Conqueror ✡

Estructura de clases del juego:

El juego tiene como núcleo de su funcionamiento a la clase **Game** que es un Enumerable de *Status* para evitar que el juego sea infinito, obteniendo ante cada jugada un nuevo estado del juego. Es la encargada de hacer todo lo relacionado con el manejo del juego. Según el nivel que se le pase a la nueva instancia de juego se definirán los tipos de jugadores. Un 0 indica que serán dos humanos. Un 1 que serán un humano y un jugador virtual, mientras que un 2 es para crear un juego entre dos jugadores virtuales. Esto se hace en la función InitializePlayers.
En su interior cuenta además con los siguientes métodos:

- ✔ GameOver: Determina si el juego ha terminado o no según las reglas definidas.
- ✔ IsValid: Determina si una jugada es válida o no según las reglas definidas.
- ✔ GetWinner: Obtiene el ganador del juego.
- ✔ StabilizeLife: Impide que las vidas tomen un valor negativo. Si ese fuera el caso, se establecen en 0.
- ✔ ChangeTurns: Se interacmbian los jugadores y al jugador en turno se le añade una carta y un charm.
- ✔ Input: Recibe la carta seleccionada por el jugador y la activa si fue una jugada válida. Por las reglas del juego, si el jugador fue un humano no se le permitirá realizar ninguna otra acción y pasará inmediatamente a esperar por la próxima jugada.
- ✔ PlayIA: Realiza la jugada del jugador virtual si el jugador en turno fuera de este tipo. Devuelve si realizó o no el movimiento.
- ✔ Activate: Obtiene el nuevo contexto luego de lanzar la carta, actualiza el estado de juego a partir de los cambios ocurridos y estabiliza la vida de los jugadores.
- ✔ Launch: Elimina la carta de la mano del jugador que la activó e interpreta su efecto, obteniendo un nuevo contexto a partir de él.
- ✔ Output: Mostraría los resultados del juego. GetEnumerator: Si no ha terminado el juego, intercambia los turnos. Si el jugador en turno es un jugador virtual, activa su carta y vuelve a cambiar el turno. Devuelve el estado de juego actual.

```
/// <summary>
/// Obtiene la siguiente jugada
/// </summary>
/// <returns>El nuevo estado de juego</returns>
public IEnumerator<Status> GetEnumerator() {
        if (!GameOver()) {
            ChangeTurns();
            if (PlayIA()) {
                ChangeTurns();
            }
            yield return this.st;
        }
        else
        yield break;

}
```

La clase **Status** es la que define el estado actual del juego. Cuando se crea, lo hace a partir de un número de jugadores. Tiene una lista de *PlayerStatus* que es quien lleva los estados actuales de los jugadores. La cantidad de elementos de esta lista estará definida por el número de jugadores. Esta clase cuenta con dos funciones:

ChangePlayers: Se encarga de intercambiar los estados de los jugadores para que de esta forma el jugador en turno siempre sea el que ocupa la posición 0 en la lista de estados.

UpdateStatus: Actualiza el estado actual del juego a partir de los valores del Contexto que fueron actualizados al activar un efecto.

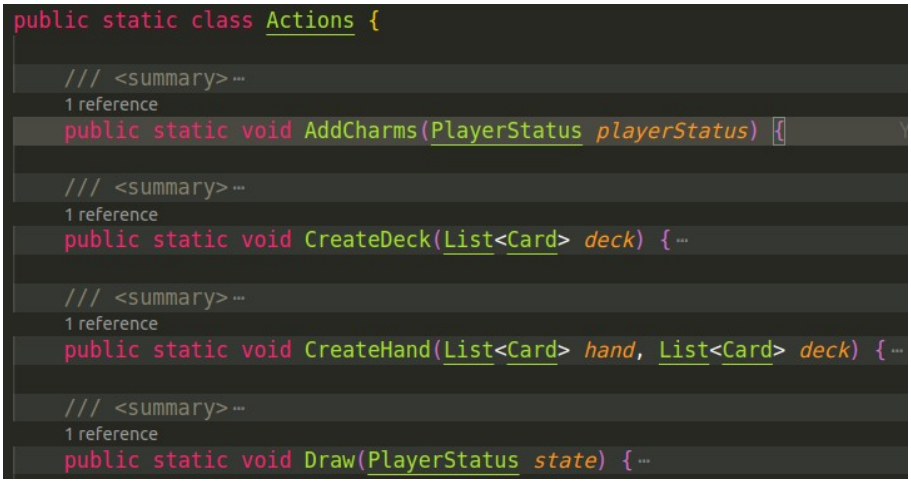
Cada instancia de la clase **PlayerStatus** define el estado en que se encuentra un jugador. Tiene un *player* que es el propio jugador, vida, charms, una lista de *cartas* en la mano del jugador y una lista de *cartas* del mazo del jugador.

```
public class Status {

    /// <summary>      You, 3 minutes ago • Uncommitted changes ⋯
    35 references
    public List<PlayerStatus> playerStatuses { get; private set; }

    /// <summary>⋯
    1 reference
    public Status(int nPlayers) {
        playerStatuses = new();
        for (int i = 0; i<nPlayers; i++) {
            playerStatuses.Add(new PlayerStatus());
        }
    }
```

```
public class PlayerStatus {
    /// <summary>⋯
    9 references
    public Player player;
    /// <summary>⋯
    6 references
    public List<Card> playerDeck = new();
    /// <summary>⋯
    10 references
    public List<Card> playerHand = new();
    /// <summary>⋯
    14 references
    public int life;
    /// <summary>⋯
    10 references
    public int charms;
```

La clase **Actions** tiene un conjunto de funciones que se utilizarán solo para realizar acciones dentro del juego. Es por ello que en su definición es estática.

Sus métodos permiten añadir o quitar cartas (Draw, RemoveCard), charms (AddCharms), crear un deck (CreateDeck), una mano (CreateHand) y barajar una lista de cartas (Shuffle).



La clase **Character** define un personaje que se almacena en una base de datos. Cada personaje tendrá un nombre, una foto y un Id que lo hace único.
La clase **Player** representa un jugador. Esta clase es abstracta porque no se pueden declarar jugadores sin saber qué tipo de jugadores son. Hereda sus características de *Character* porque cada jugador será creado a partir de un personaje.
La clase **PlayerHuman** y la clase **PlayerIA** ambas heredan de la clase *Player* sus propiedades.
La clase *PlayerIA* se diferencia de *PlayerHuman* en que tiene una función llamada SelectIACard para seleccionar la carta que va a activar el jugador virtual.

La clase **Card** define el elemento carta en el juego. Una carta tendrá un nombre, un id que la hace única, una foto, un valor de rareza que define qué tan frecuentemente debe aparecer la carta, un coste de charms, un texto de descripción y un efecto. Los charms de la carta y su rareza nunca podrán ser negativos.

La clase **Config** se emplea para establecer configuraciones iniciales del juego como la vida con que empiezan los jugadores y la cantidad de charms.

La clase **Database** se encarga de manejar los elementos almacenados en las bases de datos. Esto comprende las cartas y los personajes que se guardan en archivos .json una vez son creados. Tiene una lista de *cartas* y una lista de *personajes* que se rellenan cuando se crea un objeto de esta clase mediante los métodos InitCards y InitCharacters. Tiene dos funciones para almacenar las cartas y los personajes cuando se crean, que son StoreCard y StoreCharacter. Los métodos GetCard y GetCharacter permiten buscar entre las cartas y los personajes ya almacenados uno en específico y lo devuelven si fue encontrado. Las funciones GetLastId y UpdateId permiten obtener el último Id en una de las bases de datos, ya sea de cartas o de personajes, y actualizar el último Id existente respectivamente.

```
1 reference
public void StoreCard(Card cd) {
    InitCards();
    Cards.Add(cd);
    var options = new JsonSerializerOptions { WriteIndented = true };
    string jsonString = JsonSerializer.Serialize(Cards, options);
    File.WriteAllText(Config.PathCard, jsonString);
}

/// <summary>…
2 references
public Character GetCharacter(int id) {
    foreach (var item in Characters) {
        if (id == item.Id) {
            return item;
        }
    }
    Utils.Error("Personaje no encontrado");
    return null;
}
```

La clase **Manager** es la que va a crear las cartas y los personajes usando los métodos CreateCard y CreateCharacter.

La clase **Utils** contiene funciones para crear nuevos *Contextos* que utiliza el intérprete de efectos y para interpretar los efectos directamente. Para ello son los métodos InterpretEffect y CreateScope. También contiene una función para lanzar excepciones ante errores llamada Error.

La interfaz **IGraphics** que implementa **Game** cuenta de dos funciones: Input y Output. Esto es para garantizar que aquel que emplee la clase Game tenga que definir las funciones que serán empleadas para mostrar el juego y para leer las entradas de los usuarios. En nuestro caso el método Output tiene una implementación vacía porque para lograr su propósito hemos usado Blazor como framework web para la interfaz gráfica.

```
interface IGraphics {

    /// <summary>
    /// Metodo para que el juego pueda recibir la carta a activar
    /// </summary>
    /// <param name="card">La carta jugada</param>
    1 reference
    public void Input(Card card);

    /// <summary>
    /// Metodo para mostrar los cambios que ocurren en el juego en una interfaz
    /// </summary>
    0 references
    public void Output();
}
```

Intérprete:

Objetivo: Modificar valores del estado del juego con el fin de variar el estado del juego. Estos valores serían la vida de los jugadores, los puntos para lanzar cartas (charms), y poder llamar algunas funciones predefinidas para agregar variedad a los efectos. Esto se realizará implementando un intérprete para un lenguaje con varias restricciones, pero pudiendo realizar tareas como crear variables enteras y estructuras condicionales y ciclos.

Características:

- Al final de toda línea debe ir un ;
- Los bloques de código deben empezar por { y finalizar en } y todos los bloques excepto el último de terminar en };
- No hay tipado de variables. El único tipo que existe es int y en las estructuras con condición se trabaja internamente con un tipo bool.
- Existen funciones predefinidas y varibles constantes predefinidas.
- No existen comentarios.
- El `Context` es general, o sea, una varible creada en un if estará accesible fuera de él.

Sintaxis:

- ✔ Operaciones algebraicas: **+ - / * ( )**
- ✔ Operaciones booleanas: | & < > **==**
- ✔ Asignación: **=**
- ✔ Estructuras condicionales: if(condicion){}
- ✔ Estucturas iterativas: while(condicion){}
- ✔ Uso de funciones predefinidas: fs()
- ✔ Declaración y uso de variables enteras.
- ✔ Dentro de las estructuras condicionales no se pueden poner paréntesis.

Implementación:
Un **Token** es un objeto que tiene un tipo un valor que se obtiene del código escrito.
El **Lexer** es el encargado de leer el string de código e ir convirtiéndolo en *tokens* y se va construyendo un AST (árbol de sintaxis abstracta). El encargado de esto es el **Parser**. En el *Lexer* y *Parser* se encuentran los errores en tiempo de compilación, enviando entonces una excepción. En el *Lexer* estarían los relacionados a la escritura correcta de los *tokens* del lenguaje por ejemplo:

En el *Parser* se encontrarían errores de sintaxis como por ejemplo:

```
if (3)) {}
```

Luego, al estar construido el AST, se le pasa a la clase **Interpreter** encargada de recorrer este árbol e ir ejecutando cada instrucción donde lleva un **Context** que guarda las variables en tiempo de ejecución.

Ejemplos de código:

```
EnemyLife= 2;
a = 4;
if (EnemyLife == 2 | MyLife > 2) {
    EnemyLife = a;
    ChangeHands();
};
while (EnemyLife > 1) {
    EnemyLife = EnemyLife - a;
}
```