

CASTLEVANIA – BATTLECARDS

Reporte del proyecto de programación BattleCards:

Desarrolladores del proyecto:

Adrián Hernández Castellanos – C112

Equipo # 35

Laura Martir Beltrán – C111

BattleCards es un proyecto desarrollado con la intención de simular un videojuego de cartas del estilo Tradings Cards Game, en el cual el jugador o los jugadores poseen cartas determinadas con habilidades únicas que deben aprender a usar y desarrollar una táctica de juego que le permita eliminar las cartas del enemigo. En nuestro caso las reglas del juego son sencillas:

- El turno de cada jugador se divide en dos sesiones, jugar una carta y realizar una acción.
- Si el campo está lleno, no se puede jugar ninguna carta de la mano.
- Si no hay cartas cercanas a una carta del jugador, no se podrá atacar o realizar acciones.
- Si todas las cartas de un jugador son destruidas, el jugador pierde la partida.
- Si no quedan cartas en las manos de los jugadores y no es posible realizar acciones, gana el jugador con más cartas en el campo. En el caso en que ambos tengan la misma cantidad de cartas, la partida queda en empate.

Para profundizar en los aspectos más importantes del desarrollo de nuestro proyecto, vamos a realizar una profundización en los siguientes temas:

- Aspectos básicos del programa
- Modos de juego
- Jugador virtual
- Jerarquía de clases
- Lenguaje de programación
- Retoques finales

Aspectos básicos del programa:

Nuestro juego no es extenso ni mucho menos complejo, lo cual hace que las partidas sean rápidas e interesantes. Al ser ejecutado en una consola su único requisito consiste en tener instalado el Sistema Operativo Windows con .Net Core 6.0. Al abrir la carpeta raíz del juego encontraremos los scripts programados en C#, el archivo Readme que posee las instrucciones y modo de empleo del lenguaje y este documento. Para ejecutar la aplicación de consola, debemos acceder a la ruta bin\Release\net6.0\Castlevania Battlecards.exe. En esta misma carpeta se encuentra la carpeta edit, que contiene a los archivos Cards.txt (archivo que contiene a las cartas principales del juego) y Editor.txt (archivo que contiene a las cartas editadas por el usuario a través del lenguaje).

Este equipo > TOSHIBA EXT (E:) > Castlevania Battlecards					
	Nombre	Fecha de modifica...	Tipo	Tamaño	
do	.vscode	25/12/2022 21:11	Carpeta de archivos		
	bin	25/12/2022 21:22	Carpeta de archivos		
	IA	27/12/2022 20:27	Carpeta de archivos		
	obj	27/12/2022 18:11	Carpeta de archivos		
	screen	26/12/2022 10:18	Carpeta de archivos		
os	src	27/12/2022 13:52	Carpeta de archivos		
ys Rainbo	Castlevania Battlecards	25/12/2022 21:11	Archivo de origen ...	1 KB	
il (D:)	Castlevania	28/12/2022 02:37	Archivo de origen ...	2 KB	

Este equipo > TOSHIBA EXT (E:) > Castlevania Battlecards > bin > Release > net6.0					
	Nombre	Fecha de modifica...	Tipo	Tamaño	
o	edit	26/12/2022 11:38	Carpeta de archivos		
	publish	25/12/2022 21:22	Carpeta de archivos		
is	Castlevania Battlecards.deps	27/12/2022 18:12	Archivo de origen ...	1 KB	
	Castlevania Battlecards.dll	27/12/2022 21:12	Extensión de la ap...	58 KB	
s Rainbo	Castlevania Battlecards	27/12/2022 21:12	Aplicación	146 KB	
	Castlevania Battlecards.pdb	27/12/2022 21:12	Archivo PDB	32 KB	
	Castlevania Battlecards.runtimeconfig	27/12/2022 18:12	Archivo de origen ...	1 KB	

Nuestro juego funciona a base de entradas que le proporciona el usuario a través de la consola, y actúa en consecuencia. Al ejecutar la aplicación, se mostrará la información inicial del programa y una lista de opciones a elegir.

Modos de juego:

El juego admite modo de un jugador y modo de dos jugadores, además de presentar opciones de mostrar un breve tutorial y una lista de todas las cartas guardadas.

Al comenzar la partida se otorgan a cada jugador 5 cartas aleatorias de todas las cartas disponibles, incluidas las creadas por el usuario. El juego va avanzando por turnos haciendo una jugada y una acción en cada uno, hasta que eventualmente un jugador se quede sin cartas o la partida termine en empate.



El juego muestra, aparte de las cartas de cada jugador, un campo que será el tablero del juego. Las cartas posicionadas en el tablero pueden realizar acciones sobre cartas aliadas y enemigas. Las acciones que cada carta realiza dependen exclusivamente de la carta en cuestión, pero generalmente estas acciones se pueden catalogar en ataques y recuperaciones.



Cada carta puede realizar una acción sobre las cartas que la rodean en el campo, siendo posible que todas las cartas puedan atacar a la posición central del campo, pero muy pocas cartas pueden atacar a una carta situada en una esquina. Aparte de esta funcionalidad, la estrategia de este juego se basa en que si puedes “ver” a tu enemigo, entonces él también puede verte a ti. El truco es usar la carta más conveniente en la posición más óptima para ella.

Jugador virtual:

En el modo de juego Jugador vs Computadora conocerás a nuestro jugador virtual. Consiste en una serie de métodos contenidos en una interfaz que se usan durante el turno de la Computadora, la cual no es el mejor jugador del mundo. Su comportamiento consiste en intentar hacer el mayor daño posible al jugador e intentar jugar las cartas en la posición más protegida posible, sin ningún tipo de estrategia alterna ni ninguna forma de despistar al oponente. Aún así, cabe destacar que nos ha ganado en no pocas partidas de prueba que hemos realizado antes de realizar la entrega, así que o bien nosotros somos extremadamente malos en nuestro propio juego (no lo creo) o bien nuestro jugador virtual realmente es capaz de generar competencia utilizando una estrategia de juego específica que no es la más óptima a pesar de su eficacia.

Jerarquía de clases:

Inicialmente hemos definido una clase Card, que será nuestro prototipo de carta en el juego. Esta clase contendrá varios parámetros sencillos, y sus valores predeterminados llamados Bases, que nos ayudaran a controlar dichos valores en el juego para evitar que nuestras cartas sobrepasen el límite de salud, energía y daño del que disponen.

```
public class Card
  67 references
  public string Name;
  5 references
  public Action[] Actions;
  71 references
  public int Health;
  64 references
  public int Energy;
  60 references
  public int Damage;
  31 references
  public int BaseHealth;
  29 references
  public int BaseEnergy;
  29 references
  public int BaseDamage;
```

Los únicos parámetros que recibirá el constructor serán un entero por cada habilidad y su base, ya que al momento de crear la carta no se encuentra en la partida, y la base y el parámetro pueden tomar el mismo valor.

También contiene un array de Action, que no es más que una clase que representa una acción y contiene a dos arrays de otras clases llamadas Conditions y Effects.

```
public class Action //Esta clase rep
{
  9 references
  public string ID;
  4 references
  public Condition[] Conditions;
  4 references
  public Effect[] Effects;
```

Las Conditions y los Effects son dos clases que contienen respectivamente cadenas de caracteres (strings) que representan una condicional (booleano) y un efecto (asignación).

```

// Referencias
public class Condition //Esta clase representa una condición que se debe o no cumplir
{
    16 references
    public string ID;
    3 references
    public bool Value;
    4 references
    public string Expression; //Este valor debe ser analizado durante el juego para definir el valor del value
    2 references
    public Condition(string ID, string Expression)
    {
        id=ID;
        Value=false;
        this.Expression=Expression;
    }
}

// 32 references
public class Effect //Esta clase representa el efecto que tendrá una carta
{
    11 references
    public string ID;
    3 references
    public string Expression;
    2 references
    public Effect(string ID, string exp)
    {
        id=ID;
        Expression=exp;
    }
}

```

Además, para la mayoría de métodos utilizados nos hemos creado un script Functions donde se encuentran métodos útiles a lo largo de la ejecución del programa, previamente catalogados según su función en el código.

Lenguaje de programación:

Definitivamente esta sección es la más extensa de todo nuestro proyecto, y la que nos costó días enteros sentados frente a una pantalla. Para analizar toda la estructura del lenguaje pasaremos por diferentes scripts contenidos en la carpeta src dentro de la carpeta raíz del proyecto.

La siguiente imagen muestra el prototipo de la sintaxis de nuestro lenguaje de programación, que de nada nos va a servir si no hemos leído el apartado [Readme.dm](#) que se encuentra en la carpeta raíz del proyecto.

```

Key ==> card, action, effect, condition

Definition ==> Anything. (cabe con el ; al final)

Open and Closed Symbol ==> [ ] (usar después de una definición para parametrizar sus propiedades)

Word, Value and separator ==> card Test1. { health: 100, energy: 50, damage: 25, actions: (attack){defend}{selfHeal}, } -> Cada acción debe estar entre ()
action selfHeal. {conditions: [(lifeLessThan50){energyMoreThan15}], effects: (healLife25){decreaseEnergy10}, } -> Cada condition y effect debe estar entre {}
condition ifEnergyMoreThan50. {value: caster.health>50,}
condition energyMoreThan15. {value: caster.energy>15|caster.energy-15,} -> Entre condicionales se introduce el operador "|" (or), que determina
effect healLife25. {value: caster.health+(caster.health*25),} a una condicional como verdadera si alguna de las dos se cumple
effect decreaseEnergy10. {value: caster.energy-(caster.energy-15),} -> Introducir un operador (and) no es necesario entre condicionales y
efectos, solo debe declararse otra condicional o efecto

==> Para que una carta pueda activar una acción, todas sus condicionales deben cumplirse (true)
==> Cuando esto ocurre, todos sus efectos se ejecutan sobre las cartas definidas

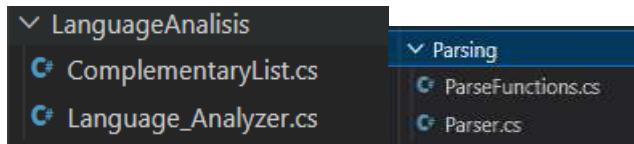
***Expresiones válidas para condicionales:
{caster,target}.(health,energy,damage){<,>,<=,>=,"int",[]}>->caster.(health,energy,damage),target.(health,energy,damage)
{player,enemy},{anyCard,allCards},{health,energy,damage}{<,>,<=,>=,"int",[]}>{Cualquier expresión ya mencionada}
field,cards{<,>,<=,>=,"int",[]}>{<="int"><=0}

***Expresiones válidas para efectos:
{caster,target}.(health,energy,damage)-{("int",[])>{caster,target}.(health,energy,damage){+,-,"/"}("int",[])>{caster,target}.(health,energy,damage)

```

Dentro de la carpeta src podremos encontrar varias carpetas, la primera de ellas que vamos a analizar se llama LanguageAnalysis, la cual se encarga de analizar la sintaxis de cada palabra escrita en nuestro lenguaje. Para ello, en la clase Keywords contiene todo tipo de clasificaciones de palabras que va otorgando según la iteración del lenguaje.

Tras asegurarnos de tener nuestro lenguaje bien escrito sintácticamente, tenemos que cerciorarnos de que lo que está escrito tenga sentido, que algunos nombres de campos no estén repetidos, que el contexto tenga definido todos los objetos que se implementan en la creación de una carta, etc... Para eso es el método `Lexical_Analyzer`, el cual también devuelve una lista de errores que se presentan tras realizar toda la revisión.



Tras hacer la revisión del lenguaje y comprobar que efectivamente no hay problemas con su interpretación, tenemos que parsear (convertir las cadenas de caracteres en datos que nuestro lenguaje base C# pueda interpretar). Para ello está la clase `Parser`, que contiene todo el procedimiento de creación de Cartas, Acciones, Condiciones y Efectos, junto con una lista de errores en caso de que algo salga mal durante el parseo. La clase `ParseFunctions` contiene todo tipo de métodos útiles para esta tarea. Tras guardarse todas las cartas y comprobar que no existe ningún error durante el parseo, debemos implementar una última clase llamada `GetCards`, cuya única función es la más extensa de todo el lenguaje, ya que se encarga de darle sentido a las expresiones de las condicionales y efectos creados como complementos de las acciones. No obstante esto es una revisión parcial que verifica que todo esté bien escrito en dichas expresiones, pues al tratarse de un lenguaje que crea condiciones que dependen del estado de la partida, el sentido a dichas expresiones se le otorgará durante la propia partida a través de la clase `InGameEvaluators` que se encuentra en el script `Functions`, cuando disponga de todas las variables necesarias para ello.

Terminando este análisis las cartas se guardan en un arreglo de cartas que contiene tanto a las cartas por defecto como a las cartas editadas, y es este arreglo del cual salen aleatoriamente las cartas de cada partida. Posteriormente a este análisis, el cual se realiza en el método de la interfaz `GameLaunch`, se le da comienzo al juego y en caso de que el código contenga algún error, se mostrará en pantalla al iniciar.

```
static void Main(string[] args)
{
    //Extraer contenido de cartas del juego base
    string code = new StreamReader("edit\\Cards.txt").ReadToEnd();
    //Analizar contenido de cartas del juego base
    (Card[] GameCards, List<string> GameErrors) = GameLaunch.Get(code);
    //Extraer contenido de edición
    string Edit = new StreamReader("edit\\Editor.txt").ReadToEnd();
    (Card[] EditCards, List<string> EditErrors) = GameLaunch.Get(Edit);
    if(GameErrors.Count != 0)
    {
        Console.WriteLine("El archivo edit\\Cards.txt no es válido, no se puede iniciar el juego...");
        foreach(string x in GameErrors)
        {
            Console.WriteLine(x);
        }
        Console.ReadLine();
        throw new Exception();
    }
}
```

```

Card[] AllCards = new Card[]{};
foreach(Card x in GameCards)
{
    AllCards=EnumerableFunctions.AddCard(AllCards, x);
}
if(EditErrors.Count==0)
{
    foreach(Card x in EditCards)
    {
        AllCards=EnumerableFunctions.AddCard(AllCards, x);
    }
}
//Comienza el juego
Screen.GameManager(AllCards,EditErrors);

```

Retoques finales:

Para terminar, hemos decidido implementar la aplicación de visualización gráfica a modo de consola. La interfaz Screen contiene todo tipo de métodos para visualizar elementos de la partida, así como la implementación del sistema de turnos y las condiciones de victoria y derrota de cada jugador.

Finalmente, la clase principal y la que ejecuta el programa es la clase Castlevania, que se encuentra en la carpeta raíz del programa.

Esto es todo por ahora...