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.

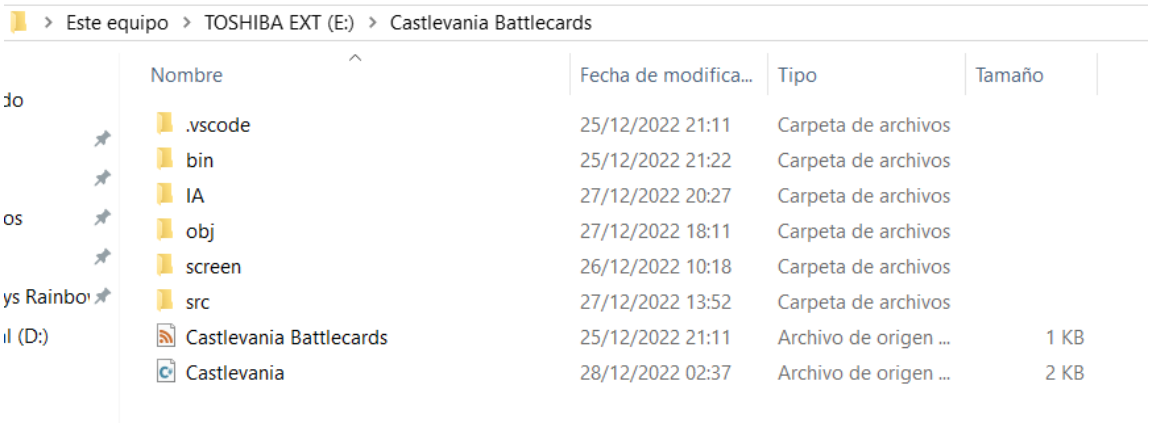
-Si no quedan cartas en las manos de los jugadores y no es posible realizar acciones, gana el jugador con más cartas en el campo. En el caso en que ambos tengan la misma cantidad de cartas, la partida queda en empate.

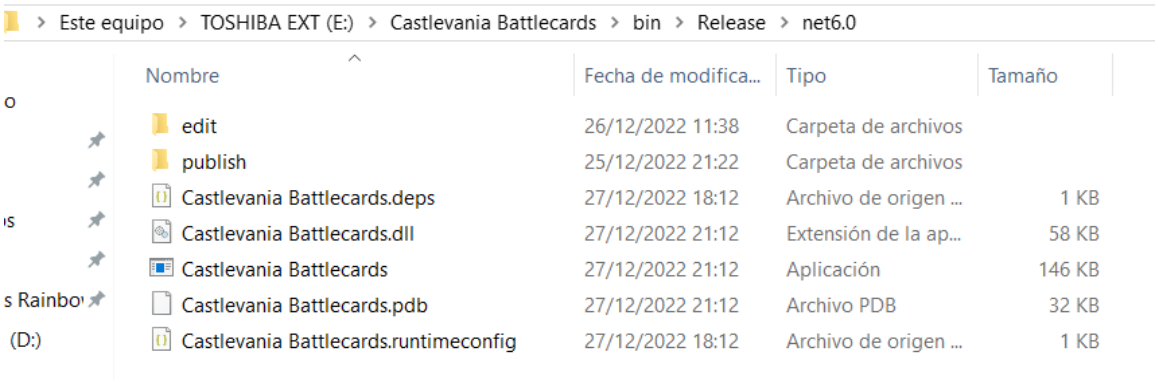
Para profundizar en los aspectos más importantes del desarrollo de nuestro proyecto, vamos a realizar una profundización en los siguientes temas:

- 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).





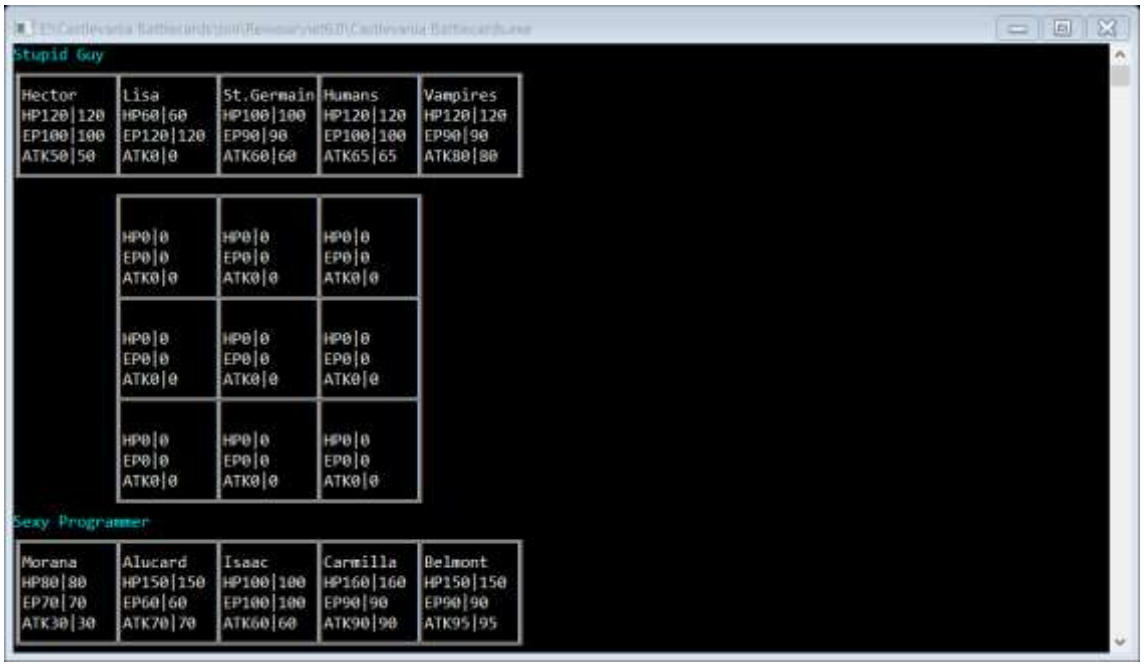
Nuestro juego funciona a base de entradas que le proporciona el usuario a través de la consola, y actúa en consecuencia. Al ejecutar la aplicación, se mostrará la información inicial del programa y una lista de opciones a elegir.

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.

Los modos de Jugador vs Jugador y Jugador vs IA inician las partidas otorgando 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 IA conocerás a Piolín, nuestro jugador virtual. Consiste en una serie de métodos contenidos en una interfaz que se usan durante el turno de la IA. Piolín, como su nombre aparenta, no es el mejor jugador del mundo, y para cada partida que juega su comportamiento se puede comparar con el de un niño pequeño (de ahí el nombre). Este 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 Piolín 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.

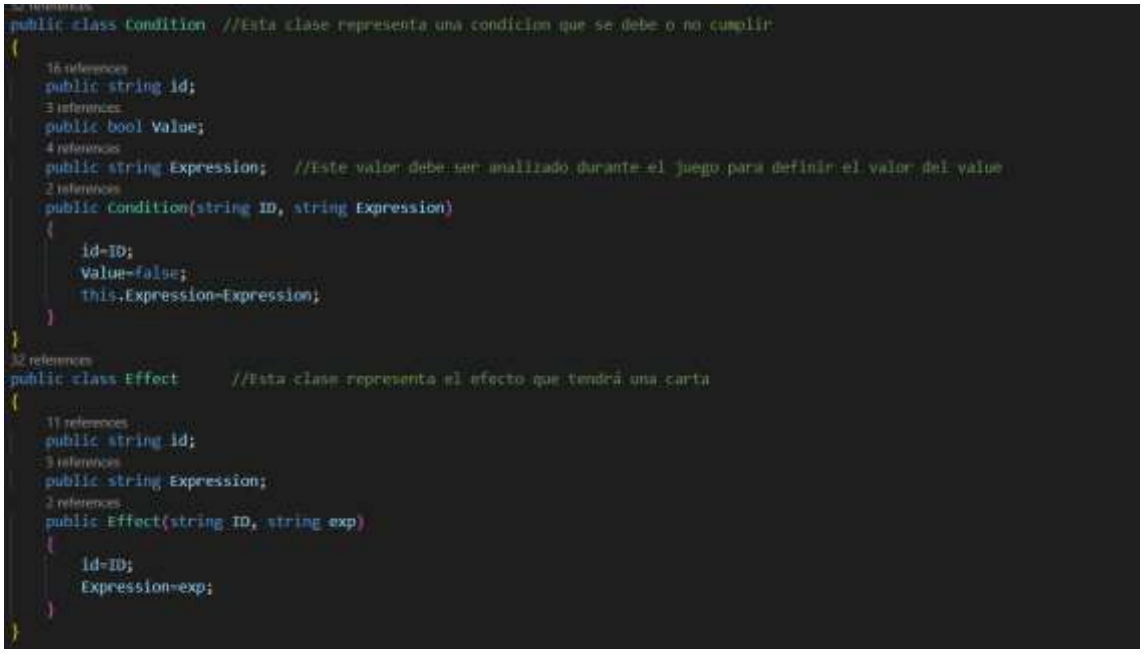
```
public class Card

    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 re
{
    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).



```
public class Condition  //Esta clase representa una condicion 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 algunos métodos sin especificar nos hemos creado una clase Methods donde se encuentran métodos muy repetidos o útiles a lo largo de la ejecución del programa.

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.

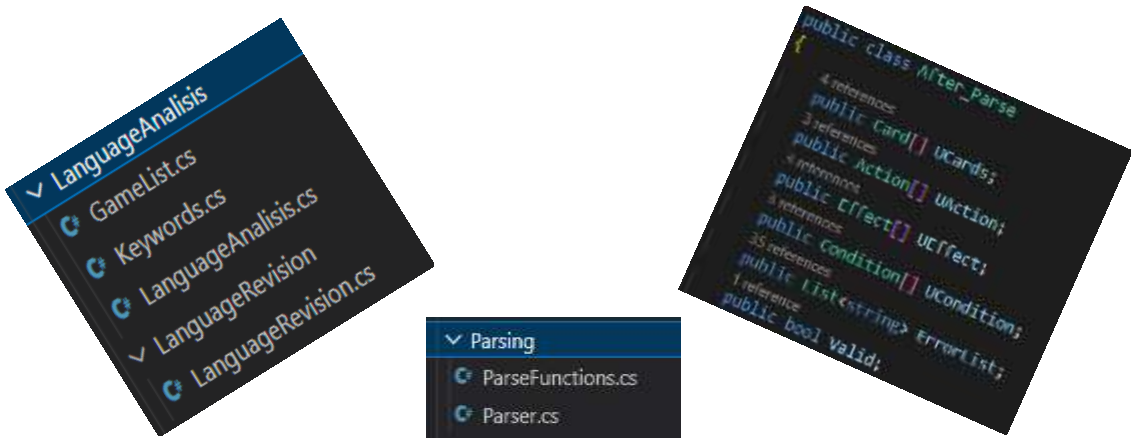
La siguiente imagen muestra el prototipo de la sintaxis de nuestro lenguaje de programación, que de nada nos va a servir si no hemos leído el apartado Readme.dm que se encuentra en la carpeta raíz del proyecto.

```
Key ==> card, action, effect, condition

Definition ==> Anything.        (ojo con el . al final)

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

Word, Value and Separator ==> card Testi. { health: 100, energy: 50, damage: 25, actions: {attack}{defend}{selfheal},}        => cada action debe estar entre {}
                              action SelfHeal. {conditions: {LifeLessThan50}{EnergyMoreThan15}, effects: {HealLifeIn25}{DecreaseEnergyIn15},}  => cada condition y effect debe estar entre {}
                              condition LifeLessThan50. {value: caster.health<50,}
                              condition EnergyMoreThan15. {value: caster.energy>15}caster.energy-=15,}   => entre condicionales se introduce el operador "|" (or), que determina
                              effect HealLifeIn25. {value: caster.health=(caster.health+25),}                  a una condicional como verdadera si alguna de las dos se cumple
                              effect DecreaseEnergyIn15. {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){<,>,=,<=,>=,}(Cualquier expresión ya mencionada)
        field.cards{<,>,=,<=,>=,}(3<="int"<=9)

***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 LanguageAnalisis, 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 en la clase

LanguageAnalisis. Al finalizar, se guardan todos los errores de compilación en una lista que el usuario puede revisar posteriormente para arreglar estos errores.

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 la clase LanguageRevision, la cual también contiene 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. Posteriormente las cartas y objetos creados se guardan temporalmente en una lista de una clase llamada GameList. Tras guardarse todas las cartas ahí y comprobar que no existe ningún error durante el parseo, debemos implementar una última clase llamada After_Parse, 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 Evaluate, 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 se le da comienzo al juego y en caso

de que el código contenga algún error, se mostrará en pantalla al iniciar.

```
//Extraer contenido de cartas del juego base
StreamReader St = new StreamReader("edit\\Cards.txt");
string code = St.ReadToEnd();
//Analizar contenido de cartas del juego base
LanguageAnalisis Code01 = new LanguageAnalisis(code);
LanguageRevision Code02 = new LanguageRevision(Code01);
ParserCheck Code03 = new ParserCheck(Code02);
After_Parse Code04 = new After_Parse(Code03);
//Extraer contenido de edición
StreamReader Ed = new StreamReader("edit\\Editor.txt");
string Edit = St.ReadToEnd();
Card[] AllCards = new Card[]{};
foreach(Card x in Code04.UCards)
{
    AllCards=Methods.AddCard(AllCards, x);
}
List<string> EditErrors = new List<string>();
if(Edit!="")
{
    LanguageAnalisis Edit01 = new LanguageAnalisis(code);
    LanguageRevision Edit02 = new LanguageRevision(Code01);
    ParserCheck Edit03 = new ParserCheck(Code02);
    After_Parse Edit04 = new After_Parse(Code03);
    foreach(Card x in Edit04.UCards)
    {
        AllCards=Methods.AddCard(AllCards, x);
    }
    EditErrors=Edit04.ErrorList;
}
//Comienza el juego
```

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…