PROYECTO FINAL 2025

JUAN BATTAGLIO QUINTANA, JUAN GUASP TIMONER

ASIGNATURA: DISEÑO DE SOFTWARE

GRUPO: B

GRADO: 2

CURSO ACADÉMICO: 2024-25

PROYECTO FINAL 2025

Índice

PROYECTO FINAL 2025

Descripción del juego

Este proyecto final de asignatura se llama DISOBR[1]. Consiste en un juego de tipo RPG y con modalidad Battle Royale, fuertemente inspirado en el proyecto final de la asignatura de Programación Orientada a Objetos del mismo curso. Un videojuego RPG consiste en que tú asumes el rol de un personaje de un tipo a elegir. Cada tipo tiene sus propias características, formas de atacar y mecánicas distintas, aun compartiendo aspectos básicos de los personajes de un videojuego como la vida, la fuerza y el nombre que el usuario puede elegir.

En nuestro caso, tenemos al guerrero, al mago y al arquero. El guerrero tiene una única forma de atacar con su espada, pero tiene una vida mayor que los demás al poseer armadura que le otorga protección. El mago realiza ataques con maná (puntos de magia) y cuando se agotan no puede atacar. Aun así, tiene una cierta cantidad de pociones que le restauran sus puntos de magia y que, dependiendo de la dificultad que escoja el jugador, tiene más o menos pociones. Y el arquero puede atacar tanto con un arco como con una daga. El arco realiza más daño, pero tiene un número limitado de flechas. Como el mago, el arquero también puede curarse con vendas que varían según el nivel de dificultad.

Siguiendo la modalidad Battle Royale, te permite encontrarte con ciertos jugadores y/o enemigos y poder interactuar con ellos mediante ataques. Al existir la probabilidad de que en un mapa te encuentres con ciertos enemigos y con otros no. La probabilidad de que un personaje juegue en ese turno es al azar, con lo cual, existe la posibilidad de estar varios turnos sin atacar.

Mediante ataques de jugadores tanto creados por el usuario como creados por bots, en el momento que solo quede uno con vida, se proclamará ganador y se podrá guardar en un archivo '.txt' la fecha en la que ganó, su nombre y sus estadísticas.

Es un juego que se juega en la consola, en el cual se han aplicado gran parte de los patrones de diseño estudiados en la asignatura, que nos han permitido flexibilizar muy positivamente la creación de personajes, la alterabilidad del código para poder hacer modificaciones y la legibilidad de los programas.

[1] Diseño de Sofware Battle Royale.

PROYECTO FINAL 2025

Manual de instrucciones

Para jugar al juego, se necesita solamente un ordenador portátil, tener los programas y un *Java Runtime Environment* instalados y acceder al archivo Main.java. Ahí, gracias al patrón Facade, solamente con ejecutar el programa se instanciará un objeto de la clase *GameController* y se llamará a su método inicial. Al llamarse al método, deberá empezar a prestar atención a la salida del terminal, que le guiará a lo largo de la partida y las múltiples opciones. Estas opciones se marcarán con números. Cuando le toque escoger una opción, pulse el número asociado a la opción y pulse Enter, y se realizarán las funciones correspondientes a la opción que ha escogido. Las primeras opciones que escogerá serán las de inicar una partida nueva, guardarla en caso de que haya jugado una y salir del programa.

Cuando se inicia una partida, se le realizan las configuraciones iniciales del juego, como el número de jugadores o bots que quiere utilizar, la dificultad del juego (normal o difícil). La dificultad normal establece un juego base. La dificultad difícil establece un mayor nivel de vida para los enemigos, además de que los jugadores usuarios tendrán menos opciones de curación o restauración de puntos (en caso de ser un arquero o mago). Después de establecer los personajes, comienza el juego.

Se tiene que saber también que los jugadores tienen que ser como mínimo dos, un jugador y un bot, o dos jugadores usuarios. Es una de las condiciones esenciales que se gestionan en el método inicial del juego.[2] Los personajes creados por los usuarios se llamarán J0, J1, J2, etc. De la misma manera, los personajes creados por la máquina se llamarán M0, M1, M2, etc.[3]

Comenzando así con el juego, se apreciarán las múltiples mecánicas en función del tipo de personaje que se tenga. Cuando toque realizar una acción, se podrá elegir una opción con el teclado (1 o 1 y 2) en función del personaje y las funcionalidades que tiene. Mediante ataques, curaciones y el desarrollo de la partida, los jugadores se van quedando con más o menos vida.

Cuando solamente quede un jugador con vida, se le declarará ganador y la partida habrá acabado, volviendo a la primera opción del principio, permitiéndote así guardar el ganador de la partida en un fichero o jugar la siguiente. También se puede salir del juego en el preferido caso.[4]

[2] Imagen de comienzo de partida en el anexo.
[3] Imagen de continuación de partida en el anexo
[4] Imagen de final de partida adjuntada en el anexo

PROYECTO FINAL 2025

Memoria

El proyecto comenzó con las decisiones de los patrones de aplicar y la decisión de inspirarnos en nuestro proyecto de la asignatura de POO, dándonos la base del concepto y el código de la clase *GameController* pero tocó estructurar los cambios con la implementación de los patrones de diseño, cuya principal consecuencia fue la de dividir en muchas más clases y patrones el proyecto, para asegurarnos poder tener flexibilidad en el código y organización, cambiando muchos aspectos previos como se tenían en la creación de personajes y sus instanciaciones en forma de objetos.

Así, para ponernos manos a la obra, establecimos los patrones que íbamos a usar en el programa y el repositorio de GitHub donde guardamos el código que fuimos desarrollando. Así, decidimos ceñirnos en gran parte al enunciado e implementar todos los patrones menos el Template, que debido a nuestra lógica de azar y el hecho de que decidimos utilizar la lógica del azar, la misma que en el anterior proyecto, prescindimos de usar ese patrón para implementar el patrón **Factory** para poder generar los enemigos en función de su dificultad, si normal o difícil.

La primera fase del desarrollo consistió en constituir los personajes y realizar la base del juego. Para los personajes utilizamos el patrón **Abstract Factory** para el tipo de personajes de los enemigos, además de un patrón **Factory** en función de la dificultad de los jugadores creados por la máquina[5]. Así, tanto para jugadores amigos y enemigos, se implementó el patrón **Decorator** para crear los personajes base y el tipo de personajes que actuaran como componente decorador, permitiendo añadir más tipos de personajes con distintas funcionalidades. En la base del juego, implementamos el patrón **Facade**, para dejar una clase Main elegante que solo con una llamada a un método ya se pueda gestionar una partida por completo, otorgando comodidad y optimización al jugador o al desarrollador que leyere el código.

Empezando con el desarrollo de la clase *GameController*, gran parte del código estaba influenciado y es muy similar al del anterior proyecto, a excepción del método que regula las acciones y ataques de los personajes. En ese método, realizamos una serie de sentencias if que regulaban las distintas estrategias que realizaban los distintos personajes en función del tipo de personaje (arquero/guerrero/mago) o (bot/usuario). Así, tocó planear e implementar el patrón **Strategy** para cada clase de personajes. Se pudo implementar este patrón correctamente, y la clase *GameController* se gestionó de manera que no hubiera errores de gestión de

[5] Teniendo en cuenta la opción de poder tomar a jugadores enemigos como jugadores creados por el usuario ya que se puede jugar entre dos o cinco personajes creados por distintos jugadores usuarios.

personajes para no llegar a excepciones, bucles infinitos o parones en el juego, aun quedando revisar un par de cuestiones como el guardado de los datos del ganador de una partida.

Por último, quisimos desarrollar un modelo que enviara notificaciones por pantalla en función de la vida de los personajes después de un turno. Esas notificaciones el estado de la vida en función del personaje atacado y si su vida está en un 50%, 20% y 0%. Se intentó utilizar para esta funcionalidad los patrones **State** y **Observer**, pero se decidió prescindir de su implementación. Para acabar con el desarrollo, pulimos los aspectos de la partida, como un problema con la gestión de la vida y la lista de personajes jugadores en la partida, además de hacer el diagrama UML de las clases y métodos.

Podemos concluir diciendo que hemos podido perfeccionar le juego que hicimos en el anterior cuatrimestre. Se ha hecho un código más limpio, más estructurado y organizado en paquetes y con las distintas clases acorde a los múltiples patrones que había que implementar. Estos cinco patrones implementados se han implementado exitosamente y le han otorgado el funcionamiento y optimización deseada para el curso de la partida y el programa, habiendo aprendido en tiempo real el procedimiento de la creación e implementación de patrones de diseño que ayudan a mejorar nuestro nivel de programación y lógico.

Diagrama UML y explicación de patrones

En este proyecto se han implementado cinco patrones de diseño distintos, que han afectado tanto a la creación de los personajes de distintos tipos y funcionalidades como a la manera en la que ellos interactúan entre ellos.

Factory: Se utilizó el patrón Factory para la creación de enemigos según la dificultad seleccionada por el jugador (normal o difícil). Esto permite encapsular la lógica de creación en clases concretas que devuelven enemigos adaptados a cada nivel de dificultad, facilitando futuras ampliaciones o ajustes del comportamiento de los bots sin modificar el código cliente.

Abstract Factory: El patrón Abstract Factory fue clave para crear bots de diferentes tipos (guerrero, mago, arquero) con sus atributos y mecánicas propias. Este patrón permitió generar familias completas de personajes, asegurando que todas sus características (ataque, defensa, curación, etc.) sean coherentes entre sí según el tipo. Así se separa la lógica de creación de personajes del resto del sistema.

Decorator: Para añadir funcionalidades específicas a los personajes (como ataques mágicos, uso de flechas o curaciones), se usó el patrón Decorator. Partiendo de un personaje base, se “decoró” con habilidades adicionales propias de su clase. Esto permitió extender las capacidades de los personajes sin tener que modificar su código original ni crear muchas subclases distintas.

PROYECTO FINAL 2025

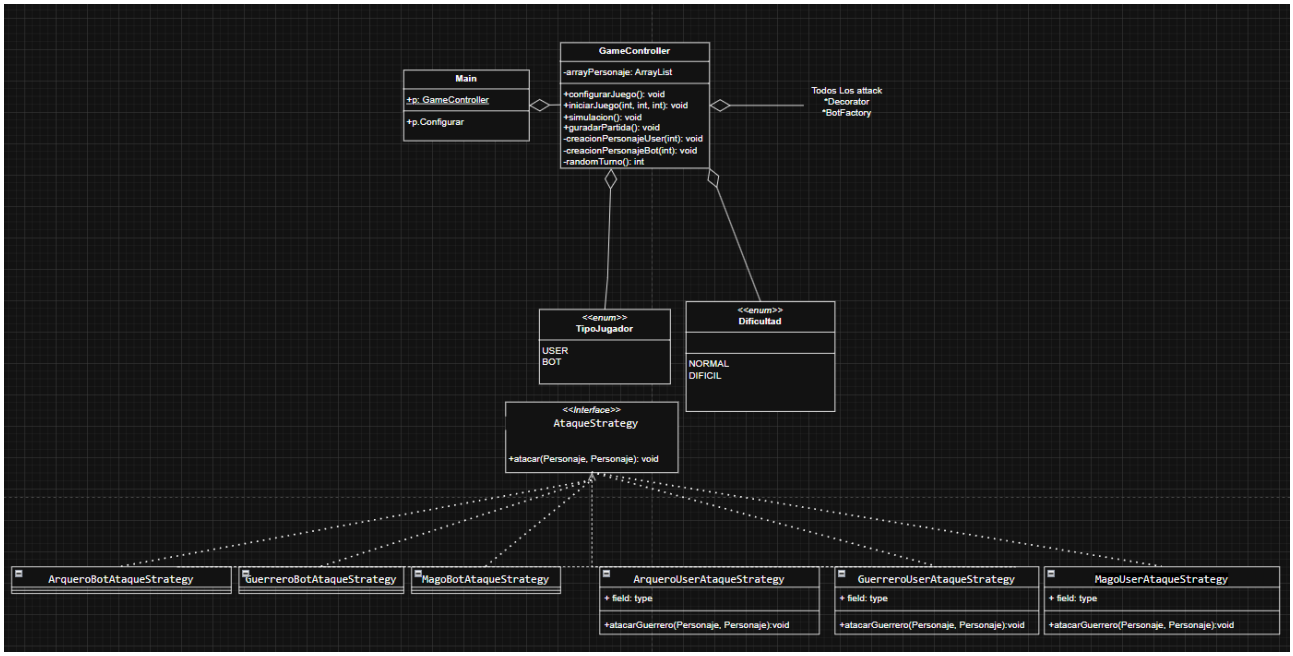
Diagrama UML y explicación de patrones

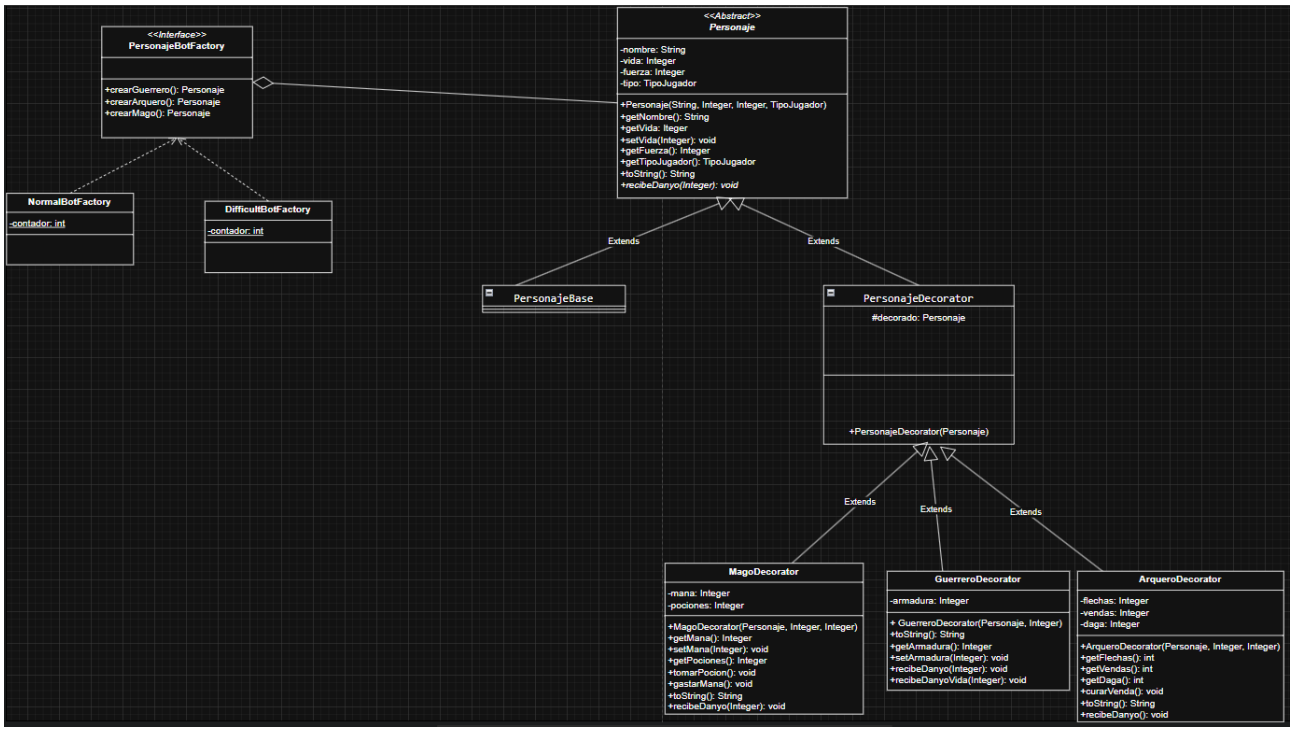
Facade: La clase *GameController* implementa el patrón Facade para simplificar la interacción con el sistema. Desde la clase *Main*, basta con una única llamada al método principal del controlador para iniciar todo el flujo del juego. Así se oculta la complejidad del subsistema y se mejora la organización general del código y su facilidad de uso.

Strategy: El patrón Strategy se utilizó para definir comportamientos diferentes de ataque y toma de decisiones, tanto en jugadores como en bots. Según el tipo de personaje (usuario o máquina, guerrero, mago o arquero), se selecciona una estrategia distinta que define cómo actúa el personaje en su turno. Esto permite cambiar el comportamiento sin alterar la estructura de las clases principales.

Gracias al uso de estos patrones, el juego no solo es funcional, sino también flexible y mantenible. Se ha logrado dividir correctamente las responsabilidades entre clases, favoreciendo la reutilización y facilitando futuras ampliaciones del sistema.

Diagrama UML




<<Interface>>
PersonajeBotFactory
+crearGuerrero(): Personaje
+crearArquero(): Personaje
+crearMago(): Personaje
<<Abstract>>
Personaje
-nombre: String
-vida: Integer
-fuerza: Integer
-tipo: TipoJugador
+Personaje(String, Integer, Integer, TipoJugador)
+getNombre(): String
+getVida: Iteger
+setVida(Integer): void
+getFuerza(): Integer
+getTipoJugador(): TipoJugador
+toString(): String
+recibeDanyo(Integer): void
NormalBotFactory
-contador: int
DifficultBotFactory
-contador: int
Extends
Extends
PersonajeBase
PersonajeDecorator
#decorado: Personaje
+PersonajeDecorator(Personaje)
Extends
Extends
Extends
MagoDecorator
-mana: Integer
-pociones: Integer
+MagoDecorator(Personaje, Integer, Integer)
+getMana(): Integer
+setMana(Integer): void
+getPociones(): Integer
+tomarPocion(): void
+gastarMana(): void
+toString(): String
+recibeDanyo(Integer): void
GuerreroDecorator
-armadura: Integer
+ GuerreroDecorator(Personaje, Integer)
+toString(): String
+getArmadura(): Integer
+setArmadura(Integer): void
+recibeDanyo(Integer): void
+recibeDanyoVida(Integer): void
ArqueroDecorator
-flechas: Integer
-vendas: Integer
-daga: Integer
+ArqueroDecorator(Personaje, Integer, Integer)
+getFlechas(): int
+getVendas(): int
+getDaga(): int
+curarVenda(): void
+toString(): String
+recibeDanyo(): void

PROYECTO FINAL 2025

Anexos

1. Comienzo del juego

```
---BIENVENIDO AL JUEGO---

Elija una opcion de las siguientes:

1. Empezar Partida Nueva
2. Guarda la Partida
3. Salir
---> 1

-------COMIENZA LA PARTIDA!!!-------


-------

Escoge el nivel de dificultad:
1- Normal 2-Dificil
---> 1

-------

Cuantos jugadores quieres?
---> 1
Cuantos bots quieres?
---> 0
Debe haber al menos un jugador usuario y un bot, o al menos dos jugadores usuario.

-------

Cuantos jugadores quieres?
---> 2
Cuantos bots quieres?
---> 0
```

2. Continuación de partida (Guerrero contra Arquero)

```
NOMBRE: J1
VIDA: 100
FUERZA: 20
ARMADURA: 30
Que deberia hacer el guerrero?
1- Atacar
--->
1
El jugador J1 ataca a M0
NOMBRE: M0
VIDA: 40
FUERZA: 20
DAGA: 10
FLECHAS: 20
VENDAS: 5
Que deberia hacer el arquero?
1- Atacar
2- Consumible

...
Como deberia atacar el arquero?
1- Arco
2- Daga

...
El jugador M0 ataca a J1 con arco
Has gastado una flecha
```

3. Final de partida

```
NOMBRE: J1
VIDA: 90
FUERZA: 20
ARMADURA: 0
Que deberia hacer el guerrero?
1- Atacar
--->
1
El jugador J1 ataca a M0
M0 esta muerto


---------

El ganador es: J1

---------
NOMBRE: J1
VIDA: 90
FUERZA: 20
ARMADURA: 0

---BIENVENIDO AL JUEGO---

Elija una opcion de las siguientes:

1. Empezar Partida Nueva
2. Guarda la Partida
3. Salir
---> 1
```