

# Informe del Proyecto de Dominó:

## Integrantes del equipo:

-David Barroso Rey.... Grupo C211

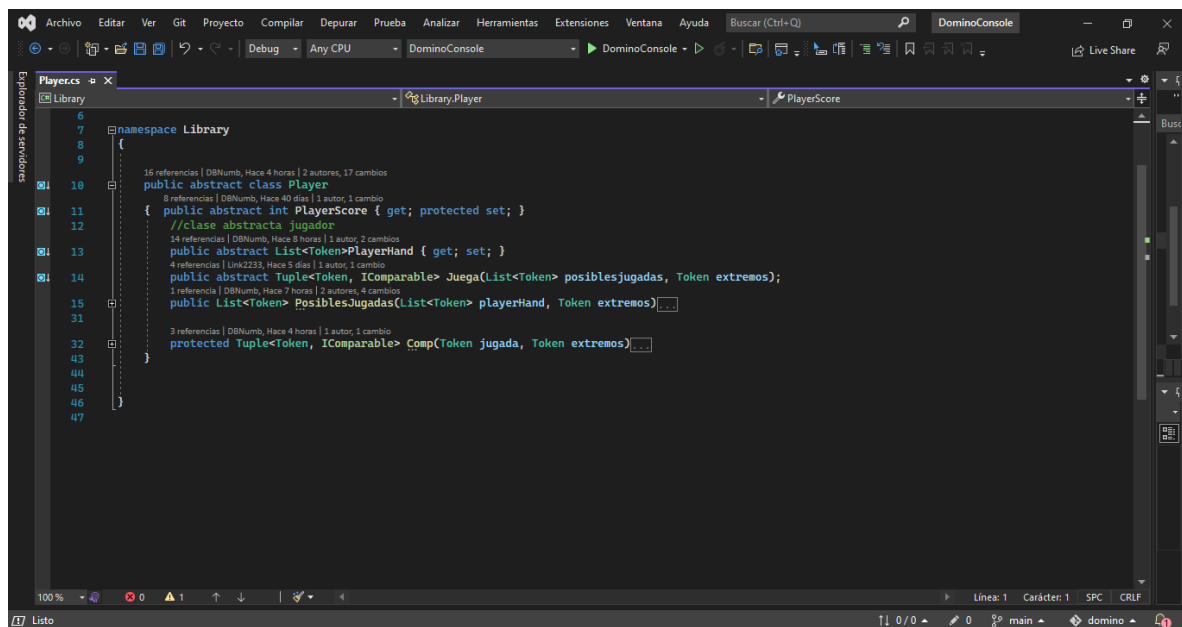
-José Damian Tadeo Escarrá.... Grupo C212

En nuestro Proyecto de programación tenemos la orden de realizar un simulador de Dominó utilizando el lenguaje de Programación C#, que permita realizar cambios en el juego y dé como resultado una simulación válida, también debemos implementar diferentes tipos de jugadores con sus respectivas estrategias. En este informe se hablará sobre los métodos de desarrollo del proyecto y algunas de las técnicas usadas en este.

Este proyecto contará con una biblioteca de clases y una aplicación de consola para la interfaz gráfica; entre las clases de la biblioteca encontraremos algunas como Board, Player y todos sus herederos, entre otras que mostraremos y explicaremos a continuación.

Clases empleadas en la biblioteca de clases del proyecto para su funcionamiento:

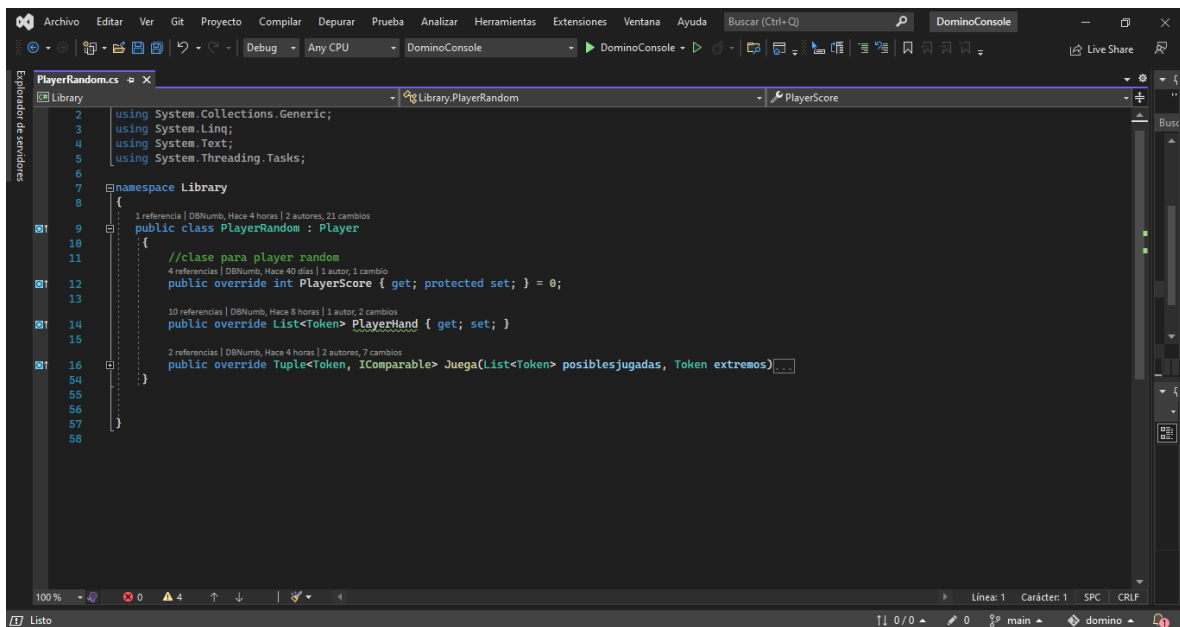
Clase abstracta Player:



Clase que nos permite instanciar cualquier tipo de jugador que herede de ella como un Player, esto nos permite trabajar con cualquier tipo de jugador sin tener que hacer sus implementaciones específicas para comportamientos que todos cumplen, como sería el de tener una mano o sacar sus posibles jugadas, estas son funciones que todos los jugadores realizarán independientemente de su tipo por lo cual lo ponemos en esta clase. La Clase Player desarrolla 2 métodos que serán la misma implementación para todos los tipos de players, en este caso son PosiblesJugadas que es un método que recibe la mano del jugador y una “ficha” que sería formada por los extremos, este método devuelve una lista de tokens (fichas) que serían las que puede jugar el jugador dados los extremos, en caso de querer implementar algún tipo de jugador tramposo o similar que quiera jugar un “forro” se le puede

hacer un override en su clase al método y de esa manera implementar su propia estrategia de sacar sus jugadas. El otro método no abstracto que define la clase es el método Comp que coge los extremos y la jugada escogida por el jugador y devuelve la jugada y el extremo por el que se jugará.

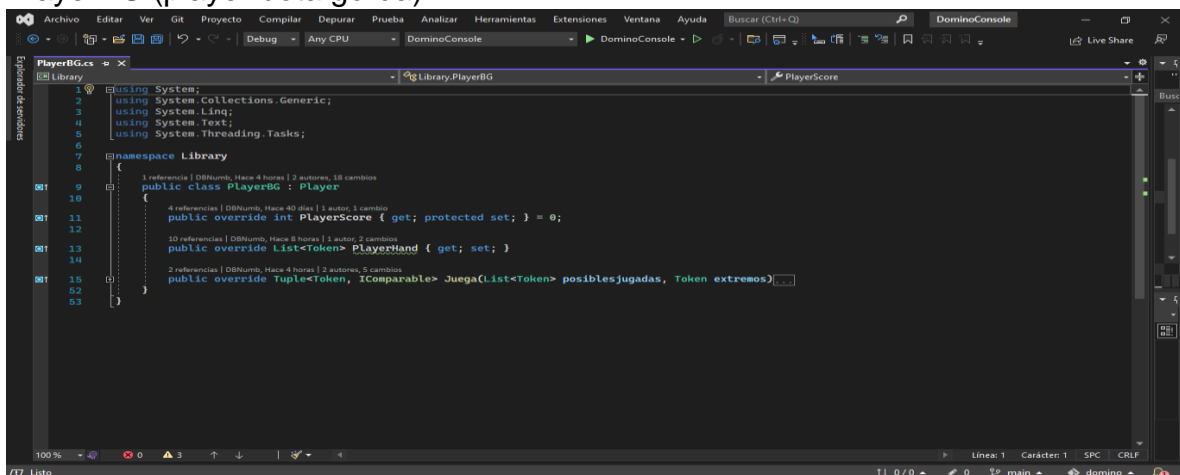
### Player Random:



```
1 using System.Collections.Generic;
2 using System.Linq;
3 using System.Text;
4 using System.Threading.Tasks;
5
6 namespace Library
7 {
8     1 referencia | DBNumb, Hace 4 horas | 2 autores, 21 cambios
9     public class PlayerRandom : Player
10     {
11         //clase para player random
12         4 referencias | DBNumb, Hace 40 días | 1 autor, 5 cambios
13         public override int PlayerScore { get; protected set; } = 0;
14         10 referencias | DBNumb, Hace 8 horas | 1 autor, 2 cambios
15         public override List<Token> PlayerHand { get; set; }
16         2 referencias | DBNumb, Hace 4 horas | 2 autores, 7 cambios
17         public override Tuple<Token, IComparable> Juega(List<Token> posiblesjugadas, Token extremos)...
```

Player Random hereda de la clase abstracta Player por lo cual toma métodos definidos y comportamientos de este. Lo que diferencia a este tipo de jugador es que para jugar escoge una ficha random entre sus posibles jugadas y la juega.

### PlayerBG (player bota gorda):

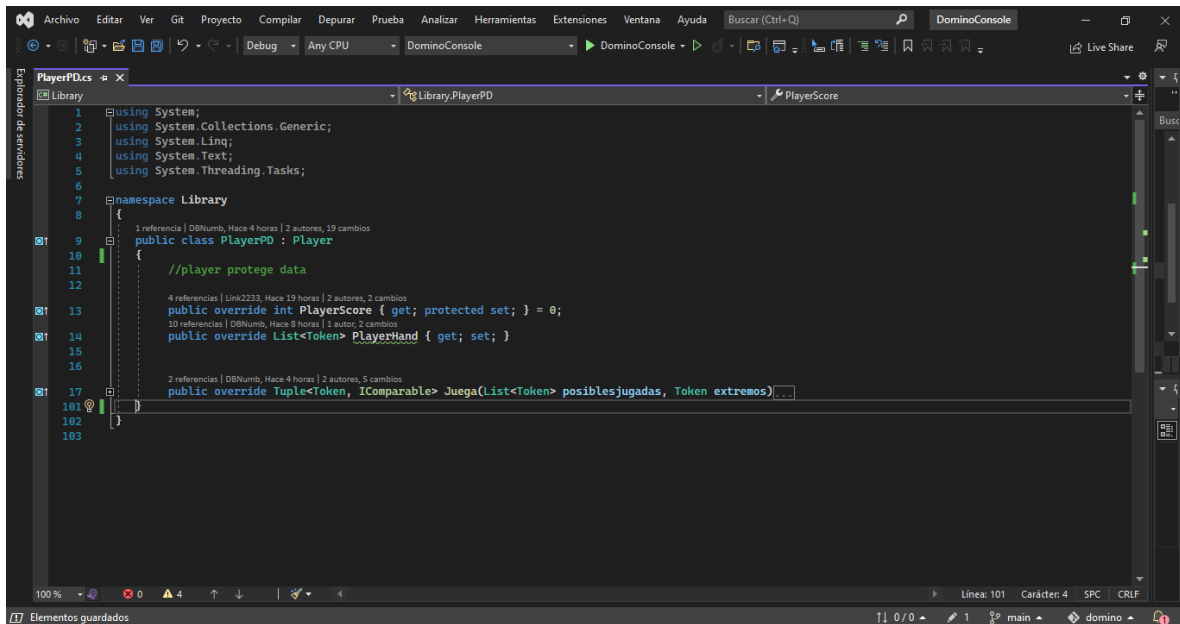


```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace Library
8 {
9     1 referencia | DBNumb, Hace 4 horas | 2 autores, 18 cambios
10     public class PlayerBG : Player
11     {
12         4 referencias | DBNumb, Hace 40 días | 1 autor, 1 cambio
13         public override int PlayerScore { get; protected set; } = 0;
14         10 referencias | DBNumb, Hace 8 horas | 1 autor, 2 cambios
15         public override List<Token> PlayerHand { get; set; }
16         2 referencias | DBNumb, Hace 4 horas | 2 autores, 5 cambios
17         public override Tuple<Token, IComparable> Juega(List<Token> posiblesjugadas, Token extremos)...
```

Este es otro de los tipos de jugadores definidos, lo que lo diferencia de los demás es que su estrategia para jugar es escoger entre sus posibles jugadas la ficha que

más puntos tenga entre ellas para poder restar la mayor cantidad de puntos a su mano en cada jugada, de esa forma si se tranca el juego tiene la menor puntuación posible comparado a si el mismo jugador hubiera jugado fichas diferentes.

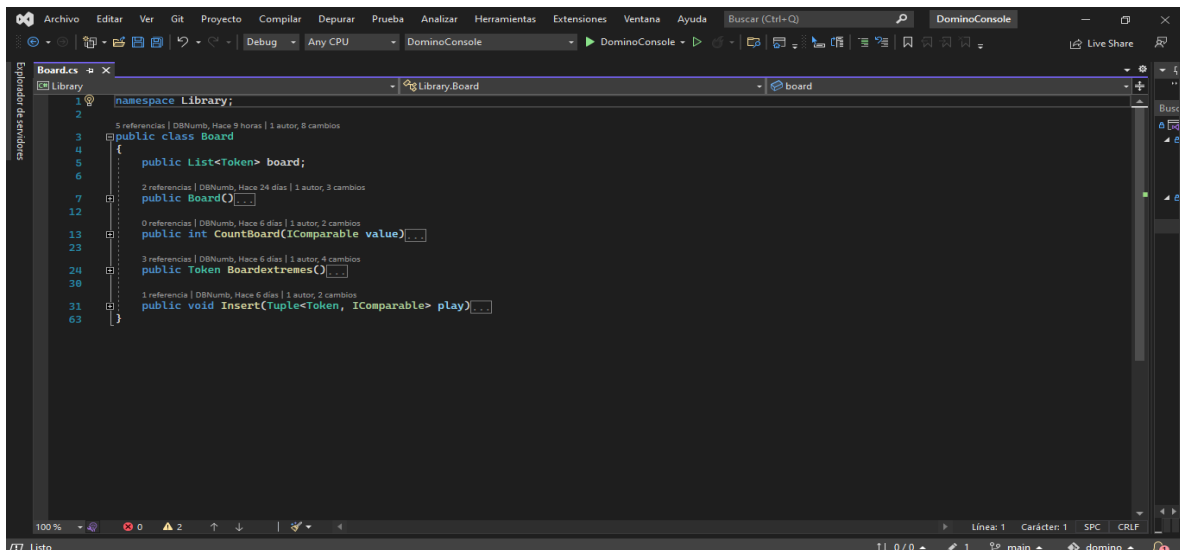
### JugadorPD (Jugador Protege Data):



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace Library
8 {
9     public class PlayerPD : Player
10     {
11         //player protege data
12
13         public override int PlayerScore { get; protected set; } = 0;
14         public override List<Token> PlayerHand { get; set; }
15
16         public override Tuple<Token, IComparable> Juega(List<Token> posiblesjugadas, Token extremos)
17     {
18     }
19 }
```

Tipo de jugador que juega protegiendo su “data” o sea, protegiendo la ficha que más abunde en su mano.

### Clase Board:

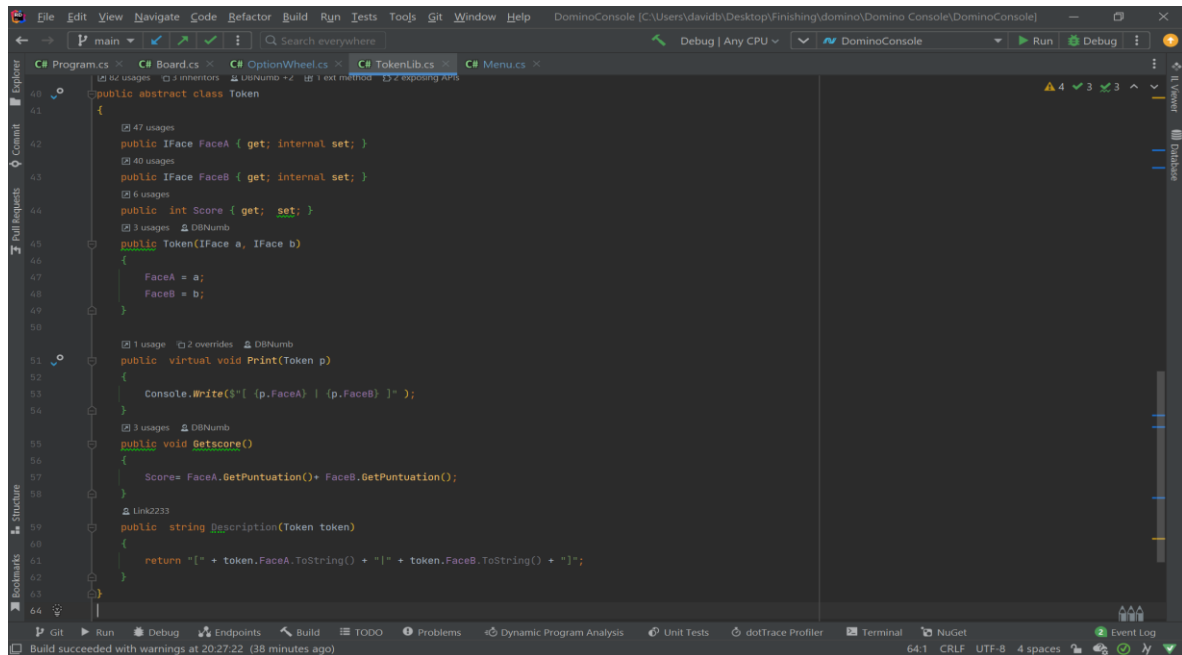
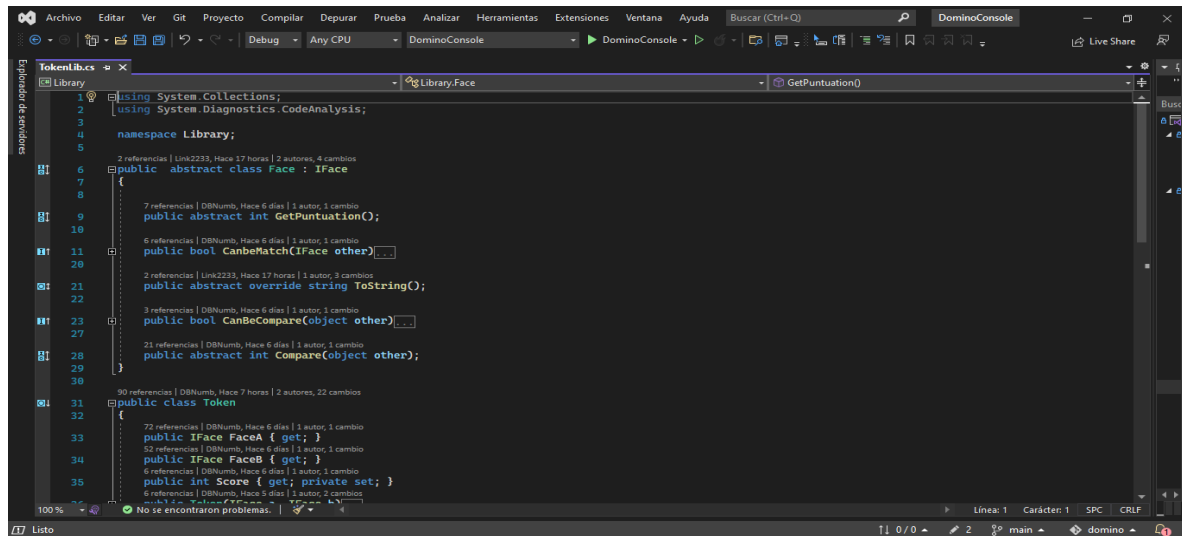


```
1 namespace Library;
2
3 public class Board
4 {
5     public List<Token> board;
6
7     public Board()
8     {
9     }
10
11     public int CountBoard(IComparable value)
12     {
13     }
14
15     public Token Boardextremes()
16     {
17     }
18
19     public void Insert(Tuple<Token, IComparable> play)
20     {
21     }
22 }
```

La clase Board simula el tablero de juego, este será una lista de fichas y la clase tendrá varios métodos para realizar determinadas funciones del tablero, entre ellas

están BoardExtremes que saca en una “ficha” los extremos del tablero, el método Insert que le inserta a la lista de fichas (tablero) la ficha que se le pase como parámetro por el extremo que se le pase por parámetro. El método CountBoard toma como parámetro un valor y por cada ficha que tenga al menos una cara igual a ese valor suma un contador, de esta manera devuelve la cantidad de fichas con ese valor en el tablero.

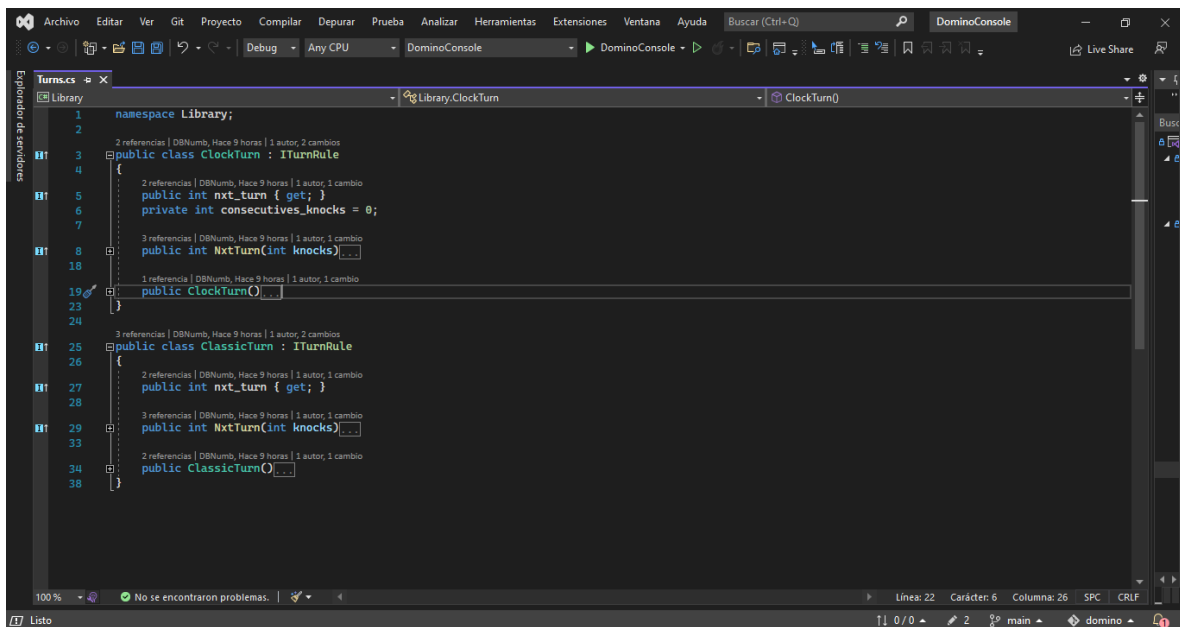
## Biblioteca TokenLib:



En esta biblioteca encontramos lo relacionado a las definiciones de las fichas y sus caras, por una parte tenemos a la clase Face que implementa la interface IFace y sería una cara de la ficha, en esta clase podemos encontrar métodos como GetPuntuacion que retorna la puntuación de la cara que se esté evaluando en el

momento, define como abstracto un override del método ToString para que toda clase que herede de Face tenga que implementar su propio ToString, tenemos el método Compare que toma otra cara y en caso de ser iguales retorna 0, si la ficha es menor a la pasada por parámetro se retorna -1, en el caso contrario se retorna 1. Por otro lado, tenemos la clase Token que recibe en su constructor 2 caras y forma una tupla de caras para formar una ficha, tiene los métodos Print, Description y GetScore, el método Print es abstracto y le corresponde la representación de la ficha en “x” interfaz gráfica(en nuestro caso, una aplicación de consola).

Biblioteca con la implementación de las Reglas de turnos:

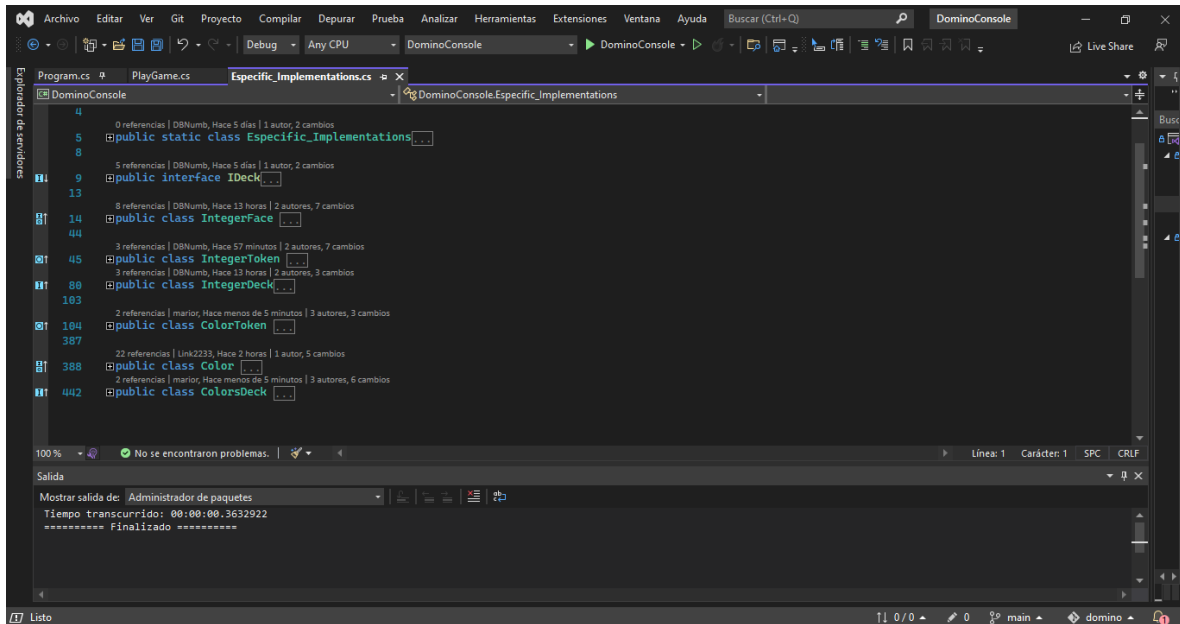


```
1 namespace Library;
2
3 public class ClockTurn : ITurnRule
4 {
5     public int nxt_turn { get; }
6     private int consecutives_knocks = 0;
7
8     public int NxtTurn(int knocks)
9     {
10         // ...
11     }
12
13     public ClockTurn()
14     {
15         // ...
16     }
17 }
18
19
20
21
22
23
24
25 public class ClassicTurn : ITurnRule
26 {
27     public int nxt_turn { get; }
28
29     public int NxtTurn(int knocks)
30     {
31         // ...
32     }
33
34     public ClassicTurn()
35     {
36         // ...
37     }
38 }
```

En esta biblioteca encontramos las diferentes reglas de turnos que implementan la interface ITurnRule. En este caso tenemos ClassicTurn que es el modo de turno clásico, se juega en un solo sentido y si se pasa un jugador se salta al siguiente que pueda jugar, y el ClockTurn que se juega en un sentido a menos que se pase un jugador, si un jugador se pasa hace que le vuelva a tocar jugar al jugador anterior al que se pasó.

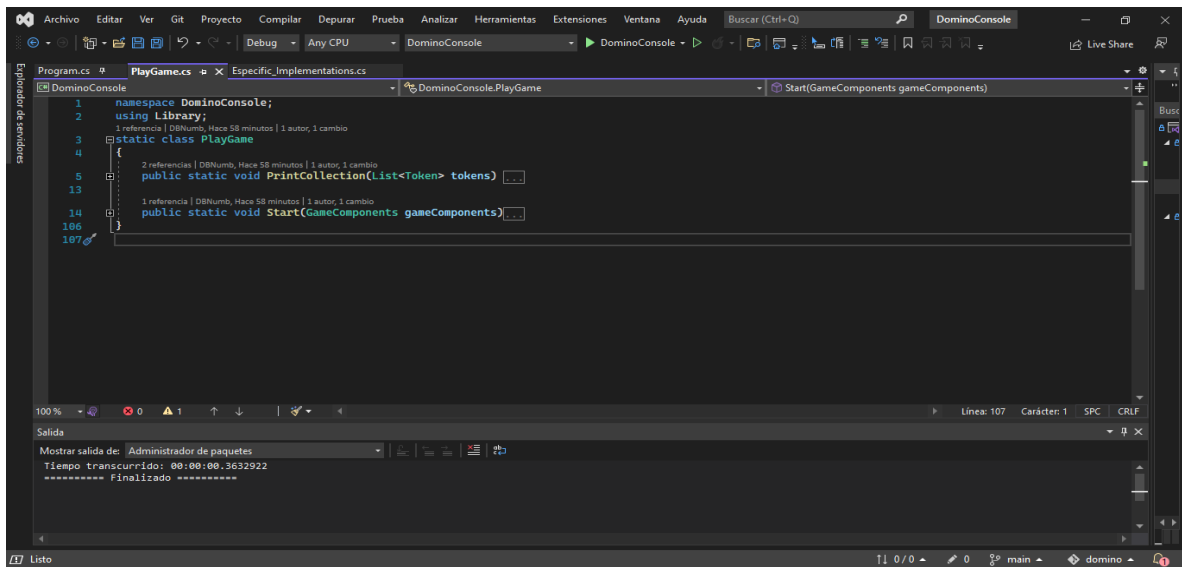
La aplicación de consola en este caso es la interfaz gráfica del proyecto y la que se encarga de interactuar con el usuario, esta nos mostrará a través de diferentes clases y métodos los estados del tablero e imprimirá el “log” para actualizar las jugadas entre otros. En esta tenemos varias bibliotecas de clases, entre ellas Program, PlayGame y Especific\_Implementations.

Especific\_Implementations:



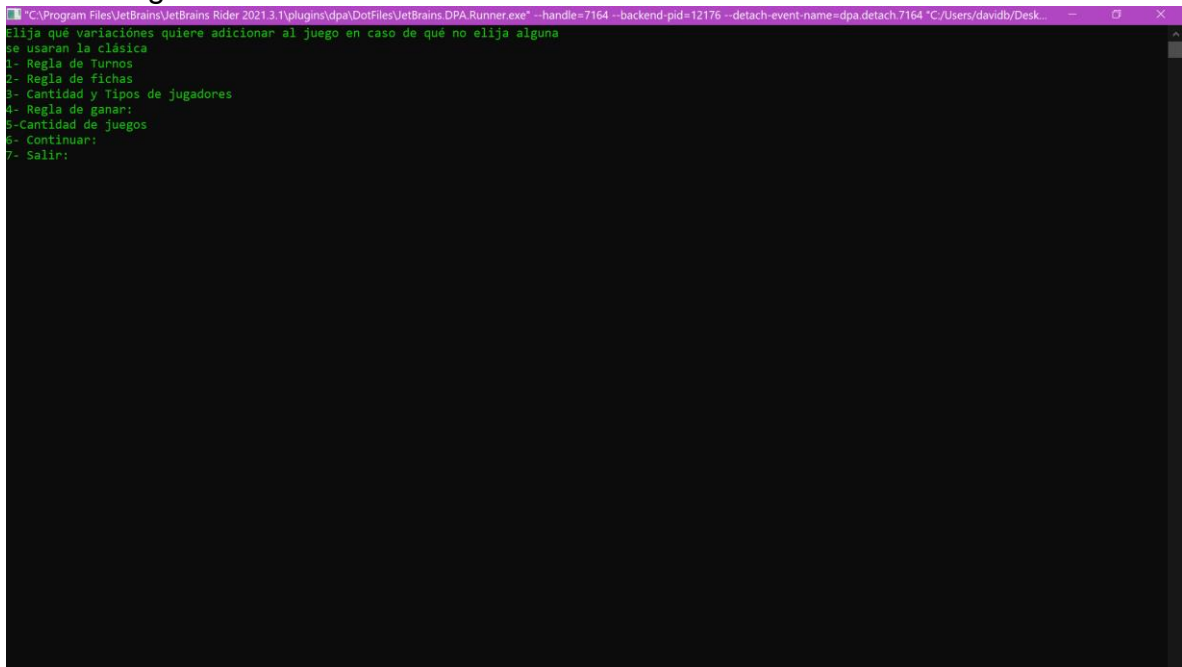
En esta biblioteca encontraremos lo referido a la creación de los diferentes tipos de “Decks” y las implementaciones específicas de los tipos de fichas y caras. En este caso podemos ver `IntegerFace`, `IntegerToken` e `IntegerDeck`, por un lado, que hacen posible la creación de caras de enteros, fichas de enteros y colección de fichas de enteros y por el otro lado tenemos `Color`, `ColorToken` y `ColorDeck` que posibilitan la creación de caras, fichas y colecciones de fichas de colores. En el caso de las implementaciones de enteros se les pasa un valor y con ese valor se hace una cara, con 2 caras se hace una ficha y con varias fichas un Deck. En el caso de los colores se le pasa un string con el nombre del color, el código RGB que lo caracteriza y el código decimal que lo distingue por ejemplo `new Color (“azul”, 0, 0, 255, 255)`.

Clase `PlayGame`:



Esta clase cuenta con los métodos PrintCollection y Start. El método start se encarga de iniciar el juego y el método PrintCollection recibe una lista de fichas y las imprime en cada llamado.

## Clase Program:



La clase Program es la que se encarga de enlazar todas las clases y métodos para hacer posible el juego y la interacción con el usuario.



### Juego con Fichas de Colores:



### Juego con Fichas de números:

