Desarrollo del Juego "Adventures C# ANASS EL MORABIT"

1. Introducción

- **Descripción general del juego: Este es un juego de aventuras desarrollado en C#, donde el jugador comienza controlando un personaje estándar (Jugador). El juego consta de tres mapas diferentes, cada uno con enemigos únicos. El primer mapa contiene 5 enemigos básicos, el segundo mapa cuenta con 6 enemigos especiales, y el tercer mapa es el escenario de la batalla final contra un jefe (Boss).**

  El objetivo principal del jugador es vencer a los enemigos en combate, avanzando de un enemigo a otro hasta llegar al jefe final. Además, si el jugador gana tres combates consecutivos, puede desbloquear un personaje especial dentro del juego, eligiendo entre diferentes clases como Mago, Arquero o Guerrero, cada uno con habilidades únicas.

- **Tecnologías utilizadas: C# como lenguaje de programación, Visual Studio Code**

2.Estructura General del Juego

- **Objetivo del juego:**
  El objetivo principal del juego es explorar tres mapas distintos, enfrentarse a una serie de enemigos y superar los desafíos que presentan. A medida que el jugador derrota a los enemigos, avanza a través de cada mapa, culminando en una batalla final contra un jefe (Boss). Además, al ganar tres combates consecutivos, el jugador puede desbloquear un personaje especial (Mago, Arquero, Guerrero), lo que le otorga nuevas habilidades para continuar la aventura. El jugador debe subir de nivel, mejorar sus estadísticas y utilizar sus habilidades estratégicamente para superar todos los obstáculos.
- **Interfaz de usuario:**
  La interfaz del juego se desarrollará en la consola, utilizando texto para interactuar con el jugador. El juego mostrará información relevante sobre el estado del jugador (salud, nivel, enemigos derrotados, etc.) y los eventos del juego (como los combates, el cambio de personaje, etc.).
  El jugador interactúa mediante comandos de texto, tales como seleccionar acciones en los combates (atacar, defender, usar habilidades), avanzar entre mapas y elegir las decisiones durante la aventura. El menú principal y las opciones estarán representados con texto en consola, y se utilizarán mensajes y decoraciones estilizadas para mejorar la experiencia visual dentro de las limitaciones de la consola.

3. Flujo del Juego

Lógica del Juego

El juego es una experiencia de aventura dinámica donde el jugador debe enfrentarse a varios enemigos a través de combates y estrategias, mientras avanza por distintos mapas. La jugabilidad se basa en la mecánica de tirar dados, gestionar puntos de habilidad y subir de nivel. Aquí están las reglas principales y la lógica que guía el desarrollo del juego:

Inicio del Juego: El jugador comienza el juego controlando a un personaje básico. El primer paso es lanzar dos dados de 6 caras para determinar el daño de ataque que puede infligir al enemigo en su turno. Esto introduce un elemento de azar en cada combate, ya que la tirada de los dados afecta el resultado del ataque.

Determinación del Nivel de Ataque: El nivel de ataque del jugador no solo depende de la tirada de los dados, sino también de sus puntos de habilidad. Estos puntos se multiplican por un factor de 0.10 y se suman al resultado de la tirada. Esto significa que a medida que el jugador acumula más puntos de habilidad, puede aumentar su poder de ataque, lo que lo hace más fuerte en el combate.

Subida de Nivel y Mejora del Personaje: Cada vez que el jugador derrota a un enemigo, gana experiencia y sube de nivel. Con cada nivel, el jugador se hace más fuerte, y este progreso es esencial para enfrentar enemigos más difíciles a lo largo del juego. Además, si el jugador pasa tres rondas sin perder, tiene la opción de elegir uno de los tres personajes especiales disponibles: Mago, Arquero o Guerrero. Esto le otorga habilidades únicas que facilitan el avance en el juego.

Estructura de los Mapas: El juego está dividido en tres mapas que el jugador debe superar. El objetivo principal es derrotar a todos los enemigos en cada mapa:

- **`Primer Mapa:`** El jugador se enfrentará a 6 enemigos básicos. Estos enemigos son relativamente fáciles, pero ofrecen una oportunidad para que el jugador mejore sus habilidades y acumule experiencia.
- **`Segundo Mapa:`** El jugador se enfrentará a 5 enemigos especiales. Estos enemigos son más fuertes y presentan desafíos más complejos, requiriendo una mayor estrategia y mejores habilidades para derrotarlos.
- **`Tercer Mapa:`** En este mapa, el jugador se enfrentará al **Boss final**, un enemigo mucho más fuerte que requiere todo el poder y la destreza adquiridos a lo largo del juego para ser derrotado.

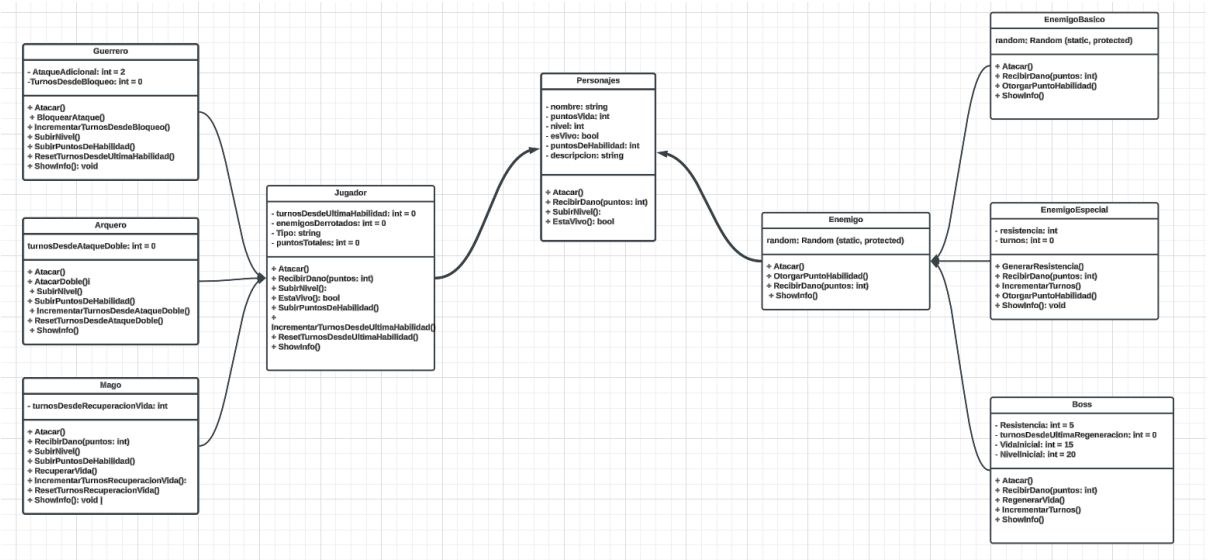
Victoria Automática por Nivel Superior: Si el jugador tiene un nivel superior al del enemigo, gana la batalla automáticamente. Esto le da una ventaja estratégica, ya que permite al jugador ganar batallas sin depender del azar de los dados, siempre que haya invertido tiempo y esfuerzo en subir de nivel.

Sorpresas durante el Juego: Cada turno ofrece una "sorpresa" para el jugador, que puede ser un beneficio o un perjuicio. Por ejemplo, el jugador puede recibir una bonificación de habilidad, ganar puntos de vida adicionales o, en algunos casos, perder puntos de vida. Estas sorpresas agregan una capa de incertidumbre y hacen que cada partida sea única, obligando al jugador a adaptarse constantemente a nuevas situaciones.

Objetivo del Juego: El objetivo principal del jugador es derrotar a todos los enemigos en cada uno de los tres mapas. A medida que avanza, enfrentará enemigos cada vez más poderosos, lo que hace que las decisiones estratégicas y el manejo de recursos, como puntos de habilidad y puntos de vida, sean cruciales para el éxito. Finalmente, debe derrotar al Boss en el tercer mapa para completar el juego.

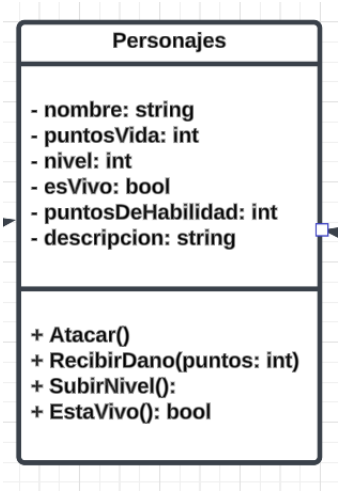
Finalización del Juego: Para ganar el juego, el jugador debe superar todos los enemigos de los tres mapas y mejorar su personaje a través de la experiencia y las habilidades adquiridas. El avance del jugador no solo depende de su habilidad para lanzar los dados, sino también de cómo maneja sus puntos de habilidad, sube de nivel y se adapta a las sorpresas del juego. El jugador debe destruir todos los enemigos y derrotar al Boss final para alcanzar la victoria.

4. Clases Principales



4.1 Clase Personajes (Personajes.cs)

Diagrama UML:



La clase `Personajes` es la clase base para representar personajes en un juego. Contiene atributos comunes a todos los personajes, como su nombre, puntos de vida, nivel, si está vivo o no, sus puntos de habilidad y una descripción opcional.

Atributos:

- **`nombre`: El nombre del personaje.**
- **`puntosVida`: La cantidad de vida del personaje. Si llega a 0, el personaje muere.**
- **`nivel`: El nivel actual del personaje.**

- `esVivo`: Un valor booleano que indica si el personaje está vivo.
- `puntosDeHabilidad`: Los puntos de habilidad del personaje, utilizados para modificar sus acciones.
- `descripcion`: Una descripción del personaje, con un valor predeterminado si no se especifica.

Métodos:

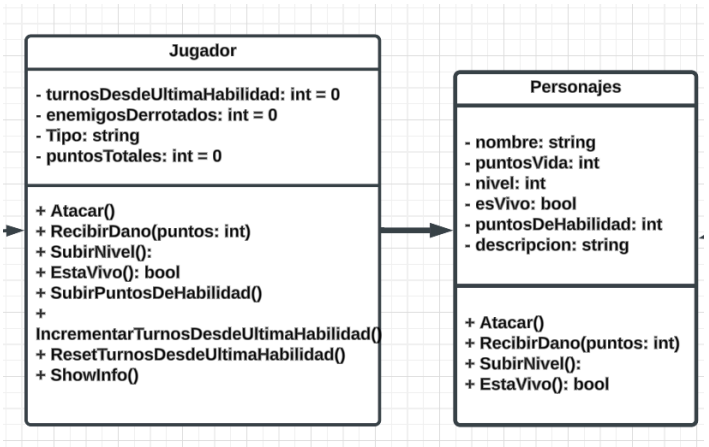
- `Atacar()`: Realiza una tirada de ataque, sumando el valor de dos dados y un bono basado en los puntos de habilidad.
- `RecibirDano(int puntos)`: Recibe daño, reduciendo los puntos de vida del personaje. Si la vida llega a 0 o menos, el personaje muere.
- `SubirNivel()`: Incrementa el nivel del personaje.
- `EstaVivo()`: Verifica si el personaje sigue vivo (es decir, si tiene puntos de vida positivos).

Funcionalidad:

La clase proporciona las funcionalidades esenciales para los personajes del juego, incluyendo atacar, recibir daño, subir de nivel y verificar si siguen vivos. Esta clase está diseñada para ser heredada, lo que permite que otras clases como `Guerrero`, `Enemigo`, etc., añadan sus comportamientos específicos mientras comparten las funcionalidades básicas definidas en `Personajes`.

4.2 Jugador (Jugador.cs)

Diagrama UML:



La clase Jugador es una extensión de la clase base Personajes, diseñada específicamente para representar un jugador dentro del juego. Además de los atributos y métodos heredados de Personajes, la clase Jugador incluye características específicas como puntos de habilidad y la capacidad de subir de nivel, lo que le permite evolucionar a lo largo del juego.

Atributos:

- `nombre`: El nombre del jugador.
- `puntosVida`: La cantidad de vida del jugador. Si llega a 0, el jugador muere.
- `nivel`: El nivel actual del jugador.
- `esVivo`: Un valor booleano que indica si el jugador está vivo.
- `puntosDeHabilidad`: Los puntos de habilidad del jugador, que pueden ser utilizados para mejorar el rendimiento en el juego (por ejemplo, en el ataque o defensa).
- `descripcion`: Una descripción del jugador, que explica sus habilidades o historia. Si no se proporciona, se usa un valor predeterminado.
- `puntosTotales`: Los puntos acumulados por el jugador durante el juego, utilizados para determinar su progreso.
- `turnosDesdeUltimaHabilidad`: Un contador que lleva el número de turnos desde que el jugador utilizó su última habilidad especial.

Métodos:

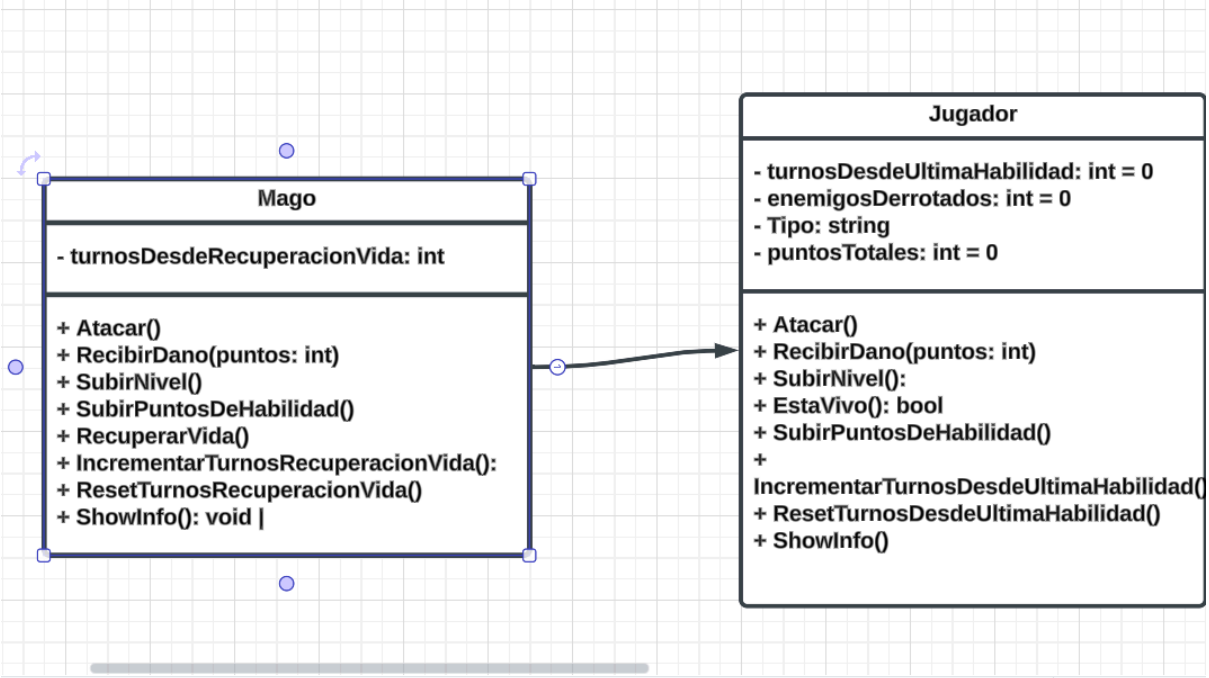
- `Atacar()`: Realiza una tirada de ataque, sumando el valor de dos dados y un bono basado en los puntos de habilidad del jugador.
- `RecibirDano(int puntos)`: Recibe daño, reduciendo los puntos de vida del jugador. Si la vida llega a 0 o menos, el jugador muere.
- `SubirNivel()`: Incrementa el nivel del jugador, mejorando sus habilidades o atributos.
- `EstaVivo()`: Verifica si el jugador sigue vivo (es decir, si tiene puntos de vida positivos).
- `SubirPuntosDeHabilidad()`: Permite al jugador ganar puntos de habilidad, mejorando sus capacidades.
- `ResetTurnosDesdeUltimaHabilidad()`: Reinicia el contador de turnos desde la última habilidad utilizada.
- `ShowInfo()`: Muestra la información detallada del jugador, incluyendo su nombre, nivel, puntos de vida, puntos de habilidad, entre otros.

Funcionalidad:

La clase `Jugador` proporciona la base para un jugador dentro del juego, con la capacidad de atacar, recibir daño, subir de nivel y manejar puntos de habilidad. Los jugadores pueden tener habilidades especiales y un sistema de puntos acumulados que les permite avanzar. Esta clase extiende la funcionalidad básica de la clase `Personajes` y se puede personalizar aún más con comportamientos específicos, como las habilidades del Guerrero, Mage, etc.

4.3 Clase Mago (Mago.cs)

Diagrama UML:



La clase `Mago` extiende la clase `Jugador` y está diseñada para representar a un personaje con habilidades mágicas, enfocado en el ataque y la recuperación de vida. Además de los atributos y métodos heredados de `Jugador`, la clase `Mago` incluye habilidades específicas como la recuperación de vida y un bono de ataque adicional en cada tirada.

Atributos:

- **`turnosDesdeRecuperacionVida`: Un contador que lleva la cantidad de turnos desde la última recuperación de vida del mago. Este atributo es clave para activar la habilidad especial de recuperar vida cada tres turnos.**
- **`Tipo`: El tipo de personaje, en este caso, "Mago".**
- **`puntosTotales`: Puntos acumulados durante el juego, representando el progreso o logros del mago.**
- **`descripcion`: Una descripción personalizada del mago, detallando sus habilidades iniciales y características.**

Métodos:

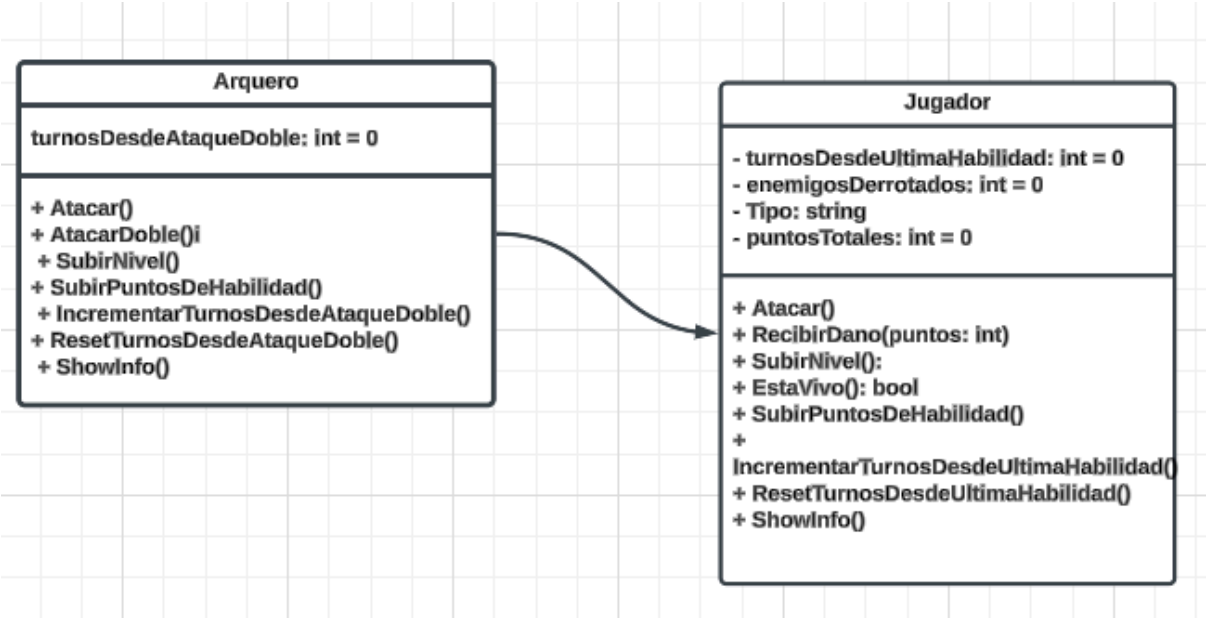
- `Atacar()`: **Realiza una tirada de ataque sumando el valor de dos dados y un bono fijo de +4. Además, se suma un bono adicional según los puntos de habilidad del mago.**
- `RecibirDano(int puntos)`: **Recibe daño de acuerdo a los puntos especificados. Llama al método base para actualizar los puntos de vida.**
- `SubirNivel()`: **Incrementa el nivel del mago, mejorando sus habilidades y atributos. Este método utiliza la implementación base de la clase `Jugador`.**
- `SubirPuntosDeHabilidad()`: **Permite que el mago gane puntos de habilidad, lo que mejora su rendimiento en el juego.**
- `RecuperarVida()`: **La habilidad especial del mago que le permite recuperar 2 puntos de vida cada tres turnos, si ha derrotado a un enemigo. Esta habilidad se activa tras haber transcurrido tres turnos desde la última recuperación.**
- `IncrementarTurnosRecuperacionVida()`: **Aumenta el contador de turnos desde la última vez que el mago recuperó vida.**
- `ResetTurnosRecuperacionVida()`: **Reinicia el contador de turnos desde la última recuperación de vida.**
- `ShowInfo()`: **Muestra la información detallada del mago, incluyendo su nombre, puntos de vida, nivel, puntos de habilidad, la cantidad de turnos desde la última habilidad usada, y su habilidad de recuperación de vida.**

Funcionalidad:

La clase `Mago` está diseñada para ser un personaje con un fuerte enfoque en el ataque y la regeneración. Tiene un bono de ataque fijo (+4) en cada tirada, lo que le otorga una ventaja ofensiva. Además, su habilidad especial de recuperar vida cada tres turnos lo hace más resiliente en batallas prolongadas. Esta clase sigue la estructura de `Jugador` y hereda sus métodos, pero con capacidades adicionales que la hacen única para enfrentar los desafíos del juego.

4.4 Clase Arquero (Arquero.cs)

Diagrama UML:



La clase `Arquero` extiende la clase `Jugador` y representa a un personaje especializado en ataques precisos y la capacidad de realizar ataques dobles, lo que lo convierte en un personaje ágil y peligroso en combate. Este personaje cuenta con un ataque mejorado y una habilidad única que puede activarse tras un intervalo de turnos.

Atributos:

- **`turnosDesdeAtaqueDoble`: Un contador que lleva el registro de los turnos transcurridos desde el último ataque doble. Es fundamental para determinar si la habilidad especial está disponible.**
- **`Tipo`: Define el tipo de personaje como "Arquero".**
- **`puntosTotales`: Representa los puntos acumulados durante el juego, que reflejan el progreso del personaje.**
- **`descripcion`: Una descripción que resalta las características iniciales del arquero, como su vida inicial y su habilidad para realizar ataques dobles.**

Métodos:

- **`Atacar()`: Realiza un ataque estándar con un bono de +3 y un adicional basado en los puntos de habilidad. Este método sobrescribe la versión base del método en `Jugador`.**
- **`AtacarDoble()`: Habilidad especial que permite realizar dos tiradas consecutivas y sumarlas al daño total. Solo puede usarse una vez cada tres turnos. Si la habilidad no está disponible, muestra un mensaje indicando el tiempo restante para su activación.**

- `IncrementarTurnosDesdeAtaqueDoble()`: Incrementa el contador de turnos desde el último ataque doble, permitiendo llevar el control para determinar si la habilidad especial está disponible.
- `ResetTurnosDesdeAtaqueDoble()`: Reinicia el contador de turnos una vez que el ataque doble ha sido ejecutado.
- `SubirNivel()`: Incrementa el nivel del arquero, mejorando sus atributos y habilidades. Utiliza el método base de `Jugador`.
- `SubirPuntosDeHabilidad()`: Permite aumentar los puntos de habilidad del arquero para mejorar sus capacidades de combate.
- `ShowInfo()`: Muestra información detallada del arquero, como sus puntos de vida, nivel, turnos desde el último ataque doble, enemigos derrotados, y una descripción general.

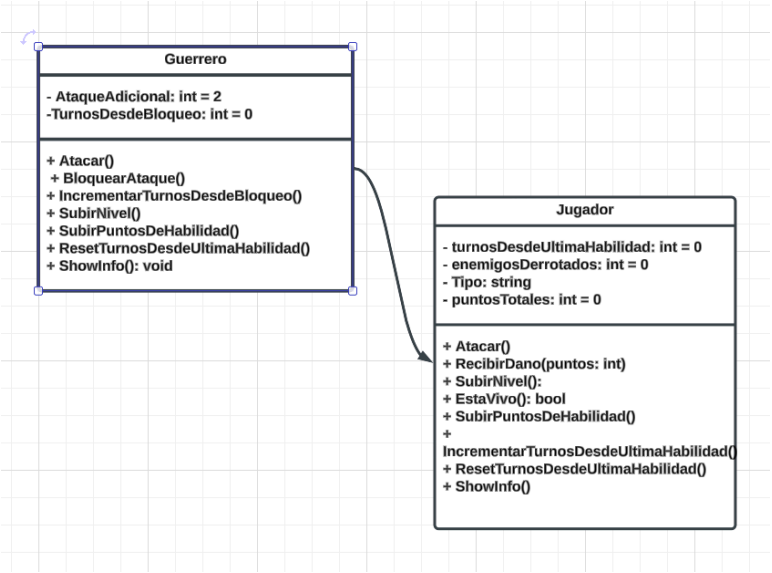
Funcionalidad:

La clase `Arquero` combina precisión y versatilidad en combate. Su ataque estándar incluye un bono fijo de +3, y su habilidad especial de ataque doble le permite realizar dos tiradas consecutivas, aumentando significativamente el daño potencial. Sin embargo, esta habilidad requiere un intervalo de tres turnos para recargarse, lo que equilibra su poder.

El `Arquero` es ideal para jugadores que buscan un personaje ágil y estratégico, capaz de infligir grandes cantidades de daño en momentos clave mientras mantiene un equilibrio entre ataque y defensa.

4.5 Clase Guerrero (Guerreo.cs)

Diagrama UML:



La clase `Guerrero` extiende la clase base `Jugador` y representa un personaje robusto y enfocado en el combate cuerpo a cuerpo. Este personaje combina ataques constantes con la capacidad de bloquear ataques enemigos periódicamente, lo que lo hace ideal para una estrategia de resistencia y daño constante.

Atributos:

- **`AtaqueAdicional`: Proporciona un bono fijo de +2 puntos en cada tirada de ataque, lo que refuerza su capacidad ofensiva.**
- **`TurnosDesdeBloqueo`: Lleva el conteo de los turnos transcurridos desde el último uso de la habilidad especial de bloqueo. Permite determinar cuándo esta habilidad está disponible.**
- **`Tipo`: Define al personaje como un "Guerrero".**
- **`descripcion`: Detalla las características iniciales del Guerrero, como su vida alta, el bono en ataques y la habilidad para bloquear.**

Métodos:

- **`Atacar()`: Sobrescribe el método base para incluir el bono de ataque adicional de +2 y un bono basado en los puntos de habilidad. Realiza una tirada de dos dados y aplica los bonos correspondientes para calcular el daño infligido.**
- **`BloquearAtaque()`: Habilidad especial que permite bloquear un ataque enemigo cada tres turnos. Si la habilidad está disponible, resetea el contador y bloquea el ataque. De lo contrario, informa al jugador cuántos turnos faltan para que esté disponible nuevamente.**
- **`IncrementarTurnosDesdeBloqueo()`: Incrementa el contador de turnos transcurridos desde el último bloqueo, ayudando a gestionar la disponibilidad de la habilidad especial.**
- **`SubirNivel()`: Incrementa el nivel del Guerrero utilizando el método base, mejorando sus atributos y habilidades generales.**
- **`SubirPuntosDeHabilidad()`: Aumenta los puntos de habilidad del personaje, lo que mejora su desempeño en combate.**
- **`ResetTurnosDesdeUltimaHabilidad()`: Reinicia el contador de habilidades especiales al estado inicial.**
- **`ShowInfo()`: Presenta una descripción completa del Guerrero, incluyendo su vida, nivel, habilidades, turnos desde el último bloqueo, enemigos derrotados, y otros atributos clave.**

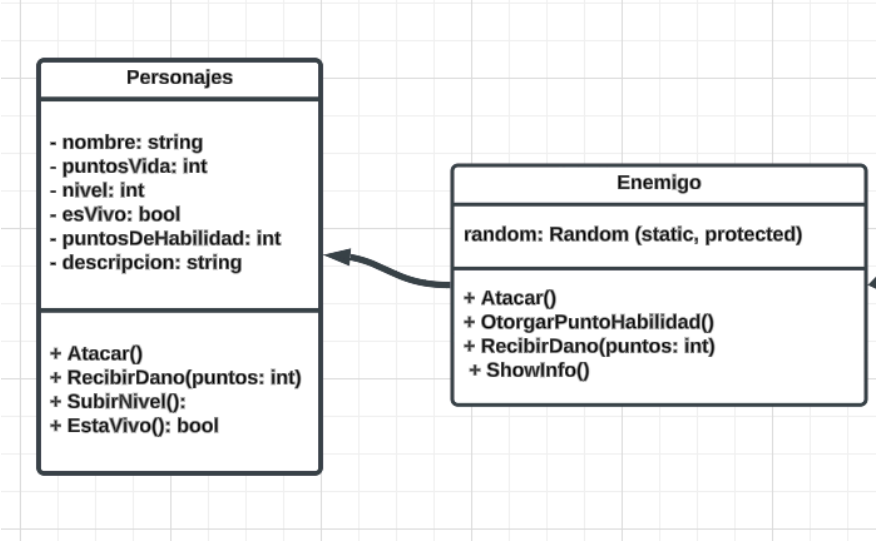
Funcionalidad:

El Guerrero está diseñado para ser un personaje versátil que combina ataque constante y defensa estratégica. Su habilidad de bloqueo le permite mitigar el daño recibido cada tres turnos, mientras que su ataque estándar siempre recibe un bono de +2, aumentando su efectividad en combate.

Este personaje es ideal para jugadores que buscan una experiencia equilibrada entre ofensiva y defensa, sobresaliendo en enfrentamientos prolongados y en escenarios donde resistir ataques es clave para la victoria.

4.6 Clase Enemigo (Enemigo.cs)

Diagrama UML:



La clase `Enemigo` extiende la clase base `Personajes` y modela a los adversarios que el jugador enfrentará en el juego. Se caracteriza por la aleatoriedad en sus atributos y habilidades, lo que agrega variabilidad y desafío a cada encuentro.

Atributos:

- **`random`: Una instancia de la clase `Random` utilizada para generar valores aleatorios para atributos como vida, nivel y tiradas de ataque.**
- **`puntosDeHabilidad`: Se inicializa en 0, ya que los enemigos no poseen habilidades especiales propias.**

Constructor:

- **Inicializa un enemigo con:**
  - **`vida`: Un valor aleatorio entre 5 y 12, representando la salud del enemigo.**
  - **`nivel`: Un valor aleatorio entre 0 y 15, definiendo su dificultad.**
  - **`esVivo`: Siempre comienza como `true`, indicando que el enemigo está vivo al inicio.**
  - **`puntosDeHabilidad`: Fijado en 0, ya que los enemigos no cuentan con un ataque base fijo ni habilidades avanzadas.**
- **Aleatoriedad: Los valores generados aleatoriamente aseguran que cada enemigo sea único, aumentando la rejugabilidad.**

Métodos:

1. **`Atacar()`:**

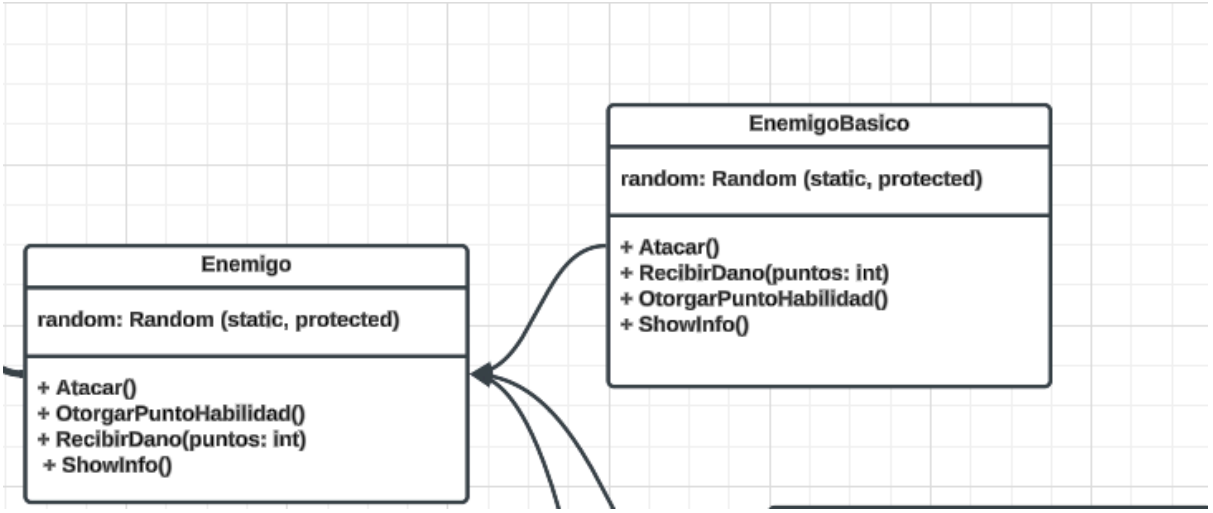
    - **Realiza una tirada de dos dados de 6 caras para determinar el daño infligido por el enemigo.**
    - **El resultado no tiene modificadores adicionales, lo que simplifica el cálculo del ataque.**
2. **`RecibirDano(int puntos)`:**
    - **Reduce los puntos de vida del enemigo según el daño recibido.**
    - **Si los puntos de vida llegan a 0 o menos, el enemigo muere (`esVivo = false`) y su vida se ajusta a 0 para evitar valores negativos.**
3. **`OtorgarPuntoHabilidad()`:**
    - **Simula una probabilidad del 50% para otorgar un punto de habilidad al jugador que derrota al enemigo.**
    - **Se utiliza un valor aleatorio para decidir si se otorga el punto.**
4. **`ShowInfo()`:**
    - **Presenta información sobre el enemigo, incluyendo su nombre, nivel, puntos de vida, si está vivo, y otros detalles relevantes.**
    - **Utiliza `DecoracionUtils` para mostrar los datos en un formato estilizado.**

Funcionalidad:

- **Aleatoriedad y Simplicidad**: Los enemigos son generados con atributos aleatorios, lo que ofrece variedad en los encuentros. Aunque sus ataques y habilidades son básicos, su diseño complementa la jugabilidad proporcionando desafíos variables al jugador.
- **Interacción con el Jugador**: La habilidad para otorgar puntos de habilidad introduce un elemento de recompensa adicional, incentivando la derrota de enemigos.
- **Extensibilidad**: Esta clase puede ser fácilmente extendida para crear enemigos más complejos con habilidades especiales o patrones de comportamiento únicos.

4.7 Clase EnemigoBasico (EnemigoBasico .cs)

Diagrama UML:



La clase `EnemigoBasico` extiende la clase `Enemigo` y representa un tipo de enemigo estándar y simple en el juego. Esta implementación utiliza principalmente la funcionalidad predeterminada de la clase padre, sin agregar comportamientos o atributos nuevos.

Constructor:

- **`EnemigoBasico(string nombre)`:**
  - **Invoca al constructor de la clase `Enemigo` y hereda su lógica.**
  - **Los atributos de puntos de vida (entre 5 y 12), nivel (entre 0 y 15) y estado inicial (`esVivo = true`) se generan aleatoriamente como en la clase base.**
  - **Razonamiento: Mantiene la simplicidad y consistencia en la generación de enemigos básicos.**

Métodos Sobrescritos:

1. **`Atacar()`:**
   - **Llama al método `Atacar()` de la clase base `Enemigo`.**
   - **Funcionalidad: Realiza una tirada de dos dados para determinar el daño infligido, sin modificar el comportamiento estándar.**
2. **`RecibirDano(int puntos)`:**
   - **Llama al método `RecibirDano()` de la clase base.**

- **Funcionalidad: Reduce los puntos de vida del enemigo básico según el daño recibido y verifica si sigue vivo.**

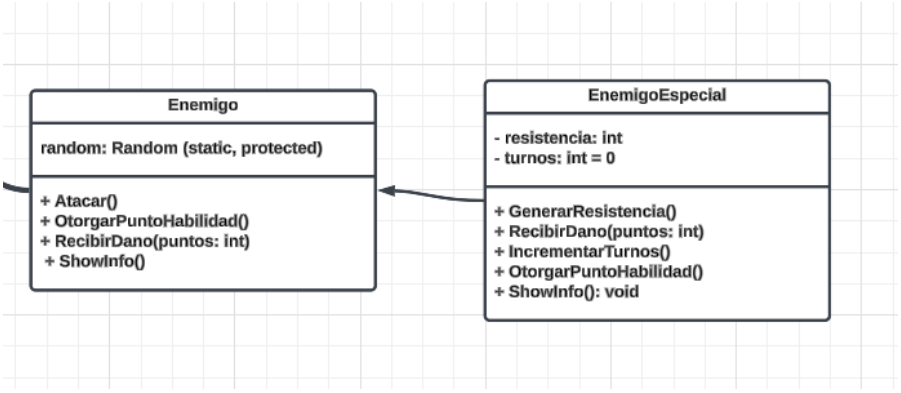
3. **`OtorgarPuntoHabilidad()`:**
   - **Llama al método de la clase base `Enemigo`.**
   - **Funcionalidad: Simula una probabilidad del 50% para otorgar un punto de habilidad al jugador que lo derrota.**
4. **`ShowInfo()`:**
   - **Sobrescribe el método `ShowInfo()` de la clase base para personalizar la presentación de información del enemigo básico.**
   - **Diferencia principal: Cambia el título para especificar que es un "Enemigo Básico" y mantiene el resto de los datos estándar (nombre, puntos de vida, nivel, estado vivo, etc.).**

Funcionalidad:

- **`Simpleza y Estándar:`**
  - **Esta clase es una implementación directa de la clase `Enemigo`, diseñada para representar enemigos comunes sin habilidades ni características avanzadas.**
- **`Uso Principal:`**
  - **Los enemigos básicos son ideales para encuentros iniciales o desafíos estándar en el juego, proporcionando oponentes que no requieren estrategias complejas.**
- **`Extensibilidad:`**
  - **Puede servir como punto de partida para crear enemigos más específicos o con habilidades únicas, sobrescribiendo los métodos según sea necesario.**

4.8 Clase EnemigoEspecial (EnemigoEspecial.cs)

Diagrama UML:



La clase `EnemigoEspecial` extiende la funcionalidad de la clase `Enemigo` para representar un enemigo más avanzado con características únicas, como resistencia al daño y habilidades progresivas.

Constructor:

- **`EnemigoEspecial(string nombre)`:**
  - **Llama al constructor de la clase base `Enemigo`.**
  - **Atributos iniciales como puntos de vida, nivel y ataque base se generan aleatoriamente como en la clase base.**
  - **Se agrega un atributo exclusivo, `resistencia`, que se inicializa utilizando el método `GenerarResistencia()`.**
  - **Razonamiento: Proporciona al enemigo una resistencia adicional para reducir el daño recibido, haciéndolo más difícil de derrotar.**

Atributos Específicos:

1. **`resistencia`:**
   - **Representa la capacidad del enemigo para reducir el daño recibido en cada ataque.**
   - **Se inicializa con un valor aleatorio entre 0 y 5 usando el método `GenerarResistencia()`.**
   - **La resistencia disminuye progresivamente cada turno para simular el desgaste del enemigo.**
2. **`turnos`:**
   - **Cuenta los turnos que el enemigo ha estado en combate.**
   - **Útil para manejar efectos basados en el tiempo, como reducir la resistencia.**

Métodos Sobrescritos:

1. **`RecibirDano(int puntos)`:**
   - **Modifica el comportamiento del método base:**
     - **Reduce el daño recibido usando la resistencia del enemigo.**
     - **Garantiza que el daño no sea negativo usando `Math.Max`.**
     - **Llama al método base para aplicar el daño resultante y actualizar los puntos de vida.**
   - **La resistencia disminuye en 1 punto por cada turno, simulando el desgaste durante el combate.**
   - **Funcionalidad: Hace al enemigo más resistente al principio del combate, pero se debilita con el tiempo.**
2. **`OtorgarPuntoHabilidad()`:**
   - **Llama al método de la clase base para mantener la probabilidad del 50% de otorgar puntos de habilidad al jugador.**

3. `ShowInfo()`:
    - **Sobrescribe el método de la clase base para mostrar atributos específicos del enemigo especial, como la resistencia.**

Métodos Propios:

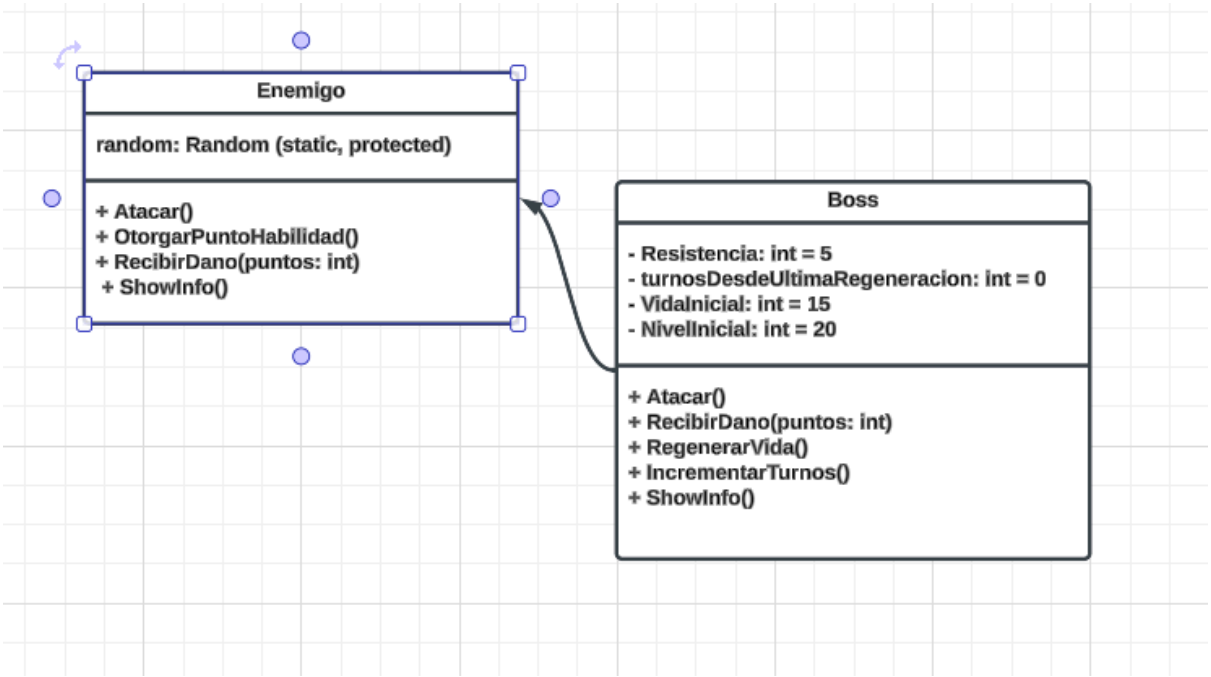
1. `GenerarResistencia()`:
    - **Genera un valor aleatorio entre 0 y 5 para inicializar la resistencia del enemigo.**
    - **Razonamiento: Proporciona variedad en las estadísticas de resistencia para cada instancia de enemigo especial.**
2. `IncrementarTurnos()`:
    - **Incrementa el contador de turnos.**
    - **Puede utilizarse para manejar otros comportamientos basados en el tiempo en el futuro.**

Funcionalidad:

- **Defensivo y Progresivo:**
    - **El atributo `resistencia` agrega un componente estratégico, haciendo al enemigo más difícil de derrotar al inicio del combate.**
    - **La resistencia decrece con el tiempo, equilibrando su dificultad.**
- **Uso Principal:**
    - **Diseñado para encuentros más desafiantes, donde el jugador debe gestionar recursos y estrategias para vencerlo.**
- **Extensibilidad:**
    - **Puede servir como base para enemigos aún más avanzados, con habilidades adicionales o resistencias específicas.**

4.9 Clase Boss. (EnemigoBoss.cs)

Diagrama UML:



La clase `Boss` es una extensión de la clase `Enemigo`, diseñada para representar un enemigo final o jefe. Incluye habilidades especiales, estadísticas más altas y mecánicas avanzadas que lo hacen más desafiante para el jugador.

Constructor:

- **`Boss(string nombre)`:**
  - **Llama al constructor de la clase `Enemigo` con valores específicos:**
    - **Vida inicial fija: 15 puntos.**
    - **Nivel inicial fijo: 20.**
  - **Establece propiedades exclusivas:**
    - **`VidaInicial`: Para mantener un límite en la regeneración.**
    - **`NivelInicial`: Para garantizar que siempre sea un enemigo de alto nivel.**

Atributos Específicos:

1. **`Resistencia`:**
   - **Fija en 5 puntos.**
   - **Reduce el daño recibido en cada ataque, lo que lo hace más duradero.**
2. **`turnosDesdeUltimaRegeneracion`:**
   - **Lleva un conteo de turnos desde la última vez que regeneró vida.**
   - **Habilidad única para aumentar la longevidad del `Boss`.**

3. `VidaInicial` y `NivelInicial`:
    - **Valores constantes para garantizar que el jefe tenga estadísticas estables y avanzadas.**

Métodos Sobrescritos:

1. `Atacar()`:
    - **Realiza un doble ataque, lanzando dos tiradas de dos dados de 6 caras cada una.**
    - **Suma ambas tiradas para calcular el daño total.**
    - **Funcionalidad: Representa una ofensiva poderosa, característica distintiva de los jefes.**
2. `RecibirDano(int puntos)`:
    - **Modifica el daño recibido al aplicar resistencia.**
    - **Usa `Math.Max` para asegurarse de que el daño nunca sea negativo.**
    - **Llama al método base para actualizar los puntos de vida.**

Métodos Propios:

1. `RegenerarVida()`:
    - **Recupera 2 puntos de vida si han pasado al menos 3 turnos desde la última regeneración.**
    - **Reinicia el contador de turnos tras la regeneración.**
    - **Garantiza que los puntos de vida no excedan el valor inicial (máximo permitido).**
2. `IncrementarTurnos()`:
    - **Incrementa el contador de turnos desde la última regeneración.**
    - **Facilita el manejo de habilidades basadas en el tiempo.**

Métodos de Información:

1. `ShowInfo()`:
    - **Sobrescribe el método de la clase base para mostrar:**
        - **Resistencia.**
        - **Contador de turnos desde la última regeneración.**
        - **Otros atributos relevantes como vida, nivel, y estado (si está vivo).**

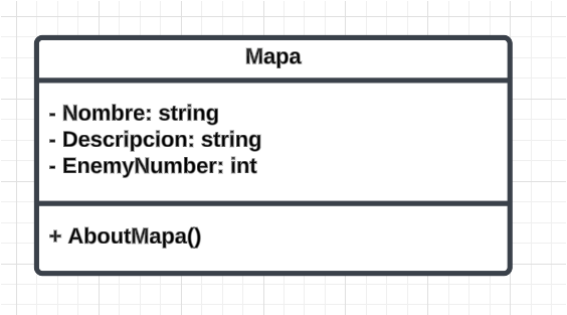
Funcionalidad:

- **Resistencia y Regeneración:**
    - **Resistencia: Mitiga el daño recibido, haciéndolo más difícil de derrotar rápidamente.**

    - **Regeneración: Le permite recuperar vida periódicamente, prolongando los combates.**
- **Doble Ataque:**
    - **Permite infligir un daño significativamente mayor en un solo turno.**
- **Rol en el Juego:**
    - **Diseñado como el enemigo más desafiante.**
    - **Combina mecánicas defensivas y ofensivas avanzadas.**
    - **Ideal como culminación de una batalla épica.**

5. Clases Adicionales:

5.1 Clase Mapa



La clase `Mapa` representa un escenario o lugar dentro del juego, proporcionando detalles básicos sobre su nombre, descripción y cantidad de enemigos presentes.

Atributos:

1. **`Nombre`:**
    - **Define el nombre del mapa.**
    - **Representa cómo se identificará el escenario dentro del juego.**
2. **`Descripcion`:**
    - **Proporciona información narrativa o temática sobre el mapa.**
3. **`EnemyNumber`:**
    - **Indica la cantidad de enemigos presentes en el mapa, estableciendo su nivel de dificultad.**

Constructor:

- **`Mapa(string nombre, string descripcion, int Enemies)`:**
    - **Inicializa los atributos con valores proporcionados por el usuario.**
    - **Define el contexto del mapa (nombre, descripción y enemigos).**

Método Principal:

- `AboutMapa()`:
  - **Presenta la información del mapa de forma estilizada:**
    - **Nombre.**
    - **Descripción.**
    - **Número de enemigos.**
  - **Ideal para mostrar al jugador un resumen del lugar antes de entrar o explorarlo.**

Uso en el Juego:

- **Sirve como una clase base para definir diferentes tipos de mapas o escenarios.**
- **Posibilita la personalización de cada área del juego mediante atributos y descripciones específicas.**

6. Funciones Principales

6.1 Play:(Jugar.cs)

Métodos de la Clase `Play`

1. `JugarPerimaraMapa(Jugador jugador, List<string> frasesMotivadoras)`

   - **Propósito**: Este método gestiona el primer mapa del juego. El jugador se enfrenta a varios enemigos básicos y debe superarlos para avanzar al siguiente mapa.

- **Importancia**: Es la primera interacción significativa del jugador con el mundo del juego. Aquí, el jugador empieza a experimentar los desafíos y las mecánicas de combate. Si el jugador pierde en este mapa, el juego termina.

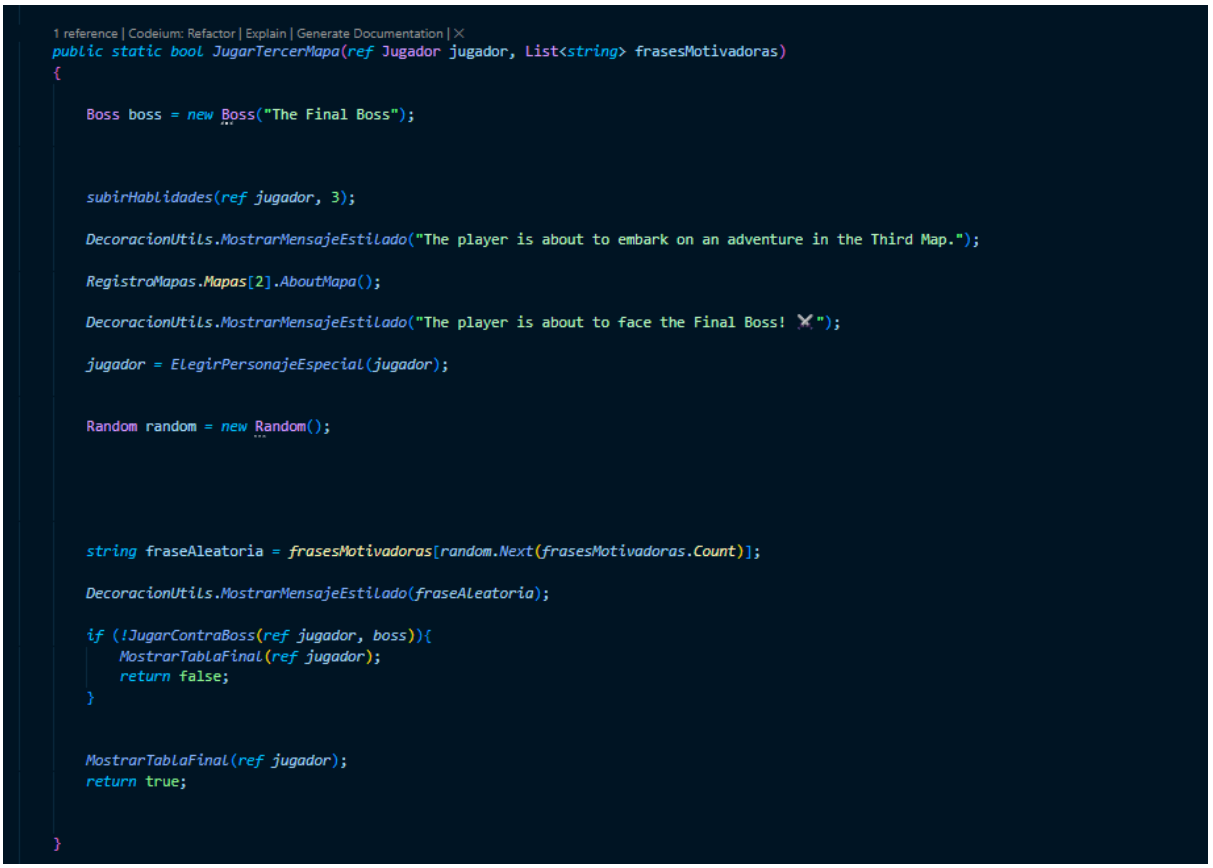
2. **`JugarContraEnemigoBasico(Jugador jugador, EnemigoBasico enemigoBasico)`**
   - **Propósito**: Gestiona el combate contra un enemigo básico. El jugador lucha contra un enemigo específico, y el combate se resuelve con turnos. El jugador puede ganar o perder, y dependiendo del resultado, el juego puede continuar o terminar.
   - **Importancia**: Este método es central para el progreso del jugador a lo largo de cada mapa. Dependiendo del tipo de jugador (por ejemplo, un guerrero, arquero, o mago), las mecánicas de combate pueden variar, pero siempre se basa en la misma estructura de turno.
3. **`JugarSegundoMapa(Jugador jugador, List<string> frasesMotivadoras)`**

```
1 reference | Codeium: Refactor | Explain | Generate Documentation | ×
public static bool JugarSegundoMapa(ref Jugador jugador, List<string> frasesMotivadoras)
{
    subirHablidades(ref jugador, 2); // Subir habilidades del jugador

    List<EnemigoEspecial> enemigos = new List<EnemigoEspecial>();

    enemigos.Add(new EnemigoEspecial("Orco Juvenil"));
    enemigos.Add(new EnemigoEspecial("Zombie Despistado"));
    enemigos.Add(new EnemigoEspecial("Murciélago Nocturno"));
    enemigos.Add(new EnemigoEspecial("Esqueleto Frágil"));
    enemigos.Add(new EnemigoEspecial("Araña Venenosa"));

    Random random = new Random();
    DecoracionUtils.MostrarMensajeEstilado("The player is about to embark on an adventure in the Second Map.");

    RegistroMapas.Mapas[1].AboutMapa();

    foreach (EnemigoEspecial EnemigoEspecial in enemigos)
    {
        string fraseAleatoria = frasesMotivadoras[random.Next(frasesMotivadoras.Count)];
        DecoracionUtils.MostrarMensajeEstilado(fraseAleatoria);
        if (!JugarContraEnemigoEspecial(ref jugador, EnemigoEspecial))
        {
            MostrarTablaFinal(ref jugador);
            return false;
        }
        if (jugador.nivel % 3 == 0)
        {
            jugador = ElegirPersonajeEspecial(jugador);
        }


    }

    MostrarTablaFinal(ref jugador);

    return true;

}
```

   - **Propósito**: Similar a `JugarPerimaraMapa`, este método gestiona el segundo mapa del juego. El jugador enfrentará a nuevos enemigos y desafíos.
   - **Importancia**: Mantiene el flujo del juego después de que el jugador ha superado el primer mapa. Aquí, los desafíos podrían aumentar en dificultad, y el jugador podría enfrentarse a nuevas mecánicas o enemigos más fuertes.
4. **`JugarContraEnemigoEspecial(Jugador jugador, EnemigoBasico enemigoEspecial)`**
   - **Propósito**: Este método maneja combates contra enemigos especiales, que son más poderosos o presentan un reto único para el jugador.

- **Importancia**: Agrega variedad y complejidad al juego. Los enemigos especiales pueden tener habilidades únicas o mecánicas de combate más avanzadas, lo que hace que el jugador deba adaptar su estrategia.

5. `JugarTercerMapa(Jugador jugador, List<string> frasesMotivadoras)`



```
1 reference | Codeium: Refactor | Explain | Generate Documentation |
public static bool JugarTercerMapa(ref Jugador jugador, List<string> frasesMotivadoras)
{

    Boss boss = new Boss("The Final Boss");



    subirHablidades(ref jugador, 3);

    DecoracionUtils.MostrarMensajeEstilado("The player is about to embark on an adventure in the Third Map.");

    RegistroMapas.Mapas[2].AboutMapa();

    DecoracionUtils.MostrarMensajeEstilado("The player is about to face the Final Boss! ⚔️");

    jugador = ElegirPersonajeEspecial(jugador);


    Random random = new Random();




    string fraseAleatoria = frasesMotivadoras[random.Next(frasesMotivadoras.Count)];

    DecoracionUtils.MostrarMensajeEstilado(fraseAleatoria);

    if (!JugarContraBoss(ref jugador, boss)){
        MostrarTablaFinal(ref jugador);
        return false;
    }


    MostrarTablaFinal(ref jugador);
    return true;


}
```

- **Propósito**: Este método gestiona el tercer y último mapa del juego, donde el jugador enfrenta desafíos finales antes de llegar al jefe final.
- **Importancia**: Es el penúltimo paso en la aventura del jugador. El tercer mapa puede representar el aumento máximo en la dificultad del juego, preparando al jugador para el enfrentamiento final.

6. `JugarContraBoss(Jugador jugador, EnemigoBasico boss)`
   - **Propósito**: Este método organiza el enfrentamiento del jugador contra el jefe final del juego, que es mucho más fuerte que los enemigos anteriores.
   - **Importancia**: El combate contra el jefe final es el clímax del juego. El jugador debe aplicar todo lo aprendido a lo largo de la aventura, y la victoria sobre el jefe final generalmente marca el final del juego.
7. `SubirHabilidades(Jugador jugador)`

```
2 references | Codeium: Refactor | Explain | Generate Documentation | ×
public static void subirHabilidades(ref Jugador jugador, int mapaActual)
{
    //Si el jugador ha llegado a la mapa 3, subir habilidades y el nivel
    if (mapaActual == 2)
    {
        jugador.puntosDeHabilidad = jugador.puntosDeHabilidad + 2; // Suma 2 puntos de habilidad
        jugador.nivel = jugador.nivel + 3; // Suma 3 niveles
        jugador.puntosVida = jugador.puntosVida + 5; // Suma 5 puntos de vida

    }

    if (mapaActual == 3)
    {
        jugador.puntosDeHabilidad = jugador.puntosDeHabilidad + 3; // Suma 3 puntos de habilidad
        jugador.nivel = jugador.nivel + 4; // Suma 4 niveles
        jugador.puntosVida = jugador.puntosVida + 7; // Suma 7 puntos de vida
    }
}
```

- **Propósito**: Este método maneja la mejora de las habilidades del jugador, permitiéndole incrementar sus estadísticas y habilidades a medida que avanza en el juego.
- **Importancia**: Es crucial para el progreso del jugador, ya que permite mejorar sus capacidades y enfrentar desafíos cada vez mayores. Es un componente central del sistema de RPG del juego.

8. `EventoSorpresa(Jugador jugador)`

```
4 references | Codeium: Refactor | Explain | Generate Documentation | ×
static void EventoSorpresa(ref Jugador jugador)
{
    Random random = new Random();
    int evento = random.Next(1, 4); // Generar un número aleatorio entre 1 y 3

    switch (evento)
    {
        case 1: // Trampa
            DecoracionUtils.MostrarMensajeEstilado("💥 You've fallen into a trap! You lose 3 health points.");
            if(jugador.puntosVida <= 3){
                jugador.puntosVida = 4; //las trampas no deben matar al jugador
            }
            jugador.RecibirDano(3);
            break;
        case 2: // Cofre
            DecoracionUtils.MostrarMensajeEstilado("🎁 You've found a chest! You gain 10 health points.");
            jugador.puntosVida = jugador.puntosVida + 10;

            break;
        case 3: // Evento positivo
            DecoracionUtils.MostrarMensajeEstilado("✨ A benevolent spirit grants you an extra skill point.");
            jugador.puntosDeHabilidad = jugador.puntosDeHabilidad + 1;
            break;
    }
}
```

- **Propósito**: Este método activa eventos sorpresa que pueden ocurrir durante el juego, añadiendo elementos aleatorios y situaciones inesperadas.
- **Importancia**: Los eventos sorpresa mantienen el juego dinámico y emocionante. Pueden ser positivos o negativos, como encontrar un objeto valioso o perder puntos de salud, lo que introduce incertidumbre en el juego.

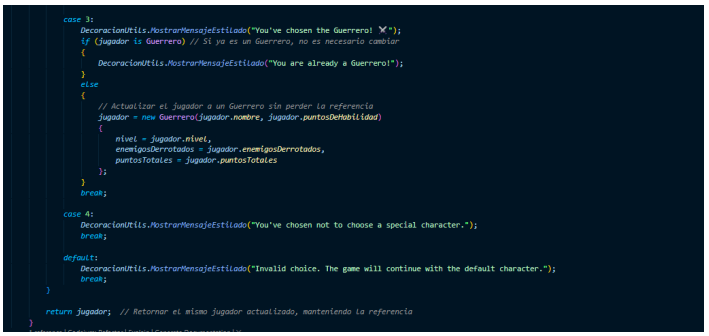
9. **`MostrarTablaFinal(Jugador jugador)`**

```
6 references | Codeium: Refactor | Explain | Generate Documentation | ×
public static void MostrarTablaFinal(ref Jugador jugador)
{
    // Decorar la tabla final con bordes y formato estilizado
    string titulo = "🏆 Final Scoreboard 🏆";
    string[] contenido = {
    $"👤 Player: {jugador.nombre}",
    $"🔢 Level: {jugador.nivel}",
    $"👾 Enemies Defeated: {jugador.enemigosDerrotados}",
    $"🏅 Total Points: {jugador.puntosTotales}",  // Mostrar puntos totales
};

    // Llamamos al método de decoración para mostrar la tabla estilizada
    DecoracionUtils.MostrarConBordes(titulo, contenido);
}


}
```

- **Propósito**: Muestra la tabla final con estadísticas del jugador, como su puntuación, nivel y habilidades.
- **Importancia**: Al final del juego, se proporciona una retroalimentación visual del progreso del jugador, lo cual es importante para la satisfacción del jugador y para motivar futuras partidas.

10. **`ElegirPersonajeEspecial(Jugador jugador)`**

```
3 references | Codeium: Refactor | Explain | Generate Documentation | ×
public static Jugador ElegirPersonajeEspecial(Jugador jugador)
{


    // Mostrar opciones de personajes especiales
    DecoracionUtils.MostrarMensajeEstilado("Choose your new character: 👑");
    DecoracionUtils.MostrarMensajeEstilado("1. Mago");
    DecoracionUtils.MostrarMensajeEstilado("2. Arquero");
    DecoracionUtils.MostrarMensajeEstilado("3. Guerrero");
    DecoracionUtils.MostrarMensajeEstilado("4. No Special Character");

    int opcionPersonaje = Convert.ToInt32(Console.ReadLine());

    // Asignar el personaje especial según la opción del jugador
    switch (opcionPersonaje)
    {
        case 1:
            DecoracionUtils.MostrarMensajeEstilado("You've chosen the Mago! 🧙‍♂️");
            if (jugador is Mago) // Si ya es un Mago, no es necesario cambiar
            {
                DecoracionUtils.MostrarMensajeEstilado("You are already a Mago!");
            }
            else
            {
                // Actualizar el jugador a un Mago sin perder la referencia
                jugador = new Mago(jugador.nombre, jugador.puntosDeHabilidad)
                {
                    nivel = jugador.nivel,
                    enemigosDerrotados = jugador.enemigosDerrotados,
                    puntosTotales = jugador.puntosTotales
                };
            }
            break;

        case 2:
            DecoracionUtils.MostrarMensajeEstilado("You've chosen the Arquero! 🏹");
            if (jugador is Arquero) // Si ya es un Arquero, no es necesario cambiar
            {
                DecoracionUtils.MostrarMensajeEstilado("You are already an Arquero!");
            }
            else
            {
                // Actualizar el jugador a un Arquero sin perder la referencia
                jugador = new Arquero(jugador.nombre, jugador.puntosDeHabilidad)
                {
                    nivel = jugador.nivel,
                    enemigosDerrotados = jugador.enemigosDerrotados,
                    puntosTotales = jugador.puntosTotales

                };
            }
            break;

        case 3:
```



```
        case 3:
            DecoracionUtils.MostrarMensajeEstilado("You've chosen the Guerrero! ⚔️");
            if (jugador is Guerrero) // Si ya es un Guerrero, no es necesario cambiar
            {
                DecoracionUtils.MostrarMensajeEstilado("You are already a Guerrero!");
            }
            else
            {
                // Actualizar el jugador a un Guerrero sin perder la referencia
                jugador = new Guerrero(jugador.nombre, jugador.puntosDeHabilidad)
                {
                    nivel = jugador.nivel,
                    enemigosDerrotados = jugador.enemigosDerrotados,
                    puntosTotales = jugador.puntosTotales
                };
            }
            break;

        case 4:
            DecoracionUtils.MostrarMensajeEstilado("You've chosen not to choose a special character.");
            break;

        default:
            DecoracionUtils.MostrarMensajeEstilado("Invalid choice. The game will continue with the default character.");
            break;
    }

    return jugador;  // Retornar el mismo jugador actualizado, manteniendo la referencia
}
```

- **Propósito**: Permite al jugador elegir un personaje especial o mejorar su personaje actual, lo que puede ser una ventaja estratégica durante el juego.
- **Importancia**: Esta función ofrece al jugador la posibilidad de personalizar su experiencia de juego, eligiendo entre diferentes personajes con habilidades especiales. Esto puede afectar cómo el jugador aborda los combates y las decisiones del juego.

7. Funciones Adicionales

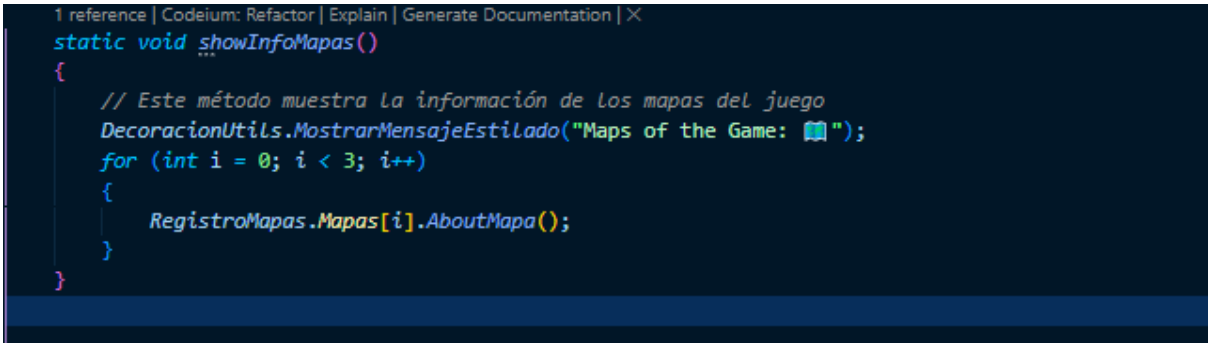
1. `showInfoPersonajesEspeciales()`

```
1 reference | Codeium: Refactor | Explain | Generate Documentation | ×
static void showInfoPersonajesEspeciales()
{
    // Este método muestra la información de los personajes especiales del juego (Mago, Arquero y Guerrero)
    DecoracionUtils.MostrarMensajeEstilado("Special Characters of the Game:");
    for (int i = 0; i < 3; i++)
    {
        PersonajesEpeciales.Jugadores[i].ShowInfo();
    }
}
```

- **Descripción**: Muestra la información de los personajes especiales disponibles en el juego.
- **Funcionamiento**:
    - Usa la clase `PersonajesEpeciales` para acceder a un arreglo de personajes (como Mago, Arquero y Guerrero).
    - Llama al método `ShowInfo()` de cada personaje para mostrar sus características o habilidades específicas.
    - Utiliza `DecoracionUtils.MostrarMensajeEstilado()` para presentar los mensajes de forma atractiva y estilizada en la consola.

```
93 references | Codeium: Refactor | Explain
public static class DecoracionUtils
{
    // Método para mostrar texto con un efecto de escritura animada
    12 references | Codeium: Refactor | Explain | ×
    public static void MostrarConBordes(string titulo, string[] contenido)
    {
        string borde = new string('=', titulo.Length + 4);

        // Mostrar el título en rojo
        Console.ForegroundColor = ConsoleColor.Red;
        EscribirLinea($"╔{borde}╗");
        EscribirLinea($"║  {titulo}  ║");
        EscribirLinea($"╚{borde}╝");
        Console.ResetColor();

        // Mostrar el contenido en azul
        Console.ForegroundColor = ConsoleColor.Blue;
        foreach (var linea in contenido)
        {
            EscribirLinea($"  > {linea}");
        }
        Console.ResetColor();
        Console.WriteLine();
    }

    // Método auxiliar para escribir texto letra por letra con un retraso
    7 references | Codeium: Refactor | Explain | ×
    public static void EscribirLinea(string texto)
    {

        foreach (char c in texto)
        {
            Console.Write(c);
            Thread.Sleep(10); // Ajusta el tiempo de retraso en milisegundos
        }
        Console.WriteLine();
    }

    // Método para mostrar un mensaje destacado (ej: Start Game, End Game)
    81 references | Codeium: Refactor | Explain | ×
    public static void MostrarMensajeEstilado(string mensaje)
    {
        string borde = new string('*', mensaje.Length + 6);

        Console.ForegroundColor = ConsoleColor.Green;
        EscribirLinea($"***** {borde} *****");
        Console.ForegroundColor = ConsoleColor.Magenta;
        EscribirLinea($"***   {mensaje.ToUpper()}   ***");
        Console.ForegroundColor = ConsoleColor.Green;
        EscribirLinea($"***** {borde} *****");
        Console.ResetColor();

        Console.WriteLine();
    }
}
```

2. `showInfoMapas()`



- **Descripción**: Muestra la información sobre los mapas disponibles en el juego.
- **Funcionamiento**:
    - Utiliza la clase `RegistroMapas` para acceder a un arreglo de mapas.
    - Llama al método `AboutMapa()` de cada mapa para obtener una descripción del mapa correspondiente.
    - Al igual que en la función anterior, utiliza `DecoracionUtils.MostrarMensajeEstilado()` para mostrar los detalles de los mapas de manera estilizada.

3. `IniciarJuego()`

```
public static void IniciarJuego()
{
    List<string> frasesMotivadoras = new List<string>
{
    "⚔️ Heroes aren't born, they are forged in the heat of battle.",
    "🌟 The path of a warrior is paved with challenges and triumphs.",
    "🛡️ Stand firm, for the storm only makes you stronger.",
    "🔥 Your courage ignites the flames of hope in the darkest of times.",
    "⚡ Each battle won is a step closer to immortality.",
    "🏹 Precision and patience are the marks of a true archer.",
    "🔮 Magic flows through those who dare to dream and believe.",
    "💪 Strength is not just physical; it's the will to keep fighting.",
    "🧠 Strategy and wisdom conquer brute force every time.",
    "🚶 The journey of a thousand miles begins with a single step."
};
    // Guardar el tiempo de inicio
    DateTime tiempoInicio = DateTime.Now;

    // Crear al jugador
    Jugador Player = new Jugador("Player", 10, 1, true, 0, "Player");

    // Empezar el juego desde el Primer Mapa y verificar si el jugador sigue adelante
    if (!Play.JugarPerimaraMapa(ref Player, frasesMotivadoras))
    {

        MostrarTiempoTranscurrido(tiempoInicio);
        // Si el jugador pierde, finalizar el juego
        return;
    }

    // Mostrar el tiempo transcurrido después de terminar el primer mapa
    MostrarTiempoTranscurrido(tiempoInicio);

    // Continuar con el Segundo Mapa si el jugador ha superado el primero
    if (!Play.JugarSegundoMapa(
       ref Player, frasesMotivadoras))
    {

        MostrarTiempoTranscurrido(tiempoInicio);
        // Si el jugador pierde, finalizar el juego
        return;
    }

    // Mostrar el tiempo transcurrido después de terminar el segundo mapa
    MostrarTiempoTranscurrido(tiempoInicio);

    // Finalmente, jugar el Tercer Mapa si el jugador ha superado los dos anteriores
    if (!Play.JugarTercerMapa(ref Player, frasesMotivadoras))
    {
        MostrarTiempoTranscurrido(tiempoInicio);
        // Si el jugador pierde en el tercer mapa, finalizar el juego
        return;
    }

    // Mostrar el tiempo transcurrido después de terminar el tercer mapa
    MostrarTiempoTranscurrido(tiempoInicio);

    // Si el jugador supera todos los mapas, mostrar un mensaje de victoria
    DecoracionUtils.MostrarMensajeEstilado("🎉 Congratulations! You've completed your adventure and proven your worth!");

}
```

- **Descripción**: Gestiona la secuencia de eventos para iniciar y jugar el juego, desde el primer mapa hasta el tercero, y calcular el tiempo transcurrido.
- **Funcionamiento**:
    - Se inicializa una lista de frases motivadoras que se mostrarán al jugador durante el juego.
    - Se registra el tiempo de inicio del juego con `DateTime.Now` para calcular el tiempo transcurrido.
    - Se crea un objeto `Jugador` con los parámetros iniciales del jugador, como su nombre, nivel, salud, etc.
    - Luego, el juego avanza a través de tres mapas consecutivos:
        - **Primer mapa**: Llama a `Play.JugarPerimaraMapa()`. Si el jugador pierde en este mapa, el juego termina.

        - **Segundo mapa**: Si el jugador ha ganado el primer mapa, pasa al segundo con `Play.JugarSegundoMapa()`.
        - **Tercer mapa**: Si el jugador ha superado los dos primeros mapas, juega el tercer mapa con `Play.JugarTercerMapa()`.
    - Después de cada mapa, se calcula y muestra el tiempo transcurrido con la función `MostrarTiempoTranscurrido()`.
    - Si el jugador completa los tres mapas, se muestra un mensaje de victoria.

4. `MostrarTiempoTranscurrido(DateTime tiempoInicio)`

```
6 references | Codeium: Refactor | Explain | Generate Documentation | ✕
public static void MostrarTiempoTranscurrido(DateTime tiempoInicio)
{
    // Calcular el tiempo transcurrido
    TimeSpan tiempoTranscurrido = DateTime.Now - tiempoInicio;
    string tiempoFormateado = tiempoTranscurrido.ToString(@"hh\:mm");
    // Mostrar el tiempo transcurrido
    DecoracionUtils.MostrarMensajeEstilado($"Time elapsed so far: {tiempoFormateado} ⌛ ");

}
```

- **Descripción**: Calcula y muestra el tiempo transcurrido desde que el jugador comenzó el juego.
- **Funcionamiento**:
    - Usa `DateTime.Now - tiempoInicio` para obtener un `TimeSpan` que representa el tiempo transcurrido.
    - Convierte el `TimeSpan` en una cadena con el formato `hh:mm` (horas y minutos).
    - Muestra este tiempo en la consola utilizando `DecoracionUtils.MostrarMensajeEstilado()`.

5. `ReglasJuegoMenu (ReglasJuego.cs)`

1. `AboutGame()`

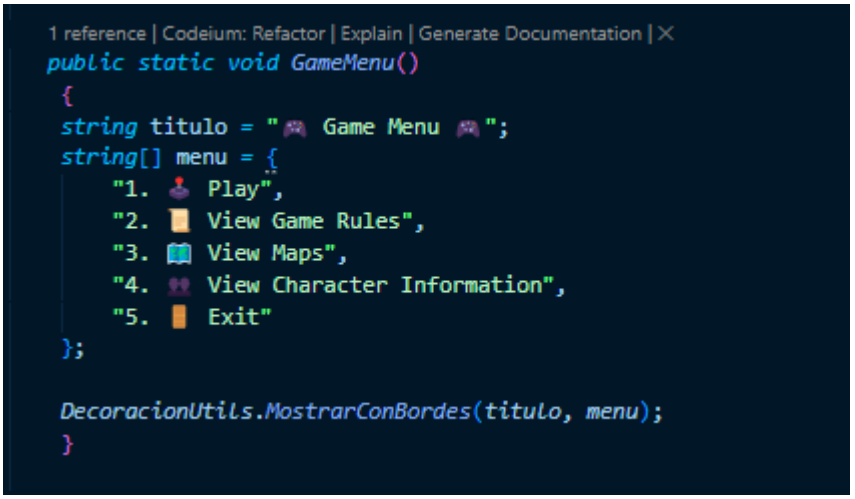
```
public static void AboutGame()
{
    string titulo = "📜 Game Rules 📜";
    string[] reglas = {
        "1. 🎲 The player starts by rolling two 6-sided dice to determine the damage dealt to the enemy.",
        "2. 🏹 The player's attack level is determined by the dice roll, plus skill points multiplied by 0.10.",
        "3. 💡 Skill points add to the dice roll result, making the player stronger as they accumulate more skill points.",
        "4. 🏆 For each enemy defeated, the player levels up and becomes stronger, preparing for tougher challenges.",
        "5. 💪 If the player has a higher level than the enemy, they win the battle automatically without needing to roll the dice.",
        "6. 🗺️ The game consists of three maps with the following structure:",
        "   - First Map: Fight against 6 basic enemies.",
        "   - Second Map: Battle 5 special enemies that are tougher and require strategy.",
        "   - Final Map: Defeat the Boss in the last map to complete the adventure.",
        "7. ⚡ If the player passes 3 rounds (one round for each map) without losing, they gain the ability to choose one of the 3 special characters: Wizard, Archer, or Warrior.",
        "8. 🎯 Defeating all enemies in each map, including the Boss in the final round, is the objective to complete the game."
    };

    DecoracionUtils.MostrarConBordes(titulo, reglas);
}
```

Esta función muestra las reglas principales del juego al jugador. La información se presenta en formato de lista, donde se explica paso a paso cómo funciona el juego. Algunas de las reglas clave incluyen:

- El jugador lanza dos dados para determinar el daño.
- Los puntos de habilidad mejoran el daño basado en los dados.
- Al derrotar enemigos, el jugador sube de nivel y mejora su personaje.
- El juego se desarrolla a través de tres mapas, cada uno con una serie de enemigos más fuertes.
- Después de pasar 3 rondas sin perder, el jugador puede elegir entre tres personajes especiales: Mago, Arquero o Guerrero.

La función `DecoracionUtils.MostrarConBordes()` se utiliza para mostrar las reglas con un formato estilizado.

2. `GameMenu()`

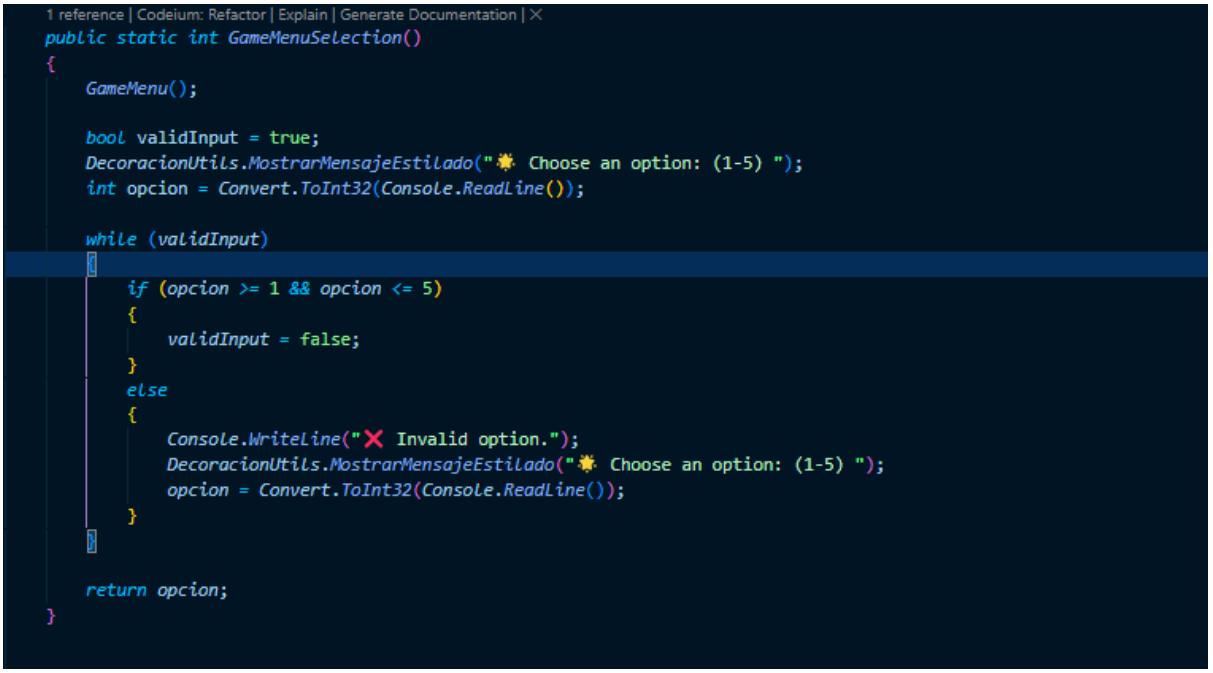


Esta función genera y muestra el menú principal del juego. El menú ofrece las siguientes opciones:

- **1. 🕹️ Play**: Iniciar una nueva partida.
- **2. 📜 View Game Rules**: Ver las reglas del juego.
- **3. 🗺️ View Maps**: Ver la información de los mapas disponibles en el juego.
- **4. 👥 View Character Information**: Ver la información sobre los personajes disponibles.
- **5. 🚪 Exit**: Salir del juego.

El método también utiliza `DecoracionUtils.MostrarConBordes()` para mostrar el menú de manera estilizada.

3. `GameMenuSelection()`



```
1 reference | Codeium: Refactor | Explain | Generate Documentation |
public static int GameMenuSelection()
{
    GameMenu();

    bool validInput = true;
    DecoracionUtils.MostrarMensajeEstilado("🌟 Choose an option: (1-5) ");
    int opcion = Convert.ToInt32(Console.ReadLine());

    while (validInput)
    {
        if (opcion >= 1 && opcion <= 5)
        {
            validInput = false;
        }
        else
        {
            Console.WriteLine("❌ Invalid option.");
            DecoracionUtils.MostrarMensajeEstilado("🌟 Choose an option: (1-5) ");
            opcion = Convert.ToInt32(Console.ReadLine());
        }
    }

    return opcion;
}
```

Esta función es responsable de mostrar el menú y obtener la selección del jugador. Primero, se llama a `GameMenu()` para mostrar las opciones, luego solicita que el jugador ingrese un número entre 1 y 5 (la opción que desea seleccionar). Si el jugador ingresa una opción inválida, el programa sigue pidiendo una opción válida hasta que el jugador ingresa un número correcto. Finalmente, devuelve la opción seleccionada por el jugador.

4. `IniciarCronometro(int TiempoMaximo)`

Esta función maneja un temporizador para el juego. Al iniciar el cronómetro, se le pasa un parámetro `TiempoMaximo`, que representa los segundos que el jugador tiene para completar la ronda o mapa. Cada segundo se va contando, y se muestra un mensaje indicando el tiempo restante. El cronómetro también permite al jugador pausar presionando cualquier tecla (aunque no afecta el conteo del tiempo, solo lo detiene momentáneamente). Si el tiempo se agota, la función marca que el tiempo ha terminado y muestra un mensaje indicando que el tiempo ha pasado. La función retorna un valor booleano que indica si el tiempo se ha agotado.
