Reporte

Alumnos

- `*Sebastián Suárez Gómez*`

- `*Héctor Miguel Rodríguez Sosa*`

Instrucciones para abrir

Consola:
```
cd App
dotnet run
```
Blazor:
```
cd BlazorApp
dotnet watch run
```

Introducción

Este reporte contiene la ficha técnica del proyecto de 2do semestre de los estudiantes mencionados anteriormente. El proyecto consiste en un modelo de un juego de dominó de manera ¨modular¨ (más de ello a continuación), incluyendo varios bots de jugadores que se adaptan al juego de manera automática.

La idea detrás del proyecto es crear un modelo de juego de dominó tal que se puedan modificar varias formas del juego como la manera en la que se pueda ganar, la forma en la que se reparten las fichas, por solo citar algunas. Anteriormente se refirió al juego como modular; esto es así porque el juego en sí tiene una clase principal llamada GameObject, tal que se estructura de la siguiente forma:

```
public class GameObject
{
    public int MaxHandSize { get; }
    public int BoardSize { get; set; }
    public List<Round> Rounds { get; set; }
    public IPlayer[] Players { get; }
    public IPlayer CurrentPlayer { get; set; }
    public IPlayer? Winner { get; set;}
    public Settings Settings{ get;}
}
```

Esta clase recibe en su constructor todas las opciones diferentes que puede recibir el juego o módulos, la mayoría bajo la clase de Settings que los recoge. En total son 5: Winnable, HandCounter, Shuffler, Board y Rounder. Estos módulos son representados por la interfaz de la cual hereda cada implementación de cada uno de ellos. Cada uno funciona por su cuenta y no tiene dependencia el uno del otro para su funcionamiento, de manera que pueden usarse diferentes formas de cada módulo y no afecta a otro que se use. Más de cada módulo a continuación.

Los únicos que por su brevedad no serán mencionados a continuación son MaxHandSize y BoardSize, los cuales son la mayor cantidad de fichas que se puede tener en la mano y el doble máximo de las fichas.

La clase GameObject tiene un método principal, AutoPlay. El cual como su nombre indica hace q el juego se juegue por si solo, de esa manera cualquier interfaz gráfica solo se tiene que preocupar por hacer algo con el tablero o las rondas

Piece

```
public abstract class Piece
{
    public int Left { get; set; }
    public int Right { get; set; }

    public abstract bool CanPlay(Board board);
    public Piece Rotate()
    {
        int c = Left;
        Left = Right;
        Right = c;
        return this;
    }
    public int GetValue() => Left + Right;
    public abstract bool MatchLeft(int num);
    public abstract bool MatchRight(int num);
    public bool Match(int num) => MatchLeft(num) || MatchRight(num);
    public override string ToString() => $"[{Left}|{Right}]";
}
```

Esta clase abstracta representa como su propio nombre dice la ficha, el token con que se juega. Básicamente esta clase tiene varios métodos ya predefinidos pero el método CanPlay deberá adaptarse para ver cómo se puede jugar una ficha en un determinado tablero. Las propiedades Right y Left representan el valor de cada lado de la ficha, el método GetValue suma ambos lados, el método Match responde a los métodos MatchRight y MatchLeft, los cuales a su vez varían en dependencia de la implementación de la ficha.

- RegularPiece donde es la manera estándar de colocar una ficha (que el lado de colocar coincida con alguno de los extremos del tablero)

- DoubleEvenPiece donde la única manera de colocar una ficha doble en el tablero es que haya un número par de fichas ya colocadas

Player

```
public abstract class BasePlayer : IPlayer
{
    public List<Piece> Hand { get; protected set; }
    public abstract Piece Play(Board board);

    protected IEnumerable<Piece> GetPossiblePieces(Board board) =>
                    this.Hand.Where(piece => piece.CanPlay(board));
    public override string ToString() => GetType().Name;
    public BasePlayer(){
        Hand = new();
    }
}
```

Son todos los posibles jugadores que pueden haber en el juego. Por ahora nuestra implementación del juego solo dispone de funcionalidad para bots. La propiedad Hand representa todas las fichas que se encuentran en la mano del jugador y el método Play devuelve la ficha que este jugador va a jugar en el tablero y a la vez la quita de la mano del jugador. Por ahora las implementaciones que tenemos de jugadores bots de Domino son:

- El mítico jugador temidos por todos: BotaGorda o PlayerMostValue para los finos, que siempre jugará la ficha con más valor de su mano

- El PlayerRandom que siempre jugará una ficha aleatoria de su mano

- El PlayerRanValue o también llamado en los círculos selectos como el Borracho, un bot que siempre jugará como BotaGorda menos en ciertas circunstancias (por lo general es que lo llama la mujer)

- Un jugador SemiInteligente que mira la cantidad de fichas que se han jugado y juega acorde a ellas

Round

```
public class Round
{
    public IPlayer Player { get; set; }
    public Piece Piece { get; set; }
}
```

En esencia no tiene mucho que decir esta clase. Simplemente lleva cada par de jugador y ficha jugada. En la clase juego existe una propiedad de una lista de estas para llevar de manera precisa todas las jugadas de cada jugador en

el juego. Por regla convencional siempre que un jugador juega una ficha se añade un nuevo par a esta lista y si un jugador no puede jugar ninguna ficha, el equivalente a pasarse es jugar una ficha null.

Board

```
public abstract class Board
{
    public List<Piece> PiecesOnBoard { get;  set; }
    public List<Piece>? Deck { get; set; }
    public abstract List<Piece> Generate(int maximumInput);
    public override string ToString() => GetType().Name;
}
```

Esta clase tiene 2 propiedades: una lista de todas las piezas que se han colocado en el tablero y todas las que o no se han repartido o no se han colocado en el tablero.

Ademas tiene el método Generate. Este método en su implementación debe generar todas las posibles fichas de cierto tipo y agregarlas al Deck, usando como máximo doble el número que se le pasa. Esto lo hace el juego de manera automática una vez que se crea, usando su propiedad de BoardSize como número máximo; por tanto es necesario que esta implementación haga eso.

En realidad la implementación de cada Board no varía mucho, simplemente lo que cambia es que el método Generate usa los diferentes tipos de fichas que existen implementadas, por eso la función ficha => tablero es inyectiva.

HandCounter

```
public interface IHandCounter
{
    int GetHandValue(IPlayer player);
}
```

Es una interfaz con implementaciones bastante sencillas de entender. Recibe en su único método un jugador y devuelve la manera en la que la implementación quiere contar el valor de cada ficha de la mano de ese jugador. Por ahora solo tenemos 2 implementaciones de esta interfaz:

- RegularHandCounter es la implementación estándar de esta interfaz ya que simplemente devuelve la suma del valor de todas las fichas de la mano de un jugador

- DoubleDoubleHandCounter funciona de manera similar a su versión estándar salvo por el hecho de que si la ficha es doble (por doble entender que ambas caras son iguales) su valor se duplica

Winnable

```
public interface IWinnable
{
    public bool Won { get; set; }
    public bool EndCondition(IGame game);
    public IPlayer Winner(IGame game);
}
```

Esta interfaz representa todas las formas en las que se ganar un juego, incluido la forma en la que se elige el ganador de dicho juego. Consta de dos métodos principales: EndCondition y Winner. Al final de cada ronda se revisa si se cumple EndCondition y si el resultado es verdadero se pasa a elegir un ganador.

El método EndCondition recibe como parámetro el juego que se está jugando y da igual como sea su implementación pero en esencia debería determinar si un juego se ha terminado o no. Hay varias formas en las que un juego puede terminar: ya sea porque se trancó y porque alguien lo ganó. En todas nuestras implementaciones se verifica cada una pero en el caso de que haya sido ganado el juego entonces la propiedad de Won cambia a verdadero y se debe elegir un ganador.

El método Winner como su propio nombre indica devuelve al ganador del juego. Por lo menos en nuestras implementaciones, se chequea siempre la propiedad de Won primero, si esta es verdadera entonces significa que el último en jugar fue el que ganó el juego y por tanto se devuelve ese mismo. Si es falsa entonces devuelve el jugador tal que la suma de la puntuación de cada ficha de su mano es la menor. Esta suma se hace acorde con el IHandCounter que tenga el juego. Por ahora las implementaciones que tenemos de esta interfaz son:

- RegularWinnable donde gana el jugador que no tenga fichas en su mano

- DropDoubleBlank donde gana el jugador que haya jugado la mítica ficha Doble Blanco

Rounder

```
public interface IRounder
{
    public IPlayer NextPlayer(IGame game);
}
```

Esta interfaz es la encargada de decir quien es el siguiente jugador en la cola del juego. No tiene mucha magia, simplemente se llama al final de cada Round y varía en dependencia de la implementación. Por ahora las implementaciones que tenemos son:

- Clockwise donde se juega en sentido de las manecillas del reloj

- CounterClockwise donde se juega en contra de las manecillas del reloj

- SkipTurn donde si un jugador se pasa entonces el siguiente en jugar sera el anterior, pero en circunstancias normales funciona en el sentido de las manecillas del reloj

Shuffler

```
public interface IShuffler
{
    public void Shuffle(IGame game);
    public void ShufflePlayer(IPlayer player, IGame game);
}
```

Es la encargada de repartir las fichas a cada jugador al principio de cada partida. Igual que el anterior este no tiene mucha magia, simplemente se llama al principio de cada partida y depende mucho de la implementación. Por ahora las implementaciones que tenemos son:

- RegularShuffler donde se asigna a cada jugador una ficha al azar del mazo del tablero

- SortedByLeftShuffler donde se ordena el mazo del tablero de acuerdo al valor de la izquierda de la ficha y siempre se le da al jugador la primera ficha de esa lista y se va quitando

- SortedByValueShuffler funciona igual que el anterior pero se ordena por el valor de la ficha

Observaciones

La idea detrás de hacer el modelo de juego de dominó se basa en 2 principios básicos: flexibilidad y personalización. Esta forma de modelar un juego de dominó de manera modular es extremadamente personalizable, incluso más a medida que va avanzando el tiempo, ya que permite acoplar cada módulo a medida que nos vaya pareciendo la necesidad. Pero sin perder la generalidad se vuelve flexible porque cada módulo no tiene que depender de otro formando así un Loose-Coupling, que permite que por muy diferente que sea un módulo de otro siempre se vaya a poder implementar sin tener que cambiar nada en la interfaz o en la biblioteca.