Diseño de software: Práctica final

ALUMNOS: Daniel Peralta Llamas, Juan Rodriguez Córdoba, Rafael Ceres Martínez

GRADO: MAIS 2ºA

Índice

1. Descripción general del juego

Para el proyecto final de esta asignatura hemos hecho un juego por turnos. Creas un personaje con unas estadísticas base y te vas haciendo más fuerte mientras derrotas cuatro tipos de enemigos de cuatro mundos distintos. Los patrones usados han sido :

-Facade : Para manejar el transcurso de la partida y facilitar la experiencia del usuario

-Abstract factory method: Para la creación de enemigos

-State : Para poder recibir e infligir estados alterados

-Singleton : Para la clase que calcula el daño y para el abstract factory method

-Strategy : Para definir distintas estrategias que podrán seguir los enemigos

-Template : Para manejar el comportamiento de los enemigos

-Decorator : Para añadir mejoras a las acciones tanto de personajes como las de los enemigos

2. Manual de uso

Para jugar al juego, hay que ejecutar el main de la clase *PlayGame*, que se encuentra en la carpeta del patrón Facade.

Cuando ejecutas el programa te pedirá el nombre del jugador. Se te informará de tus estadísticas y empezará el bucle en el que te pedirá lo que quieres hacer, luego el enemigo (puede variar el orden en función de la velocidad del enemigo) , se pasará de turno y así hasta que mueras o quieras salir. Las opciones que el jugador tiene en cada turno son atacar, física o mágicamente. Defenderse , curarse o usar un item. El enemigo es controlado por la máquina y una vez derrotado se pasa a una pantalla de selección en la que se podrá elegir entre dos opciones, un item o una mejora para algún aspecto del personaje. Cada cuatro enemigos aparecerá un jefe y si lo derrotas pasarás de mundo. Una vez que derrotas al jefe del mundo final se te darán dos opciones, salir de la partida o seguir con la que tienes. Al elegir continuar mantendrás tus objetos y atributos, pero volverás a empezar desde el primer mundo con enemigos cada vez más poderosos.

3. Strategy

Hemos usado el patrón Strategy para controlar el comportamiento del enemigo. Este puede realizar cuatro acciones en su turno: Ataque físico, ataque mágico, curación y protección. Dependiendo de la estrategia que tenga, tendrá más probabilidad de elegir una u otra.

Estos son los componentes principales del diseño:

-Contexto: La clase ***Enemy*** tiene una relación de agregación con *EnemyBehaviorStrategy*. La estrategia es intercambiable y se proporciona en el constructor (y también hay una por defecto).

-Superclase común: La interfaz ***EnemyBehaviorStrategy*** define el método *PerformAction*, que será implementado de forma distinta por cada una de las estrategias concretas. Antes de pasar a las estrategias concretas, hemos puesto por debajo una clase abstracta llamada ***AbstractEnemyBehaviorStrategy***, que introduce un nuevo método llamado *chooseAction*. Este método recibe las probabilidades de ejecutar cada acción, coge un número aleatorio y en función de este elige la acción, teniendo en cuenta las probabilidades. Hemos implementado este método con el fin de no repetir este proceso en cada una de las estrategias concretas.

-Estrategias concretas: Hay un total de cuatro estrategias concretas, cada una centrada principalmente en una o dos acciones, con probabilidades distintas de ejecutarlas:

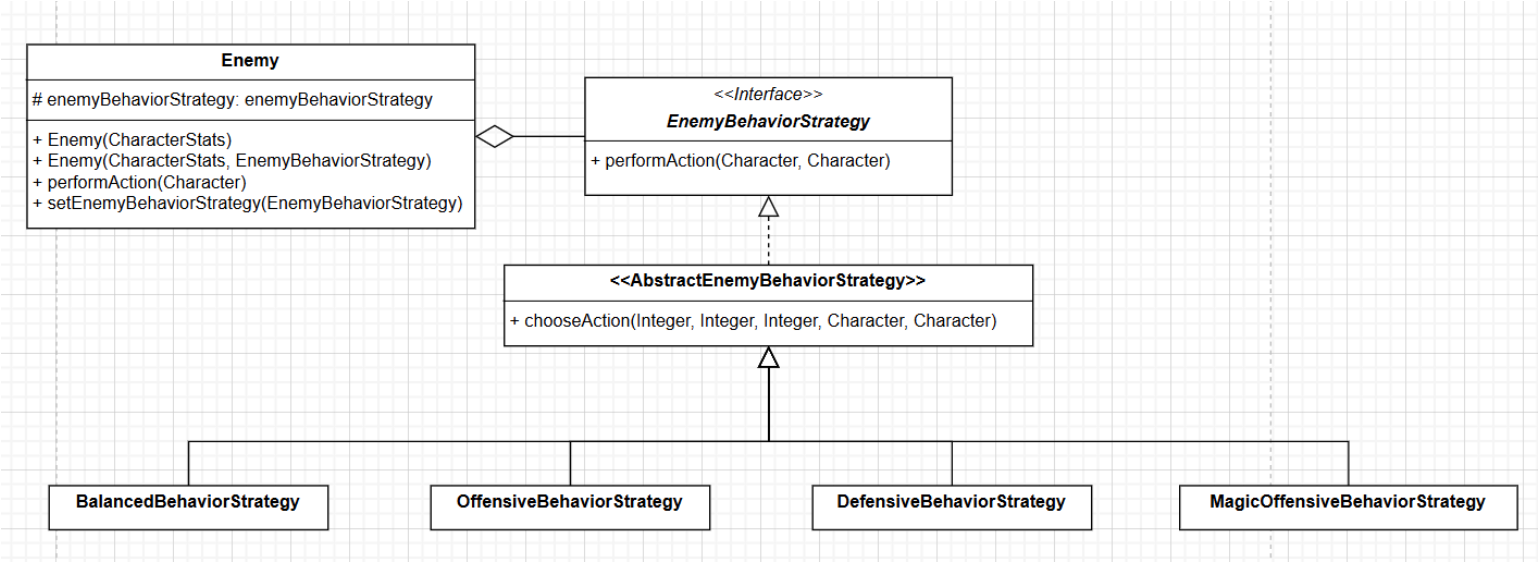
- **-*OffensiveBehaviorStrategy*:** Tiene un 60% de probabilidades de realizar un ataque físico, 20% de uno mágico, 10% de curarse y 10% de protegerse.
- **-Magic*OffensiveBehaviorStrategy*:** Tiene un 20% de probabilidades de realizar un ataque físico, 60% de uno mágico, 10% de curarse y 10% de protegerse.
- **-*DefensiveBehaviorStrategy*:** Tiene un 20% de probabilidades de realizar un ataque físico, 10% de uno mágico, 20% de curarse y 50% de protegerse.

***-BalancedBehaviorStrategy*:** Tiene un 25% de probabilidad de hacer cada acción.

Cada una de estas estrategias llama al método *chooseAction* de su padre con distintos valores para las probabilidades.

La estrategia que adopta cada enemigo depende del tipo de enemigo que sea y del mundo en el que esté, lo definimos en la implementación del patrón Abstract Factory.

A continuación está el diagrama UML con todas las clases e interfaces:



4. Decorator

El propósito de la implementación del patrón Decorator es añadir mejoras a las acciones, tanto del personaje como de los enemigos.

Estos son los componentes principales:

-Superclase común: Tanto los componentes base como los decoradores implementan la interfaz ***ActionComponent***, que tiene métodos para describir la acción, realizarla, buscar un decorador concreto (con el fin de mejorar decoradores ya existentes) y buscar el componente base (para ver de qué tipo es la acción).

-Componente base: La interfaz ***BaseActionComponent*** tiene un método que devuelve el tipo de la acción en forma de enumerado *ActionType*. Este enumerado tiene un atributo String que indica de forma más legible el tipo de acción.
Hay cuatro componentes base que implementan esta interfaz, uno para cada tipo de acción: *BaseAttackAction*, *BaseMagicAction*, *BaseHealingAction* y *BaseGuardAction*. Cada uno de estos implementa el método *PerformAction* de forma distinta, haciendo uso de la clase *BattleCalculator* que explicamos en el apartado del patrón Singleton.

-Decoradores: Todos los decoradores extienden de la clase *AbstractActionComponentDecorator*, que a su vez implementa la interfaz *ActionComponentDecorator*. Los decoradores tienen como atributo el componente al que decoran y el nivel de la mejora, ya que serán más efectivas cuanto mayor sea su nivel. Una acción no puede ser decorada con un decorador de la misma clase dos veces, en todo caso se sube el nivel de la mejora correspondiente buscando el decorador. Los decoradores de las acciones son variados:

> **-*SkillBoostComponentDecorator*:** Aplica una mejora general a la acción. Si es ataque, hace más daño; si es protección, absorbe más daño; etc.
>
> **-*AbsorbDamageComponentDecorator*:** Para las acciones de hacer daño, el personaje que las hace también se cura.

***-InflictSlowdownComponentDecorator* e *InflictParalysisComponentDecorator*:** Para las acciones de hacer daño, además les da probabilidad de infligir un efecto alterado. Esta probabilidad es mayor cuanto mayor sea el nivel de la mejora.

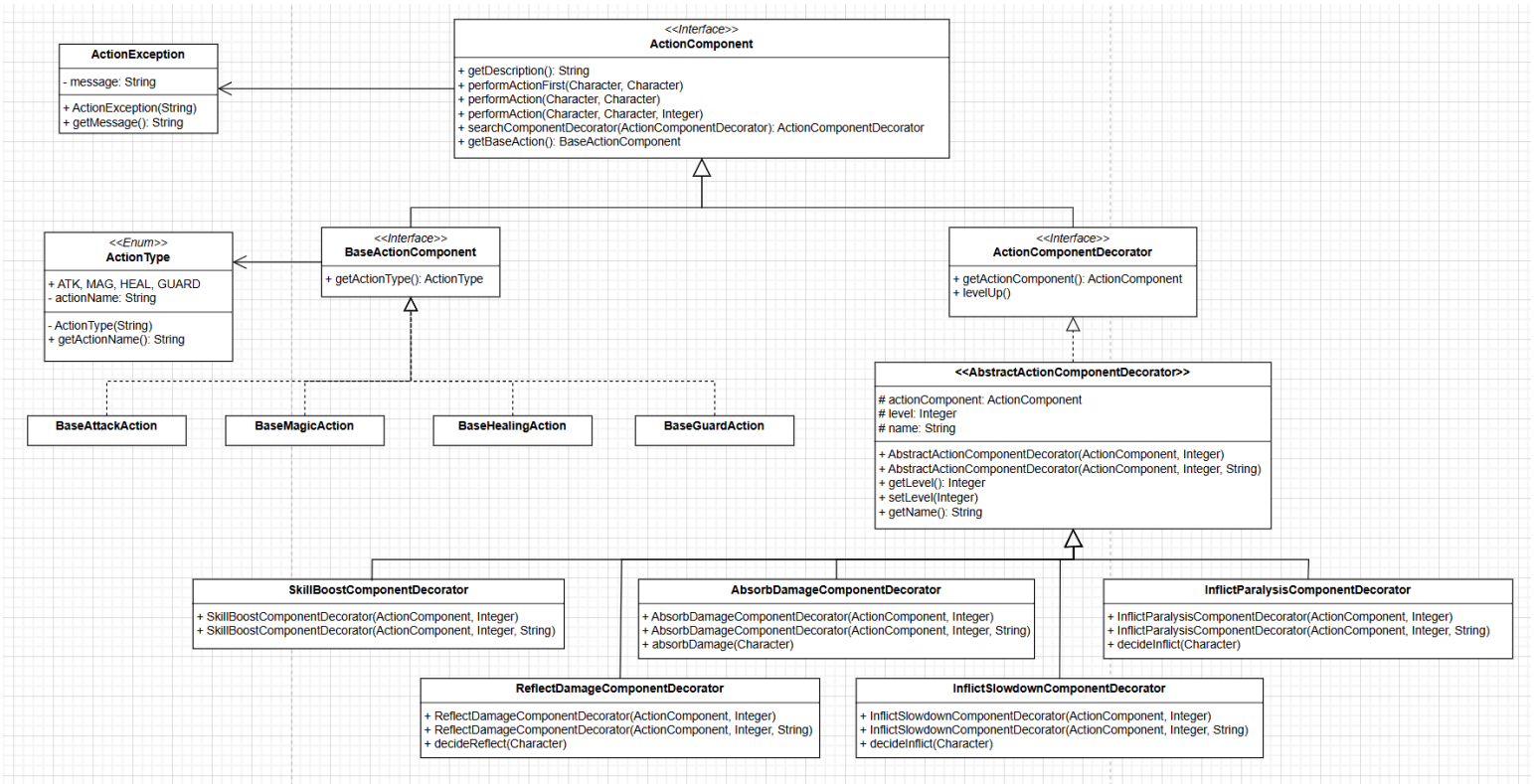
-ReflectDamageComponentDecorator: Para la acción de protegerse, da además la probabilidad de reflejar el ataque al personaje que ataca.

Cuando se llama al método *performAction* de un decorador, este hace el efecto que tenga que hacer y llama al *performAction* de su componente, hasta llegar a la base. El método *performAction* está sobrecargado, ya que puede tener o no mejora de la habilidad (la otorgada por *SkillBoostComponentDecorator*). Si al decorador que aplica esta mejora le llega la versión ya mejorada, lanzará una excepción de tipo *ActionException*, ya que estaríamos en el caso de que un componente ha sido decorado dos veces con el mismo decorador. Este caso no se va a dar en el juego, ya que cuando el jugador elige una mejora y ya la tiene, lo que hacemos es subirla de nivel.

En el juego, el jugador tiene como atributo una acción de cada tipo, que será la que realice cuando decida. Estas acciones empiezan como base y se pueden mejorar al terminar cada combate, ya que al jugador se le ofrece aleatoriamente una de las mejoras antes mencionadas para una acción también aleatoria.

En cuanto a los enemigos, las mejoras que tienen sus acciones dependen de su tipo y del mundo en el que estén, por lo que vienen dadas por el Abstract Factory. Los enemigos también tienen como atributo una acción de cada tipo, y la acción que realizan en su turno se decide según su estrategia, como hemos explicado en el apartado del patrón Strategy.

A continuación está el diagrama UML correspondiente a la implementación del patrón Decorator:



5. State

Hemos implementado el patrón state tanto para los estados del jugador como los del enemigo.

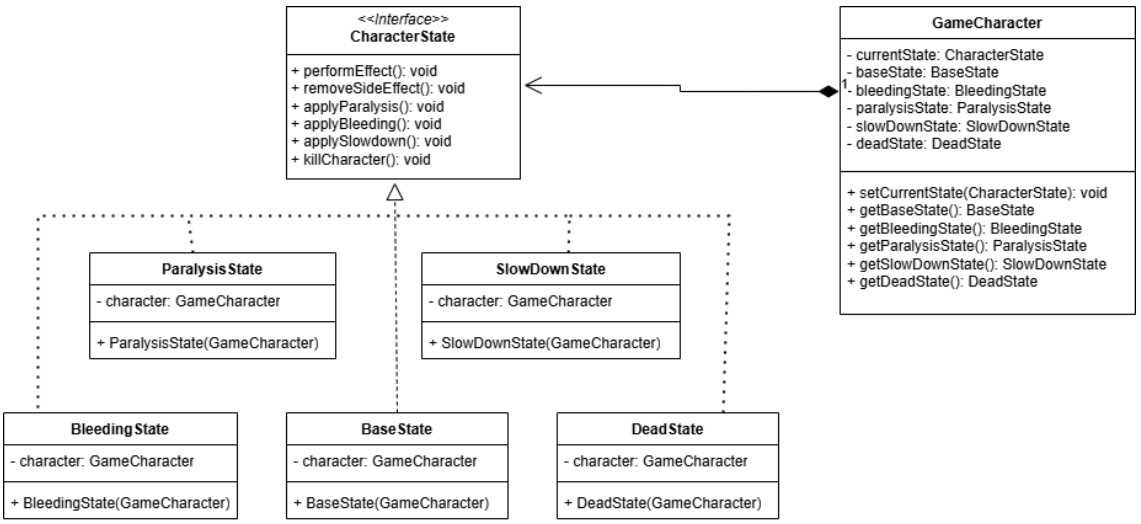
Sus componentes principales son:

-Superclase común: Interfaz ***CharacterState***. Contiene el método principal de cada estado (*performEffect()*) y cinco métodos de transiciones entre estos. (Quitar efecto vuelve al estado Base, aplicar parálisis pasa a Paralizado, aplicar ralentización pasa a Ralentizado, aplicar sangrado pasa a ConSangrado, matar jugador pasa a Muerto)

-Estados Concretos: Clases ***BaseState, BleedingState, ParalysisState, SlowDownState, DeadState.*** Estados concretos que implementan la superclase común ***CharacterState***. Cada estado realiza un efecto distinto mediante el método *performEffect()*.

-Contexto: Clase ***GameCharacter***. Actúa de contexto teniendo una instancia de cada estado más un estado actual al que delega mediante composición las acciones correspondientes

A continuación está el diagrama UML correspondiente a la implementación del patrón State:



6. Abstract Factory

Para la instanciación de enemigos hemos usado el abstract Factory Method. Tenemos una clase Abstracta, “Enemy” que definirá el comportamiento de los enemigos haciendo uso del template method explicado más adelante.

Como cliente tenemos la clase “Enemy Factory Context” que será la encargada de delegar a la fábrica correspondiente la creación de los enemigos, y de incrementar las estadísticas para las partidas que se quieran continuar después del jefe final.

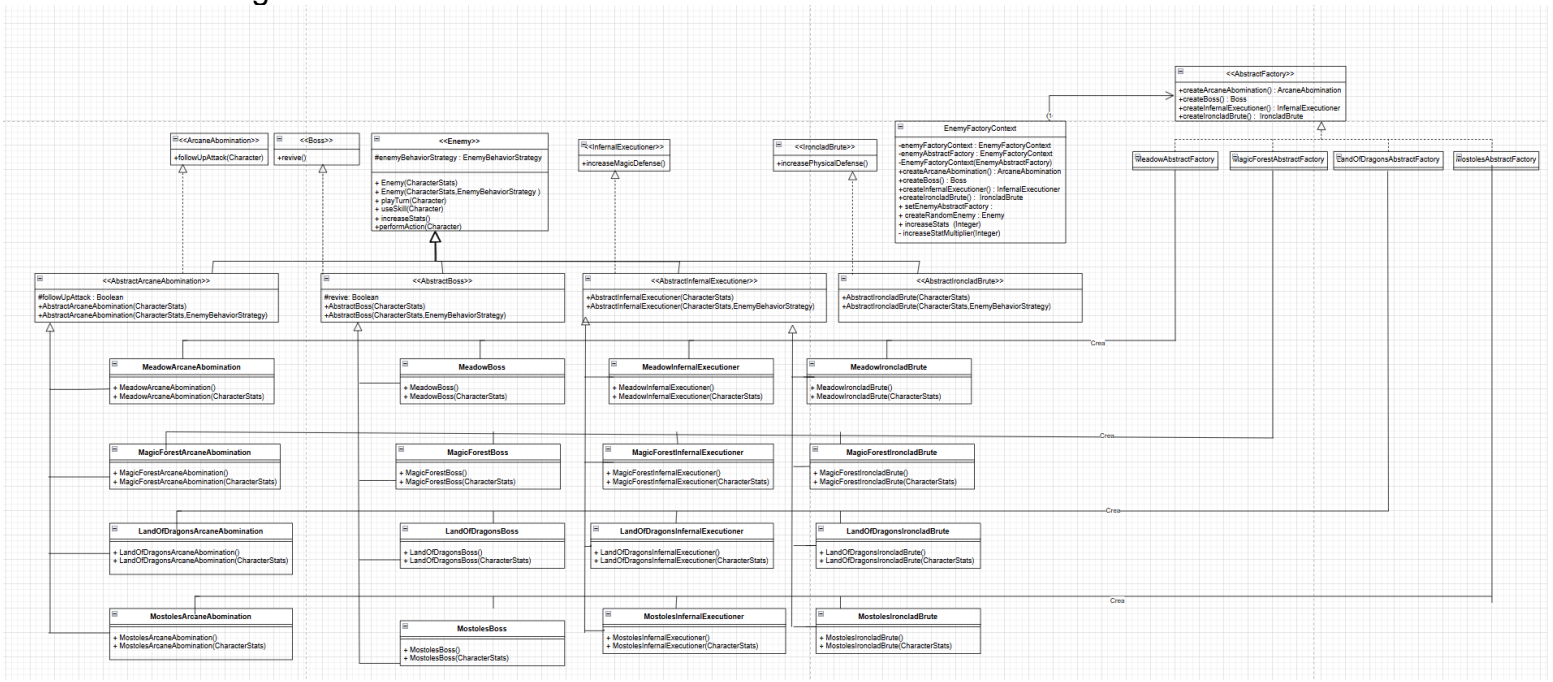
Como factoría abstracta tenemos la clase “Enemy Abstract Factory” que tiene todos los métodos de creación de enemigos. Tenemos 4 factorías concretas , una por cada mundo (Meadow Abstract Factory,Magic Forest Abstract Factory, Land Of Dragon Abstract Factory,Móstoles Abstract Factory), con los métodos de creación de cada enemigo.

Para que el código sea más corto, simple y legible hemos añadido una clase abstracta para cada tipo de enemigo en la que se encuentran todos los atributos y métodos que son comunes a todos los enemigos de un tipo.

Los productos abstractos serían las interfaces de cada tipo de enemigo en la que está el método de su habilidad característica (Ironclad Brute se sube la defensa

física,Arcane Abomination hace un ataque adicional,Boss puede revivir una vez y el Infernal Executioner se sube la defensa mágica).

Como productos concretos tenemos las 16 clases que definen a cada tipo de enemigo de cada mundo.

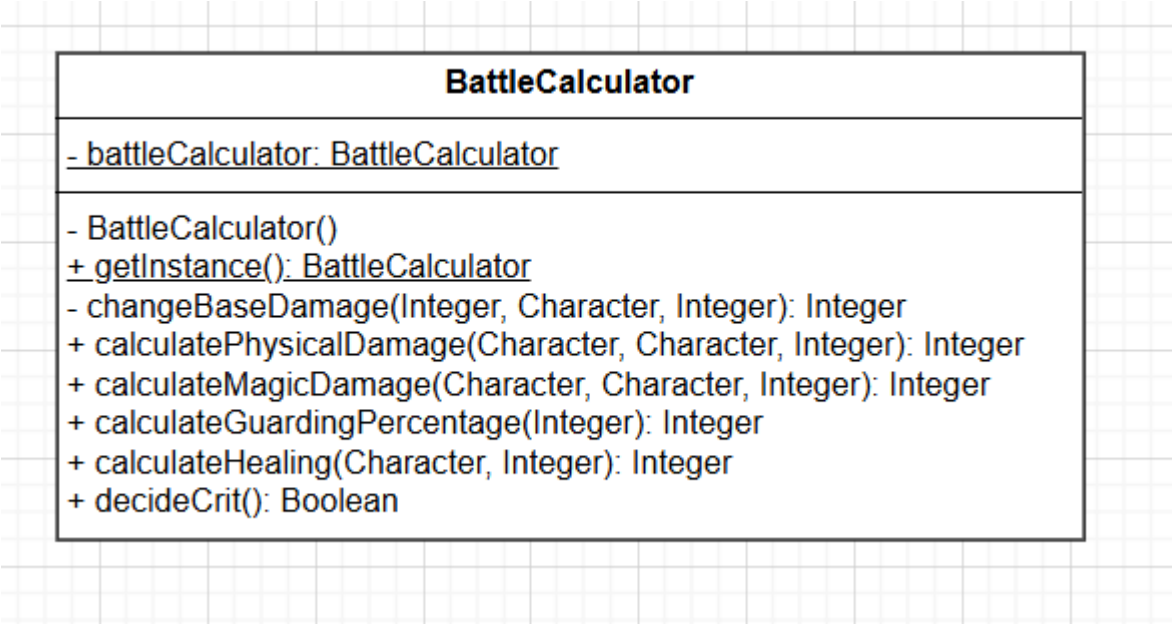


7. Singleton

Hemos aplicado el patrón Singleton para la clase ***BattleCalculator***, que es la que calcula el daño hecho a los enemigos, si un golpe es crítico, el daño absorbido al protegerse, etc.

Hemos usado instanciación temprana o *static*: La calculadora tiene un atributo estático de su tipo que creamos al principio, y un getter estático que lo devuelve.

El diagrama UML es muy sencillo, ya que solo involucra una clase:



8. Template Method

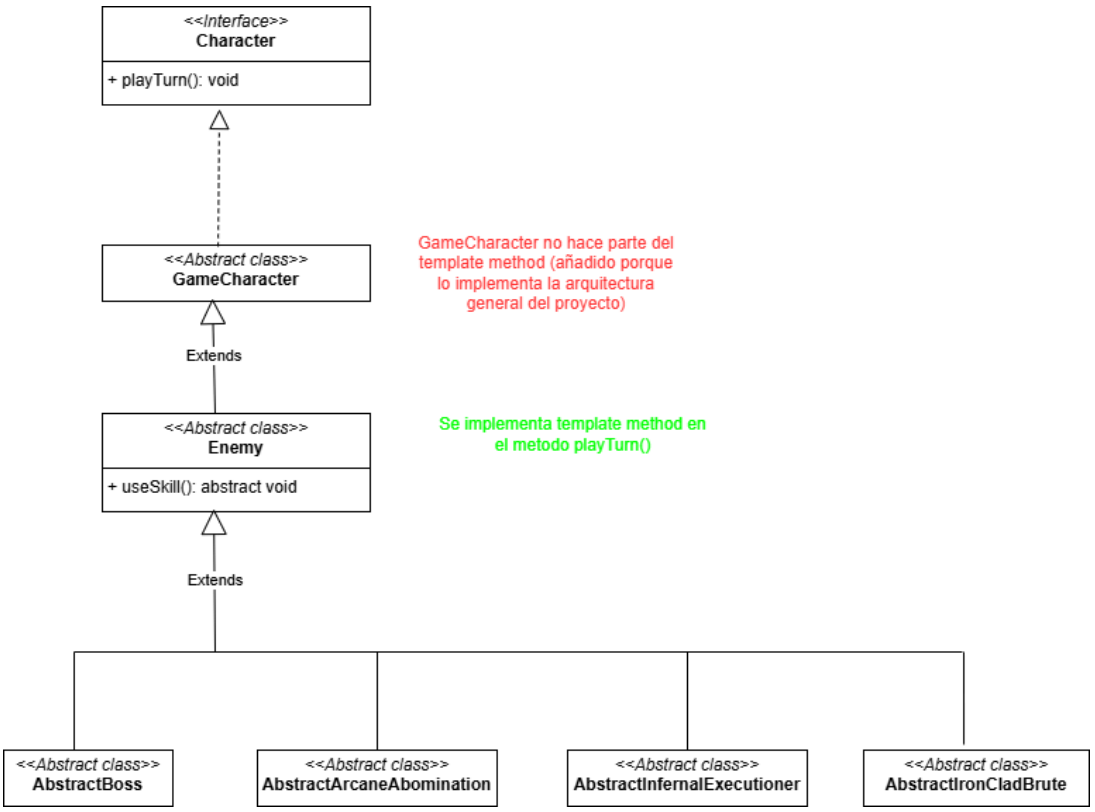
Para los enemigos hemos implementado un template method ya que todos estos siguen el mismo patrón:

Sus componentes son los siguientes:

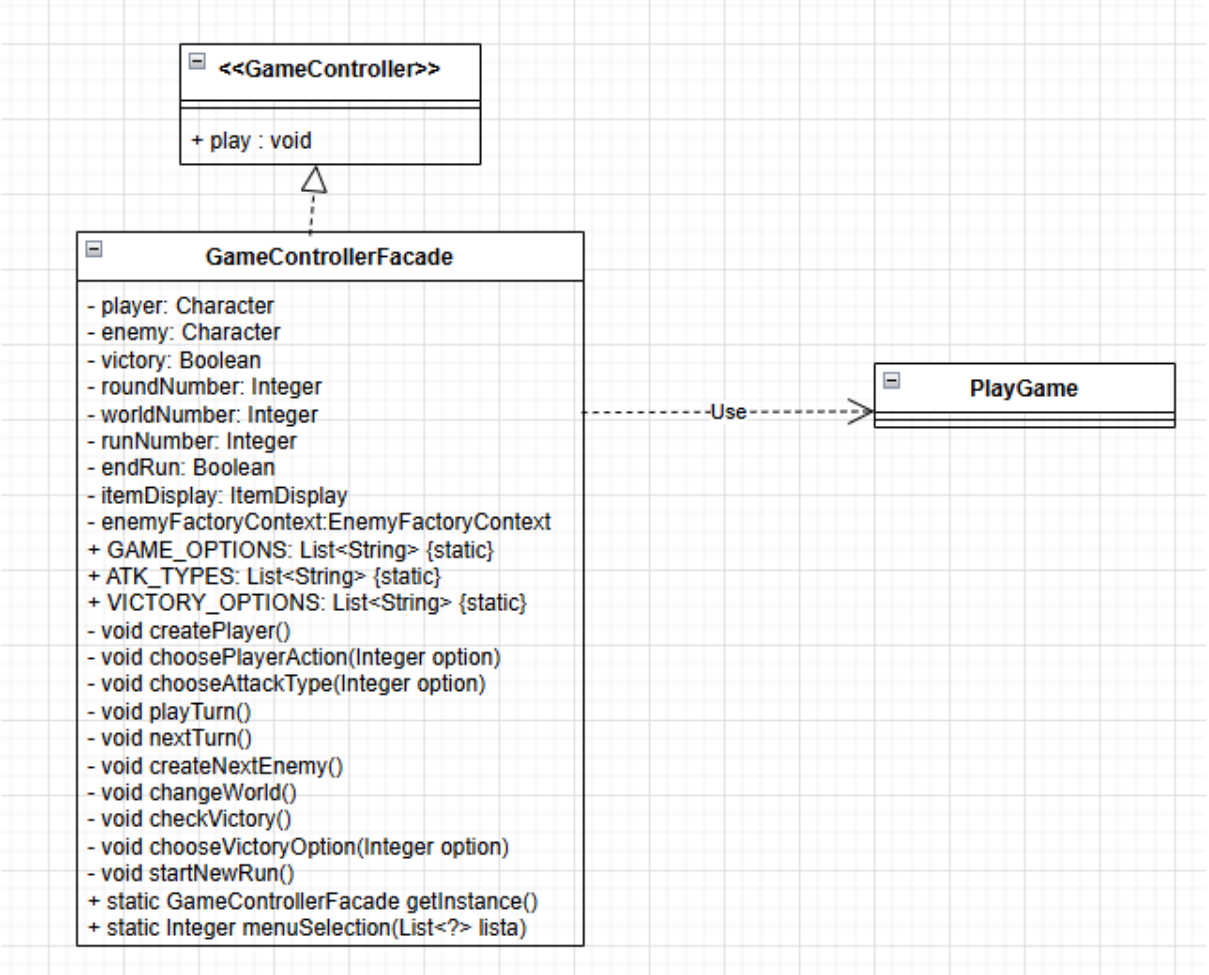
-Clase abstracta con template method: Clase abstracta ***Enemy.*** Desarrolla el método *playTurn()* donde hace llamadas al método abstracto *useSkill()* (Cada enemigo implementa una habilidad diferente pero todos tienen el mismo patrón: Primero realiza acción y luego usa habilidad)

-Clases Concretas: (En este caso abstractas ya que de estas salen sus concretas por cada mundo) Clases abstractas ***AbstractBoss, AbstractInfernalExecutioner, AbstractArcaneAbomination, AbstractIronCladBrute***. Cada una de ellas desarrolla el método *useSkill()*

A continuación está el diagrama UML correspondiente a la implementación del patrón Template method:


<<Interface>>
Character
+ playTurn(): void
<<Abstract class>>
GameCharacter
GameCharacter no hace parte del template method (añadido porque lo implementa la arquitectura general del proyecto)
Extends
<<Abstract class>>
Enemy
+ useSkill(): abstract void
Se implementa template method en el metodo playTurn()
Extends
<<Abstract class>>
AbstractBoss
<<Abstract class>>
AbstractArcaneAbomination
<<Abstract class>>
AbstractInfernalExecutioner
<<Abstract class>>
AbstractIronCladBrute

9. Facade



Para simplificar la experiencia de usuario y que simplemente tenga que llamar al método “play” hemos usado el patrón de fachada.

Aunque no sea necesario para el patrón hemos optado por añadir una interfaz ***GameController*** que tiene el método play.

La clase GameControllerFacade implementa la interfaz mencionada, es la encargada de crear la partida y confirmar que todo vaya bien. Para instanciarlo hemos usado el patrón singleton con una implementación static.

Tiene varias listas y un constructor estático para inicializarlas. Dichas listas serán las que muestran información al usuario sobre qué opciones tiene para jugar, atacar o una vez que ha ganado.

Para mostrar estas listas tenemos el método “menuSelection” que recibe una lista genérica, la muestra y maneja las elecciones del usuario. Tiene atributos para

gestionar la información del jugador, el enemigo y distintos parámetros de la partida como el número de ronda,de mundo…

El método “play” es un bucle del que no se sale hasta que quiera salir o el jugador muera. Este bucle repite las acciones de mostrar tanto al jugador como al enemigo, elegir la acción del jugador, jugar turno, pasar de turno y comprobar si ha ganado.

Una vez que has ganado se te da la opción de seguir jugando pero aumentando la dificultad. En playGame simplemente se instancia un gameController y se llama al método play.