Proyecto de Programación II. Facultad de Matemática y Computación. Universidad de La Habana. Curso 2022

> Brian Ameht Inclan Quesada. C-211
> Dariel Martinez Perez. C-211

El objetivo del presente proyecto es elaborar un programa que permita acoplar e interpretar los diferentes tipos de desarrollos del juego de Dominó. Con la intención de que el código sea fácil de mantener y ampliar con el tiempo, nos hemos basado en los principios SOLID para eliminar malos diseños que devengan en la refactorización del código fuente por parte de otro programador hasta que sea legible y extensible, ademas de que presenta una estrategia global del desarrollo ágil y adaptativo de software.

Implementación:

Para nuestra implementación, dividimos nuestro código en dos carpetas: “Domino_Server” y “DominoEngine”. En “DominoEngine” se encuentran las clases que implementan el juego, y en “Domino_Server” se encuentran las clases que implementan la interfaz gráfica.

DominoEngine:

Aquí se encuentra toda la jerarquía de clases que nos permite armar la estructura sobre la cual esta elaborado el juego. Con diferentes tipos de classlibs en las cuales se interpretan funcionalidades y desarrollan así el transcurso del juego. Comencemos explicando qué objetos usamos para nuestro juego:

Ficha<T>

```
public List<T> Valores

public virtual int Valor

public bool IsDoble()
```

Con esta clase queremos interpretar lo que se puede entender como una ficha en un juego de domino, la cual se caracteriza por los valores

los cuales son totalmente genéricos, ademas posee propiedades como el peso que posee la misma para un determinado juego

Board<T>

```
public void Repartir()

public IEnumerator<Iestado<T>> Jugar()

public void Reset()

public IEnumerator<Iestado<T>> GetEnumerator()

IEnumerator IEnumerable.GetEnumerator()
```

Board es una clase genérica sobre la cual “ocurre el juego”, es la encargada de arrancar el motor del juego en su método **Jugar()** el cual ejecuta una serie de estados con toda la información que hay por cada jugada que se hace. Ademas de esto posee un método repartir para así otorgarles las fichas a los jugadores según el criterio por el cual se entregaran las fichas a cada jugador.El método **Reset()**, actualiza el tablero para empezar un juego nuevo.

Jugador<T>

```
public Jugador(String Nombre, Istrategy<T> Strategy)

public string Nombre

public List<IFicha<T>> Hand

public void AddMano(List<IFicha<T>> Posible_Pieces, IRules<T>
reglas)
public void Hand_Reset()

public int Valor(IValor_Hand<T> Criterio)

public (int, IFicha<T>) Jugar(Iestado<T> estadoactual)

public bool Contains(Iestado<T> estadoactual)

public void AddFicha(List<IFicha<T>> Posible_Pieces)
```

Como bien sabemos, el jugador es quien juega el papel fundamental en este juego, por ende el cual es quien lo hace divertido, en nuestra

implementacion del mismo, viene siendo como una cascara que espera a que le pasen su contenido ya que decidimos que cada jugador se le pase como parámetro en su constructor una estrategia que es la que lo va a caracterizar, ademas de esto entre lo mas importante posee las propiedades Hand y Nombre que como tal indican una es para obtener acceso a su mano y la otra a su nombre respectivamente. El método **AddMano()** se emplea para que el jugador recoja la mano que le corresponde según un criterio dado para cojer las fichas, **HandReset()** elimina todas las fichas que tiene el jugador en su mano, **Valor()** según un criterio de pesos entre fichas obtiene el peso que tiene la mano de ese jugador para el juego, **Jugar()** devuelve un entero y una ficha en la cual el entero es el lugar de la mesa por donde se va a jugar y la ficha a jugar respectivamente según el criterio pasado por las reglas(en caso de que no haga trampa), en caso de no llevar en entero es un valor negativo, **Contains()** dado el estado en el cual se encuentra el juego, verifica según distintas condiciones si puede tirar una ficha o no, **AddFicha()** agrega de una serie de fichas, una a la mano.

Estados<T>

Clase encargada de almacenar en que estado se encuentra la partida despues de una jugada, como por ejemplo la cantidad de fichas que hay en la mesa o la cantidad de fichas de un jugador, si se paso o no etc.

Manager_Rounds<T> , Manager_Points<T>

Clases encargadas en el manejo de la cantidad de rondas y los ganadores de las mismas en el caso de **Manager_Round** y de verificar y otorgar ganadores de un juego en cuanto a la cantidad de puntos especificada en caso de **Manager_Points**.

Estrategias de los Jugadores

Como bien comentamos arriba un la estrategia es lo esencial para el jugador, adelante veremos los diferentes tipos de estrategias que creamos en nuestro juego, todas implementando la interfaz **Istrategy<T>** la cual posee el método **Jugar()**

```
public interface Istrategy<T> where T : IComparable
{
  (int, IFicha<T>) Jugar(Iestado<T> estadoactual, List<IFicha<T>> Hand);
}
```

Random_Player<T>

Este jugador como bien lo indica su principal misión es jugar random ante cualquier situación, para ello seleccionamos entre las fichas de la mano las posibles a jugar, de esas escojemos una random, ya una vez que tenemos esa ficha verificamos todos los lugares posibles por el cual se puede lanzar, y de estos se selecciona uno random y se lanza la ficha

```
public (int, IFicha<T>) Jugar(Iestado<T> estadoactual, List<IFicha<T>>
Hand)
```

Botagorda<T>

Este jugador esencialmente juega para soltar la ficha que mayor peso tenga en su mano según un IComparer

```
protected bool organizado = false;
public (int, IFicha<T>) Jugar(Iestado<T> estadoactual, List<IFicha<T>>
Hand)
```

Pro_Player<T>

Aunque no es el jugador inteligente que todos esperamos, este jugador elige una de las dos estrategias de arriba segun el estado en el que se encuentre el juego

Criterios y Reglas:

ICouple<T>

Con esta interfaz intentamos modelar el problema de las parejas en el juego, el método **Couples()** que devuelve un diccionario en el cual los keys son las teams y los values los jugadores que pertenecen a cada team.

```
Dictionary<int, List<IJugador<T>>> Couples(List<IJugador<T>>
List_Of_Players)
```

IValor_Hand<T>

Cualquier clase que implemente esta interfaz tendrá solo un objetivo, y sera el de asignar a cada jugador un peso especifico de su mano en el juego devolviéndolo así en el método **Valor()** que recibe como parámetros una lista de fichas y devuelve un entero con el peso de esa lista de fichas.

```
int Valor(List<IFicha<T>> item)
```

En este caso se puede implementar para calcular el valor usual de la mano en el juego o calcular por su valor promedio, como hacemos en el juego.

IPosible_Pieces<T>

Interfaz que encapsula la idea de generar todas las posibles piezas de un juego, se puede implementar de diferentes maneras ya que si quieres puedes excluir o agregar e incluso clonar y repetir fichas, a gusto e como quieran implementarla, en nuestro caso hemos hecho dos implementaciones, una es la usual del domino clásico y otra que le hemos llamado **Posible_Pieces_Simetric** basándonos en el lenguaje de la lógica y las relaciones simétricas en las cuales si (x,y) pertenece a la relación, entonces (y,x) también pertenece a la relación.

IRepartir<T>

Basándonos en que todos los jugadores deben saber de que forma se esta repartiendo o eligiendo las fichas que le corresponde a cada cual, hemos creado esta interfaz que en su implementacion mediante el método **Repartir()** devuelve una cantidad de fichas a cada jugador según como se desee repartir en ese mometo

```
List<IFicha<T>> IRepartir<T>.Repartir(List<IFicha<T>> Posible_Pieces,
int CantFichaXjugador)
```

IUnion<T>

Interfaz que encapsula la forma en la que esta correcto unir una ficha con otra, ya sea por sus colores o por cualquier tipo que sea IComparable, con esto seria posible unir fichas de tipo entero con fichas de tipo string sin ningun problema ya que como ambos se pueden

comparar solo queda hacer una pequeña implementacion donde esto sea posible.

ITipeOfGame<T>

Aqui dejamos que el programador pueda construir su juego a gusto utilizando las clases e interfaces anteriores, como bien lo dice **ITipeOfGame** es una interfaz que tiene un método que devuelve un **IEnumerable<IEstados<T>>** lo cual se hace para que con facilidad cada vez que se realice una acción en el juego se actualicen los estados y los diferentes parámetros y poder devolverlos con la instrucción yield return de una manera sencilla. En nuestro juego solo hicimos dos implementaciones de la misma, pero como bien sabe existe una infinidad de juegos que puede crear a partir de su implementacion, es solo dejar volar la mente.

Rules<T>

Y por ultimo las reglas, Rules es una clase genérica encargada de modelar las reglas del juego mediante una serie de propiedades y métodos implementados a partir de la interfaz genérica **IRules<T>** para verificar que el juego se desarrolle al margen de lo previsto por el usuario.

```
public ITipeOfGame<T> TipeOfGame { get; private set; }

public IValor_Hand<T> Criterio { get; private set; }

public IPosible_Pieces<T> Posible_Pieces { get; private set; }

public IRepartir<T> Repartir { get; private set; }

public int Cantidad { get; private set; }

public ICouple<T> Couples { get; private set; }

public IComparer<int> Comparar { get; private set; }

public IUnion<T> Compare { get; private set; }
```

Como se ve en la imagen eta compuesta de múltiples propiedades las cuales son los tipos de los que hablamos anteriormente para que en ellas se pueda tener todo tipo de criterio acerca de cualquier evento que suceda en el juego y así los jugadores lo sepan. Son propiedades de solo

lectura ya que ningún jugador debe acceder a una de estas reglas y cambiarlas en el transcurso del juego.
Esta clase también posee diferentes tipos de métodos como se ve adelante.

```
  public Iestado<T> Actualizar_Board(int Posicionn, IFicha<T> Ficha,
Iestado<T> estado)

    public (IJugador<T>, List<IJugador<T>>) Winner(List<IJugador<T>>
jugadors)

    public (IJugador<T>, List<IJugador<T>>) Winner(IJugador<T> jugador,
List<IJugador<T>> Jugadores)

    public Iestado<T> Not_Pieces(IJugador<T> jugador, Iestado<T> estado)

    public bool IsValid(int Posicionn, IFicha<T> Ficha, Iestado<T>
estado)
```

En el caso de **Actualizar_Board()** devuelve un nuevo estado del juego luego de recibir una ficha, la posición por la que se va a jugar esa ficha y el estado en el que se encuentra el juego, según estos parámetros se modelan una serie de pasos que actualizan de una forma u otra el estado en el que se encontraba el juego en el momento del evento.
Winner() es un método con dos sobrecargas en las cuales devuelven el jugador ganador y su equipo, la primera recibe una lista de jugadores con el fin de si se tranca el juego, verificar según los criterios de comparacion quien es el ganador y así el equipo al que pertenece, el segundo recibe un jugador y la lista de los jugadores en mesa, con el fin de que si un jugador se pega y gana, elegir el equipo al que pertenece el mismo.
Not_Pieces() método encargado de que si un jugador se pasa actualizar el estado y guardar las fichas que no lleva cada jugador en un diccionario de jugadores(keys) y fichas a las que se ha pasado(values)

IsValid() verifica si dado una ficha, una posición por l que se va a jugar esa ficha y el estado en que se encuentra el juego, es posible ejecutar ese movimiento según las reglas y diferentes criterios sobre los que se juegan.

DominoServer:

Aquí se encuentra la implementación de la parte gráfica del proyecto la cual esta desarrollada en Blazor utilizando HTML y CSS para el diseño de las páginas y los recursos utilizables. La implementación se basa en los siguientes aspectos conexos a la biblioteca de clases de DominoEngine:

```
public class Params
{
    //Propiedades de las paginas
    public int Mode { get; set; }
    public List<int>? _Numbers;
    public int _NumbOP { get; set; }
    public List<Istrategy<int>> _PlayerType { get; set; } = new List<Istrategy<int>>{};
    public static ICouple<int>? _Teams { get; set; }
    public static IRepartir<int>? _Repart { get; set; }
    public string? _VictoryP { get; set; }
    public static IUnion<int>? _UnionType { get; set; }
    public static IComparer<int>? _Comparer { get; set; }
    public static IValor_Hand<int>? _ValorHandType { get; set; }
    public static IPosible_Pieces<int>? _Pieces { get; set; }
    public static ITipeOfGame<int>? _GameType { get; set; }
```

La clase **Params** contiene todos los parametros modificables en el juego. Estas opciones se establecen en la página **Main_Menu** a través de formularios introducidos en HTML y se recogen en el grupo de métodos de dicha página para ser utilizados posteriormente cuando la partida comience.

```
@code{
    private int numbOP;
    protected void OnValidSubmit(){
        System.Console.WriteLine("ok");
    }
    public void _Gamemode(ChangeEventArgs gm) => Params._Gamemode(gm.Value.ToString());
    public void _NumbOP(ChangeEventArgs numb){
        Params._NumberOfPlayers(numb.Value.ToString());
        numbOP = Params._NumbOP;
    }
    public void _PlayersT(ChangeEventArgs type) => Params._PlayersT(type.Value.ToString());
    public void _Teams(ChangeEventArgs tm) => Params._MyTeams(tm.Value.ToString());
    public void _Repart(ChangeEventArgs rp) => Params._MyRepart(rp.Value.ToString());
    public void _PuntRules(ChangeEventArgs punt) => Params._PuntRules(punt.Value.ToString());
    public void _Union(ChangeEventArgs union) => Params._Union(union.Value.ToString());
    public void _Comparer(ChangeEventArgs comp) => Params._MyComparer(comp.Value.ToString());
    public void _ValorHand(ChangeEventArgs value) => Params._ValorHand(value.Value.ToString());
    public void _Pieces(ChangeEventArgs piece) => Params._PossiblePieces(piece.Value.ToString());
    public void _GameType(ChangeEventArgs type) => Params._TypeOfGame(type.Value.ToString());
}
```

La página **GameOn** se encarga de inicializar todos los párametros recogidos en la clase **Params** para dar comienzo a la partida. Después de creadas las instancias de clases correspondientes a la parte lógica se

generan, a través del método **OnInitialized(),** 4 jugadores con nombre aleatorio, se establecen las reglas del juego y se prepara el tablero.

```
protected override async void OnInitialized()
{
    Random random = new Random();

    IRules<int> Rules0 = new Rules<int>(Repartir_Usual, ValorHand,
        Params.Mode, Posible_Pieces, game0, teams, comparer, union);

    for (var i = 0; i < Params._NumbOP; i++)
    {
        jugadores.Add(new Jugador<int>(Params.Bots[random.Next(0, 10)], Params._PlayerType[i]));
    }

    board = new Manager_Points<int>(Params._Numbers.ToArray(), jugadores, Rules0, int.Parse(Params._VictoryP));
}
```

El juego comienza al presionar el botón **Jugar** que hace una llamada al método homónimo, el cual se encarga de imprimir el estado de la partida y el tablero a medida que se recorre el **IEnumerable Pieces_In_Board**. El **Task OnChange()** actualiza las fichas que aparecen en pantalla a medida que la partida avanza y los métodos **ImageMaker()** dibujan las fichas insertando imágenes predefinidas de la carpeta **wwwroot**. En la carpeta **wwwroot** también se almacenan los estilos de CSS visibles en la página dentro del archivo **site.css**.

```
protected async Task Jugar()
{
    string text = "";
    foreach (var item in board)
    {
        text = $"Tab: [";
        foreach (var item1 in item.Pieces_In_Board)
        {
            text += " " + $"{item1.Value}" + " ";
        }
        text += "]";
        text += $"{item.Progreso}" + "\n";

        moves.Add(text);
        piece_in_table = item;
        last_play = item.Pieces_Played[item.Pieces_Played.Count-1];

        OnChange();

        await Task.Delay(1000);

        StateHasChanged();
    }
}
```