

Informe del Proyecto de Dominó:

Integrantes del equipo:

-David Barroso Rey.... Grupo C211

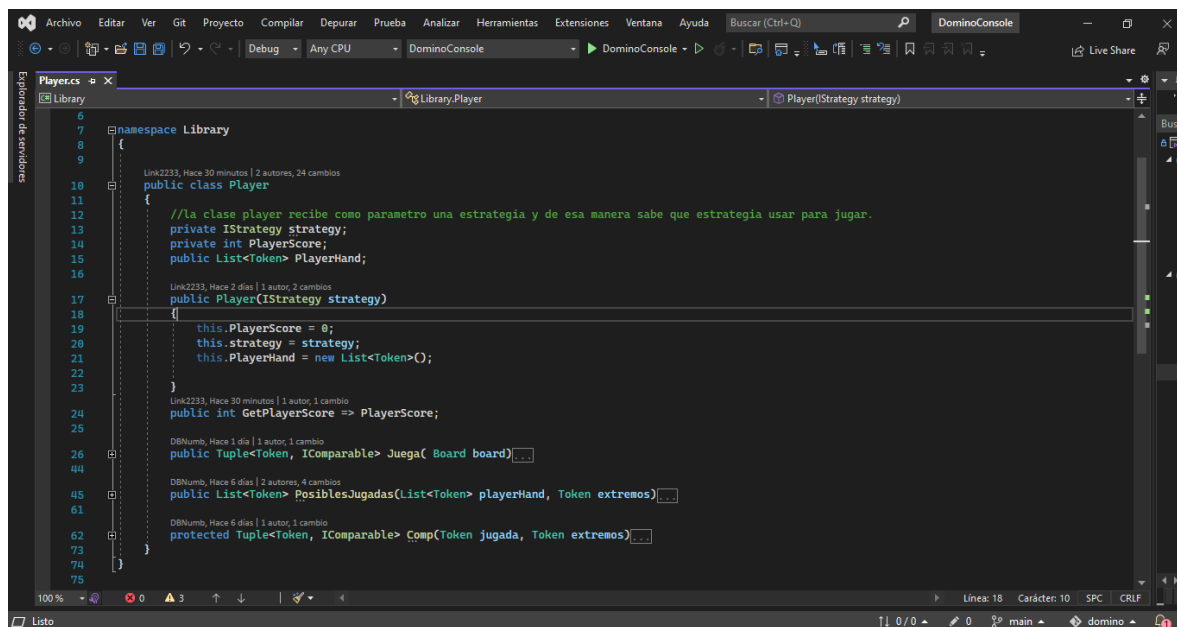
-José Damian Tadeo Escarrá.... Grupo C212

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, entre otras que mostraremos y explicaremos a continuación.

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

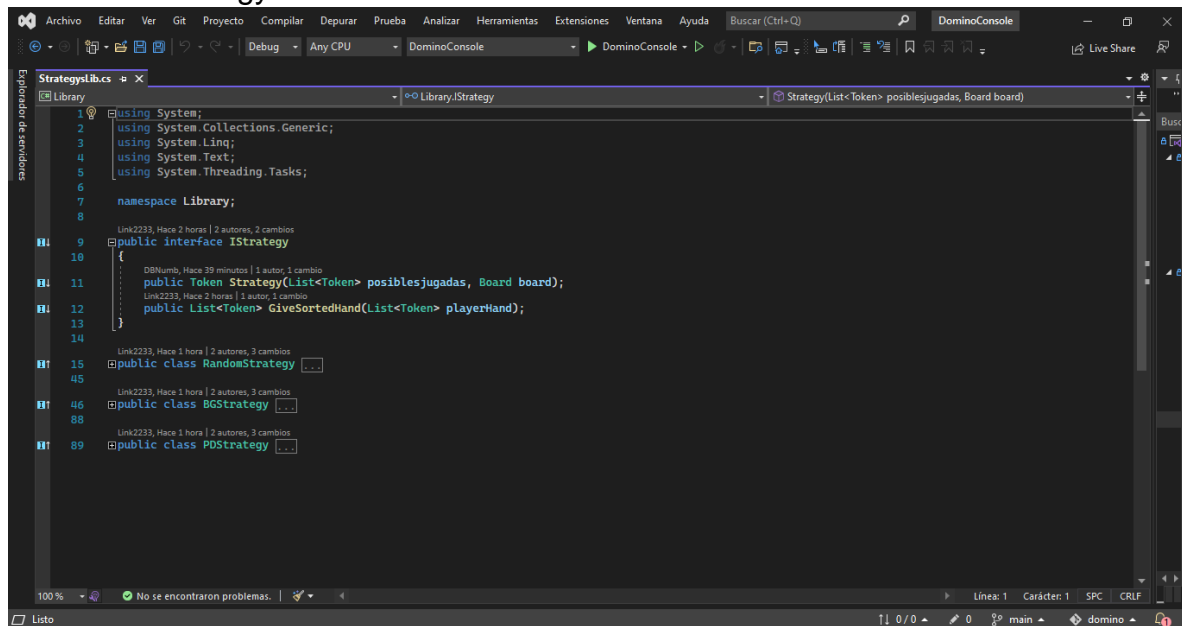
Clase Player:



```
6 namespace Library
7 {
8     Link2233, Hace 30 minutos | 2 autores, 24 cambios
9     public class Player
10     {
11         //la clase player recibe como parametro una estrategia y de esa manera sabe que estrategia usar para jugar.
12         private IStrategy strategy;
13         private int PlayerScore;
14         public List<Token> PlayerHand;
15
16         Link2233, Hace 2 dias | 1 autor, 2 cambios
17         public Player(IStrategy strategy)
18         {
19             this.PlayerScore = 0;
20             this.strategy = strategy;
21             this.PlayerHand = new List<Token>();
22         }
23
24         Link2233, Hace 30 minutos | 1 autor, 1 cambio
25         public int GetPlayerScore => PlayerScore;
26
27         DBNumb, Hace 1 dia | 1 autor, 1 cambio
28         public Tuple<Token, IComparable> Juega( Board board)
29
30         DBNumb, Hace 6 dias | 2 autores, 4 cambios
31         public List<Token> PosiblesJugadas(List<Token> playerHand, Token extremos)
32
33         DBNumb, Hace 6 dias | 1 autor, 2 cambios
34         protected Tuple<Token, IComparable> Comp(Token jugada, Token extremos)
35     }
36 }
```

Clase que recibe una estrategia en el momento de instanciarla, esta estrategia definirá el tipo de juego que realizará el jugador. Player tiene una propiedad GetPlayerScore que da como resultado la sumatoria de los puntos de todas las fichas jugadas por este, entre sus métodos podemos encontrar el método Juega que se encarga de devolver una tupla con la jugada y el extremo donde se jugará, el método PosiblesJugadas recibe una lista de jugadas y una "ficha" en la que se encuentran ambos extremos donde se puede jugar, este método devuelve una lista de posibles jugadas validas a insertar en al menos uno de los 2 extremo.

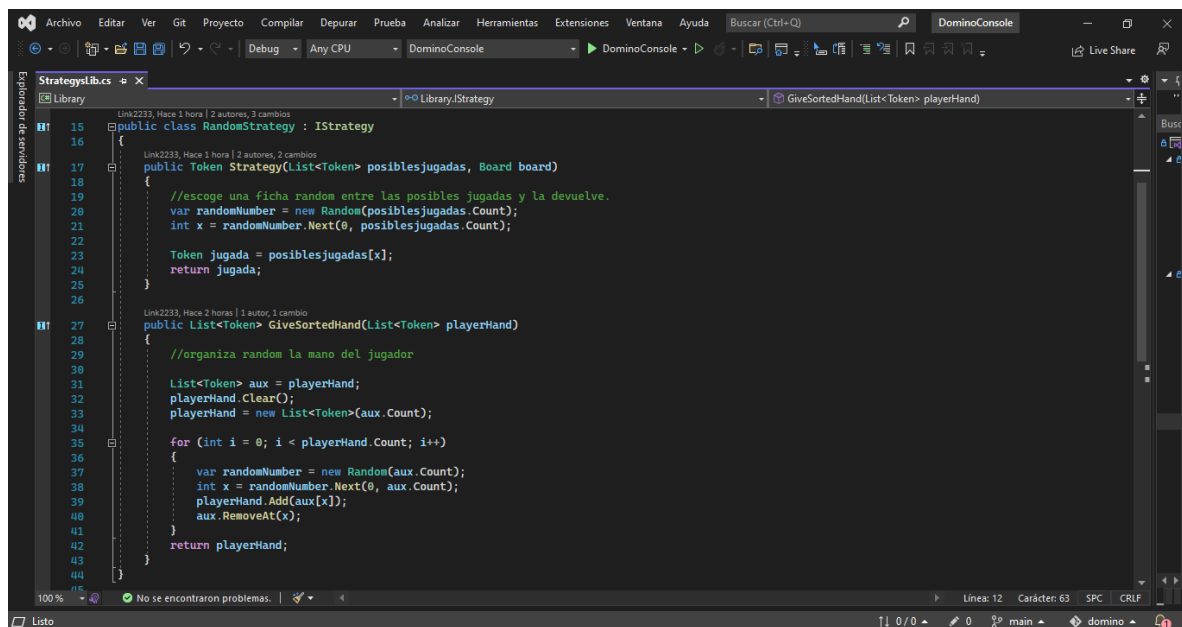
Interface IStrategy:



```
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 interface IStrategy
10 {
11     public Token Strategy(List<Token> posiblesjugadas, Board board);
12     public List<Token> GiveSortedHand(List<Token> playerHand);
13 }
14
15 public class RandomStrategy
16 {
17 }
18
19 public class BGStrategy
20 {
21 }
22
23 public class PDStrategy
24 {
25 }
```

Interface que permite que cada clase que la implemente defina su propia estrategia para pasarle a los jugadores y su propia forma de ordenar sus manos.

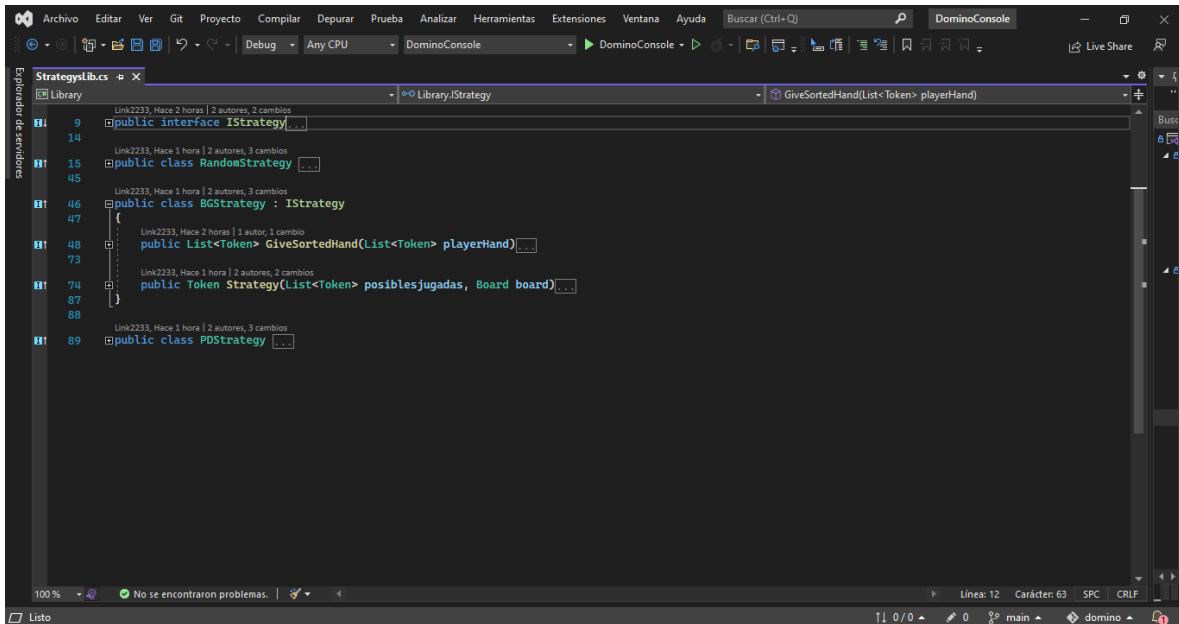
Estrategia Random:



```
15 public class RandomStrategy : IStrategy
16 {
17     public Token Strategy(List<Token> posiblesjugadas, Board board)
18     {
19         //escoge una ficha random entre las posibles jugadas y la devuelve.
20         var randomNumber = new Random(posiblesjugadas.Count);
21         int x = randomNumber.Next(0, posiblesjugadas.Count);
22
23         Token jugada = posiblesjugadas[x];
24         return jugada;
25     }
26
27     public List<Token> GiveSortedHand(List<Token> playerHand)
28     {
29         //organiza random la mano del jugador
30
31         List<Token> aux = playerHand;
32         playerHand.Clear();
33         playerHand = new List<Token>(aux.Count);
34
35         for (int i = 0; i < playerHand.Count; i++)
36         {
37             var randomNumber = new Random(aux.Count);
38             int x = randomNumber.Next(0, aux.Count);
39             playerHand.Add(aux[x]);
40             aux.RemoveAt(x);
41         }
42
43         return playerHand;
44     }
45 }
```

La estrategia Random implementa la interface IStrategy por lo cual toma métodos de esta. Lo que diferencia a este tipo de estrategia es que para jugar escoge una ficha random entre las posibles jugadas y la devuelve.

BGStrategy (estrategia bota gorda):



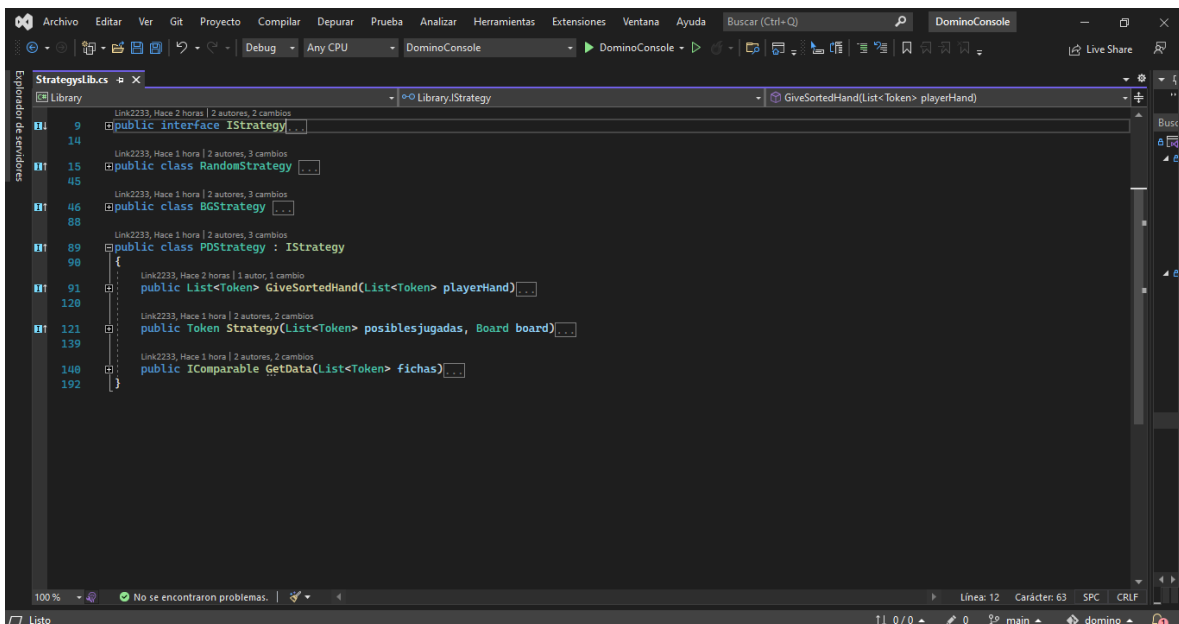
```

9  public interface IStrategy
14
15  public class RandomStrategy
45
46  public class BGStrategy : IStrategy
47  {
48      public List<Token> GiveSortedHand(List<Token> playerHand)
73
74      public Token Strategy(List<Token> posiblesjugadas, Board board)
87
88  }
89  public class PDStrategy

```

Este es otro de los tipos de estrategia definidas, lo que la diferencia de las demás es que para escoger la jugada busca entre sus posibles jugadas la ficha que más puntos tenga 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.

PDStrategy (Estrategia Protege Data):



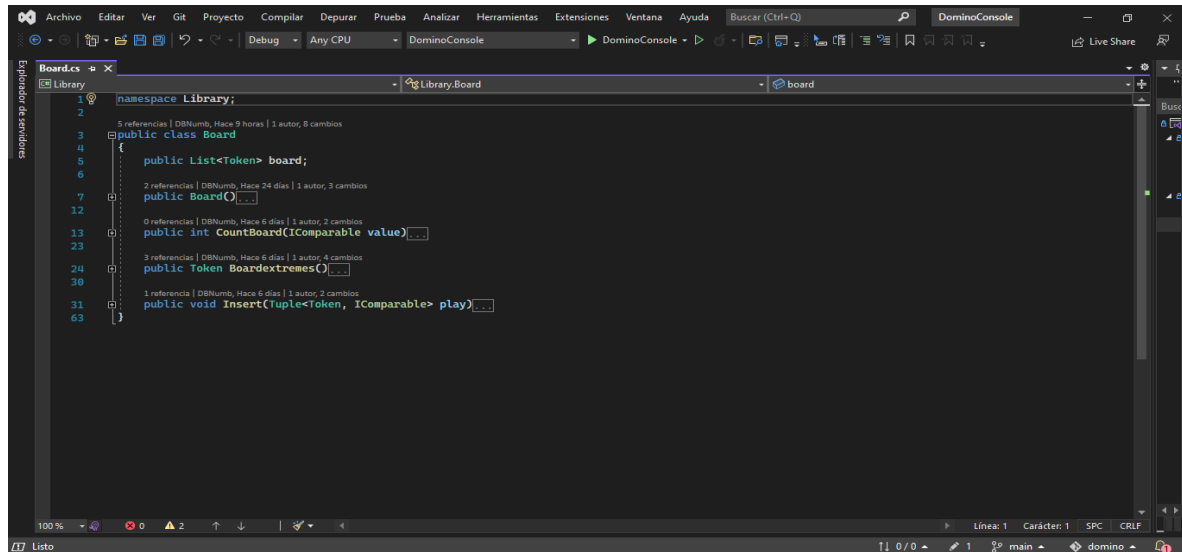
```

9  public interface IStrategy
14
15  public class RandomStrategy
45
46  public class BGStrategy
88
89  public class PDStrategy : IStrategy
90  {
91      public List<Token> GiveSortedHand(List<Token> playerHand)
120
121      public Token Strategy(List<Token> posiblesjugadas, Board board)
139
140      public IComparable GetData(List<Token> fichas)
192

```

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

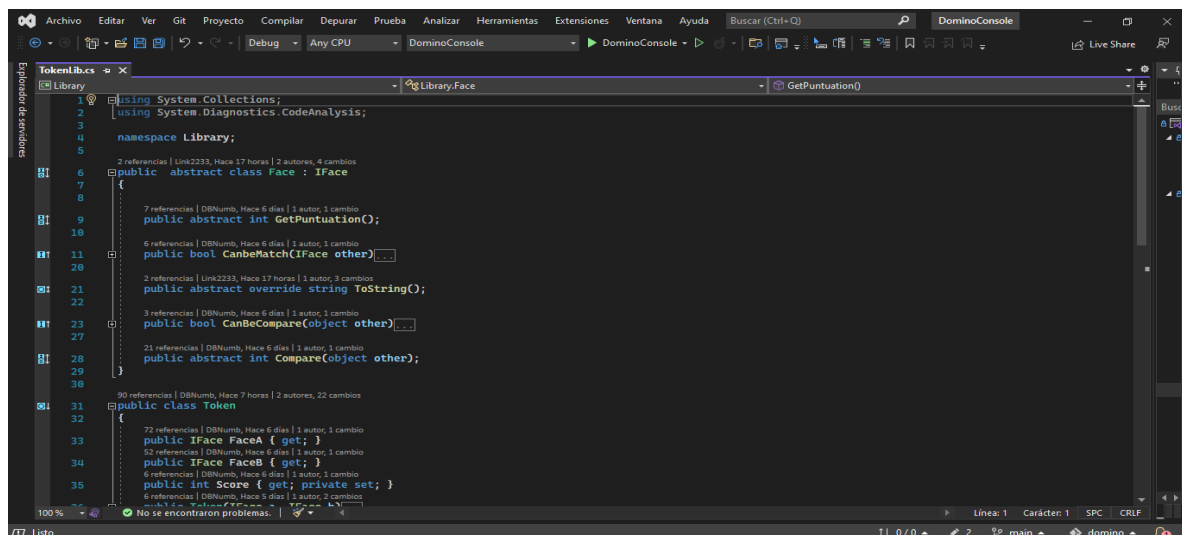
Clase Board:



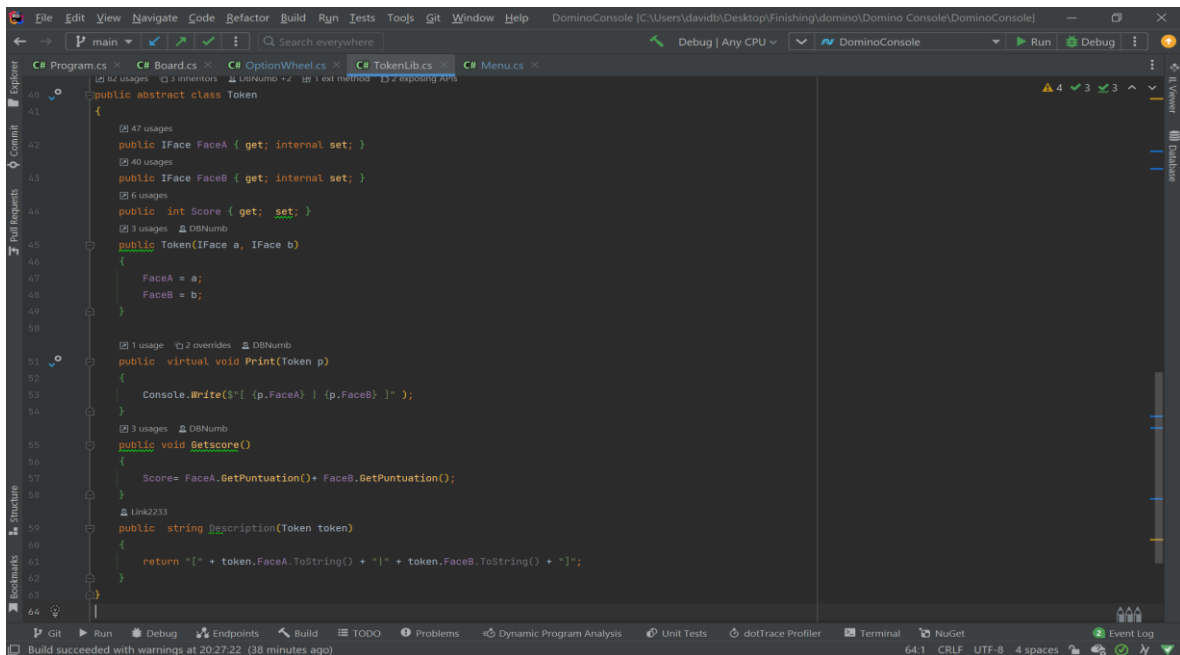
```
1 namespace Library;
2
3 public class Board
4 {
5     public List<Token> board;
6
7     public Board()
8     {
9     }
10
11     public int CountBoard(Comparable value)
12     {
13     }
14
15     public Token Boardextremes()
16     {
17     }
18
19     public void Insert(Tuple<Token, Comparable>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



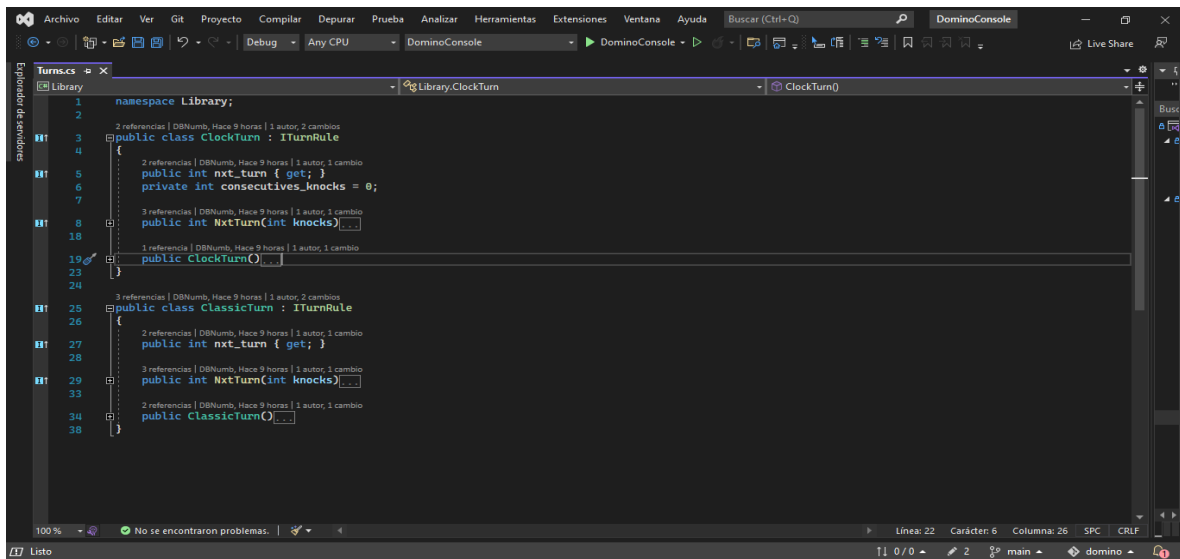
```
1 using System.Collections;
2 using System.Diagnostics.CodeAnalysis;
3
4 namespace Library;
5
6 public abstract class Face : IFace
7 {
8     public abstract int GetPuntuacion();
9
10     public bool CanBeMatch(IFace other)
11     {
12     }
13
14     public abstract override string ToString();
15
16     public bool CanBeCompare(object other)
17     {
18     }
19
20     public abstract int Compare(object other);
21 }
22
23 public class Token
24 {
25     public IFace FaceA { get; }
26     public IFace FaceB { get; }
27     public int Score { get; private set; }
28
29     public Token(IFace faceA, IFace faceB)
30     {
31     }
32 }
```



```
40 public abstract class Token
41 {
42     [47 usages]
43     public IFace FaceA { get; internal set; }
44     [40 usages]
45     public IFace FaceB { get; internal set; }
46     [6 usages]
47     public int Score { get; set; }
48     [3 usages]
49     public Token(IFace a, IFace b)
50     {
51         FaceA = a;
52         FaceB = b;
53     }
54     [1 usage] [2 overrides] [DBNumb]
55     public virtual void Print(Token p)
56     {
57         Console.WriteLine($"[{p.FaceA} | {p.FaceB}]");
58     }
59     [3 usages] [DBNumb]
60     public void GetScore()
61     {
62         Score = FaceA.GetPuntuation() + FaceB.GetPuntuation();
63     }
64     [Link2233]
65     public string ToString()
66     {
67         return "[" + token.FaceA.ToString() + "|" + token.FaceB.ToString() + "]";
68     }
69 }
```

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 GetPuntuation 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:

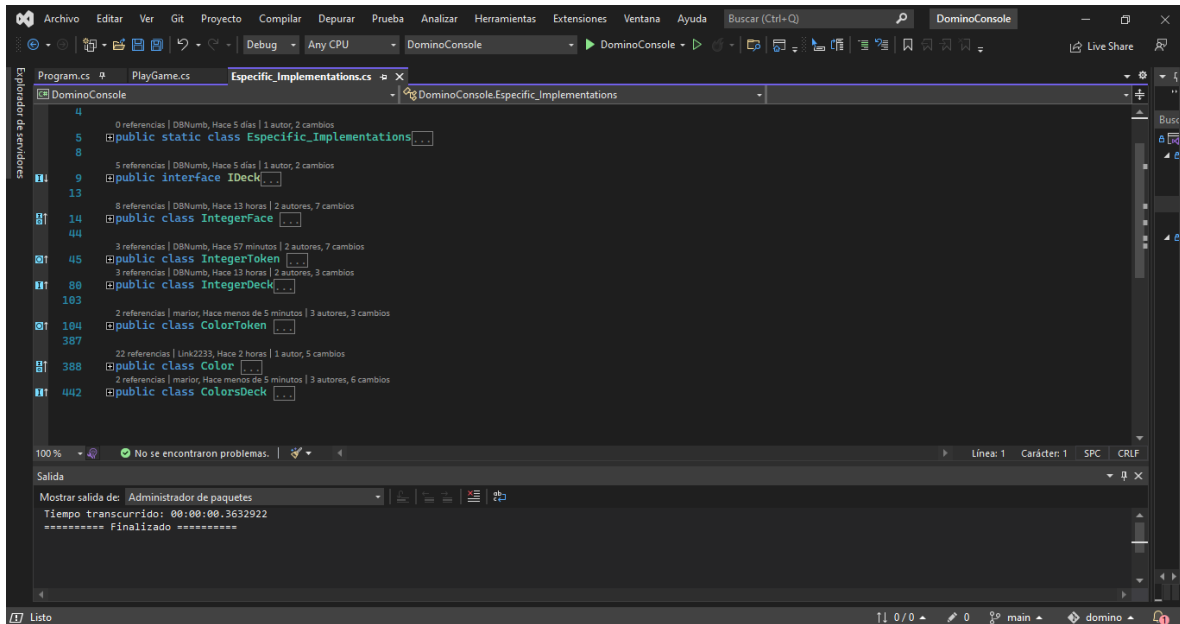


```
1 namespace Library;
2
3 public class ClockTurn : ITurnRule
4 {
5     2 referencias | DBNumb, Hace 9 horas | 1 autor, 1 cambio
6     public int nxt_turn { get; }
7     private int consecutive_knocks = 0;
8
9     3 referencias | DBNumb, Hace 9 horas | 1 autor, 1 cambio
10    public int NxtTurn(int knocks) {...}
11
12    1 referencia | DBNumb, Hace 9 horas | 1 autor, 1 cambio
13    public ClockTurn() {...}
14
15
16    3 referencias | DBNumb, Hace 9 horas | 1 autor, 2 cambios
17    public class ClassicTurn : ITurnRule
18    {
19        2 referencias | DBNumb, Hace 9 horas | 1 autor, 1 cambio
20        public int nxt_turn { get; }
21
22        3 referencias | DBNumb, Hace 9 horas | 1 autor, 1 cambio
23        public int NxtTurn(int knocks) {...}
24
25        2 referencias | DBNumb, Hace 9 horas | 1 autor, 1 cambio
26        public ClassicTurn() {...}
27    }
28 }
```

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`:

Juego con Fichas de Colores:

