La solución está compuesta por tres proyectos:

BattleCardsLibrary

CardDeveloper1

UserInterface

BattleCardsLibrary se encarga de controlar la lógica del juego, contando con el archivo **Game.cs** en el cual se define la clase Game. Esta clase contiene lo que el juego necesita para ser jugado como una colección de cartas, dos jugadores y algo que le permita decidir qué hacer con cada acción que los jugadores decidan hacer con una carta, siendo ese algo el método CardActionReceiver. La clase Game también contiene los métodos siguientes:

`CheckAndChangePhaseAndCurrentPlayer`: el cual es llamado cuando un jugador manda al juego la acción EndTurn.Este método actualiza la fase y el jugador actuales del juego en dependencia de su estado.

`GameIsOver`: se encarga de chequear si el juego terminó y en caso de que esto suceda devuelve true.

`GetFirstPlayerByDiceThrowing`: se encarga de determinar qué jugador comenzará el juego simulando un lanzamiento de dados.

`CreateAPlayerInstance`: crea una instancia de la clase HumanPlayer o AIPlayer en dependencia de los parámetros ingresados en UIPlayer a través de la ventana SetPlayers de UserInterface antes de comenzar el juego.

`GenerateRandomDeck`: genera un deck aleatorio para un jugador en dependencia de la cantidad de cartas que haya en la lista AllCardsCreated, que contiene a todas las cartas con las que el juego cuenta para jugar, sin poder contener más de 20 cartas un mismo deck.

En el archivo **Actions.cs** se encuentran los métodos para ejecutar las acciones relacionadas con la fase de batalla: Attack, Heal y DirectAttack.

BattleCardsLibrary también cuenta con una carpeta llamada Utils en la cual se encuentran los enum utilizados por la solución BattleCards y otra carpeta con el nombre de Player, en la cual se encuentran las clases e interfaces relacionadas con los jugadores.

Además, BattleCardsLibrary tiene a los archivos **IEvaluate.cs, ICard.cs, ISpellCard.cs** y **IMonsterCard.cs** . Cada uno contiene una interfaz con el mismo nombre del archivo, que fueron creadas para poder utilizar las cartas y evaluarlas sin tener implementadas las clases Card, SpellCard, MonsterCard, ni ninguna de las expresiones que implementan la interfaz IEvaluate, las cuales son utilizadas en el proyecto CardDeveloper1.

CardDeveloper1 es un proyecto para crear cartas y guardarlas, lo que implica dada una fuente de información determinada (la cual se representa en el proyecto mediante la interfaz ICardCreatorSource) obtener el objeto que necesita como parámetro CardCreator para crear una nueva carta, pasando primero por un proceso de validación de la misma y una vez creada la carta guardarla. Para desarrollar este proceso

se utilizan instancias de las interfaces ICardCreatorSource, ICardCreator, ICardValidator y ICardSaver, contenidas en la carpeta CardDeveloper.

En este segundo proyecto también se encuentra la carpeta ICardEvaluator, en la cual hay un archivo llamado **Interpreter.cs**, el cual contiene la clase Interpreter, que dado un string que represente a una expresión y el tipo de la carta que lo contiene es capaz de devolver la expresión que dicho string representa, lanzando una excepción en caso de encontrar algún error en la sintaxis o semántica del string ingresado. Los distintos tipos de expresiones se encuentran en esta carpeta, cada uno en un archivo .cs con el nombre de la expresión.

En CardDeveloper1 también está la carpeta Exceptions, en la cual están todas las excepciones que pueden ser lanzadas en el proceso de comprobar que una carta es válida. Estas excepciones son clases que heredan de Exceptions que fueron utilizadas para que el código en el proceso de chequear si una carta es válida se entendiera con mayor facilidad.

En la carpeta Cards de este proyecto se encuentran las clases Card, MonsterCard y SpellCard. Estas son instancias de las interfaces ICard, IMonsterCard y ISpellCard respectivamente, lo cual permite que el juego pueda utilizar las cartas creadas por este proyecto sin tener las propias clases de las cartas, sino haciendo referencia a las interfaces que ellas implementan.

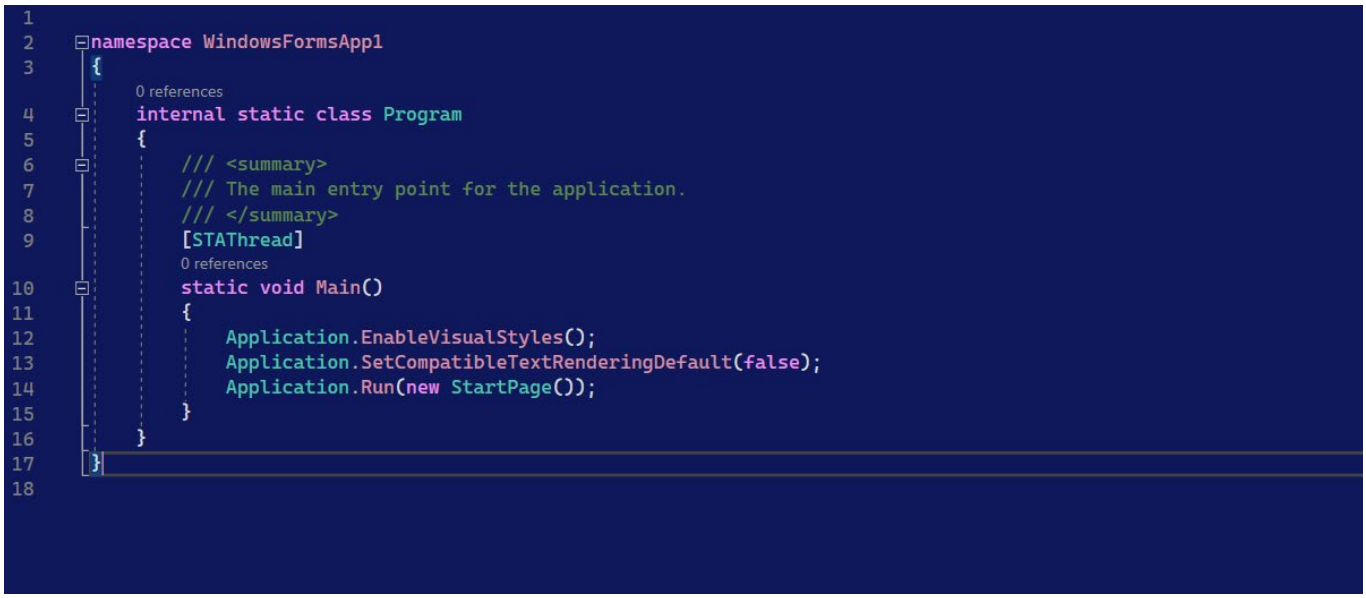
La clase Card(que se encuentra en el archivo **Card.cs)** tiene el método `CheckIfValueIsNumber` para chequear que los valores de todas las propiedades que son numéricas sean precisamente números. También cuenta con el método `GetExpressionOrDefaultValueAsConstant`, el cual determina si la carta que se está creando tiene una expresión asociada a Attack, Heal o Defend y en caso de que esto ocurra llama al método BuildExpression de la clase Interpreter para construir la expresión y volverla a evaluar, mientras que en caso contrario le asigna a Attack, Heal o Defend el valor que le corresponde en el diccionario CardProperties al valor de Attack, Heal o Defend en el diccionario ExtraPropertiesDefaultValue respectivamente. ExtraPropertiesDefaultValue es un diccionario que le asigna a propiedades de cartas otras propiedades, lo cual se utiliza en casos como este en el cual no tendríamos un valor para cierta propiedad y queremos adjudicarle un valor por defecto. Por la manera en que el juego fue diseñado los valores de Attack, Heal y Defend por defecto son los valores de Damage, HealingPowers y Armour, los cuales no conocemos hasta el momento en que el usuario los ingresa, por lo tanto, el diccionario ExtraPropertiesDefaultValue asigna a estas tres propiedades la propiedad cuyo valor le corresponde por defecto, para después obtener el valor de dichas propiedades en el diccionario CardProperties.

El método `SetValues` es el encargado de establecer la propiedad que le corresponde a Attack, Heal y Defend respectivamente en el diccionario ExtraPropertiesDefaultValue.

El método `GetCardDescription` obtiene la descripción de una carta como un string dado un string[] `cardDefinition` que es la definición de la carta que brinda ICardCreatorSource después de procesar el contenido ingresado por el usuario. Este string[] `cardDefinition se` obtiene llamando al método `GetCardDefinition` de la instancia de `ICardCreatorSource` que reciba CardDeveloper, la cual recibe del proyecto UserInterface, en dependencia de la vía por la cual haya sido creada la carta, en este caso contando con dos clases: `DataProcessorForPlainText y DataProcessorForFriendlyUI.`

UserInterface se encarga de la parte gráfica de la solución, es decir, es lo que permite visualizar el estado del juego en todo momento e interactuar con él. Este proyecto es una aplicación de Windows Forms que

cuenta con archivos .cs que representan cada una de las ventanas que un jugador tiene l posibilidad de visualizar en el transcurso de las distintas etapas del juego, como crear cartas o iniciar un nuevo juego.



```
namespace WindowsFormsApp1
{
    0 references
    internal static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        0 references
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new StartPage());
        }
    }
}
```

La imagen que se ve arriba es del código que se encuentra en el archivo **Program.cs**, el cual contiene el método Main del proyecto y declara que la aplicación inicia mostrando la ventana StartPage, creando una nueva instancia del objeto StartPage a ejecutar cuando se ejecute la aplicación. UserInterface es el proyecto que ejecuta toda la solución, por decirlo de una forma, porque cuando se ejecuta este llama a BattleCardsLibrary y CardDevoper1 para iniciar un nuevo juego o crear cartas.

En el archivo Helper del proyecto se encuentra la clase Helper, que contiene al método ShowMessage el cual crea y muestra un objeto de tipo MessageBox con el mensaje correspondiente, siendo este uno de error en caso de que la carta no pueda ser creada (caso en el que se especifica el error cometido) o uno de información para notificar que la carta pudo crearse, si su creación fue satisfactoria.

Los archivos **CreateCardPage.**cs y **CreateCardPlainText.cs** tienen la misma funcionalidad que es crear una carta, pero ofrecen al usuario diferentes modalidades para su creación, por lo tanto implementan la interfaz ICardCreatorSource de forma diferente. Ambos archivos contienen al método next_bt_Click, el cual crea un CardValidator, que recibe la instancia de ICardCreatorSource correspondiente y una instancia de CardCreator.

En las imágenes de abajo se muestran las diferencias entre ambos archivos que se acaba de explicar.

Método next_bt_Click de **CreateCardPage.cs**

```
private void next_bt_Click(object sender, EventArgs e)

    //Gather all data and call method to evaluate expression
    ICardCreatorSource cardSource = new DataProcessorForFriendlyUI(this.panel1.Controls.OfType<TextBox>(), monster_card_rb.Checked, spell_card_rb.Checked);
    ICardValidator cardValidator = new CardValidator(cardSource, CardCreator.Instance);

    ValidationResponse validationResponse = cardValidator.ValidateCard();

    Helper.ShowMessage(validationResponse);

    if (validationResponse.ValidationResult == ValidationResult.Ok)
    {
        CardSaver.Instance.SaveCard(cardValidator.Card.Description, cardValidator.Card.Name);
        previousForm.Show();
        Hide();
    }


7 references
public class DataProcessorForFriendlyUI : ICardCreatorSource

    public IEnumerable<TextBox> Textboxes;
    public bool Monster;
    public bool Spell;
    1 reference
    public DataProcessorForFriendlyUI(IEnumerable<TextBox> textboxes, bool monster, bool spell)
    {
        this.Spell = spell;
        this.Monster = monster;
        this.Textboxes = textboxes;
    }

    2 references
    public string[] GetCardDefinition()
```

```
    2 references
    public string[] GetCardDefinition()
    {
        List<string> textToProcess = new List<string>();// monster_card_rb   spell_card_rb
        if (!Monster && !Spell)
        {
            //show error because you need to choose at least oneICardtype.
            throw new InvalidCardTypeException("You must select a validICardtype.");
        }
        else
        {
            textToProcess.Add("Type");
            textToProcess.Add(Monster ? "Monster" : "Spell");
            /*textToProcess[0] = "Type";
            textToProcess[1] = monster_card_rb.Checked ? "Monster" : "Spell";*/
        }
        foreach (var textbox in Textboxes)
        {
            if (textbox.Text != string.Empty)
            {
                textToProcess.Add(textbox.Name.Replace("_value", ""));
                textToProcess.Add(textbox.Text);
                //textToProcess[i++] = textbox.Name.Replace("Value", "");
                //textToProcess[i++] = textbox.Text;
            }
        }
        string[] textToReturn = new string[textToProcess.Count];
        for (int j = 0; j < textToProcess.Count; j++)
        {
            textToReturn[j] = textToProcess[j];
        }
        return textToReturn;
    }
```

Método next_bt_Click de **CreateCardPlainText.cs**

```
private void next_bt_Click(object sender, EventArgs e)
{
    ICardValidator cardValidator = new CardValidator(new DataProcessorForPlainText(this.card_exp.Text.Split("\r\n")), CardCreator.Instance);

    ValidationResponse validationResponse = cardValidator.ValidateCard();

    Helper.ShowMessage(validationResponse);

    if (validationResponse.ValidationResult == ValidationResult.Ok)
    {
        CardSaver.Instance.SaveCard(cardValidator.Card.Description, cardValidator.Card.Name);
        previousForm.Show();
        Hide();
    }

}
```

```
public class DataProcessorForPlainText : ICardCreatorSource
{
    private string[] Text;
    1 reference
    public DataProcessorForPlainText(string[] text)
    {
        this.Text = text;
    }
    2 references
    public string[] GetCardDefinition()
    {
        string textAsString = string.Empty;
        foreach (var item in Text)
        {
            textAsString += ": " + item;
        }
        string[] cardDefinition = textAsString.Remove(0, 2).Split(": ");
        return cardDefinition;
    }
}
```

En ambos se realizan las mismas operaciones, solo que el ICardCreatorSource es diferente porque la información con la que se cuenta es diferente.

En el archivo **SetPlayer.cs** se encuentra el código correspondiente a la recolección de los datos de los jugadores con los que se va a iniciar el juego, puesto que una vez ingresados ambos jugadores se crea una nueva instancia de la clase Game de BattleCardsLibrary y se crea un nuevo Board para mostrar la ventana que representa el tablero del juego.

En el archivo **Board.cs** está la clase Board, con distintos métodos que responden a los posibles eventos que el usuario puede iniciar mediante su interacción con la ventana Board, así como métodos que actualizan el estado del juego que se muestra, a los cuales se llama dentro de esos métodos que responden a eventos.