Estructura del proyecto.

ConsoleInterface: Proyecto de tipo consola encargado de mostrar el desarrollo de una partida, así como la configuración de la misma.

Game: Biblioteca de clases que contiene implementaciones concretas de los aspectos modificables del juego.

Logic: Biblioteca donde están definidas las interfaces que se utilizaran y contiene las clases concretas que almacenan el estado del juego y controlan la ejecución del mismo.

¿Qué es una ficha?

En este proyecto se considera una ficha como un objeto que contiene dos objetos que implementan **IFace**, estos últimos definen como se comparan las fichas, que valor posee y como se representará esta. Esto permite tener una ficha [X-Y] donde X y Y podrían ser de cualquier tipo sin afectar el funcionamiento de los demás elementos.

Implementación de la clase Token que representa una ficha:

```
public class Token
{
    private IFace face1, face2;
    public Token (IFace _face1, IFace _face2)
    {
        face1 = _face1;
        face2 = _face2;
    }
   public IFace Face1 {get { return face1; } }
   public IFace Face2 {get { return face2; } }
}
```

¿Cuál será el conjunto de fichas que se utilizará?

De esto se encargarán las clases que implementen **IGenerator**, estas definen como estarán compuestas las fichas que se utilizarán, así como la cantidad. Esto solo afectará probablemente la duración del juego y como se visualizarán dichas fichas en la interfaz.

Implementación de un conjunto clásico de fichas:

```
public IEnumerable<Token> generateTokens(int n)
{
	List<Token> tokens = new List<Token>();

	for (int i = 1; i <= n; i++)
	{
		for (int j = i; j <= n; j++)
		{

		IFace face1 = new NormalFace(i);
		IFace face2 = new NormalFace(j);

		tokens.Add(new Token(face1, face2));

		}
	}
	return tokens;
}
```

¿Cómo se reparten las fichas?

Las clases que implementan **IDistribute** definen el método *HandOutTokens* que recibe como parámetros los jugadores a los que se le repartirán las fichas y el conjunto de fichas de donde se elegirán las mismas, estas implementaciones deciden si habrá fichas repetidas, la cantidad de fichas que se le repartirán a cada jugador y otras maneras de distribuir las fichas.

Implementación de *HandOutTokens* que distribuye n fichas de forma aleatoria:

```
public void HandOutTokens(IEnumerable<IPlayer> players, List<Token> tokens)

{
	Random random = new Random();
	foreach (IPlayer player in players)
	{
		for (int j = 0; j < n_players; j++)
		{
			int index = random.Next(0, tokens.Count);
			player.addToken(tokens[index]);
			tokens.Remove(tokens[index]);

		}
	}
}
```

¿Cuándo se puede jugar una ficha?

ITable define el comportamiento de una “mesa”, cualquier implementación de una “mesa” debe definir cuando una ficha “encaja” en dicha mesa y como “encaja” esta ficha con las restantes, aunque validar y agregar son comportamientos que en ocasiones no están ligados, se definió de esta forma para tener un control mayor sobre como se agrega una ficha a la “mesa”.

Validación clásica de una ficha:

```
      public bool Validate(IPlayer player, Token? token, IFace face, History
history)
        {

            if (token is null) return false;

            if (face1 == null || face2 == null) return true;

            if (!face.Equals(face1) && !face.Equals(face2)) return false;

            if (token.Face1.Equals(face) || token.Face2.Equals(face)) return
            true;

            return false;
        }
```

¿Cuál es la responsabilidad de un jugador?

IPlayer define el comportamiento de un jugador, su principal responsabilidad es seleccionar la ficha que va a jugar, aunque las clases que implementen esta interface deberán encargarse de otros aspectos como: eliminar una ficha o devolver el id de un jugador que no dejan de ser relevantes para el desarrollo de la partida.

El método encargado de devolver una ficha es:

```
public (Token, IFace) selectToken(ITable table, History history);
```

Se puede apreciar que este método devuelve la ficha que seleccionó el jugador, pero, además, retorna por qué lugar de la mesa jugará. Esto inicialmente se debe a que en el domino clásico se tiene una ficha que es válida por cualquiera de las caras disponibles para jugar, se realizó con la intención de darle al jugador la libertad de elegir por dónde quiere jugar.

¿Quién decide el orden del juego?

Otro aspecto con gran influencia en el desarrollo de una partida y con varias formas de definirse, merece tener una implementación no ligada a los demás comportamientos. Cualquier clase que pretenda definir el orden de un juego

deberá implementar **IOrder**, esta definirá *GetPlayer* que devuelve el jugador al que le toca jugar.

Ejemplo de implementación:

```
        public IPlayer GetPlayer(ITable table, IEnumerable<IPlayer> p, History
history)
        {
            IPlayer[] players = p.ToArray();
            int dif = n_players - players.Length;
            n_players -= dif;
            actual_turn -= dif;
            if (actual_turn + 1 == players.Length)
                actual_turn = -1;

            return players[++actual_turn];

        }
```

¿Acciones?

Las clases que implementen **IAction** podrán decidir qué hacer cuando un jugador elija la ficha que desea jugar, ejemplo de esto sería eliminar el jugador si este no tiene una ficha valida que jugar o agregarle una ficha cada vez que se pase.

Ejemplo de implementación donde se agrega la ficha a la mesa y se elimina del jugador:

```
        public void doSomething(IPlayer player, Token token, IFace face, ITable
table, ICollection<IPlayer> players,List<Token>  tokens,History history)
        {

            if (token != null)
            {
                table.addToken(token, face);
                player.RemoveToken(token);
            }
        }
```

¿Cuándo se acaba la partida?

Se pueden definir muchos motivos por los que una partida finaliza, el clásico, donde todos los jugadores se pasan o alguno se quedan sin fichas o podemos imaginar algo más extremista donde si un jugador se pasa dos veces se finaliza la partida. Estos comportamientos lo pueden definir cualquier clase que implemente **IFinish**.

Implementación de este último comportamiento:

```
      public bool Finish(ITable table, IEnumerable<IPlayer> p, History history)
        {
            foreach (var player in p)
            {
                int count = 0;
                string id = player.getID();
                for (int i = 0; i < history.IdHistory.Count; i++)
                {
                    if (!history.IdHistory[i].Equals(id)) continue;
                    if (history.TokensHistory[i] == null) count++;
                    if (count == 2) return true;
                }
            }
            return false;
        }
```

¿Cómo se eligen los ganadores?

Al igual que con la condición de finalizado, existen varias formas de escoger los ganadores, incluso manteniendo las demás reglas del domino clásico, este comportamiento estará definido por las clases que implementen **IWin.**

Ejemplo de implementación donde se eligen como ganadores los jugadores con menor cantidad de pases:

```
public IEnumerable<IPlayer> getWiner(ITable table, IEnumerable<IPlayer players,
History history)
 {
      List<IPlayer> winners = new List<IPlayer>();
      int min_passes = int.MaxValue;
      foreach (IPlayer player in players)
      {
        int currentCount = CountPasses(player, history);
        if (currentCount == min_passes) winners.Add(player);

        if (currentCount < min_passes) {
                winners.Clear();
                winners.Add(player);
                min_passes = currentCount;
           }
        }
            return winners;
 }
```

History

Clase encargada de almacenar las fichas jugadas y el id del jugador de dicha ficha. Esta clase será de gran ayuda para definir cuando un jugador se pasó, contar la cantidad de pases, además, se utilizará para definir las condiciones de finalizado y decidir los ganadores.

Manager

Esta clase es la encargada de obtener las implementaciones concretas de los comportamientos y controlar la ejecución de las mismas. También se encarga de notificar sobre los cambios de estado del juego.

¿Cómo se desarrolla la partida?

Consiste en una serie de pasos que se ejecutarán mientras no se haya cumplido la condición de finalizado. Primeramente, se pregunta a que jugador le toca jugar, después se obtiene la ficha que el jugador seleccionó, se valida y se agrega a dicha ficha a **History** además del id del jugador, posteriormente se ejecutarán las acciones. Cuando la condición de finalizado deje de cumplirse se obtendrán los ganadores.

Implementación del método *play:*

```
public void play() {
  I    Enumerable<IPlayer> winners = null;
       infoMonitor.Subscribe(infoHandler);

       while (!finish.Finish(table,players, history))
       {
              IPlayer player = order.GetPlayer(table,players, history);
              Token token;
              IFace face;
              (token,face) = player.selectToken(table,history);

              if (!table.Validate(player, token,face, history)) token = null;
              history.log(player.getID(), token);

              foreach(var action in actions)
              {
                     action.doSomething(player, token, face,table,
              players,tokens, history);
              }

              infoHandler.Update(table, history, players, winners,false);
       }

       winners = win.getWiner(table, players, history);
       infoHandler.Update(table, history, players, winners, true);

       infoMonitor.Unsubscribe();

}
```

Sobre la interfaz

Patrón Observer

Para mostrar el desarrollo de una partida se utiliza este patrón de diseño, asigna a la interfaz, en este caso una aplicación de consola, la única responsabilidad de representar la información contenida en un objeto de tipo **Info**. Esto da la posibilidad de implementar otras formas de mostrar el desarrollo de la partida sin comprometer el funcionamiento del resto del código.

InfoHandler que implementa **IObservable<Info>** es el encargado, mediante los métodos *Update* y *Notify* de avisar a la interfaz sobre los cambios.

InfoMonitor que implementa **IMonitor** se ocupa de mostrar los cambios. Cualquier variación de interfaz que se implemente para mostrar el desarrollo del juego deberá implementar **IMonitor**.

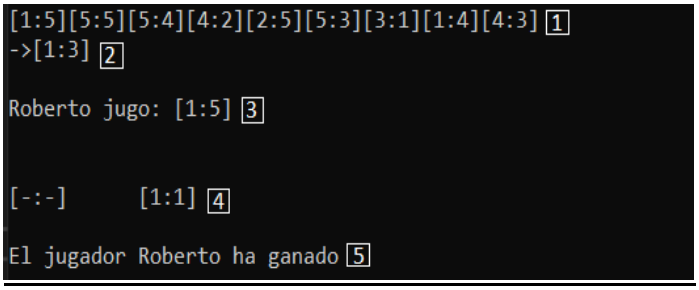
¿Cómo se configura una partida?

GameSettings se encarga de establecer los tipos(types) de las implementaciones concretas que se usarán durante la partida, para mostrar las opciones se utiliza la propiedad *Description* que esta definida en cada una de estas implementaciones. Cuando el usuario elija una de estas opciones se creará una instancia del tipo correspondiente que después se pasará a *BuildGame* para establecer las reglas y comportamientos que tendrá la partida.

Ejemplo de cómo se establecen dichos tipos:

```
private static Type[] typesofOrders = { typeof(NormalOrder),
typeof(ReverseOrder) };
```

Ejemplo de como se muestra una partida en consola:



1. Fichas de la mesa
2. Indica por que lado de la mesa se puede jugar
3. Muestra la ficha que selecciono el jugador en cada turno
4. Muestra las fichas de los jugadores.
5. Se muestran los ganadores.