

# Informe de Proyecto de Programación: Battle-Card.

Segundo Semestre.

## Integrantes:

- Orlando Rafael de la Torre Leal.
- Ernesto Alejandro Bárcenas Trujillo.

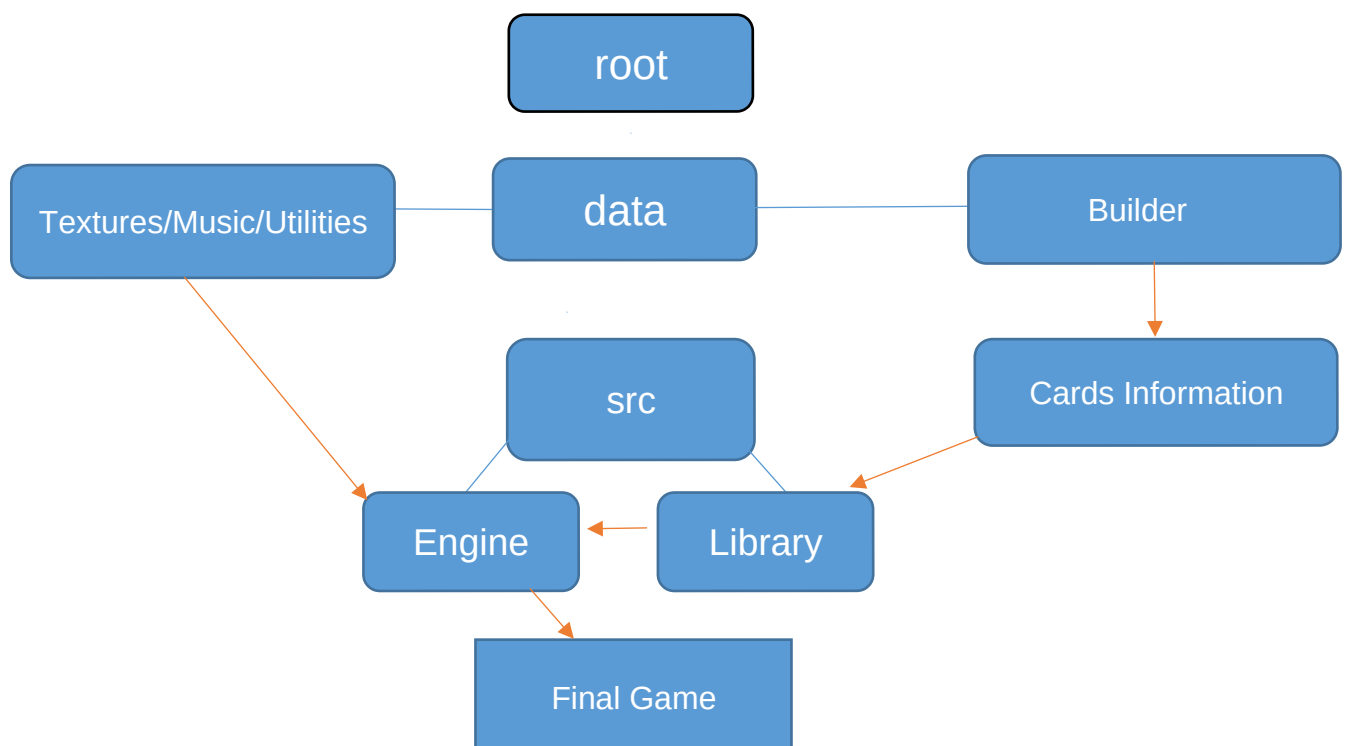
## Introducción:

Este proyecto consiste en la realización de un juego de cartas. Para la creación de este nos hemos basado en el minijuego de cartas que se encuentra en el videojuego **"The Witcher 3: The Wild Hunt"**: Gwynt, pero con una temática diferente; la ambientación y personajes serían los de la popular serie **"Game of Thrones"**.

El otro objetivo fundamental es incluir en el juego, una forma de que el usuario pueda crear sus propias cartas y efectos, con una capacidad ilimitada de combinaciones. Para ello hemos implementado un **mini-compilador**, capaz de interactuar con las variables del juego y dar instrucciones precisas, permitiendo crear prácticamente cualquier habilidad imaginable o posible dentro de las limitaciones del juego.

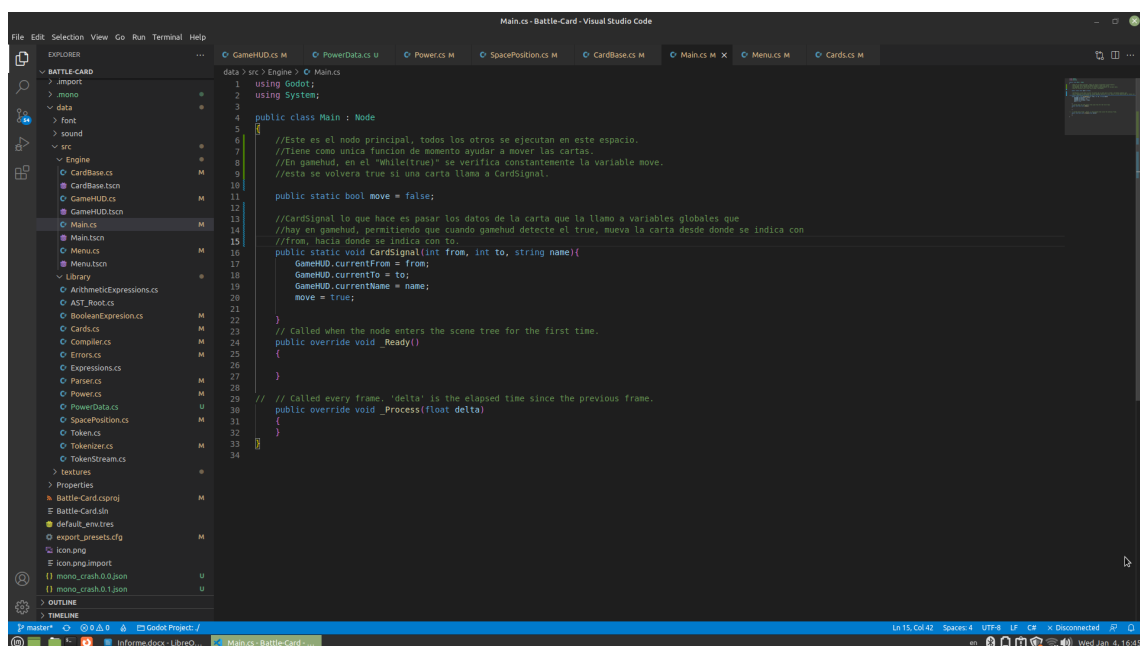
Para la visualización del juego se ha utilizado un motor de juegos **"OpenSource"** llamado Godot\*, no nos adentraremos en profundidad en el funcionamiento de dicha herramienta, pero si habrá un apartado para dar un mini-repaso técnico sobre su funcionamiento conceptual; será necesario para entender el funcionamiento del juego, pues lo que es la interacción con el mismo se realiza a través de sus interfaces.

Se ha concebido la conceptualización del *árbol* proyecto de la siguiente forma:



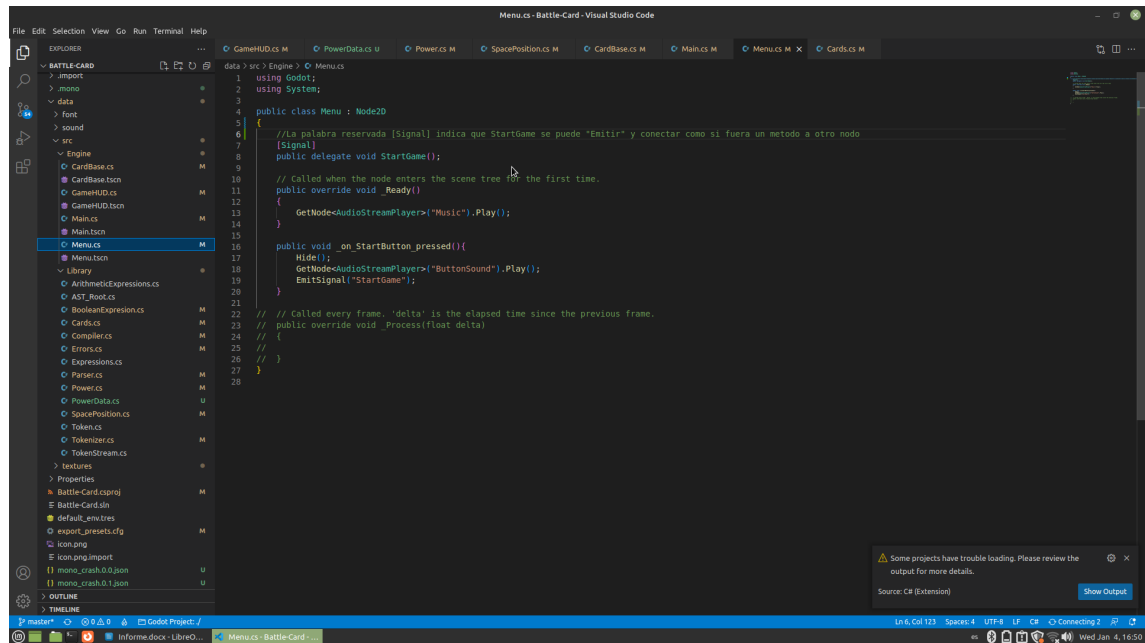
A continuación se procede a explicar el funcionamiento del motor(Godot):

Empezaremos por la clase **Main**, esto en el entendimiento abstracto del motor se considera un nodo, será el nodo principal, donde ocurrirá toda la acción, el menú y la interfaz de juego serán hijos de este nodo. La única función de **Main**, además de la mencionada anteriormente, será la de hacer de intermediario entre las cartas y la interfaz, pues las cartas son agregadas en tiempo de ejecución, y para poder hacer que estas interactúen con la interfaz de juego, se requiere de alguien que pase la información que las cartas envían, para ello tenemos un método llamado **CardSignal** que es **static** y una variable **bool** llamada **move**. **CardSignal** será llamado por las cartas, pasándole información de dónde y hacia dónde se quieren mover, además del nombre de la misma, una vez que se llama, **CardSignal** le pasa esta información a la interfaz de juego(**GameHUD**) y hace true la variable **move**; la clase **GameHUD** durante su flujo “**escucha**” a esta variable, y si es **true**, moverá la carta con los datos antes pasados.

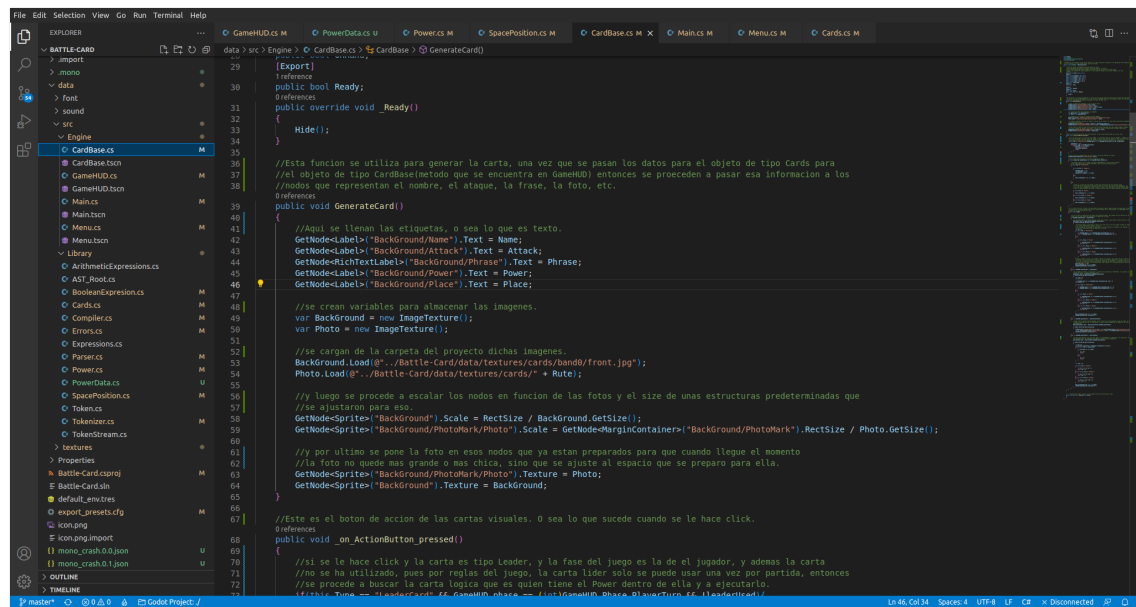


```
1 using Godot;
2 using System;
3
4 public class Main : Node
5 {
6     //Este es el nodo principal, todos los otros se ejecutan en este espacio.
7     //Tiene como unica funcion de momento ayudar a mover las cartas.
8     //En gamehud, en el "while(true)" se verifica constantemente la variable move.
9     //Esta se volvera true si una carta llama a CardSignal.
10
11     public static bool move = false;
12
13     //CardSignal lo que hace es pasar los datos de la carta que la llamo a variables globales que
14     //hay en gamehud, permitiendo que cuando gamehud detecte el true, mueva la carta desde donde se indica con
15     //from, hacia donde se indica con to.
16     public static void CardSignal(int from, int to, string name){
17         GameHUD.currentfrom = from;
18         GameHUD.currentto = to;
19         GameHUD.currentName = name;
20         move = true;
21     }
22
23     // Called when the node enters the scene tree for the first time.
24     public override void _Ready()
25     {
26     }
27
28     // Called every frame, 'delta' is the elapsed time since the previous frame.
29     public override void _Process(float delta)
30     {
31     }
32 }
33
34 }
```

Seguimos con la clase **Menu**; como su nombre lo indica será el nodo que represente el menú del juego, entre sus propiedades tiene un delegado (**StartGame**) que encima tiene la palabra reservada **[Signal]**, indicando que en el entendimiento del motor, esto es una señal que puede ser conectada con otros nodos, pasando información o iniciando una serie de instrucciones, aquí se utilizara para navegar entre los distintos nodos, pues este es el menú principal del juego.



```
data > src > Engine > Menu.cs
1 using Godot;
2 using System;
3
4 public class Menu : Node2D
5 {
6     //La palabra reservada [Signal] indica que StartGame se puede "Emtir" y conectar como si fuera un metodo a otro nodo
7     [Signal]
8     public delegate void StartGame();
9
10    // Called when the node enters the scene tree for the first time.
11    public override void _Ready()
12    {
13        GetNode<AudioStreamPlayer>("Music").Play();
14    }
15
16    public void _on_StartButton_pressed(){
17        Hide();
18        GetNode<AudioStreamPlayer>("ButtonSound").Play();
19        EmitSignal("StartGame");
20    }
21
22    // Called every frame, 'delta' is the elapsed time since the previous frame.
23    // public override void _Process(float delta)
24    // {
25    // }
26
27 }
28
```

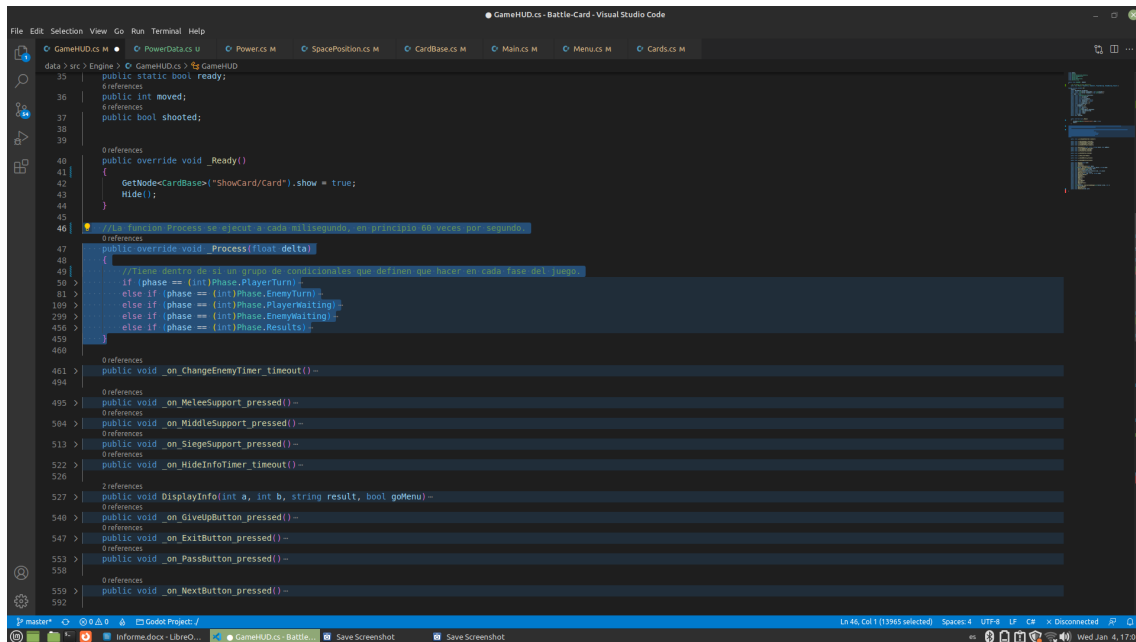


Continuamos con la clase **CardBase**, es la representación visual de las cartas logicas(**Cards**), la cual tiene dentro de si todas las propiedades que puede tener una carta del juego. Tiene una función llamada **GenerateCard()**, la cuál se encarga de que una vez se le ha pasado toda la información necesaria a **CardBase**, dicha función pasará estos datos a las estructuras o nodos para que pueda ser mostrado en pantalla. Explicación más detallada en los comentarios del código:

En **CardBase** nos encontramos también con una especie de función, que no es más que la acción que se tomará en cuanto se presione la carta, dentro pueden ocurrir muchas cosas, entre ellas que si la carta es de tipo **LeaderCard**: se ejecute su poder, si la carta se encuentra en la mano y es el turno del jugador: que se mueva la carta a su correspondiente posición y como consecuencia de ello, si la carta tiene alguna habilidad: ésta se ejecute, otra de las cosas que gestiona es que si estamos en medio de la ejecución de un poder, que funciona seleccionando algunas cartas para realizar algo con ellas, la carta estará en espera y reaccionará en dependencia del poder que se esté ejecutando.(Esto está mucho más detallado en los comentarios.).

Como último elemento del motor nos queda la interfaz de juego: **GameHUD**, la cual es más extensa y compleja, se hace muy difícil explicar de forma detallada como funciona todo, en cambio se irán mencionando los métodos y procesos que tiene y una explicación en abstracto del flujo de datos. Para una mejor comprensión de todo: revisar comentarios.

Lo primero que encontraremos será un **enum** llamado **Phase**, donde se definen las distintas fases del juego, esto será utilizado para organizar el flujo de datos. Este flujo de datos ocurre en la función **Process**, la cual se ejecuta(dependiendo del ordenador) unas 60 veces por segundo, y la misma irá ejecutando acciones en dependencia de la fase del juego en que estemos, la cual se definirá por una variable de tipo **int** llamada **phase**, que irá tomando los valores del **enum**.



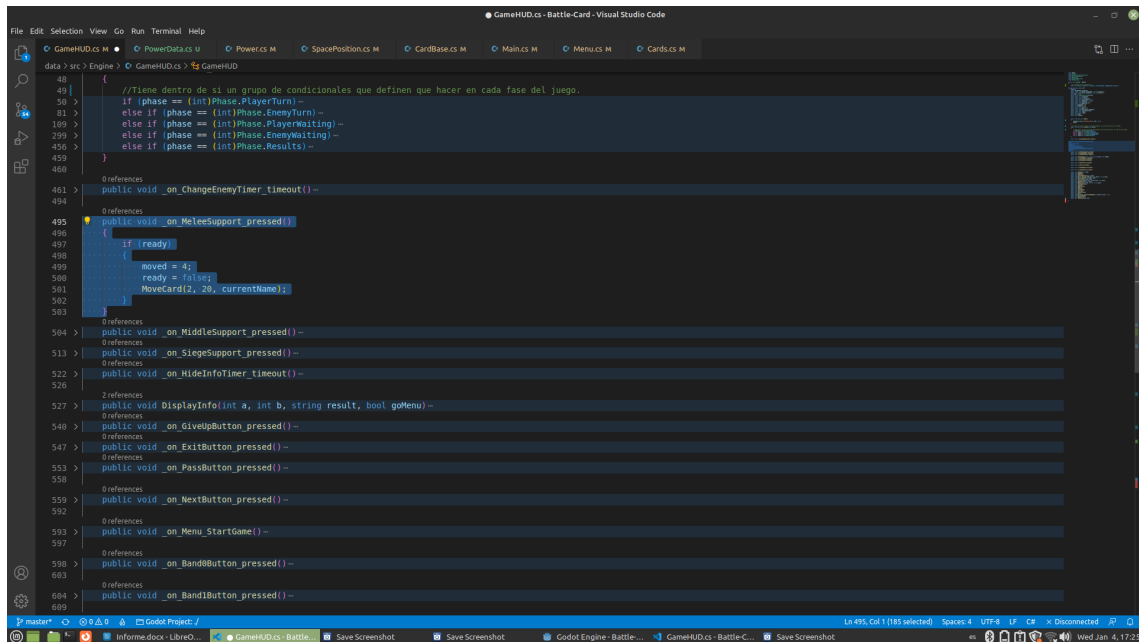
```
data > src > Engine > GameHUD.cs > GameHUD
35 public static bool ready;
36 public int moved;
37 public bool shot;
38
39
40 public override void Ready()
41 {
42     GetNodeCardBase("ShowCard/Card").show = true;
43     Hide();
44 }
45
46 //La función Process se ejecuta cada milisegundo, en principio 60 veces por segundo
47 public override void Process(float delta)
48 {
49     //Time dentro de el un grupo de condicionales que definen que hacer en cada fase del juego
50     if (phase == (int)Phase.PlayerTurn)
51     else if (phase == (int)Phase.EnemyTurn)
52     else if (phase == (int)Phase.PlayerWaiting)
53     else if (phase == (int)Phase.EnemyWaiting)
54     else if (phase == (int)Phase.Results)
55
56
57 public void on_ChangeEnemyTimer_timeout()
58
59 public void on_MeleeSupport_pressed()
60
61 public void on_MiddleSupport_pressed()
62
63 public void on_SiegeSupport_pressed()
64
65 public void on_HideInfoTimer_timeout()
66
67 public void DisplayInfo(int a, int b, string result, bool goMenu)
68
69 public void on_GiveUpButton_pressed()
70
71 public void on_ExitButton_pressed()
72
73 public void on_PassButton_pressed()
74
75 public void on_NextButton_pressed()
```

Procedemos a explicar las funciones que son acciones que el jugador hace, lo primero que se mostrará apenas se presione el botón de NewGame del menú, será la opción de escoger con que bando quieres jugar, si escoges Daenerys se llamara a la función `on_Band0Button_pressed()`, la cual indica que se eligió llamando a la siguiente función del flujo: `NewGame(bool band)` pasándole `true` como argumento, en caso de que se elija JonSnow se llama a la función `on_Band1Button_pressed()`, la cual llamará a `NewGame(bool band)` con argumento `false`.

Otra de los botones con los que el jugador puede interactuar son dos que se muestran de forma permanente, está el botón Pass(`on_PassButton_pressed()`), el cual indica que se pasa la ronda modificando una variable global llamada `playerPass` y el otro es Menu(`on_MenuButton_Pressed`), el cual llama a la función `EndGame()` y termina el juego mostrando el menú.

Los siguientes botones son los que se muestran en un cartel utilizando la función `DisplayInfo(int a, int b, string result, bool goMenu)`, que muestra un resumen de la ronda una vez esta ha acabado, en sus botones encontramos el botón Next (`on_NextButton_pressed()`) la cual si todavía no han acabado las vidas de ninguno de los dos jugadores, pasa a la siguiente ronda, en caso de que una de las vidas sea cero, llama a `DisplayInfo(int a, int b, string result, bool goMenu)` con argumento `true`, lo que hará una diferencia en el Next que lo que hará será seguir llamando a `DisplayInfo(int a, int b, string result, bool goMenu)` con argumento `true` y el siguiente botón de nombre Menu o GiveUp si existe una próxima ronda, el método que respalda este botón es `on_ExitButton_pressed()` el cual enviará al jugador al menú.

Los botones que quedan son de apoyo para las `EffectCard`, pues si el subtipo de la `EffectCard` es Support, el jugador debe ser capaz de seleccionar que fila afectar; para esto hay un botón a la izquierda de las filas del jugador, los métodos tienen por nombre `on_MeleeSupport_pressed()`, `on_MiddleSupport_pressed()`, `on_SiegeSupport_pressed()`, la función que cumplen los tres, es la de mover la carta de tipo `EffectCard` a la posición debida y pasar a una variable global de tipo `int` llamada `moved` cuál es la fila que le corresponde a ese botón para que luego cuando se ejecute el poder, saber sobre que fila usarlo.

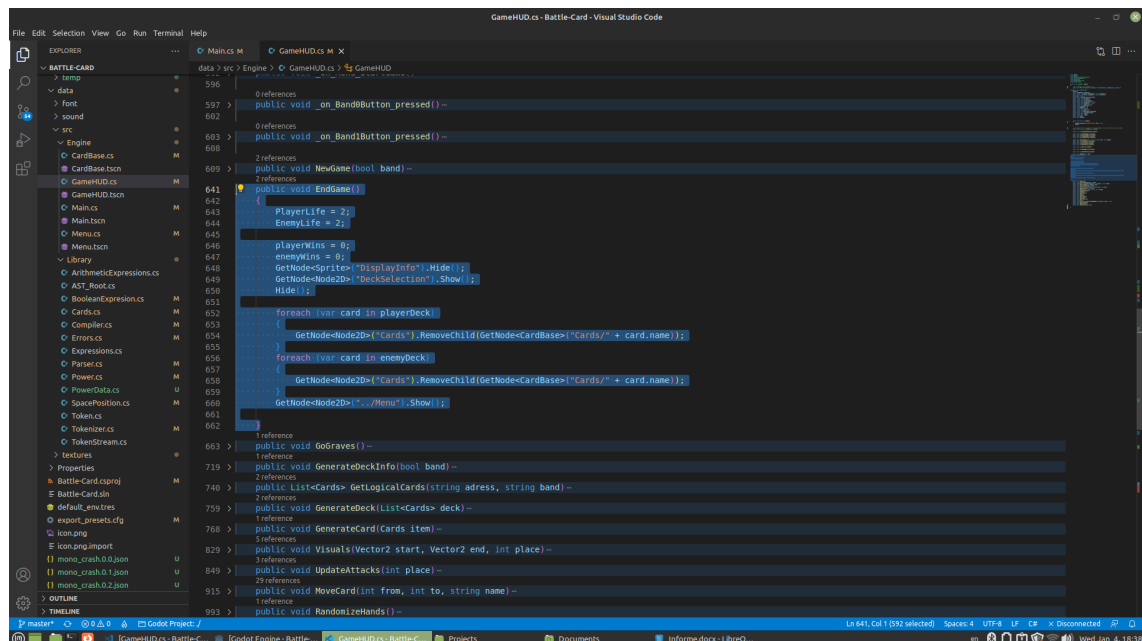


```
48 {
49     //Time dentro de si un grupo de condicionales que definen que hacer en cada fase del juego.
50     if (phase == (int)Phase.PlayerTurn) -
51     else if (phase == (int)Phase.EnemyTurn) -
52     else if (phase == (int)Phase.PlayerWaiting) -
53     else if (phase == (int)Phase.EnemyWaiting) -
54     else if (phase == (int)Phase.Results) -
55 }
56
57 0 references
58 public void _on_ChangeEnemyTimer_timeout() -
59
60 0 references
61 public void _on_MeleeSupport_pressed() -
62 {
63     if (ready)
64     {
65         moved = 4;
66         ready = false;
67         MoveCard(2, 20, currentName);
68     }
69 }
70
71 0 references
72 public void _on_MiddleSupport_pressed() -
73
74 0 references
75 public void _on_SiegeSupport_pressed() -
76
77 0 references
78 public void _on_HideInfoTimer_timeout() -
79
80 2 references
81 public void DisplayInfo(int a, int b, string result, bool goMenu) -
82
83 0 references
84 public void _on_GiveUpButton_pressed() -
85
86 0 references
87 public void _on_ExitButton_pressed() -
88
89 0 references
90 public void _on_PassButton_pressed() -
91
92 0 references
93 public void _on_NextButton_pressed() -
94
95 0 references
96 public void _on_Menu_StartGame() -
97
98 0 references
99 public void _on_Band0Button_pressed() -
100
101 0 references
102 public void _on_Band1Button_pressed() -
103 }
```

El resto de funciones se dividen en las que realizan las acciones de los poderes, las que recolectan la información inicial y sirven para crear las cartas y las que afectan a la parte visual.

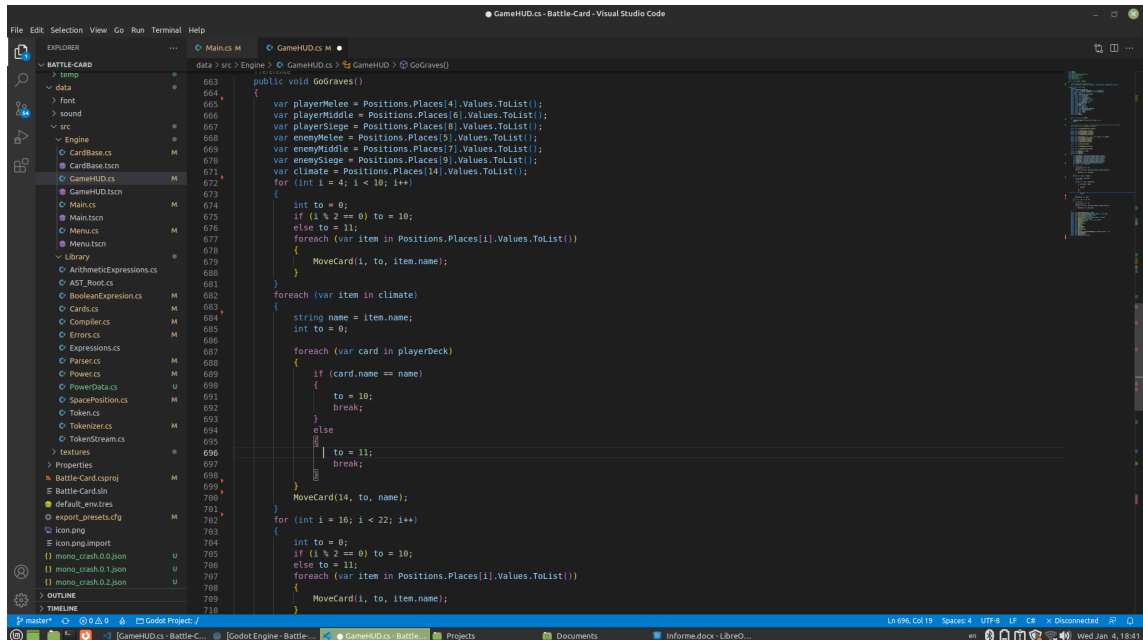
**Comencemos por las que sirven para crear las cartas pues son las primeras que se ejecutan.**

**-EndGame():** Restaura las vidas del jugador y del enemigo por si se vuelve a iniciar un juego, se restauran el número de rondas ganadas de ambos, y se procede a eliminar todas las cartas que existen en el juego, acto seguido se vuelve al menú.

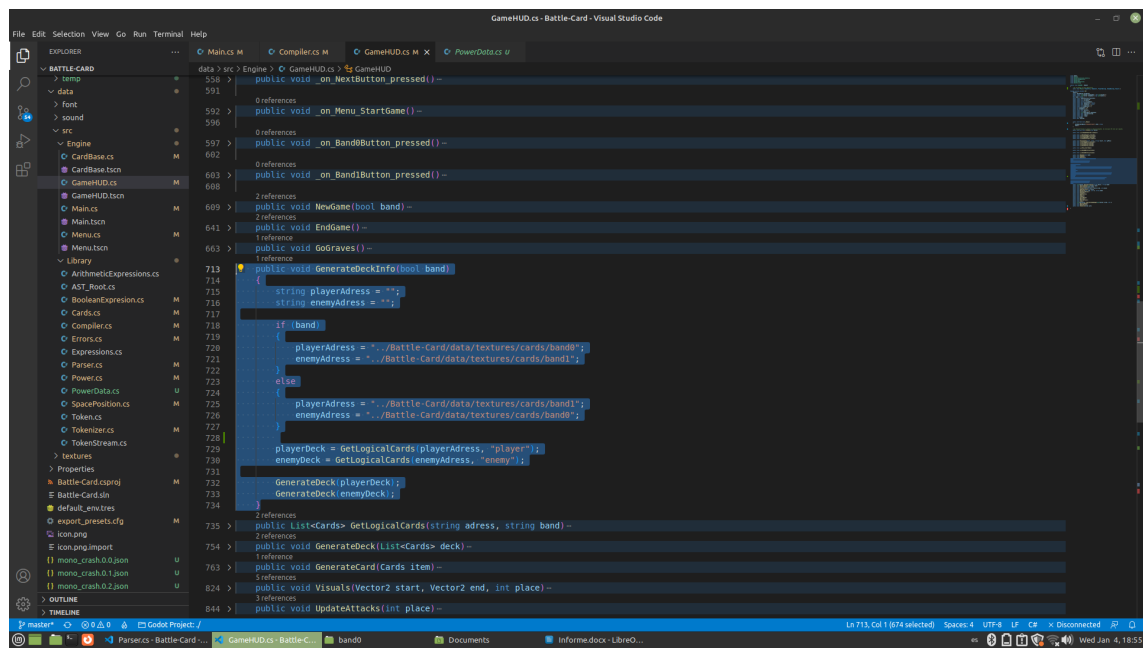


```
641 public void EndGame()
642 {
643     PlayerLife = 2;
644     EnemyLife = 2;
645
646     playerWins = 0;
647     enemyWins = 0;
648     GetNode<Sprite>("DisplayInfo").Hide();
649     GetNode<Node2D>("DeckSelector").Show();
650     Hide();
651
652     foreach (var card in playerDeck)
653     {
654         GetNode<Node2D>("Cards").RemoveChild(GetNode<CardBase>("Cards/" + card.name));
655     }
656     foreach (var card in enemyDeck)
657     {
658         GetNode<Node2D>("Cards").RemoveChild(GetNode<CardBase>("Cards/" + card.name));
659     }
660     GetNode<Node2D>("../Menu").Show();
661
662     0 references
663 public void GoGraves() -
664
665 0 references
666 public void GenerateDeckInfo(bool band) -
667
668 2 references
669 public List<Cards> GetLogicalCards(string address, string band) -
670
671 2 references
672 public void GenerateDeck(List<Cards> deck) -
673
674 0 references
675 public void GenerateCard(Cards item) -
676
677 5 references
678 public void Visuals(Vector2 start, Vector2 end, int place) -
679
680 3 references
681 public void UpdateAttacks(int place) -
682
683 2 references
684 public void MoveCard(int from, int to, string name) -
685
686 1 reference
687 public void RandomizeHands() -
688 }
```

-**GoGraves()**: Se crean listas de cartas de las distintas posiciones donde pueden existir cartas en el campo, y luego se procede a ir moviéndolas todas a sus respectivos cementerios.



-**GenerateDeckInfo(bool band)**: Éste método se encarga en función de la variable band de decidir cuáles son las direcciones donde se encuentran las cartas del jugador y del enemigo. Luego procede a inicializar las cartas lógicas llamando al método **GetLogicalCards(string adress)** el cual devuelve un **List<Cards>** que será almacenado en **playerDeck** y **enemyDeck**, ambas variables globales de **GameHUD**. Por último se procede a crear las cartas visuales utilizando **GenerateDeck(List<Cards>)**.



-**GetLogicalCards**(string adress, string band): Primero creamos un Tokenizer, luego procedemos a obtener las direcciones de los textos y almacenarlas en un array, entonces por cada dirección se crea una lista con todos los **Tokens** del texto, se crea un **TokenStream**, se inicializa un objeto de tipo **Parser**, y se parsea la carta a partir de los tokens, luego a esta carta se le agrega el bando al que pertenece y por último se agrega a **List<Cards> answer** que es la que se devolverá en la instrucción **return**.

```

713 > public void GenerateDeckInfo(bool band)
714 > {
715 >     public List<Cards> GetLogicalCards(string adress, string band)
716 >     {
717 >         Tokenizer a = Compiler.Lexically;
718 >         string[] texts = System.IO.Directory.GetFiles(adress, "*.txt");
719 >         List<Cards> answer = new List<Cards>();
720 >         foreach (var item in texts)
721 >         {
722 >             string text = System.IO.File.ReadAllText(item);
723 >             List<Token> MyList = (List<Token>)a.GetTokens(text, item);
724 >             TokenStream b = new TokenStream(MyList);
725 >             Parser c = new Parser(b);
726 >             Cards card = c.ParseCard();
727 >             card.band = band;
728 >             answer.Add(card);
729 >         }
730 >         return answer;
731 >     }
732 > }
733 >
734 > public void GenerateDeck(List<Cards> deck)
735 > {
736 >     public void GenerateCard(Cards item)
737 >     {
738 >         public void Visuals(Vector2 start, Vector2 end, int place)
739 >         {
740 >             public void UpdateAttacks(int place)
741 >             {
742 >             }
743 >             public void MoveCard(int from, int to, string name)
744 >             {
745 >             }
746 >             public void RandomizeHands()
747 >             {
748 >             }
749 >             public void EnemyIA()
750 >             {
751 >             }
752 >             public void UpdateLife()
753 >             {
754 >             }
755 >             public void Reborn()
756 >             {
757 >             }
758 >             public void Summon()
759 >             {
760 >             }
761 >         }
762 >     }
763 > }
764 >
765 >
766 >
767 >
768 >
769 >
770 >
771 >
772 >
773 >
774 >
775 >
776 >
777 >
778 >
779 >
780 >
781 >
782 >
783 >
784 >
785 >
786 >
787 >
788 >
789 >
790 >
791 >
792 >
793 >
794 >
795 >
796 >
797 >
798 >
799 >
800 >
801 >
802 >
803 >
804 >
805 >
806 >
807 >
808 >
809 >
810 >
811 >
812 >
813 >
814 >
815 >
816 >
817 >
818 >
819 >
820 >
821 >
822 >
823 >
824 >
825 >
826 >
827 >
828 >
829 >
830 >
831 >
832 >
833 >
834 >
835 >
836 >
837 >
838 >
839 >
840 >
841 >
842 >
843 >
844 >
845 >
846 >
847 >
848 >
849 >
850 >
851 >
852 >
853 >
854 >
855 >
856 >
857 >
858 >
859 >
860 >
861 >
862 >
863 >
864 >
865 >
866 >
867 >
868 >
869 >
870 >
871 >
872 >
873 >
874 >
875 >
876 >
877 >
878 >
879 >
880 >
881 >
882 >
883 >
884 >
885 >
886 >
887 >
888 >
889 >
890 >
891 >
892 >
893 >
894 >
895 >
896 >
897 >
898 >
899 >
900 >
901 >
902 >
903 >
904 >
905 >
906 >
907 >
908 >
909 >
910 >
911 >
912 >
913 >
914 >
915 >
916 >
917 >
918 >
919 >
920 >
921 >
922 >
923 >
924 >
925 >
926 >
927 >
928 >
929 >
930 >
931 >
932 >
933 >
934 >
935 >
936 >
937 >
938 >
939 >
940 >
941 >
942 >
943 >
944 >
945 >
946 >
947 >
948 >
949 >
950 >
951 >
952 >
953 >
954 >
955 >
956 >
957 >
958 >
959 >
960 >
961 >
962 >
963 >
964 >
965 >
966 >
967 >
968 >
969 >
970 >
971 >
972 >
973 >
974 >
975 >
976 >
977 >
978 >
979 >
980 >
981 >
982 >
983 >
984 >
985 >
986 >
987 >
988 >
989 >
990 >
991 >
992 >
993 >
994 >
995 >
996 >
997 >
998 >
999 >
1000 >

```

-**GenerateCard**(Cards item): Se encarga de generar la carta visual, si es tipo unidad, efecto o líder, se crea un objeto de tipo **CardBase**, se le pasan las propiedades de **item** y luego se agrega como hijo a la escena **GameHUD**: o sea, se crea un nodo que representa a la carta visual, lo cual nos permitirá manipularla utilizando el nombre lógico de la carta, pues se le pone como nombre al nodo: el nombre de la carta.

```

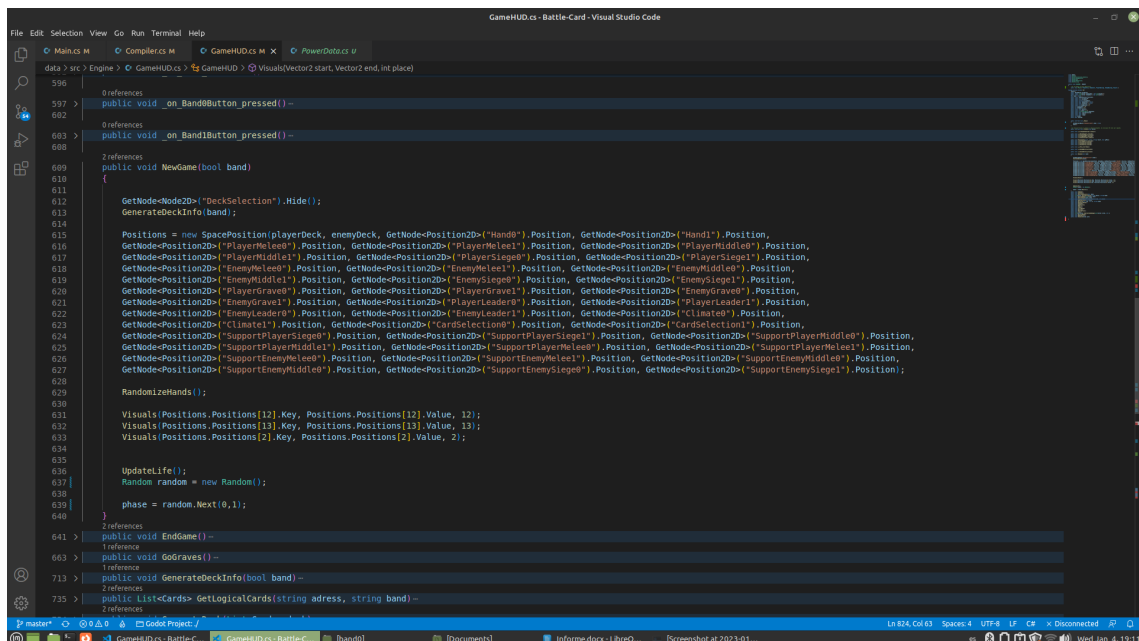
733 > public List<Cards> GetLogicalCards(string adress, string band)
734 > {
735 >     public void GenerateDeck(List<Cards> deck)
736 >     {
737 >         public void GenerateCard(Cards item)
738 >         {
739 >             if (item is UnitCard)
740 >             {
741 >                 var newItem = UnitCardItem;
742 >                 var card = (CardBase)CardScene.Instance();
743 >                 card.Name = newItem.name;
744 >                 card.Attack = newItem.damage.ToString();
745 >                 card.Place = newItem.position;
746 >                 card.Phase = newItem.phase;
747 >                 card.Route = newItem.imagePath;
748 >                 card.Type = newItem.type;
749 >                 card.Power = newItem.power.Name;
750 >                 card.GenerateCard();
751 >                 GetNode<Node2D>("Cards").AddChild(card, true);
752 >                 GetNode<MarginContainer>("Cards/CardBase").Name = newItem.name;
753 >             }
754 >             else if (item is LeaderCard)
755 >             {
756 >                 var newItem = LeaderCardItem;
757 >                 var card = (CardBase)CardScene.Instance();
758 >                 card.Name = newItem.name;
759 >                 card.Phase = newItem.phase;
760 >                 card.Route = newItem.imagePath;
761 >                 card.Type = newItem.type;
762 >                 card.Power = newItem.power.Name;
763 >                 card.GenerateCard();
764 >                 GetNode<Node2D>("Cards").AddChild(card, true);
765 >                 GetNode<MarginContainer>("Cards/CardBase").Name = newItem.name;
766 >             }
767 >             else if (item is EffectCard)
768 >             {
769 >             }
770 >         }
771 >     }
772 > }
773 >
774 >
775 >
776 >
777 >
778 >
779 >
780 >
781 >
782 >
783 >
784 >
785 >
786 >
787 >
788 >
789 >
790 >
791 >
792 >
793 >
794 >
795 >
796 >
797 >
798 >
799 >
800 >
801 >
802 >
803 >
804 >
805 >
806 >
807 >
808 >
809 >
810 >
811 >
812 >
813 >
814 >
815 >
816 >
817 >
818 >
819 >
820 >
821 >
822 >
823 >
824 >
825 >
826 >
827 >
828 >
829 >
830 >
831 >
832 >
833 >
834 >
835 >
836 >
837 >
838 >
839 >
840 >
841 >
842 >
843 >
844 >
845 >
846 >
847 >
848 >
849 >
850 >
851 >
852 >
853 >
854 >
855 >
856 >
857 >
858 >
859 >
860 >
861 >
862 >
863 >
864 >
865 >
866 >
867 >
868 >
869 >
870 >
871 >
872 >
873 >
874 >
875 >
876 >
877 >
878 >
879 >
880 >
881 >
882 >
883 >
884 >
885 >
886 >
887 >
888 >
889 >
890 >
891 >
892 >
893 >
894 >
895 >
896 >
897 >
898 >
899 >
900 >
901 >
902 >
903 >
904 >
905 >
906 >
907 >
908 >
909 >
910 >
911 >
912 >
913 >
914 >
915 >
916 >
917 >
918 >
919 >
920 >
921 >
922 >
923 >
924 >
925 >
926 >
927 >
928 >
929 >
930 >
931 >
932 >
933 >
934 >
935 >
936 >
937 >
938 >
939 >
940 >
941 >
942 >
943 >
944 >
945 >
946 >
947 >
948 >
949 >
950 >
951 >
952 >
953 >
954 >
955 >
956 >
957 >
958 >
959 >
960 >
961 >
962 >
963 >
964 >
965 >
966 >
967 >
968 >
969 >
970 >
971 >
972 >
973 >
974 >
975 >
976 >
977 >
978 >
979 >
980 >
981 >
982 >
983 >
984 >
985 >
986 >
987 >
988 >
989 >
990 >
991 >
992 >
993 >
994 >
995 >
996 >
997 >
998 >
999 >
1000 >

```



-**GenerateDeck**(List<Cards> **cards**): No es más que iterar por cada elemento de la lista llamando al método **GenerateCard**(Cards **item**).

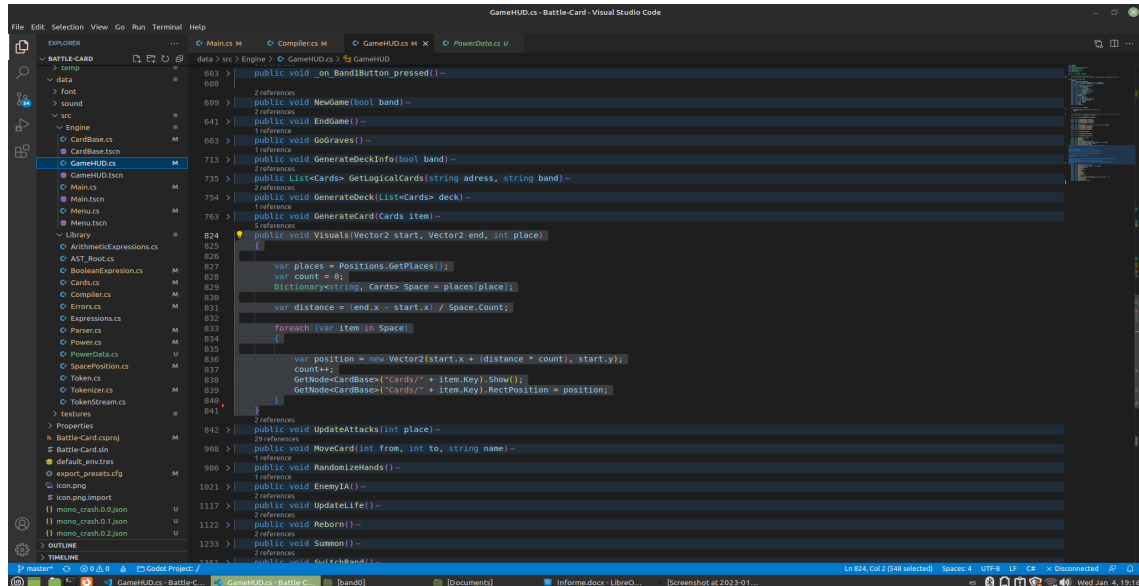
-**NewGame**(bool **band**): Lo primero que realiza es llamar al método **GenerateDeckInfo**(**band**), luego inicializa el objeto de tipo **SpacePosition** **Positions** que es una variable global de **GameHUD**, acto seguido se llama al método **RandomizeHands**(), y se llama al método **Visuals**(Vector2 **a**, Vector2 **b**, int **n**) para mostrar las cartas que se encuentran en la mano. Luego se llama a **UpdateLife()** y por último se inicializa el juego dándole un valor a **phase**(0 o 1).



```
596 0references
597 > public void on_BandButton_pressed()--
598 0references
599 > public void on_BandButton_pressed()--
600 2references
601 public void NewGame(bool band)
602 {
603     GetNode<Node2D>("DeckSelection").Hide();
604     GenerateDeckInfo(band);
605
606     Positions = new SpacePosition(playerDeck, enemyDeck, GetNode<Position2D>("Hand0").Position, GetNode<Position2D>("Hand1").Position,
607     GetNode<Position2D>("PlayerMelee0").Position, GetNode<Position2D>("PlayerMelee1").Position, GetNode<Position2D>("PlayerMiddle0").Position,
608     GetNode<Position2D>("PlayerMiddle1").Position, GetNode<Position2D>("PlayerSiege0").Position, GetNode<Position2D>("PlayerSiege1").Position,
609     GetNode<Position2D>("EnemyMelee0").Position, GetNode<Position2D>("EnemyMelee1").Position, GetNode<Position2D>("EnemyMiddle0").Position,
610     GetNode<Position2D>("EnemyMiddle1").Position, GetNode<Position2D>("EnemySiege0").Position, GetNode<Position2D>("EnemySiege1").Position,
611     GetNode<Position2D>("PlayerGravel").Position, GetNode<Position2D>("PlayerGravel1").Position, GetNode<Position2D>("EnemyGravel").Position,
612     GetNode<Position2D>("EnemyGravel1").Position, GetNode<Position2D>("PlayerLeader0").Position, GetNode<Position2D>("PlayerLeader1").Position,
613     GetNode<Position2D>("EnemyLeader0").Position, GetNode<Position2D>("EnemyLeader1").Position, GetNode<Position2D>("Climate0").Position,
614     GetNode<Position2D>("Climate1").Position, GetNode<Position2D>("CardSelection0").Position, GetNode<Position2D>("CardSelection1").Position,
615     GetNode<Position2D>("SupportPlayerSiege0").Position, GetNode<Position2D>("SupportPlayerSiege1").Position, GetNode<Position2D>("SupportPlayerMiddle0").Position,
616     GetNode<Position2D>("SupportPlayerMiddle1").Position, GetNode<Position2D>("SupportPlayerMelee0").Position, GetNode<Position2D>("SupportPlayerMelee1").Position,
617     GetNode<Position2D>("SupportEnemyMelee0").Position, GetNode<Position2D>("SupportEnemyMelee1").Position, GetNode<Position2D>("SupportEnemyMiddle0").Position,
618     GetNode<Position2D>("SupportEnemyMiddle1").Position, GetNode<Position2D>("SupportEnemySiege0").Position, GetNode<Position2D>("SupportEnemySiege1").Position);
619
620     RandomizeHands();
621
622     Visuals(Positions.Positions[12].Key, Positions.Positions[12].Value, 12);
623     Visuals(Positions.Positions[13].Key, Positions.Positions[13].Value, 13);
624     Visuals(Positions.Positions[2].Key, Positions.Positions[2].Value, 2);
625
626     UpdateLife();
627     Random random = new Random();
628     phase = random.Next(0,1);
629 }
630 2references
631 > public void EndGame()--
632 1reference
633 > public void GoGraves()--
634 1reference
635 > public void GenerateDeckInfo(bool band)--
636 2references
637 > public List<Cards> GetLogicalCards(string address, string band)--
638 2references
```

**Ahora vendrían los métodos relacionados con los visuales:**

-**Visuals(Vector2 start, Vector2 end, int place)**: Éste método se encarga de actualizar el lugar geométrico de las cartas que se encuentran en una posición(**place**) de **Positions.Places**(Variable Global de tipo **SpacePositions**), para ello se le pasan los **Vector2** que indican el inicio y el final del espacio visual donde deben estar. Este algoritmo por cada carta que haya en dicho lugar, calcula una distancia que es a la que deben estar una de otras para estar repartidas equitativamente en dicho lugar y procede a modificar las posiciones de las cartas.



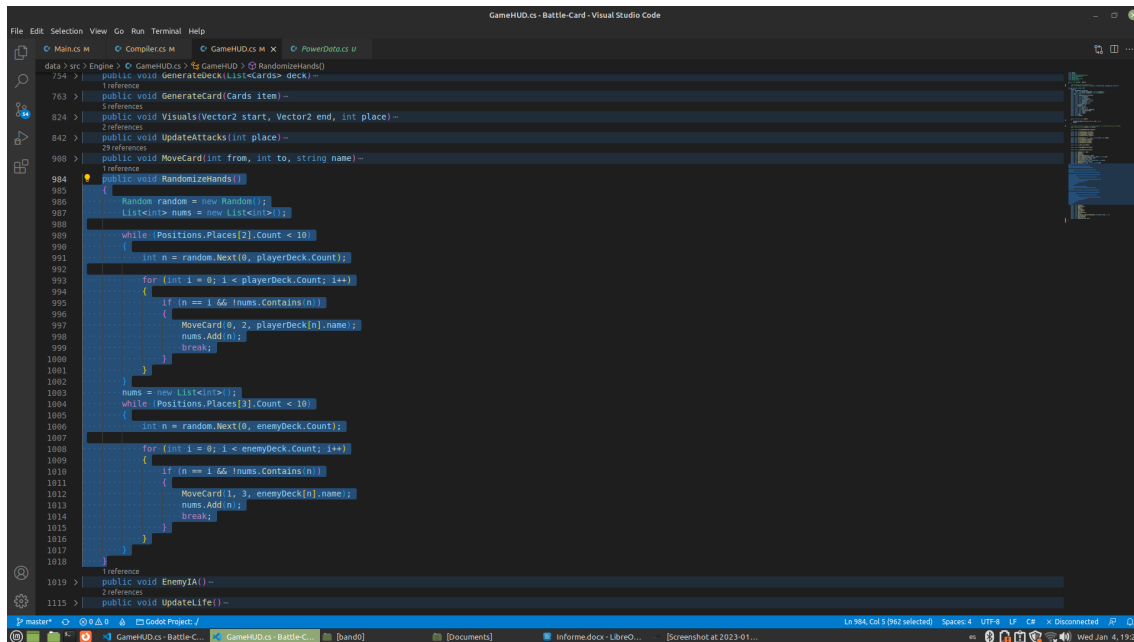
-**UpdateAttacks(int place)**: En dependencia del lugar que represente place, se ira por cada una de las cartas que estén en ese lugar acumulando sus ataques y colocándolos en la parte visual que muestra la suma total de los ataques de las cartas que están en la fila, luego se actualiza la suma global de todos los ataques de todas las filas sumando el ataque global de cada fila.

-**UpdateLife()**: Un método realmente simple que se encarga de actualizar las vidas restantes del jugador y del enemigo.

**Finalmente solo nos quedan los métodos que realizan acciones en el juego, muchos apoyándose en los visuales .**

-**MoveCard(int from, int to, string name)**: Lo primero que hace es verificar si la carta se encuentra en el lugar desde donde hay que moverla, en caso afirmativo se procede a remover en la parte lógica(**SpacePosition**) de ese lugar y a colocar en el nuevo, luego si el lugar hacia donde se movió(**to**) es la mano del jugador, a la parte visual de la carta movida(**CardBase**) se le pone en true la variable **OnHand**, si se movió hacia la posición de selección, se le pone en true la variable **Ready**. Luego si el lugar al que se movió la carta es alguno del campo, y proviene de la mano o del cementerio, tanto del jugador como del enemigo, se ejecuta el poder de dicha carta en caso de tener. Por último si el lugar desde donde se movió es alguno de los representados visualmente, se actualiza el visual llamando al método **Visuals**.

-**RandomizeHands()**: Éste método se encarga de mover 10 cartas aleatorias desde los decks a las manos de sus respectivos dueños.



```
data > src > Engine > GameHUD.cs > %GameHUD% > RandomizeHands()
734 > public void GenerateDeck(List<Cards> deck) -
1 reference
763 > public void GenerateCard(Cards item) -
3 references
824 > public void Visuals(Vector2 start, Vector2 end, int place) -
2 references
842 > public void UpdateAttacks(int place) -
29 references
908 > public void MoveCard(int from, int to, string name) -
1 reference
984 > public void RandomizeHands()
985 {
986     Random random = new Random();
987     List<int> nums = new List<int>();
988     while (Positions.Places[2].Count < 10)
989     {
990         int n = random.Next(0, playerDeck.Count);
991         for (int i = 0; i < playerDeck.Count; i++)
992         {
993             if (n == i && !nums.Contains(n))
994             {
995                 MoveCard(0, 2, playerDeck[n].name);
996                 nums.Add(n);
997                 break;
998             }
999         }
1000     }
1001     nums = new List<int>();
1002     while (Positions.Places[3].Count < 10)
1003     {
1004         int n = random.Next(0, enemyDeck.Count);
1005         for (int i = 0; i < enemyDeck.Count; i++)
1006         {
1007             if (n == i && !nums.Contains(n))
1008             {
1009                 MoveCard(1, 3, enemyDeck[n].name);
1010                 nums.Add(n);
1011                 break;
1012             }
1013         }
1014     }
1015 }
1016
1019 > 1 reference
public void EnemyAI() -
2 references
1115 > public void UpdateLife() -
```

-**Reborn()**: Éste método es llamado cuando en la cola de poderes, el primero es de tipo **RebornPower**, lo que haría es verificar si la intención del **RebornPower** es seleccionar cartas para revivir, o si simplemente pasaron una lista de nombres para revivir en caso de que se encuentren ahí. En el primer caso, se verifica si estamos en la fase de ejecución de poderes del jugador, en cuyo caso se posicionan las cartas del cementerio en la posición de selección para que el jugador elija, cada vez que se seleccione una carta el contador de cantidad de cartas máximas a revivir disminuirá, una vez llegue a cero, se terminará la ejecución del poder. En el caso de que no sea el turno del jugador, sino del enemigo, se selecciona una carta aleatoria para revivir. Si lo que se pasa es una lista de nombres para revivir, simplemente se verifica si es el turno del jugador si existen dichas cartas para revivir en el cementerio y viceversa, si es el turno del enemigo, se verifica que existan, una vez hecho se reviven, o sea, se llama al método **MoveCard**.

-**Summon()**: Éste método es llamado cuando en la cola de poderes, el primero es de tipo **SummonPower**, lo que haría es verificar cuál es la intención del poder, si es seleccionar, o si es pasar una lista de nombres para invocar desde el deck. En el primer caso se verifica si es el turno del jugador, en caso de que así sea, se posicionan las cartas en la posición de selección, cada vez que el usuario selecciona una carta, disminuye el contador del máximo de cartas a revivir, una vez llega a cero, termina la ejecución del poder, si no es el turno del jugador, se eligen cartas aleatoria para invocar. En caso de que se haya pasado una lista de nombres para invocar, simplemente se procede a ir por el deck del jugador si es su turno o del enemigo si es el de él, y verificar que dicha carta esté presente en el deck, de así ser, simplemente se mueve usando **MoveCard** a su posición en el campo.

-**SwitchBand()**: Éste método es llamado cuando en la cola de poderes, el primero es de tipo **SwitchBandPower**, lo que hace es buscar la carta por todo el campo, una vez encontrada se procede a colocar en su homólogo en el campo rival al que se encontró utilizando el método **MoveCard**.

-**Destroy()**: Éste método es llamado cuando en la cola de poderes, el primero es de tipo **DestroyPower**, el cual puede tener tres identificadores, el primero es que la carta a destruir sea de libre elección, que se haya dado una lista de nombres con cartas a destruir si están en el campo, y por último si se quiere destruir una fila entera. En el

primer caso, si es el turno del jugador, se pone true en la variable **Ready** de todas las **CardBase** que se encuentren en el campo, en cuyo caso las cartas estarán en espera para que cuando sean seleccionadas, moverse a su cementerio y disminuir el contador de la cantidad máxima de cartas definidas para destruir que una vez llegue a cero, terminará la ejecución del poder, en caso de que no sea el turno del jugador, se seleccionan el número definido de cartas a destruir de forma aleatoria, preferiblemente del campo del jugador. En el caso de que se haya pasado una lista de nombres con cartas a destruir, se verifica que estén en el campo y se envían a su respectivo cementerio, en caso de que sea una fila, simplemente se itera por dicha fila enviando las cartas a su respectivo cementerio.

**-ModifyAttack()**: Éste método es llamado cuando en la cola de poderes, el primero es de tipo **ModifyAttackPower**, el cuál puede tener cuatro identificadores, el primero es modificar el ataque de toda una fila en cierto valor, en cuyo caso se itera por las cartas de la fila modificando el ataque en dicha cantidad. El segundo es que se pase una lista con los nombres de las cartas a modificar, en cuyo caso se verifica si la carta está presente en el campo y de ser así, se modifica su daño en cierto valor. El tercer caso es cuando se invoca una **EffectCard** de tipo *Support* que modifica el daño de toda una fila, si es el turno del jugador, se espera a que este seleccione una y se ejecuta el poder en la fila que seleccionó. En caso de ser el turno enemigo, se selecciona la fila con mayor cantidad de cartas, o en caso de que no tenga ninguna carta en el campo, se elegirá siempre la melee. El cuarto caso, es cuando se da elección libre al jugador para elegir que carta quiere modificar, en cuyo caso se pone en true la variable **Ready** de todas las **CardBase** del campo si es el turno del jugador, en caso contrario, se elige al azar la cantidad de cartas definidas a modificar.

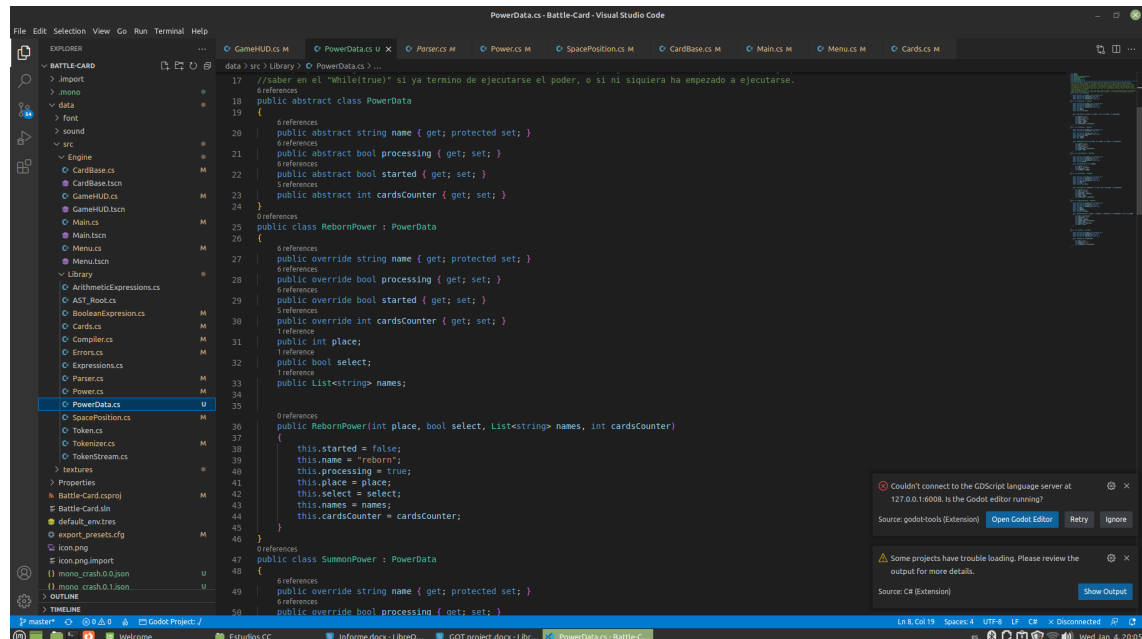
**-Draw()**: Éste método es llamado cuando en la cola de poderes, el primero es de tipo **DrawPower**, el cuál si es el turno del jugador, selecciona un número definido de cartas aleatorias y las coloca en su mano desde su deck y lo mismo en caso de que sea el turno del enemigo, solo que será a su respectiva mano desde su respectivo deck. Utilizando por supuesto el método **MoveCard**.

**-RestoreReady()**: Se encarga de poner en false la variable **Ready** de todas las **CardBase** que hay en el campo. Se utilizaría después de que el jugador terminase de seleccionar cartas como consecuencia de alguna habilidad para garantizar que no siga seleccionando cartas una vez haya terminado el poder y evitar posibles errores.

Con esto terminaría la explicación de las clases que componen el motor.

**Procedemos a mostrar las clases de Library, o sea: la lógica.**

La siguiente clase sería **PowerData**: está pensada para poder organizar la información de los poderes y poder utilizarlos de uno en uno.



La clase **SpacePosition** es una abstracción para representar los lugares lógicos y además tendrá almacenadas las posiciones de la parte visual.

### Informe del "Compilador"

El proceso de creación de cartas en nuestro juego es realizado a través de una herramienta que hemos desarrollado, una especie de mini-compilador que se encarga de leer y procesar los datos correspondientes a cada carta y utilizarlos para crear una carta dentro del juego (esto se hace a través de un pequeño lenguaje que explicaremos detalladamente mas adelante).

**\*Como funciona:**

Este proceso consta de dos partes fundamentales, una primera parte en la cual procedemos a leer el texto y convertirlos en objetos que tengan sentido dentro de nuestro lenguaje y otra en la cual le daremos un sentido a estos objetos dentro de nuestro juego (esta es una explicación muy superficial del proceso) El primer momento lo definiremos como tokenizacion y el segundo momento como parseo.

**\*Cartas:**

Las cartas son los objetos indispensable de todo juego de cartas, y en nuestro caso no es diferente. Existen diferentes tipos de cartas en nuestro juego, cada una con sus especificidades y cada una siendo representada por un tipo de dato. A su vez todas las cartas poseen determinadas características comunes y ellas son recogidas también en un tipo de dato, el tipo de dato Card (que es un tipo de dato abstracto)

El nombre y el tipo hacen referencia al nombre y tipo que posee la carta; mientras que la ruta indica la dirección de la imagen asociada a la carta y el poder es una abstracción que poseen las cartas que les permitirán interactuar con el estado del juego, como explicaremos mas adelante.

Las cartas de tipo unidad, además de poseer las propiedades generales poseen un ataque, una frase del personaje ficticio al que representa y una posición que puede ser “Cuerpo a cuerpo”, “A distancia” o “Asedio”. Estas cartas son las mas comunes.

Las cartas de tipo líder, además de las propiedades básicas poseen una frase solamente. En este caso la frase cumple la misma función que en el caso de las cartas de tipo unidad. Cada bando solo puede poseer una carta líder y estas son trasladadas en cuanto empieza el juego a posiciones especiales dentro del campo.

Las cartas de tipo Efecto, además de las propiedades generales poseen una posición en el campo. Estas cartas tienen por objetivo servir de apoyo a las cartas aliadas ya sea incrementando la efectividad de nuestras cartas o disminuyendo la de las cartas enemigas. Las posiciones que pueden ocupar son “Clima” o “Soporte”. En el caso de las cartas de clima se trasladarán hacia una región especial. En el caso de las cartas de soporte su posición será determinada a futuro por el usuario, pero estarán limitadas a tres zonas especiales de apoyo correspondientes con las zonas que ocupan las cartas de tipo unidad.



Los poderes son una abstracción que permite a las cartas interactuar con el estado del juego (fuera del tiempo de compilación en el cual la carta y el poder son creados) Cada poder posee un nombre, un conjunto de condiciones que se deben cumplir para ejecutar el poder, un conjunto de instrucciones a ejecutar, un método para revisar que se cumplan todas las instrucciones y un método para ejecutar todas las condiciones.

#### \*Condiciones:

Las condiciones son simplemente una abstracción que contiene una expresión booleana y un método que evalúa dicha expresión y se encarga de determinar la veracidad de la condición en dependencia de la veracidad de la expresión.

#### \*Instrucciones:

Las instrucciones son objetos que poseen un nombre y un conjunto de objetos agrupados en un IEnumerable. La idea es simple, el nombre de la instrucción se corresponde con una funcionalidad del juego y el IEnumerable de objetos con los parámetros que recibe la función asociada a dicha funcionalidad. Estas funcionalidades están definidas en el lenguaje.

##### \*Instruccion Destroy:

La instrucción Destroy hace referencia a la funcionalidad de destruir cartas. Puede recibir argumentos para destruir las cartas en una zona determinada del campo (ya sea el campo enemigo o el propio), un numero de cartas a seleccionar que serán destruidas o un conjunto de cartas especificas a destruir.

##### \*Instruccion Reborn:

La instrucción Reborn hace referencia a la funcionalidad de revivir cartas. Puede recibir argumentos para revivir un numero de cartas a seleccionar por el usuario o un conjunto de cartas especificas a revivir.

##### \*Instruccion Summon:

La instrucción Summon hace referencia a la funcionalidad de invocar cartas desde el deck al campo. Puede recibir argumentos para invocar un numero de cartas a seleccionar por el usuario o un conjunto de cartas especificas a invocar.

##### \*Instruccion Draw:

La instrucción Draw hace referencia a la funcionalidad de robar cartas desde el deck a la mano. Puede recibir como argumento un numero para invocar esa cantidad de cartas de modo aleatorio.

##### \*Instruccion ModifyAttack:

La instrucción ModifyAttack hace referencia a la funcionalidad de alterar el ataque de las cartas de tipo unidad. Recibe la cantidad en la cual vamos a modificar el ataque y ademas:

Un numero que indica la cantidad de cartas a escoger por el usuario para modificar o una zona del campo a la cual modificar el ataque de todas sus cartas o un conjunto de cartas especificas a modificar.

##### \*Instruccion SwitchBand:

La instrucción SwitchBand hace referencia a la funcionalidad de cambiar la posición de una carta y recibe como parámetros la carta a cambiar y el lugar de destino, que puede ser la posición MyField o EnemyField.

#### \*Expresiones:

Las expresiones son abstracciones que poseen un valor, un tipo y un método para evaluarse y pueden ser de dos tipos: Expresiones booleanas o Expresiones aritméticas. En el caso de las Expresiones booleanas la evaluación devuelve un valor booleano (verdadero o falso) y en el caso de las expresiones aritméticas. la evaluación devuelve un valor numérico. (La clase Expresión es una clase abstracta)

##### \*Expresiones booleanas:

Las Expresiones booleanas son una subclase abstracta de la clase Expresión donde el único cambio es que su tipo queda asignado como un tipo booleano. De ella heredan dos tipos de expresiones booleanas, las Expresiones booleanas binarias y las Expresiones booleanas unarias. (ambas son abstractas)

**\*Expresiones booleanas binarias:**

Las Expresiones booleanas binarias heredan de las expresiones booleanas poseen dos expresiones booleanas, una expresión izquierda y una expresión derecha (de ahí el nombre de binaria); y su evaluación depende de la evaluación previa de dichas expresiones. Existen tres ejemplos de expresiones booleanas binarias: La expresión de conjunción lógica, la expresión de disyunción lógica y el comparador binario. Todas serán detalladas mas adelante.

**\*Expresión booleana binaria And:**

La expresión booleana de conjunción (And) hereda de las expresiones booleanas binarias

y recibe como argumentos dos expresiones booleanas. La evaluación de esta expresión booleana es verdadera si y solo si se cumplen ambas expresiones binarias recibidas.

**\*Expresión booleana binaria Or:**

La expresión booleana de disyunción (Or) hereda de las expresiones booleanas binarias

y recibe como argumentos dos expresiones booleanas. La evaluación de esta expresión booleana es verdadera si y solo si se cumplen alguna de las dos expresiones binarias recibidas.

**\*Expresión booleana binaria Comparador binario:**

La expresiónbooleana binaria de comparación recibe dos expresiones aritméticas y un criterio de comparación. El criterio de comparación esta representado por un delegado BooleanComparer que recibe dos enteros y devuelve un valor booleano. La evaluación del comparador es simple, primero se proceden a evaluar ambas expresiones aritméticas y luego se pasan los resultados de estas evaluaciones como parámetros al comparador booleano en cuestión; finalmente el resultado del comparador binario sera el arrojado por el comparador booleano recibido. Este mecanismo permite reutilizar el tipo comparador binario con diferentes comparadores booleanos.

**\*Comparador booleano Igual:**

El comparador binario igual recibe dos números enteros y devuelve verdadero si ambos son iguales, de lo contrario devuelve falso.

**\*Comparador booleano Mayor:**

El comparador binario igual recibe dos números enteros y devuelve verdadero si el primer numero es mayor que el segundo, de lo contrario devuelve falso.

**\*Comparador booleano Menor:**

El comparador binario igual recibe dos números enteros y devuelve verdadero si el primer numero es menor que el segundo, de lo contrario devuelve falso.

**\*Comparador booleano Mayor o igual:**

El comparador binario igual recibe dos números enteros y devuelve verdadero si el primer numero es mayor o igual que el segundo, de lo contrario devuelve falso.

**\*Comparador booleano Menor o igual:**

El comparador binario igual recibe dos números enteros y devuelve verdadero si el primer numero es menor o igual que el segundo, de lo contrario devuelve falso.

**\*Expresiones booleanas unarias:**



Las expresiones booleanas unarias heredan de las expresiones booleanas y poseen una expresión booleana y su evaluación depende de la evaluación previa de dicha expresión.

Existen tres tipos de expresiones booleanas unarias: el Predicado Verdadero, el Predicado Falso y la expresión Existe; las cuales se analizarán a continuación.

\*Predicado verdadero:

El predicado verdadero es una expresión booleana que hereda de las expresiones unarias y tiene por particularidad que su evaluación siempre devuelve un resultado verdadero.

\*Predicado falso:

El predicado falso es una expresión booleana que hereda de las expresiones unarias y tiene por particularidad que su evaluación siempre devuelve un resultado falso.

\*Expresión existe:

La expresión existe recibe el nombre de una carta, el nombre de una zona del campo y la referencia a un método encargado de devolver en tiempo de ejecución la colección de cartas correspondiente con esa zona del campo. La evaluación de la expresión existe es bastante simple, solo consiste en invocar al método referenciado para obtener la colección de cartas de dicha zona del campo y verificar si existe una carta cuyo nombre coincida con el de la carta que recibimos; en caso positivo la expresión se evalúa como verdadera y en caso contrario como falsa.

\*Expresiones aritméticas:

Las Expresiones aritméticas son una subclase abstracta de la clase Expresión donde el único cambio es que su tipo queda asignado como un tipo numérico. De ella heredan dos tipos de expresiones aritméticas, las Expresiones aritméticas binarias y las Expresiones aritméticas unarias. (ambas son abstractas)

\*Expresiones aritméticas binarias:

Las Expresiones aritméticas binarias heredan de las expresiones aritméticas y poseen dos expresiones aritméticas (izquierda y derecha). Su evaluación consiste en aplicar alguna operación aritmética a los resultados de evaluar ambas expresiones recibidas, donde dicha operación es inherente a cada Expresión aritmética binaria.

\* Expresión aritmética binaria suma:

La expresión aritmética binaria suma hereda de la Expresión aritmética binaria. Recibe dos expresiones aritméticas y su evaluación consiste en realizar la suma de los valores obtenidos después de evaluar cada expresión aritmética.

\* Expresión aritmética binaria resta:

La expresión aritmética binaria resta hereda de la Expresión aritmética binaria. Recibe dos expresiones aritméticas y su evaluación consiste en realizar la resta de los valores obtenidos después de evaluar cada expresión aritmética.

\* Expresión aritmética binaria multiplicación:

La expresión aritmética binaria multiplicación hereda de la Expresión aritmética binaria. Recibe dos expresiones aritméticas y su evaluación consiste en realizar la multiplicación de los valores obtenidos después de evaluar cada expresión aritmética.

\* Expresión aritmética binaria división:

La expresión aritmética binaria división hereda de la Expresión aritmética binaria. Recibe dos expresiones aritméticas y su evaluación consiste en realizar la división de los valores obtenidos después de evaluar cada expresión aritmética.

**\*Expresiones aritméticas unarias:**

Las expresiones aritméticas unarias heredan de las expresiones aritméticas y poseen una expresión aritmética. Su evaluación es el resultado de la evaluación de dicha expresión aritmética. Existe un solo tipo de expresión aritmética. unaria: la expresión Number

**\*Expresión aritmética. unaria Number:**

La expresión aritmética unaria Number hereda de las Expresiones aritméticas unarias

y pueden recibir diversos parámetros ellos son: un numero directamente, el nombre de una carta o un conjunto de datos que consiste en: el nombre de una zona del campo, el nombre de una función que nos devolverá un valor numérico relacionado con la colección de cartas que contiene esa zona del campo en tiempo de ejecución y la referencia a una función que invocara a la función anterior. Si el argumento recibido es un numero, la evaluación de la expresión es el mismo numero; si el argumento recibido es el nombre de una carta y este nombre se corresponde con el de una carta de tipo unidad, la evaluación de la expresión es el ataque de dicha carta; si por ultimo el argumento recibido es el conjunto de objetos descritos anteriormente entonces la evaluación de la expresión sera el valor numérico devuelto por la función cuyo nombre se reciba.

**\*El lenguaje:**

Nuestro lenguaje posee un conjunto de operadores y un conjunto de palabras claves que otorgan al usuario la posibilidad de construir cartas para el juego completamente desde cero, siempre y cuando las cartas cumplan con ciertas reglas establecidas por el juego. El conjunto de operadores definidos en nuestro lenguaje sera listado a continuación y cada uno sera brevemente explicado:

**\*Operadores:**

- Add Este es el operador correspondiente a una expresión aritmética suma
- Mult Este es el operador correspondiente a una expresión aritmética multiplicación
- Sub Este es el operador correspondiente a una expresión aritmética resta
- Div Este es el operador correspondiente a una expresión aritmética división
- = Este es el operador usado para asignar un valor
- == Este es el operador correspondiente al comparador booleano Igual
- > Este es el operador correspondiente al comparador booleano Mayor
- < Este es el operador correspondiente al comparador booleano Menor
- >= Este es el operador correspondiente al comparador booleano Mayor o igual
- <= Este es el operador correspondiente al comparador booleano Menor o igual
- And Este es el operador correspondiente a una expresión booleana binaria And
- Or Este es el operador correspondiente a una expresión booleana binaria Or
- Not Este es el operador correspondiente a una expresión booleana unaria Not
- Compare Este es el operador correspondiente a una expresión booleana binaria Comparador binario.

- True Este es el operador correspondiente a una expresión booleana unaria  
Predicado verdadero
- False Este es el operador correspondiente a una expresión booleana unaria  
Predicado falso
- ExistCardIn Este es el operador correspondiente a una expresión booleana unaria Existe
- , Este es el operador correspondiente a un separador de valores
- ; Este es el operador correspondiente a un separador de declaraciones
- ( Este es el operador correspondiente al inicio de un signo de agrupación de valores
- ) Este es el operador correspondiente al fin de un signo de agrupación de valores
- { Este es el operador correspondiente al inicio de un signo de agrupación de condiciones
- } Este es el operador correspondiente al fin de un signo de agrupación de condiciones

A continuación presentaremos la lista de palabras claves definidas en el lenguaje y las explicaremos brevemente:

#### Palabras Claves:

- ConditionSet  
Esta palabra clave indica el conjunto de condiciones que necesita un poder para ejecutarse.
- InstructionSet  
Esta palabra clave indica el conjunto de instrucciones que ejecutara un poder.
- Condition  
Esta palabra clave indica una condición para la ejecución de un poder.
- Instruction  
Esta palabra clave indica una instrucción a ejecutar por un poder.
- Card  
Esta palabra clave indica una carta
- UnitCard  
Esta palabra clave indica una carta de tipo Unidad.
- LeaderCard  
Esta palabra clave indica una carta de tipo Lider
- EffectCard  
Esta palabra clave indica una carta de tipo Efecto
- Name  
Esta palabra clave indica la propiedad nombre de las cartas
- Attack  
Esta palabra clave indica la propiedad ataque de las cartas de tipo unidad.
- Power  
Esta palabra clave indica la propiedad power de las cartas.
- Position  
Esta palabra clave indica la propiedad posición de las cartas de tipo unidad y de tipo efecto.
- Phrase  
Esta palabra clave indica la propiedad frase de las cartas de tipo unidad y de tipo líder.
- Path  
Esta palabra clave indica la propiedad ruta de las cartas.

- Type  
Esta palabra clave indica la propiedad tipo de las cartas.
- Melee  
Esta palabra clave indica la posición “Cuerpo a cuerpo” de las cartas de tipo Unidad.
- Middle  
Esta palabra clave indica la posición “A distancia” de las cartas de tipo Unidad.
- Siege  
Esta palabra clave indica la posición “Asedio” de las cartas de tipo Unidad
- Weather  
Esta palabra clave indica la posición “Clima” de las cartas de tipo Efecto
- Support  
Esta palabra clave indica la posición “Soporte” de las cartas de tipo Efecto
- Destroy  
Esta palabra clave indica el llamado a la instrucción Destruir.
- Summon  
Esta palabra clave indica el llamado a la instrucción Summon.
- Reborn  
Esta palabra clave indica el llamado a la instrucción Reborn.
- Draw  
Esta palabra clave indica el llamado a la instrucción Draw.
- ModifyAttack  
Esta palabra clave indica el llamado a la instrucción ModifyAttack.
- SwitchBand  
Esta palabra clave indica el llamado a la instrucción SwitchBand.
- AllEnemyCard  
Esta palabra clave indica la colección de cartas (de tipo unidad) en el campo enemigo.
- AllOwnCards  
Esta palabra clave indica la colección de cartas (de tipo unidad) en el campo propio.
- HighestAttackIn  
Esta palabra clave indica la carta con mayor ataque en una zona del juego específica.
- LowestAttackIn  
Esta palabra clave indica la carta con menor ataque en una zona del juego específica.
- NumberOfCardsIn  
Esta palabra clave indica el número de cartas en una zona del juego específica.
- Damage  
Esta palabra clave indica el ataque de una carta de tipo unidad específica.
- DamageIn  
Esta palabra clave indica el ataque combinado de todas las cartas de tipo unidad en una zona específica del juego.
- OwnHand  
Esta palabra clave indica la colección de cartas en la mano propia.
- OwnMelee  
Esta palabra clave indica la colección de cartas en la zona “Cuerpo a cuerpo” propia.

- OwnMiddle  
Esta palabra clave indica la colección de cartas en la zona "A distancia" propia.
- OwnSiege  
Esta palabra clave indica la colección de cartas en la zona "Asedio" propia.
- OwnGraveryard  
Esta palabra clave indica la colección de cartas en el cementerio propio.
- OwnDeck  
Esta palabra clave indica la colección de cartas en el deck propio.
- EnemyHand  
Esta palabra clave indica la colección de cartas en la mano enemiga.
- EnemyMelee  
Esta palabra clave indica la colección de cartas en la zona "Cuerpo a cuerpo" enemiga.
- EnemyMiddle  
Esta palabra clave indica la colección de cartas en la zona "A distancia" enemiga.
- EnemySiege  
Esta palabra clave indica la colección de cartas en la zona "Asedio" enemiga.
- EnemyGraveryard  
Esta palabra clave indica la colección de cartas en el cementerio enemigo.
- EnemyDeck  
Esta palabra clave indica la colección de cartas en el deck enemigo.
- AllExistingCards  
Esta palabra clave indica la colección de todas las cartas del juego.
- MyField  
Esta palabra clave indica el campo propio.
- EnemyField  
Esta palabra clave indica el campo enemigo.
- FreeElection  
Esta palabra clave indica una zona del campo de libre elección

\*La sintaxis:

La sintaxis que definimos en nuestro lenguaje es bastante rigurosa pero a la vez bastante simple. De manera general las ideas fundamentales son las siguientes:

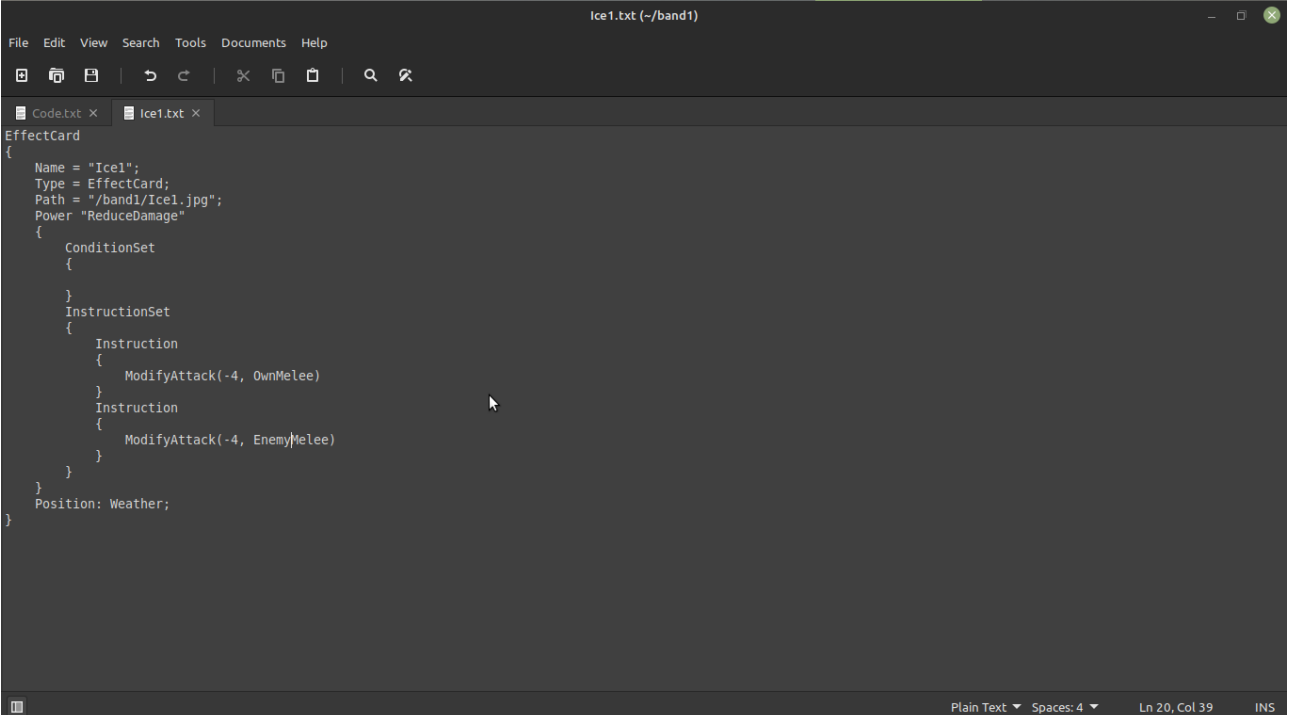
- Cada vez que se cree un objeto se agrupara el cuerpo de la declaración en llaves.
- Cada vez que se llegue al final de una asignación se colocara un punto y coma.
- Los parámetros que reciban diferentes funcionalidades irán agrupados en paréntesis y separados por comas.
- Los identificadores se colocaran entre comillas para diferenciarlos de las palabras reservadas.

La creación de las cartas se define de la siguiente manera:

```

#(Tipo de carta)
{
    Name = #(nombre de la carta (es un identificador)) ;
    Type = #(tipo de carta que debe coincidir con el declarado al comienzo) ;
    Path = #(ruta de la imagen que se asignara al visual de la carta (es un
identificador)) ; (si se asigna el identificador "Default" se asignara una imagen
por defecto)
    Power #(Nombre del poder (es un identificador)) (Si se asigna el
identificador "None" se ignorara el cuerpo definido para el poder)
    {
        ConditionSet
        {
            Condition (Pueden haber múltiples condiciones)
            {
                #(expresión booleana)
            }
        }
        InstructionSet
        {
            Instruction (Pueden haber múltiples instrucciones)
            {
                #(Llamado a una de las instrucciones definidas
anteriormente)
            }
        }
    }
}

#(Aquí irían las propiedades Phrase, Position y Attack en ese orden para las
cartas de tipo unidad, y en caso de que no sea una carta de tipo unidad se
mantendría el orden omitiendo las propiedades que no pertenezcan al tipo de
carta en cuestión (tenga en cuenta que la propiedad Phrase lleva asignado un
identificador, la propiedad Position una posición previamente definida(una
palabra clave) y la propiedad Attack lleva asignado un numero))
}
```



The screenshot shows a code editor window titled 'Ice1.txt (~/.band1)'. The editor contains a JSON-like configuration for an 'EffectCard'. The configuration includes fields for Name, Type, Path, Power, ConditionSet, InstructionSet, and Position. The ConditionSet and InstructionSet are nested objects. The InstructionSet contains two instructions: 'ModifyAttack(-4, OwnMelee)' and 'ModifyAttack(-4, EnemyMelee)'. The Position is set to 'Weather'.

```
EffectCard
{
    Name = "Icel";
    Type = EffectCard;
    Path = "/band1/Icel.jpg";
    Power "ReduceDamage"
    {
        ConditionSet
        {
        }
        InstructionSet
        {
            Instruction
            {
                ModifyAttack(-4, OwnMelee)
            }
            Instruction
            {
                ModifyAttack(-4, EnemyMelee)
            }
        }
    }
    Position: Weather;
}
```

Code.txt (~/Documents/Proyecto 2do semestre)

```
File Edit View Search Tools Documents Help
[Icons]

Code.txt x
UnitCard
{
    Name = "John Snow";
    Type = UnitCard;
    Path = "MyPath";
    Power "Wisdom"
    {
        ConditionSet
        {
            Condition
            {
                And(Compare(2,3,<),True)
            }
        }
        InstructionSet
        {
            Instruction
            {
                Draw(2)
            }
            Instruction
            {
                Draw(Card "Pepe", Card "Juanito")
            }
        }
    }
    Phrase = "I'm the King in The North";
    Position = Melee;
    Attack = 10;
}

Plain Text Spaces: 4 Ln 3, Col 23 INS
```

JonSnow.txt (~band1)

```
File Edit View Search Tools Documents Help
[Icons]

Code.txt x Ice1.txt x JonSnow.txt x
LeaderCard
{
    Name = "Jon Snow";
    Type = LeaderCard;
    Path = "/band1/JonSnow.jpg";
    Power "NeedHelp"
    {
        ConditionSet
        {
            ExistCardIn(Card "Ghost")
        }
        InstructionSet
        {
            Instruction
            {
                Destroy(EnemyMelee)
            }
            Instruction
            {
                Destroy(EnemyMiddle)
            }
            Instruction
            {
                Destroy(EnemySiege)
            }
        }
    }
    Phrase = "You All Crowned Me Your King. I Never Wanted It...";
}

Saving file '/home/erenesto/band1/JonSnow.txt'...
Plain Text Spaces: 4 Ln 24, Col 35 INS
```

El proceso de tokenización del código:

La tokenización del código es el primer proceso por el cual atraviesa nuestro "Compilador", el cual básicamente consiste en convertir el documento de texto en el cual se encuentra la definición de la carta en un conjunto de "tokens". Los tokens son objetos que poseen dos atributos, un tipo (que puede ser número, texto, palabra clave, símbolo e identificador) y un valor (no expondremos aquí la lista de valores, pues se hace bastante extensa).

El proceso se auxilia de varias herramientas que intervienen en él. La primera herramienta es la clase `Compiler`, en la cual se registran todos los operadores y las palabras claves de nuestro lenguaje, y la cual nos devuelve un objeto de tipo `tokenizer`. El `tokenizer` es una herramienta que tiene registrados los operadores y las palabras claves del lenguaje, y posee además métodos para identificar los diferentes tokens. En el `tokenizer` se define el método `GetTokens()` que nos devuelve una lista de tokens después de recibir como argumento un texto dado. Para desplazarse por el texto tenemos otra herramienta importante, la clase `TokenReader`, que contiene un conjunto de métodos para leer los caracteres, y en los cuales se apoyan los métodos definidos en el `tokenizer` para identificar los diferentes tokens. En este primer periodo saltarán errores si se escribe algo en el texto que no esté definido en nuestro lenguaje, así como en los casos en los cuales se declare un identificador y no se coloquen las comillas.

Existe para esto último una última herramienta de nuestro compilador, la estructura `CodeLocation`, la cual almacena el nombre de la carta a la que pertenece cada token, así como la línea en la que se encuentra; permitiendo así hacer más fácil la identificación y rectificación de errores.

El proceso de parseo del código:

El parseo del código es el último proceso por el cual atraviesa nuestro "Compilador", el cual consiste en dar una significación semántica al texto introducido por el usuario.

Llegados a este punto tenemos ya una lista de tokens en la cual está recogido todo el código introducido por el usuario, así que nada de lo que este ha introducido escapa de la definición de nuestro lenguaje.

Ahora es el momento de dar sentido a este conjunto de tokens, y para eso poseemos nuevamente herramientas útiles.

La primera herramienta es la clase `TokenStream`, la cual posee métodos muy útiles. Para desplazarse por la lista de tokens, así como colecciones que agrupan palabras claves de acuerdo a su naturaleza.

La segunda herramienta es el `parser`, la cual contiene al método `ParseCard()` que se encarga de crear finalmente la carta a partir de la definición dada por el usuario. El `parser`, para poder parsear la carta, contiene métodos que le permiten parsear cada tipo específico de carta, cada propiedad de las cartas, cada tipo de expresión (Aritmética y Booleana), cada instrucción definida, los conjuntos de instrucciones y de condiciones, así como los valores devueltos por las palabras claves `Damage`, `DamageIn`, y `NumberOfCardsIn`.

En esta etapa es donde se detectan errores de la naturaleza de argumentos incorrectos o símbolos y tipos de datos esperados; por ejemplo si se quisiera asignar una cadena de texto al ataque de una carta de tipo unidad o si se quisiera definir la propiedad frase en una carta de efecto, ambos casos serían reportados como errores de compilación por nuestro compilador gracias al `parser`. En estos casos también se utilizaría la estructura `CodeLocation` para simplificar el proceso de rectificación de estos errores al usuario.



Como jugar?

Es un juego de cartas por turnos donde cada jugador tiene dos vidas. Una vez que ambos pasen el turno, se calcula el daño total de cada uno y el que menos tenga pierde una vida. Luego se procede a la siguiente ronda hasta que uno de ellos quede sin vidas y se alcance el final del juego. Inicialmente ambos jugadores roban 10 cartas y A Jugar!

**\*Godot:** [GitHub - godotengine/godot: Godot Engine – Multi-platform 2D and 3D game engine](https://github.com/godotengine/godot)