

**TÍTULO: game Design Patterns**

**ALUMNO / A:** ERICK CÁRDENAS SILVA, ÁLVARO MARTÍNEZ PARPOLOWICZ Y JORGE TESCH TORRES

**GRADO:** INGENIERÍA DEL SOFTWARE Y MATEMÁTICA COMPUTACIONAL

Índice

[Descripción del juego 1](#_Toc72757061)

[Meta del juego 1](#_Toc72757062)

[Personajes 1](#_Toc72757063)

[Estrategias de los enemigos 1](#_Toc72757064)

[Estados de los personajes 2](#_Toc72757065)

[Ítems 2](#_Toc72757066)

[Manual 4](#_Toc72757067)

[Simulación de juego: 4](#_Toc72757068)

[Elección de consola: 4](#_Toc72757069)

[Elección de Interfaz grafica 5](#_Toc72757070)

[Paquete base 7](#_Toc72757071)

[State Pattern 8](#_Toc72757072)

[Decorator Pattern 9](#_Toc72757073)

[Template Method & Strategy Pattern 11](#_Toc72757074)

[Abstract Factory Pattern & Template Method 12](#_Toc72757075)

[Singleton, Strategy & Facade Patterns 14](#_Toc72757076)

# Descripción del juego

## Meta del juego

Se trata de un juego de peleas por turnos RPG, donde el jugador tendrá que enfrentar a varios enemigos en cada mundo hasta acabar el mundo 5. Para ello en cada mundo se le dejará a elegir un nuevo ítem que añadirá a su colección de ítems facilitando el camino al jugador.

## Personajes

### Personajes: Jugador

Este será el personaje manejado por el jugador que decidirá sus acciones y deberá intentar derrotar a todos los enemigos hasta pasarse el mundo 5.

### Personajes: Enemigos

Estos son personajes que el jugador tendrá que derrotar. Los enemigos portarán distintas armas, y decidirán sus acciones mediante estrategias de combate.

### Personajes: Jefes

Los jefes son enemigos más poderosos o tienen alguna habilidad que dificulta el avance del jugador.

## Estrategias de los enemigos

### Estrategia agresiva

Esta es una estrategia que revisará estadísticas de los aliados y los enemigos, pero generalmente tenderá a atacar.

### Estrategia defensiva

Esta es una estrategia que revisará estadísticas de los aliados y los enemigos, pero generalmente tenderá a defenderse.

### Estrategia “tonta”

Esta es una estrategia que no se basa absolutamente en nada y decide sus acciones aleatoriamente.

### Estrategia normal

Esta es una estrategia que revisara sus propias estadísticas y las del enemigo para decidir cuál es la mejor acción. La que salga como mejor acción no tiene por qué ser la selecciona solo que, según qué tan buena sea la decisión, esta tendrá más posibilidades de ser ejecutada. Esta estrategia solo será implementada en jefes.

### Estrategia difícil

Esta es una estrategia similar a la estrategia normal solo que también tendrá en cuenta lo buenos que son sus ítems ofensivos o defensivos. También será solo para jefes.

## Estados de los personajes

Los estados son una característica de los personajes. Estos siempre están en un estado que puede ser estándar, confuso, furioso, paralizado, envenenado, seriamente paralizado, y seriamente envenenado.

### Estándar

Este será el estado por defecto de los personajes, no tendrá ningún efecto permanecer en este estado.

### Confuso

El jugador tendrá una probabilidad de acabar realizando su acción sobre sí mismo, aunque seleccione un enemigo como objetivo

### Furioso

Si un personaje está furioso su estadística de ataque se verá multiplicada por 2, pero al acabar el estado de furioso, tendrá posibilidad de acabar en Confuso en vez de en estándar.

### Paralizado

Un personaje paralizado tendrá una probabilidad de que su acción ese turno acabe no realizándose.

### Envenenado

El envenenamiento causa que el personaje reciba daño fijo cada turno.

### Seriamente paralizado

Un personaje seriamente paralizado no realizara la acción que elija ese turno. Solo se puede acceder a este estado si te paralizan estando ya paralizado.

### Seriamente envenenado

Un personaje seriamente envenenado perderá vida, en función de su vida máxima, cada turno. Solo se puede acceder a este estado si te envenenan estando ya envenenado.

## Ítems

Los ítems son los objetos que tanto el jugador como los enemigos. Estos pueden ser:

### Ítem normales o regulares

Estos ítems solo otorgaran estadísticas al portador.

### Ítems activables

Objetos que dotan al jugador de una habilidad concreta con la que atacar/defenderse/curarse/etc.. Además, también mejoran sus estadísticas.

### Ítems usables

Estos son una variante de objetos activables pero que son de una cantidad de usos fija, como las pociones o los antídotos.

### Ítems pasivos

Estos objetos darán una ventaja pasiva al jugador además de mejorar sus estadísticas. Las ventajas pasivas otorgan bonus de estadísticas temporales cuando se cumplen ciertas condiciones (sin que el jugador deba realizar ninguna acción para activarlas).

# Manual

## Simulación de juego:

Nada más iniciar el juego este te preguntara por tu nombre de jugador, y como quieres jugar al juego, por consola o usando una interfaz gráfica.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Figura 1: Menú inicial.

## Elección de consola:

El juego te mostrara una descripción del nivel, ensenándote los enemigos que hay y sus estadísticas. Y además como es el inicio de un mundo, el juego te dejara elegir entre 3 ítems que equipar.

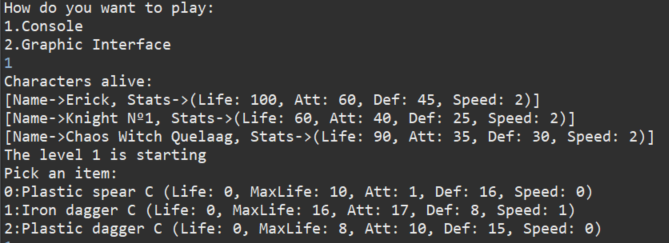


Figura 2: Consola, Simulación inicio de partida.

### Sistema de turnos por consola

Después de elegir el ítem, empezara el turno que consiste elegir acción y objetivo. En esta simulación se ha elegido ataque con espada al objetivo 1. Después de seleccionar objetivo se hará el cálculo total de daño y se reescribirán las estadísticas actuales de todos.

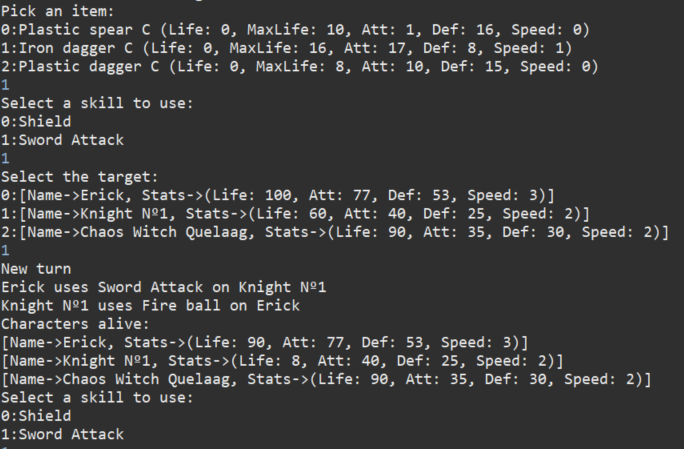


Figura 3: Consola, Sistema de turnos.

### Cambio de mundo por consola

Al comprobar que estas solo en un mundo, el juego te pasara al siguiente, generando nuevos enemigos y permitiéndote elegir un ítem para añadir a los que ya tienes.

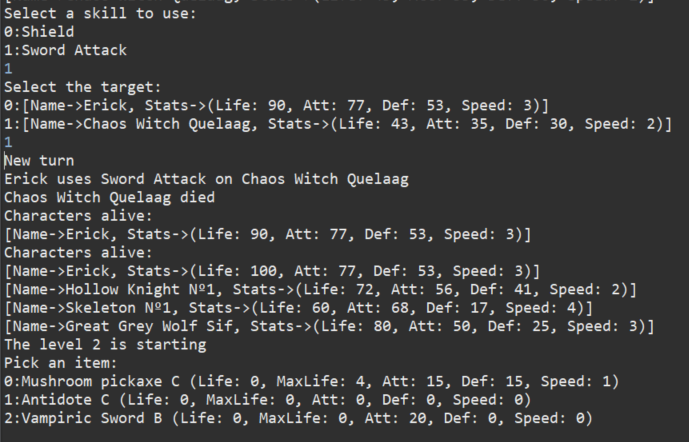


Figura 4: Consola, Cambio de mundo.

## Elección de Interfaz grafica

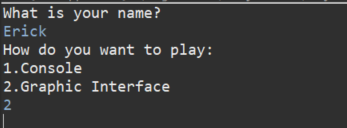


Figura 5: Elección de interfaz Gráfica.

Al seleccionar la interfaz se te abrirá un menú que está dividido en 4 secciones, superior izquierda, superior derecha, inferior izquierda e inferior derecha.

**Superior izquierda:** Allí se guardará un log de todo lo que va pasando en la partida.

**Superior derecha:** Aquí se mostrarán los personajes que hay en la sala y su vida actual.

**Inferior izquierda:** Este recuadro informara al jugador de sus estadísticas, su equipamiento y su estado actual.

**Inferior derecha:** Este será el menú donde el jugador seleccionará sus acciones, tanto a la hora de elegir ítem al empezar un mundo, como para elegir su acción y objetivo de la acción.

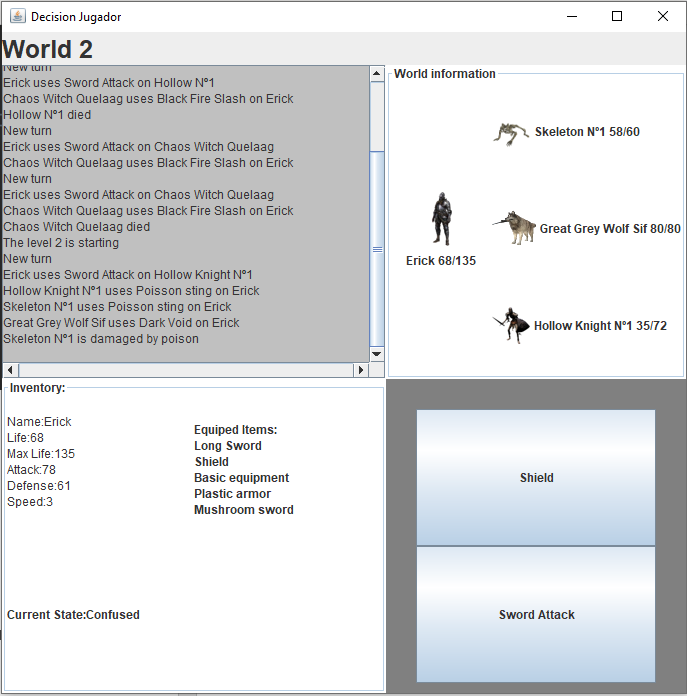


Figura 6: Interfaz Gráfica, simulación de partida.

# Paquete base

Figura X: Paquete base

Figura 7: Paquete base

El diagrama UML que se presenta no incluye ningún patrón de diseño. Aun así, se considera útil presentar las principales entidades y sus relaciones, incluidas en el paquete base, que constituirán la base del resto del proyecto.

La clase abstracta Character modela a todos los personajes del juego. Tiene como atributos un nombre, un String con la ruta en la que se encuentra el Sprite que define al personaje, un objeto tipo Item (patrón decorador) y un objeto tipo State (patrón state). Además de los getters y setters implementa métodos que delegan la ejecución por composición en alguno de los atributos mencionados.

De esta clase extienden Enemy, con la que se representa a los NPC’s del juego, y, Player, que modela al jugador. Los objetos Enemy tienen un atributo DecisionTemplate (patrón template-strategy) que modela su comportamiento. Los objetos tipo Player almacenan sus decisiones en variables tipo Character y ActiveItemDecorator (patrón decorador).

La clase Stats funciona como wrapper que aglutina todas las estadísticas que contempla el juego en una sola clase. Estas estadísticas (vida, vida máxima, ataque, defensa y velocidad) están representadas por valores enteros. Cabe destacar el método applyStats() que se usa para sumar objetos de tipo Stats entre ellos.

Por último, la clase Action es utilizada por todas las demás clases como interfaz de comunicación, es decir, los demás objetos generan mensajes en formato Action para comunicarse con otros objetos. Una acción está determinada por el personaje que la usa (user), el personaje contra el que se usa (target), la variación de estadísticas que va a producir (variation) y dos atributos de tipo enumerado que dan profundidad al juego caracterizando las acciones (actionType y skillType). Una acción cuyo valor de usuario y/o objetivo es null representa eventos del sistema. Por ejemplo, la acción provocada por el estado de envenenado se considera que es lanzada por el entorno hacia el personaje envenenado.

# State Pattern

Hemos utilizado el State Pattern a la hora de implementar cambios de estados entre los que puede pasar el personaje como pueden ser envenado, paralizado, furioso…

Nos hemos decantado por este patrón de diseño para este apartado ya que los estados realizaban acciones diferentes, por lo que cada vez que se modificara el estado del personaje, el proceso que este realizaría se vería también modificado. Sin embargo, no se ha utilizado el patrón de diseño Strategy ya que, además, no todos los estados son accesibles desde todos los demás estados. Es decir, cada estado debe implementar una lógica de cambio de estado que le permita decidir si puede/debe o no cambiar a otro estado.

Diagrama

Descripción generada automáticamente

Figura 8: Diagrama UML del State Pattern.

En el diagrama UML se puede observar los distintos componentes del patrón State. La clase State ejerce de contexto, la interfaz CharacterState define los métodos comunes de los estados concretos, la clase abstracta AbstractState hace de plantilla para los estados implementando algunos métodos y atributos por defecto, y los diferentes estados que conforman el patrón de diseño.

**State (Contexto):** Esta clase gestiona y hace transparente la lógica detrás del patrón de estados. Tiene dos atributos, un entero con la duración en turnos de cada estado y un CharacterState con el estado concreto. Este está, por defecto, inicializado al estado estándar.

**Enumerado States:** Utilizado para simplificar la comunicación en este patrón. Tiene un valor por cada estado existente.

**CharacterState(Estado):** Interfaz que se encarga de definir el comportamiento de los estados. Contiene las funciones process, effect y setSuggestion. El método setSuggestion() se utiliza para informar al estado a que otro estado le sugiere el entorno (entidad superior al contexto de este patrón) cambiar. El método process() implementa la lógica de cambio de estado basándose en la cantidad de turnos restantes, y la sugerencia de cambio de estado. Por lo tanto, que el entorno haya sugerido cambiar a estado confundido no quiere decir que el método process() vaya a cambiar a estado confundido, sino que tendrá en cuenta la sugerencia y tomará la decisión apropiada. El método effect() modifica una acción dependiendo del estado en el que se encuentre su usuario.

**AbstractState y las clases estado:** Implementan la lógica concreta de los métodos mencionados anteriormente.

# Decorator Pattern

Figura 9: Diagrama UML parcial del patrón Decorator

Figura 9: Clase Equipment desarrollada

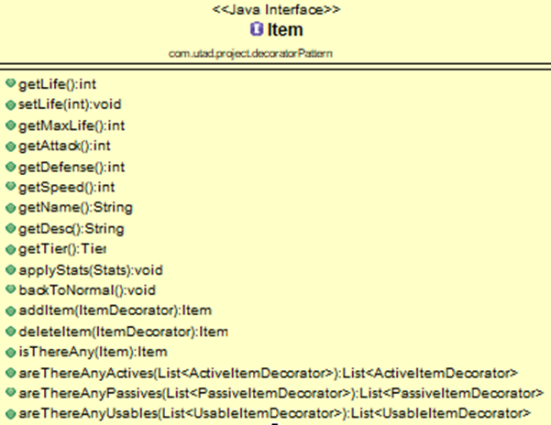


Figura 11: Interfaz Item desarrollada

Figura 10: Clase ItemDecorator desarrollada

En el proyecto hemos decidido utilizar el patrón decorador para modelar los objetos (o ítems) que puede llevar equipado un personaje. No hay un límite de objetos que puede llevar equipados y todo nuevo objeto equipable representa una mejora respecto al equipamiento anterior. Además, buscamos que no todos los ítems sean iguales y que algunos puedan darle al enemigo la capacidad de usar habilidades (por ejemplo: equipar el arco te permite lanzar flechas). Así que se decidió que el patrón decorador se usaría para gestionar tanto las estadísticas de un personaje como las habilidades que dispone.

Así, todos los ítems se agrupan bajo la interfaz decorable Item. Por definición todos los ítems deben tener: un objeto tipo Stats, que indique con qué estadísticas está mejorando el equipamiento del personaje (además de métodos para acceder a todos sus atributos y métodos propios); un nombre, que lo identifique; y, una categoría (Tier, tipo enumerado), para clasificarlos según su poder.

La clase Equipment implementa la interfaz Item y constituye el componente base del patrón. De manera general la clase Equipment tendrá un objeto Stats inicializado a cero y se utilizará para almacenar los cambios temporales en las estadísticas de un personaje (ejemplo: doble de defensa este turno) que pueden ser revertidos con el método backToNormal(), o la vida que va perdiendo el personaje.

La clase abstracta ItemDecorator implementa la interfaz Item e implementa parte de los métodos (recursivos la mayoría) comunes a todos los objetos que decoran a otros ítems. Tiene, como no podía ser de otra manera, una referencia (equipment) al objeto que está decorando, que puede ser el componente base u otro decorador.

La clase RegularItem extiende de ItemDecorator y es la versión de decorador más básica, que solo decora las estadísticas del componente base.

La clase abstracta PassiveItemDecorator es una especialización de decorador, que añade la funcionalidad de mejorar las estadísticas de la jerarquía completa de decoradores mediante el método abstracto modifyStats(). Las clases concretas Thornmail y DemonSpear, entre otras no mostradas en el UML, extienden de esta e implementan dicho método.

La clase abstracta ActiveItemDecorator es otra especialización de decorador, que añade la funcionalidad de usar habilidades con el método abstracto useSkill(). Las implementaciones concretas de este método llevan la lógica que determina la variación de estadísticas de los objetos Action. Son pequeñas calculadoras de daño que mandan información al GameManager (patrón Singleton) que calculará los daños finales. Las clases concretas Bow y LongSword son algunos de las hijas concretas de esta clase.

Por último, la clase abstracta UsableItemDecorator es una especialización de la especialización ActiveItemDecorator. Añade la característica de que solo pueden ser usadas (método useSkill() ) un numero amount de veces. Una vez agotados serán eliminados de la jerarquía de decoradores. La clase Potion es un ejemplo de hija concreta de esta clase.

A continuación, se presenta el diagrama UML completo para poder observar todos los objetos que extienden de las clases especialización de decorador, que por estética y simplicidad se han suprimido del diagrama inicial.

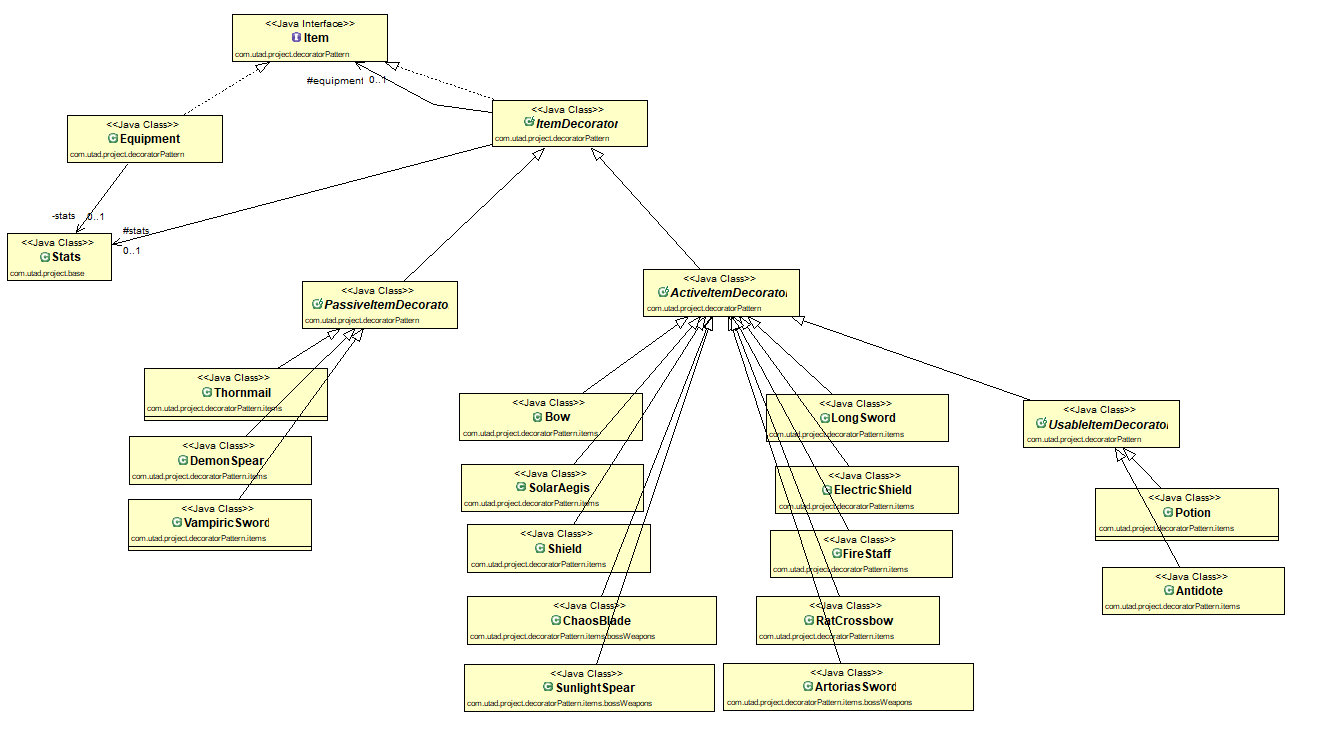


Figura 12: Diagrama UML completo del patrón decorador sin los métodos ni atributos mostrados.

# Template Method & Strategy Pattern

El sistema de elección de acciones de los enemigos está diseñado usando un Template method, y un patrón Strategy.

Nos hemos decantado por estos patrones de diseño ya que no todos los enemigos deciden su forma de actuar de la misma manera, dicho de otro modo, cada enemigo sigue su propia estrategia de combate. Además, tienen la posibilidad de adaptarse cambiando de estrategia de combate en función de los cambios que se producen en el juego. Este cambio es impredecible y se decide por tanto en tiempo de ejecución. Por otro lado, los enemigos siempre siguen un patrón para elegir sus acciones: analiza el entorno, almacena las habilidades disponibles, y selecciona la habilidad; por lo que es un esqueleto que usamos como Template.

De esta manera cada estrategia describirá solo los algoritmos de worthAttack(), worthDefense(), worthNeutral() y selectSkill().Los enemigos podrán ir cambiando de estrategias de un turno a otro mediante la función ChangeStrategy(), que utiliza al propio enemigo como contexto.

Diagrama

Descripción generada automáticamente

Figura 13: Diagrama UML del Template Method & Strategy Pattern.

En el diagrama UML de estos patrones podemos ver como los enemigos tienen todos un DecisionTemplate como atributo, que define su comportamiento / estrategia de combate. La clase abstracta DecisionTemplate es la que hace la labor de Template mediante la función final decisión(). La función analize() es una pseudo-plantilla se define en esta clase, pero está abierta a modificaciones. Las funciones worthAttack(), worthDefense(), worthNeutral() y selectSkill() deben ser definidas en cada estrategia de manera obligatoria.

Las funciones editables por cada patrón son:

**Template Method:**

Analize(): La función Analize es la que analiza el entorno y recoge las probabilidades de todas sus acciones. Se ve redefinida en la estrategia de healer donde tiene una probabilidad de heal que es una acción neutral diseñada especialmente para sanar. Este es un ejemplo de estrategia que analiza el entorno de manera distinta.

**Strategy Pattern:**

Los diferentes worths: estos devuelven un entero con añade probabilidad a su acción, por ejemplo, si dentro de la función worthAttack() se calcula que es buena opción atacar, devolverá un número muy alto.

selectSkill(): Esta función es la finalmente hace el cálculo de las probabilidades para saber de qué tipos era su acción (ofensiva, neutral o defensiva), y decide que acción debe ejecutarse.

# Abstract Factory Pattern & Template Method

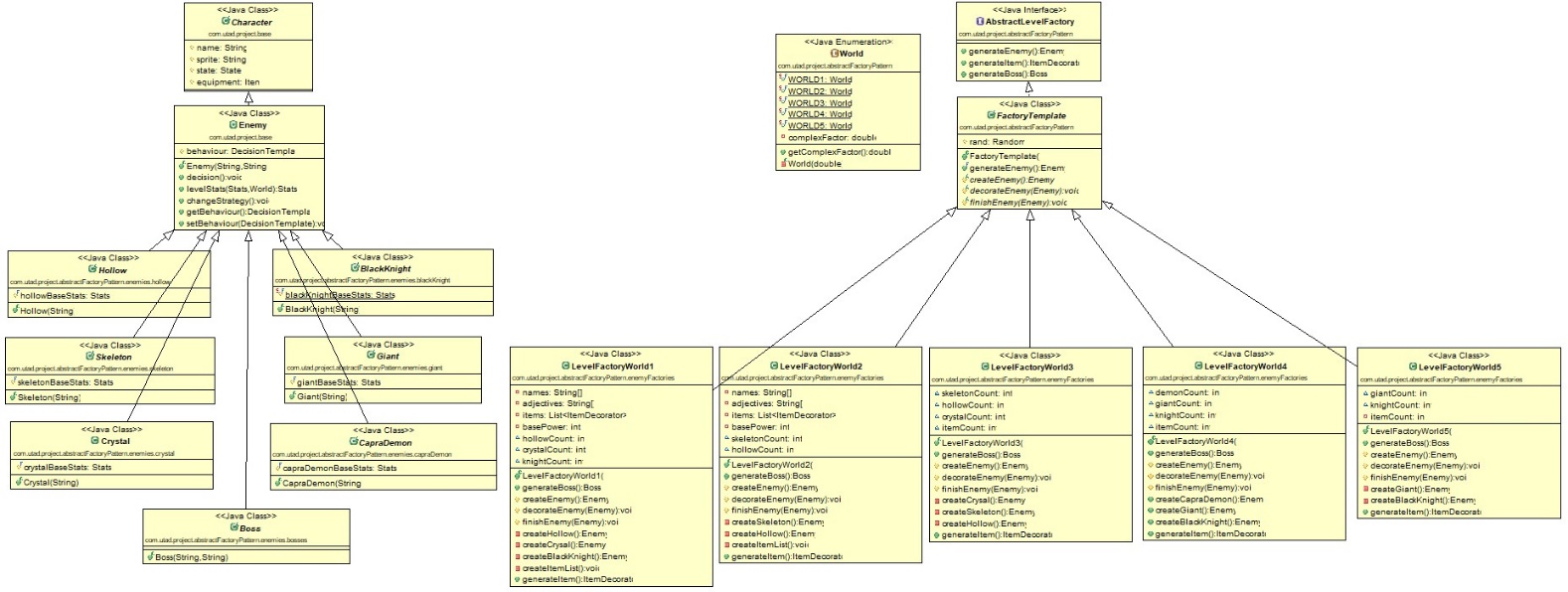


Figura 15: Diagrama UML Enemigos Concretos

Figura 14: Diagrama UML con la jerarquía de enemigos (solo clases abstractas)

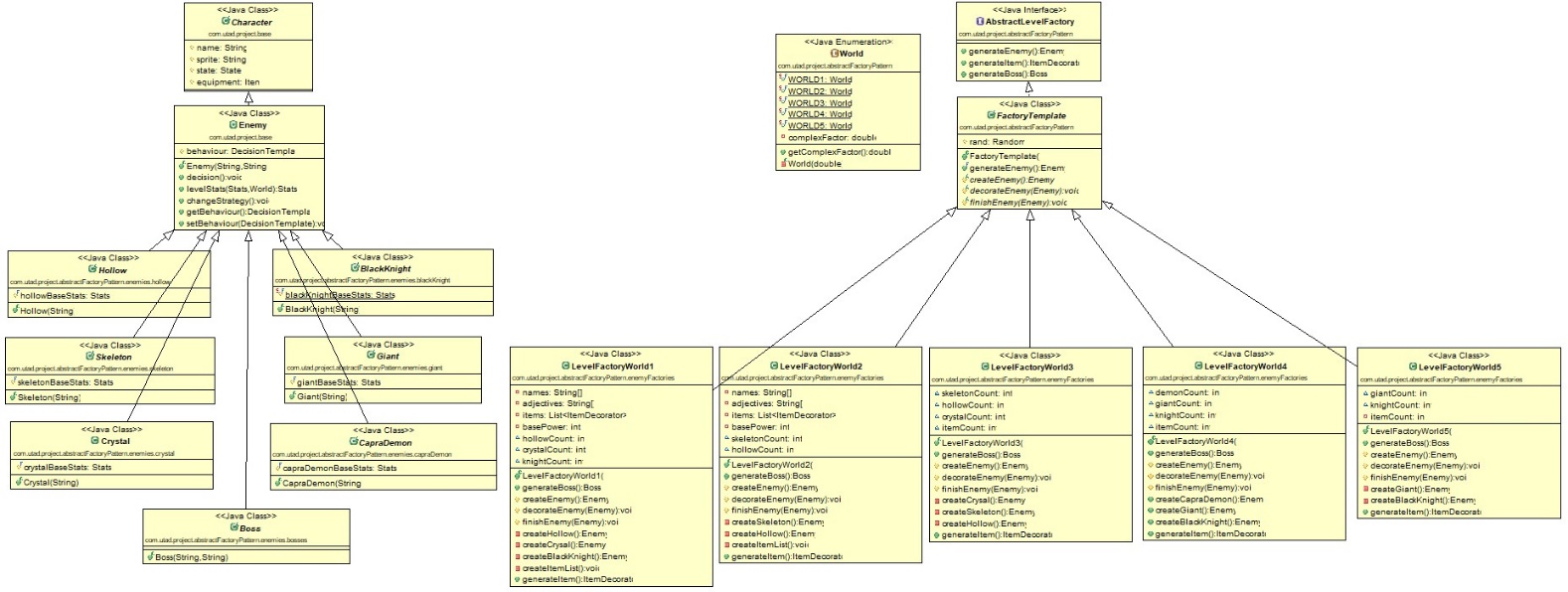


Figura 15: Diagrama UML con las factorías

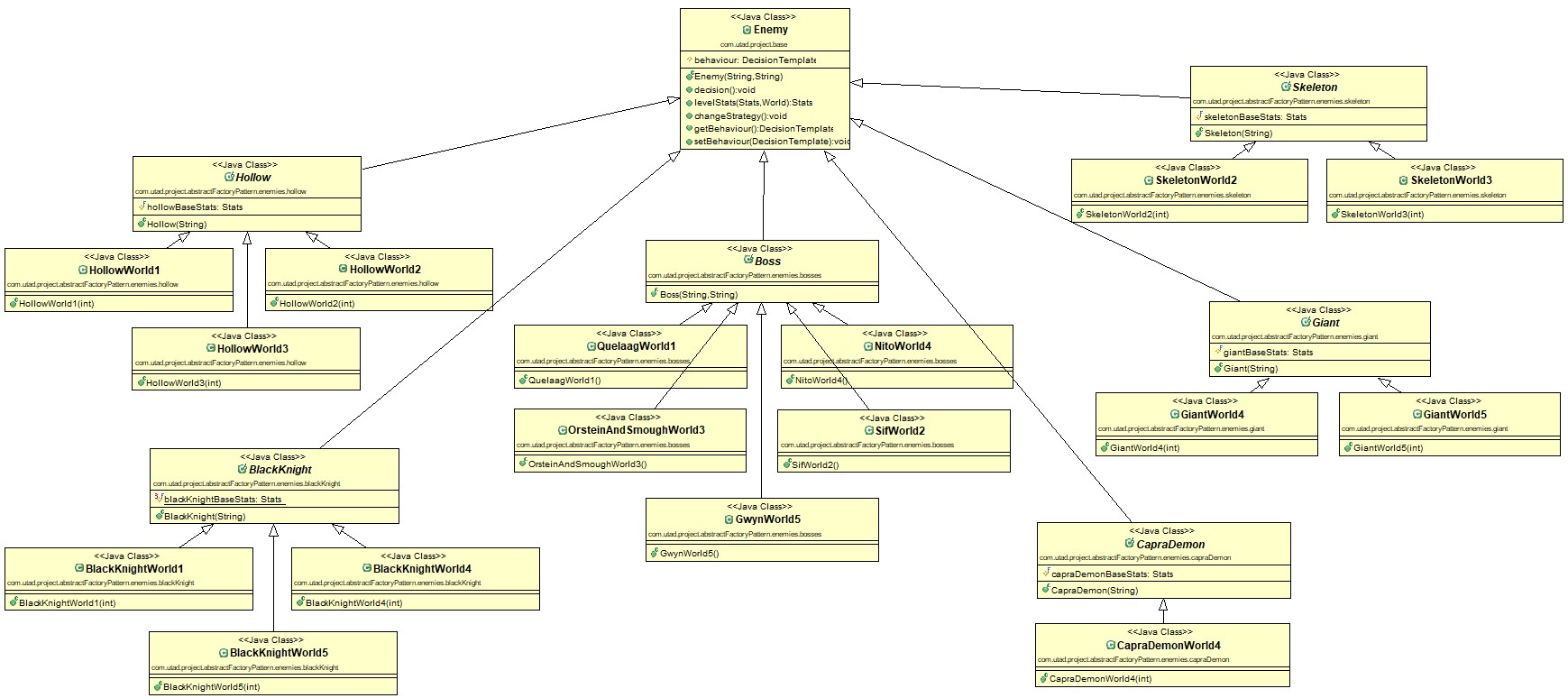


Figura 16: Diagrama UML con la jerarquía completa de enemigos

Para la generación de los enemigos a los que nos enfrentamos en el videojuego utilizamos el patrón Abstract Factory ya que este nos permite crear los diferentes conjuntos de enemigos, los cuales están relacionados entre sí, sin especificar sus clases concretas.

El Abstract Factory se compone de una interfaz general, AbstractLevelFactory, en la quedan definidos los métodos principales para generar un enemigo, un jefe final y un objeto. Sin embargo, como para generar un enemigo es necesario realizar una serie de pasos, hemos desarrollado una clase FactoryTemplate en la que queda definido el método generateEnemy() como el desarrollo final en orden de los métodos createEnemy(), que crea la instancia de un enemigo concreto; decorateEnemy(), para otorgar equipamiento a ese enemigo con el patrón Decorator; y finishEnemy(), con el que damos al enemigo un comportamiento definido por el patrón Strategy y/o cambiamos el estado inicial con el que empieza a jugar con el patrón State.

Estos últimos tres métodos, juntos a los utilizados para generar objetos y jefes, son los que quedan definidos y sobrescritos en las cinco factorías, correspondientes a los cincos mundos de los que se compone el videojuego, para así generar los diferentes objetos que, aunque pertenecen a las mismas familias, poseen elementos y características propias en función del mundo en el que nos encontremos.

Todos los enemigos que se generan extienden de una clase general para cada tipo de enemigo, las cuales a su vez extienden de la clase Enemy. En la clase general para cada tipo de enemigo quedan instanciadas las estadísticas base y la ruta de la imagen utilizada en la interfaz gráfica, excepto para los jefes ya que estos poseen estadísticas e imágenes individuales.

Finalmente, encontramos las clases concretas de cada familia de enemigos, las cuales son instanciadas por las factorías y conforman los enemigos encontrados dentro del propio videojuego. En sus contructores se les otorga un nombre especifico y se multiplican sus estadísticas base por el factor de complejidad correspondiente al mundo en el que son instanciados.

# Singleton, Strategy & Facade Patterns

Figura 17: Diagrama UML del patrón Singleton, combinado con un patrón Strategy y un Facade

La coordinación e interconexión de todas las clases explicadas anteriormente se gestionan mediante la clase GameManager. Esta implementa el patrón de diseño Singleton para asegurar que existe un y solo un gestor del juego accesible para todos los objetos. Por otro lado para reducir la complejidad de implementación del sistema de gestión, hemos decidido crear tres subsistemas (sistema de combate, sistema gráfico y sistema de generación de niveles), de tal manera que la clase GameManager actúa también de fachada de cara al resto de objetos mediante métodos como play() o informPlayer().

Puesto que se ha decidido que el jugador puede relacionarse con el juego tanto a través de la consola, como por medio de una interfaz gráfica, hemos implementado una estrategia para variar la forma de jugar, que se decide en tiempo de ejecución. Dadas las características del juego no se contempla el cambio de estrategia gráfica en medio del juego.

La clase ConsoleDisplay implementa la estrategia gráfica DisplayStrategy e interactúa con el jugador por medio de un objeto tipo Scanner e imprimiendo resultados por la consola. Por otro lado, la clase WindowDisplay implementa también esta interfaz, aunque se relaciona con el usuario por medio de un objeto tipo Window que extiende de la clase JFrame del paquete swing de java. El objeto tipo Window es nuestra propia ventana cuyos métodos en resumen la crean (constructor) y la actualizan turno a turno (demás métodos).

La clase CombatDisplay se encarga de gestionar el subsistema de combate. Teniendo en cuenta que tan solo hay creada una forma de resolver los combates, no hemos llegado a implementar un patrón Strategy como para el subsistema gráfico, aunque sería susceptible de mejora si se incluyera una distinción entre modos de juego (ejemplo: modo fácil y modo difícil).

Por último, el sistema de generación de ítems y enemigos viene dado por el patrón Abstract Factory (explicado anteriormente).

Con todo esto, la clase GameManager coordina y delega en los distintos subsistemas las tareas a realizar. En un principio también se encargaba de gestionar las excepciones que lanzaran los subsistemas, pero por simplicidad y de cara a crear nuevas estrategias que no lanzaran excepciones, decidimos filtrarlas y tratarlas antes de tener que lanzarlas.