mensajes de las estrategias concretas.
Define los métodos damagePlayer(), applyDefense() y enemyHasDied() de la interfaz EnemyStrategy como métodos abstractos, y implementan estrategias.
Por último, implementa el método applyStrategy() de la interfaz ServiceStrategy, de modo que elige el orden en el que se van a realizar las acciones: primero defensa con applyDefense(), luego daño con damagePlayer() y por último muerte con enemyHasDied().

Clase OffensiveStrategy: Clase para la estrategia concreta agresiva. Extiende de la clase abstracta AbstractEnemyStrategy.
En este caso, al tratarse de la estrategia agresiva, en los métodos applyDefense() y enemyHasDied() no hacemos nada y en el método damagePlayer() indicamos que el enemigo está atacando al jugador.

Clase DefensiveStrategy: Clase para la estrategia concreta de defensa. Extiende de la clase abstracta **AbstractEnemyStrategy**.
En este caso, al tratarse de la estrategia defensiva, en el método applyDefense() indicamos que el enemigo se está defendiendo del ataque del jugador y en los métodos damagePlayer() y enemyHasDied() no hacemos nada.

Clase DeadStrategy: Clase para la estrategia concreta de muerte. Extiende de la clase abstracta **AbstractEnemyStrategy**.
En este caso, al tratarse de la estrategia cuando el personaje muere, en el método enemyHasDied() indicamos que el enemigo ha muerto y en los métodos damagePlayer() y applyDefense() no hacemos nada.

Clase DefaultStrategy: Clase para la estrategia concreta por defecto. Extiende de la clase abstracta **AbstractEnemyStrategy**.
En este caso, al tratarse de la estrategia por defecto, en los métodos applyDefense(), damagePlayer() y enemyHasDied() no hacemos nada.



2.2 Patrón Decorator

(1) **Componente:** es la interfaz común que define las operaciones que pueden ser realizadas por los objetos tanto decorados como no decorados. En este caso este elemento está formado por la Interfaz **CharacterAction**

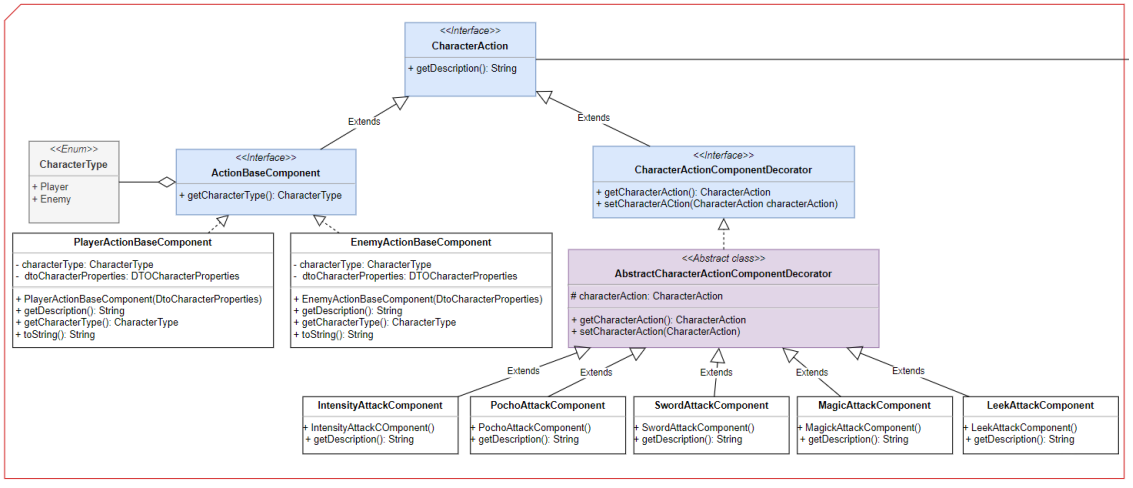
(2) **Objeto concreto:** son las clases que implementan la interfaz Componente y proporcionan la funcionalidad básica.
En este caso este elemento está formado por las clases:
- **PlayerActionBaseComponent** se utilizan para formar un personaje principal, pero solo con aquellas características necesarias para que se considere un personaje principal.
- **EnemyActionBaseComponent** se utilizan para formar un enemigo, pero solo con aquellas características necesarias para que se considere un enemigo.

Ambas clases implementan una interfaz común que contienen todos los métodos que comparten un personaje principal y un enemigo

(3) **Decorador:** es una clase abstracta que también implementa la interfaz Componente y tiene una referencia a un objeto de tipo Componente. El Decorador tiene una referencia al Componente y utiliza esa referencia para llamar al método del Componente y agregar alguna funcionalidad extra.
En este caso el decorador se llama **ActionCharacterComponentDecorator** y tiene un **componenteBase** como atributo

(4) **Decorador concreto:** son las clases que extienden de la clase Decorador y agrega funcionalidades adicionales a los objetos Componente. En este caso los decoradores concretos se pueden dividir en 2 subgrupos:
- Por un lado, se encuentran los distintos tipos de armas: **Sword**, **Magic Wand** y **Leek**, cada una de las armas tienen los métodos y atributos del decorador común del que heredan y otros nuevos únicos para cada arma.
- Por otro lado, se encuentra la clase **IntensityAttackComponent** que es el componente decorador que disminuye o aumenta la potencia de los distintos ataques.

PATÓN DECORATOR



2.3 Patrón State

Parent classes and interfaces:

La interfaz **PlayerState** define un único método, **process**, que es implementado por cada estado concreto para procesar la lógica correspondiente a ese estado.

La interfaz **PlayerStateTransition** extiende la interfaz PlayerState y define tres métodos adicionales: **actionDamagedState**, **actionParalizedState** y **actionActiveState**. Estos métodos son utilizados por el contexto (la clase Player) para realizar acciones específicas en función del estado actual del objeto.

La clase **AbstractPlayerState** es una clase abstracta que implementa la interfaz **PlayerStateTransition**, que a su vez extiende la interfaz PlayerState. Proporciona una implementación por defecto para los tres métodos definidos en PlayerStateTransition. Esta implementación simplemente muestra un mensaje de error indicando que la acción no está permitida en el estado actual. Los estados concretos pueden sobrescribir estos métodos si necesitan proporcionar una implementación diferente.

Concrete states:

La clase **ActiveState** representa un estado en el que el jugador está activo y preparado para luchar. Este estado extiende la clase **AbstractPlayerState** y proporciona su propia implementación del método **process**, que llama al método **actionActiveState**.

El constructor de ActiveState recibe una instancia del objeto Player (el contexto) y lo pasa a la clase **AbstractPlayerState** a través del constructor de la superclase. El método **process** de **ActiveState** llama al método **actionActiveState** que se define en la interfaz **PlayerStateTransition** y se proporciona una implementación concreta en **AbstractPlayerState**. Este método muestra un mensaje por consola que indica que el jugador está ahora en estado activo y preparado para luchar, utilizando el nombre del jugador proporcionado por el objeto Player pasado en el constructor.

La clase **ParalizedState** representa un estado en el que el jugador está paralizado (el jugador perdera turno). Este estado extiende la clase **AbstractPlayerState** y proporciona su propia implementación del método **process**, que llama al método **actionParalizedState**.

El constructor de **ParalizedState** recibe una instancia del objeto Player (el contexto) y lo pasa a la clase **AbstractPlayerState** a través del constructor de la superclase.

El método **process** de **ParalizedState** llama al método **actionParalizedState** que se define en la interfaz **PlayerStateTransition** y se proporciona una implementación concreta en **AbstractPlayerState**. Este método verifica si el jugador ya ha sido curado en este estado actual, en caso contrario, el jugador recupera 3 puntos de vida y se muestra un mensaje por consola que indica que el jugador está en estado de curación y su vida actual. También actualiza el booleano curado a true para evitar que el jugador recupere vida de nuevo. Si ya ha sido curado, el estado actual del jugador se actualiza a su estado activo, y se salta el estado paralizado. Esto se hace para evitar que el jugador sea inmortal.

La clase **DamagedState** representa el estado en el que el jugador ha recibido daño en su salud durante el juego. Al igual que los otros estados, implementa la interfaz **PlayerState** y extiende la clase **AbstractPlayerState**.

En este estado, se sobrescribe el método **process** y **actionDamagedState** para definir el comportamiento específico del estado. En el método **process**, se llama al método **actionDamagedState** para ejecutar las acciones correspondientes al estado.

En el método **actionDamagedState**, se imprime un mensaje indicando que el jugador ha sido dañado y su vida actual. Si la vida del jugador es menor o igual a 3 corazones y aún está vivo, se cambia el estado actual del jugador a **ParalizedState** para permitir que el jugador se cure. Si la vida del jugador es mayor que 3 corazones, el estado actual del jugador se establece en **ActiveState** para que el jugador pueda seguir jugando normalmente.

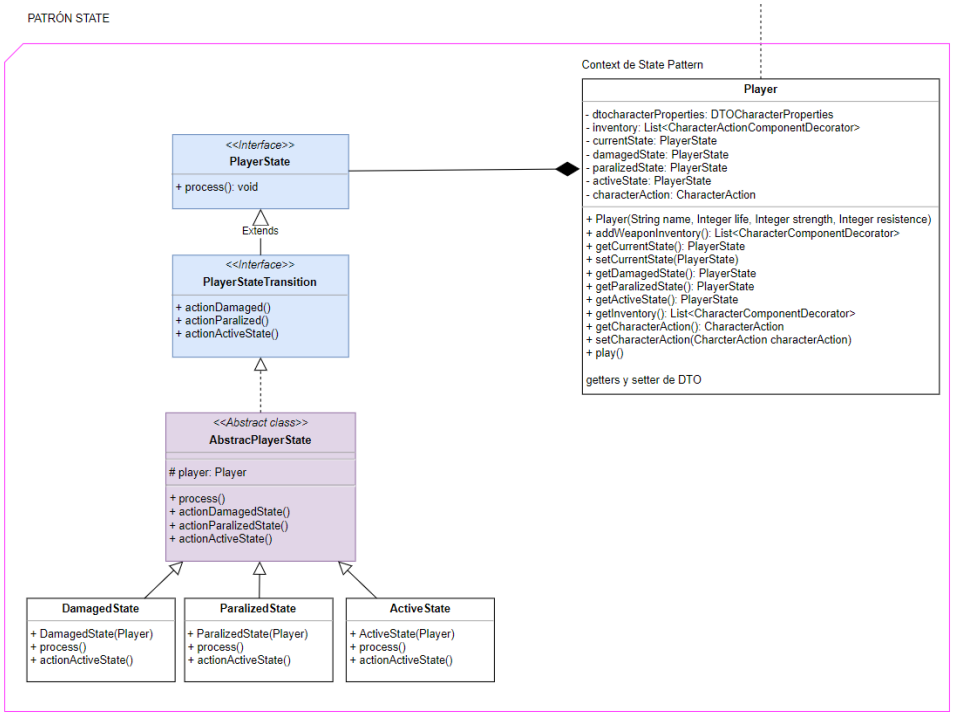
Context:
La clase Player representa un personaje en el juego y tiene diferentes estados que afectan su comportamiento según las circunstancias. Estos estados son instancias de la clase abstracta **PlayerState**, y se componen de los estados **DamagedState**, **ParalizedState** y **ActiveState**; guarda una relación de composición

dura con estas; y para su posterior modificación, se actualiza el atributo **currentState** con cada uno de los objetos de los estados de forma interna cuando lo requiera la ejecución. Además, la clase tiene un objeto DTOCharacterProperties que encapsula las propiedades básicas del personaje, como su nombre, vida, fuerza y resistencia.

La clase también tiene una lista de objetos **CharacterActionComponentDecorator** que representa las habilidades y objetos en el inventario del personaje. Esta lista se inicializa en el constructor con el método **addWeaponsInventory**, que agrega instancias de diferentes tipos de habilidades y objetos al inventario.

La clase también tiene un objeto **CharacterAction** que se trata del componente base del decorador, en el cual se va a elaborar el mensaje de ataque del jugador con el arma y el intensificador, si radica. Este objeto se inicializa en el constructor utilizando la clase **PlayerActionBaseComponent**, que implementa la interfaz **CharacterAction** y utiliza las propiedades del objeto DTOCharacterProperties para definir el comportamiento básico del personaje.

La clase Player tiene un método **play** que utiliza el estado actual del personaje para determinar su comportamiento en el juego. Los métodos getter y setter están disponibles para obtener y modificar las propiedades del personaje encapsuladas en el objeto DTOCharacterProperties.



2.4 Patrón Abstract factory

Este patrón implementa la creación de una familia de objetos (características similares). Permitiendo esto, tanto el encapsulamiento de la creación de los objetos como ocultar la lógica tras de la construcción de estos. En lugar de crear objetos directamente, se llamarán a los métodos de la fábrica para la instanciación de estos objetos.

Se basa en una jerarquía de fábricas abstractas que crean objetos relacionados. Cada **fabrica abstracta** define la interfaz para la creación de objetos y la **fábrica concreta** implementa esta interfaz.

De cara a nuestro programa, esa interfaz **AbstractFactory** que forma parte de los requisitos de patrón será **AbstractEnemyFactory**, conteniendo esta los métodos para la creación de los enemigos (createWarrior, createWarlock y createMiku)

Tendremos una **ConcreteFactory** por cada uno de los mundos con los que vamos a trabajar (**OceanEnemyFactory, ForestEnemyFactory, DesertEnemyFactory**)

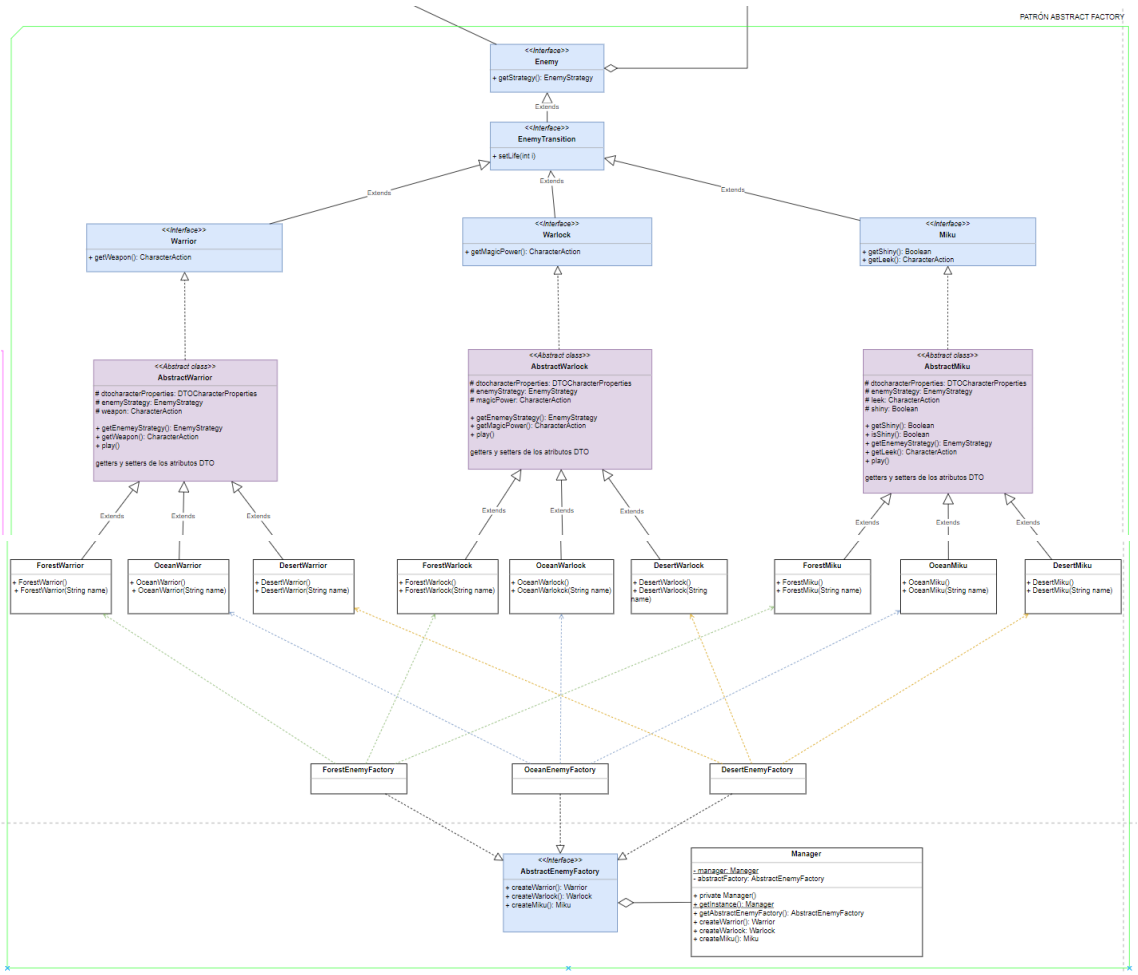
Teniendo cada uno de estos los métodos para la instanciación de los distintos enemigos, cuyas características variaran dependiendo del mundo

- **OceanEnemyFactory:** OceanWarrior, OceanWarlock, OceanMiku
- **DesertEnemyFactory:** DesertWarrior, DesertWarlock, DesertMiku
- **ForestEnemyFactory:** ForestWarrior, ForestWarlock, ForestMiju

Siendo estas clases implementaciones concretas de las clases abstractas **AbstractWarrior**, **AbstractWarlock** y **AbstractMiku** respectivamente, definiendo estas clases abstractas las características de cada enemigo
Las interfaces **Warrior**, **Warlock** y **Miku** definen el funcionamiento especifico de cada enemigo, teniendo cada una de estas un método para devolver el arma de tipo **CharacterAction**

Por último, pese a que no sea un requisito necesario por parte del patrón hemos tomado la decisión de añadir 2 clases adicionales:

- **Manager**: hemos implementado el patrón singleton en este, para tener una instancia única que se encargue de la gestión de la factoría que se usa en cada momento
- **EnemyTransition**: hemos implementado una clase intermedia entre la interfaz enemy y los distintos enemigos. Usando la técnica de **refactorización Extract Superclass,** ya que estos métodos que hemos desplazado eran comunes a todas las clases inferiores



2.5 Patrón Facade & Singleton

A través de la Clase **GameController (Singleton)**, el proyecto organiza y ejecuta la partida de una forma ordenada. Para ello, hace uso de una **Calculator (Singleton)**, que procesa todas las interacciones internas de la partida. Se utilizan clases **singleton** para evitar varias partidas simultáneas en una sola ejecución del programa.

La partida se desarrolla a través del método **game**, llamando al submétodo **startGame** de tipo bool que devolverá true cuando termine la partida con la victoria de nuestro héroe o false en el caso de su derrota, dando dicha información por pantalla.

El método **startGame** decidirá primero el orden de los combates de forma aleatoria, y llevará a nuestro protagonista al primero de los tres mundos (Forest) y comenzará el combate creando a un enemigo aleatorio usando el sub método **changeWorld**, que llama al método **play** donde tiene lugar la pelea, se asigna una **factory** para crear al enemigo a enfrentar y el tipo de dicho enemigo.

El método **play** ejecuta las rondas de combate y llama a los respectivos luchadores a realizar sus acciones hasta que solo quede uno de los dos con vida.

Esta clase además gestiona la acción del jugador gracias al método **playersTurn**, donde se lee la entrada por consola del jugador. El jugador podrá decidir entre atacar o defender. Si se defiende, aumentará su resistencia al próximo golpe. Si por el contrario ataca, se le pedirá elegir entre una selección de armas a utilizar que funcionan bajo el patrón decorator.

El usuario deberá completar los tres mundos con éxito, sobreviviendo a los combates para ganar al juego.

La clase **Calculator** realiza todas las operaciones internas del juego. Funciona también en modo **Singleton** para asegurarnos de que solo hay una única instancia por partida.

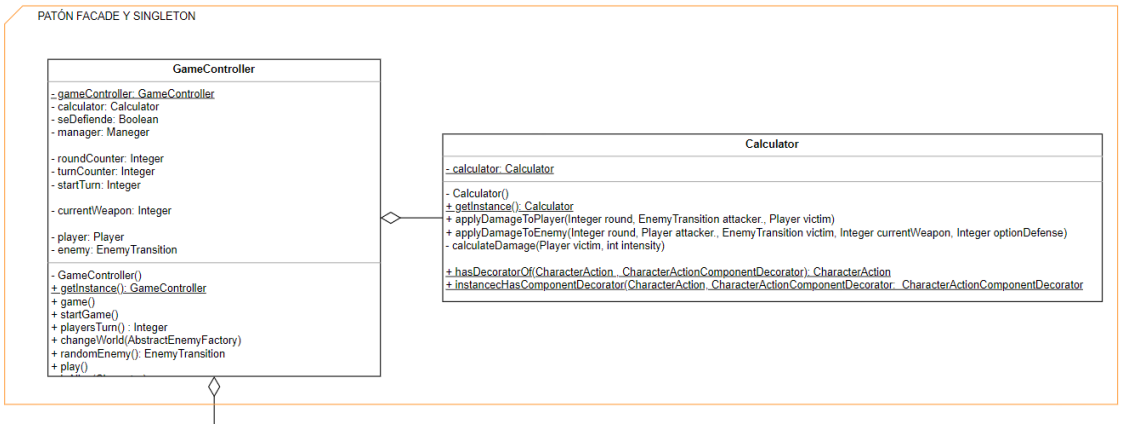
Esta clase dispone de cuatro métodos:

(1) **applyDamageToPlayer**: Se comprueba qué enemigo ha hecho el ataque y cúal ha sido la intensidad del mismo, y se realiza un daño consecuente al jugador.

(2) **applyDamageToEnemy**: Se analiza el tipo de ataque del jugador para dañar al enemigo de una forma consecuente.

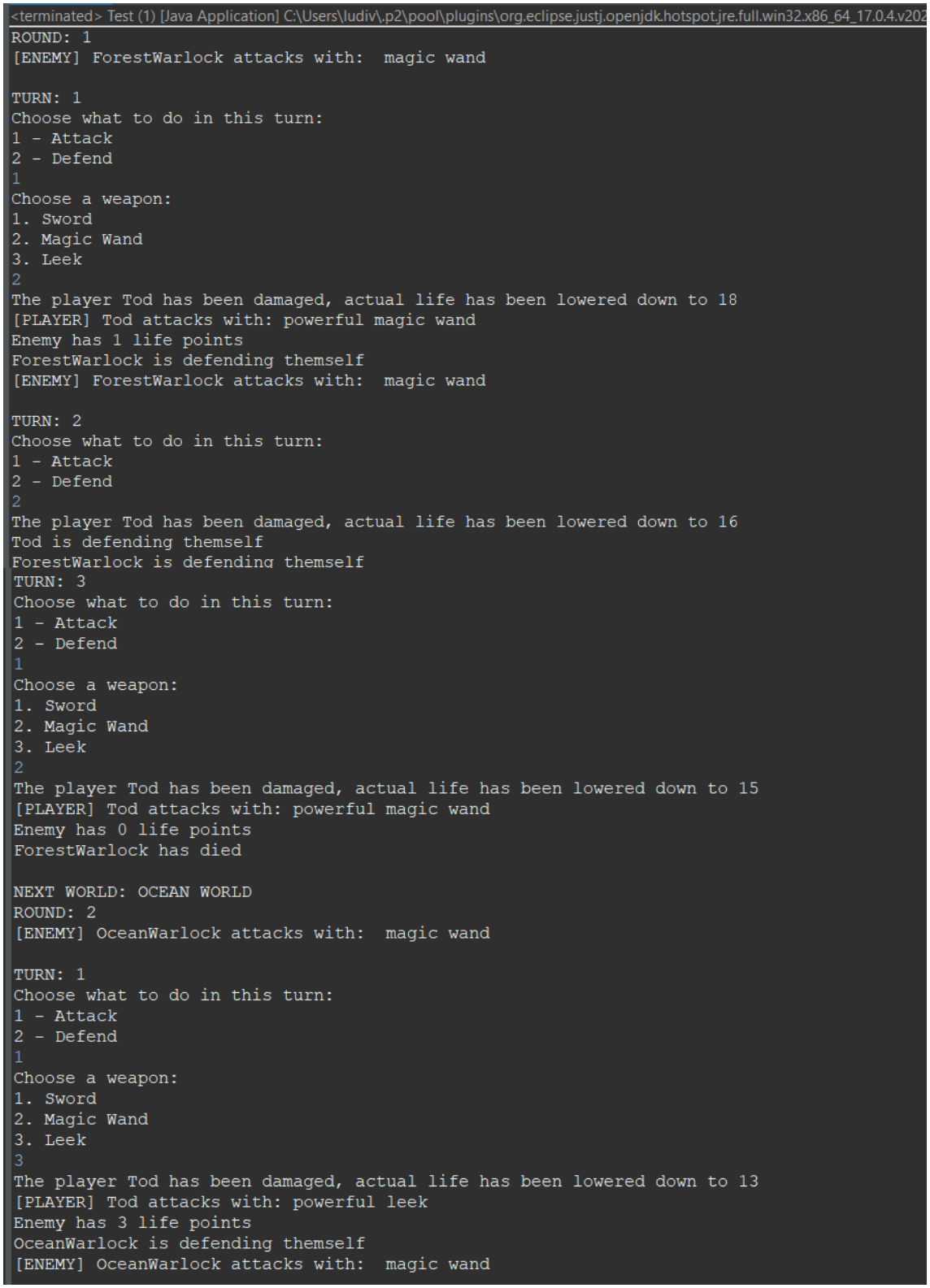
(3) **hasDecoratorOf**: A través de recursividad comprueba si hay un componente aplicado. Devuelve el objeto si está el componente buscado y devuelve Null en caso contrario

(4) **instanceHasComponentDecorator**: Utiliza el método anterior para comprobar CharacterAction



3 Simulacro ejecución programa

3.1 Derrota



```
<terminated> Test (1) [Java Application] C:\Users\ludiv\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v202
ROUND: 1
[ENEMY] ForestWarlock attacks with:  magic wand

TURN: 1
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
2
The player Tod has been damaged, actual life has been lowered down to 18
[PLAYER] Tod attacks with: powerful magic wand
Enemy has 1 life points
ForestWarlock is defending themself
[ENEMY] ForestWarlock attacks with:  magic wand

TURN: 2
Choose what to do in this turn:
1 - Attack
2 - Defend
2
The player Tod has been damaged, actual life has been lowered down to 16
Tod is defending themself
ForestWarlock is defending themself
TURN: 3
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
2
The player Tod has been damaged, actual life has been lowered down to 15
[PLAYER] Tod attacks with: powerful magic wand
Enemy has 0 life points
ForestWarlock has died

NEXT WORLD: OCEAN WORLD
ROUND: 2
[ENEMY] OceanWarlock attacks with:  magic wand

TURN: 1
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
3
The player Tod has been damaged, actual life has been lowered down to 13
[PLAYER] Tod attacks with: powerful leek
Enemy has 3 life points
OceanWarlock is defending themself
[ENEMY] OceanWarlock attacks with:  magic wand
```

```
ROUND: 3
[ENEMY] DesertWarlock attacks with: powerful magic wand

TURN: 1
Choose what to do in this turn:
1 - Attack
2 - Defend
2
The player Tod has been damaged, actual life has been lowered down to 7
Tod is defending themself
DesertWarlock is defending themself
[ENEMY] DesertWarlock attacks with: powerful magic wand

TURN: 2
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
1
The player Tod has been damaged, actual life has been lowered down to 6
[PLAYER] Tod attacks with: powerful sword
Enemy has 6 life points
DesertWarlock is defending themself

TURN: 3
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
3
The player Tod has been damaged, actual life has been lowered down to 4
[PLAYER] Tod attacks with: powerful leek
Enemy has 4 life points
DesertWarlock is defending themself
[ENEMY] DesertWarlock attacks with: powerful magic wand

TURN: 4
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
2
The player Tod has been damaged, actual life has been lowered down to 2
Player can´t attack this turn
DesertWarlock is in their default strategy
[ENEMY] DesertWarlock attacks with: powerful magic wand
```

```
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
1
The player Tod has been damaged, actual life has been lowered down to 11
[PLAYER] Tod attacks with: powerful sword
Enemy has 1 life points
OceanWarlock is defending themself
[ENEMY] OceanWarlock attacks with:  magic wand

TURN: 3
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
2
The player Tod has been damaged, actual life has been lowered down to 9
[PLAYER] Tod attacks with: powerful magic wand
Enemy has 0 life points
TURN: 4
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
2
The player Tod has been damaged, actual life has been lowered down to 2
Player can´t attack this turn
DesertWarlock is in their default strategy
[ENEMY] DesertWarlock attacks with: powerful magic wand

TURN: 5
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
1
The player Tod has been damaged, actual life has been lowered down to 0
[PLAYER] Tod attacks with: powerful sword
Enemy has 2 life points
DesertWarlock is defending themself
GAME OVER :|
```

3.2 Victoria

```
<terminated> Test (1) [Java Application] C:\Users\ludiv\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4
ROUND: 1
[ENEMY] ForestWarrior attacks with:  sword

TURN: 1
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
1
The player Tod has been damaged, actual life has been lowered down to 18
[PLAYER] Tod attacks with: powerful sword
Enemy has 0 life points
ForestWarrior has died

NEXT WORLD: OCEAN WORLD
ROUND: 2
[ENEMY] OceanWarlock attacks with:  magic wand

TURN: 1
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
1
The player Tod has been damaged, actual life has been lowered down to 16
[PLAYER] Tod attacks with: powerful sword
Enemy has 3 life points
OceanWarlock is defending themself
[ENEMY] OceanWarlock attacks with:  magic wand

TURN: 2
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
1
The player Tod has been damaged, actual life has been lowered down to 14
[PLAYER] Tod attacks with: powerful sword
Enemy has 1 life points
OceanWarlock is defending themself
[ENEMY] OceanWarlock attacks with:  magic wand

TURN: 3
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
1
```

```
[PLAYER] Tod attacks with: powerful sword
Enemy has 0 life points
OceanWarlock has died

NEXT WORLD: DESERT WORLD
ROUND: 3

TURN: 1
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
1
The player Tod has been damaged, actual life has been lowered down to 10
[PLAYER] Tod attacks with: powerful sword
Enemy has 6 life points
DesertWarlock is defending themself
[ENEMY] DesertWarlock attacks with: powerful magic wand

TURN: 2
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
1
The player Tod has been damaged, actual life has been lowered down to 8
[PLAYER] Tod attacks with: powerful sword
Enemy has 4 life points
DesertWarlock is defending themself

TURN: 3
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
1
The player Tod has been damaged, actual life has been lowered down to 6
[PLAYER] Tod attacks with: powerful sword
Enemy has 2 life points
DesertWarlock is defending themself
[ENEMY] DesertWarlock attacks with: powerful magic wand

TURN: 4
Choose what to do in this turn:
1 - Attack
2 - Defend
1
Choose a weapon:
1. Sword
2. Magic Wand
3. Leek
1
The player Tod has been damaged, actual life has been lowered down to 4
[PLAYER] Tod attacks with: powerful sword
Enemy has 0 life points
DesertWarlock has died
YAY! YOU WON :D
```

4 Diagrama UML

Debido a la longitud del esquema este será adjuntado junto con este archivo en la entrega para poder verlo con la mayor calidad posible