Proyecto de Año de Ciencias de la Computación

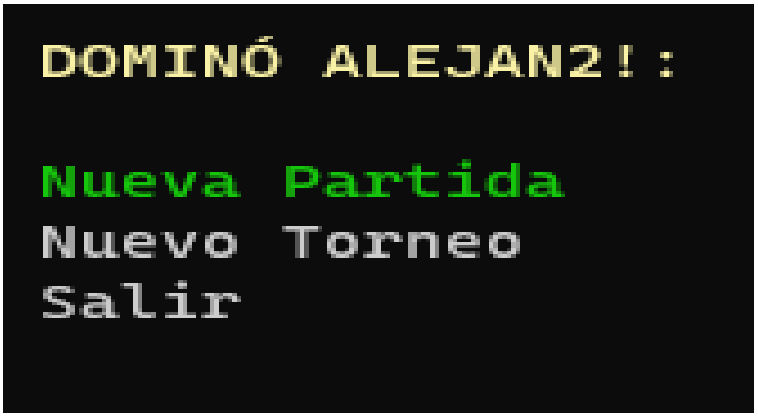
Curso 2021-2022

Integrantes:

Alejandro Solar Ruiz C212

Luis Alejandro Rodríguez Otero C211

Objetivo: Implementar un programa que permita modelar un juego de Dominó permitiendo modificar los aspectos variables de este así como la implementación de una serie de estrategias



Reporte:

El proyecto cuenta con una aplicación de consola llamada Program.cs dentro de la carpeta Main y una biblioteca de clases en la carpeta Engine que a su vez está dividida en Core e Implementations:

En Core tenemos las clases e interfaces que no dependen de un tipo de implementación concreta de cierta característica. Mientras que en Implementations tenemos las implementaciones concretas de las características que expondremos más adelante.

Mapa de la Carpeta Core:

- **Core**:
  - AuxiliarMethods:
    - Extensors:
      - IEnumerableExtensors.cs
      - SimpleCloneableExtensors.cs
    - Auxiliar.cs
    - AuxiliarGenericMethods.cs
    - Reflection.cs
  - CloneableItems:
    - Board.cs
    - Couple.cs
    - Event.cs
    - Move.cs
    - Piece.cs
    - Player.cs
    - Register.cs
  - Creators:
    - GameCreator.cs
    - PlayerCreator.cs
    - TournamentCreator.cs
  - GameManagers:
    - Game.cs
    - Round.cs
    - Tournament.cs
  - Interfaces:
    - ICloneable.cs
    - IDealer.cs
    - IDefaultGame.cs
    - IEvaluator.cs
    - IScore.cs
    - ISimpleClone.cs
    - IStopCondition.cs
    - IStrategy.cs
    - ITurnOrderer.cs
    - IValid.cs
    - IWinCondition.cs

Mapa de la carpeta Implementations:

- **Implementations**:
  - Dealers:
    - AlphabeticalDealer.cs
    - HigherDealer.cs
    - RandomDealer.cs
  - DefaultGames:
    - ClassicGamesImplementations:
      - Double6.cs
      - Double9.cs
    - ClassicGames.cs
  - Evaluators:
    - DefaultEvaluator.cs
    - DoubleEvaluator.cs
    - EvenEvaluator.cs
    - OddEvaluator.cs
  - Scores:
    - PointScore.cs
    - WinScore.cs
  - StopConditions:
    - BlockStop.cs
    - EmptyHandStop.cs
    - EqualHandsStop.cs
  - Strategies:
    - CleverStrategy.cs
    - CouplesStrategy.cs
    - DoublesStrategy.cs
    - DrunkStrategy.cs
    - GreedyStrategy.cs
    - HumanStrategy.cs
    - RandomStrategy.cs
  - TurnOrderers:
    - CouplesOrderer.cs
    - RandomOrderer.cs
    - ValueOrderer.cs
  - Valids:
    - ClassicValid.cs
    - HigherExtremeValid.cs
    - HigherValueValid.cs
  - WinConditions:
    - HigherPointsWin.cs
    - LowerPointsWin.cs
    - EmptyHandWin.cs

Analicemos el contenido de Core:

Esta carpeta se encuentra dividida en:

- AuxiliarMethods
- CloneableItems
- Creators
- GameManagers
- Interfaces

Describiremos el contenido de cada una a continuación.

AuxiliarMethods:

Como su nombre lo indica, aquí guardamos clases con métodos auxiliares que son utilizados a lo largo del programa. Aquí tenemos:

- Extensors
- Auxiliar.cs
- AuxiliarGenericMethods.cs
- Reflection.cs

Primero explicaremos el funcionamiento de métodos auxiliares utilizados en mayor o menor medida por el resto de clases e interfaces. Estos se encuentran en Auxiliar.cs, AuxiliarGenericMethods.cs y Reflection.cs.

AuxiliarGenericMethods.cs:

Selector: Devuelve un objeto de tipo T y recibe un string subtitulo y un IEnumerable de opciones, estas opciones se imprimen en consola y le corresponde al usuario seleccionar una opción mediante las flechas direccionales.

ConfigSelector: Se encarga de llenar una lista de uno de los aspectos variables del juego, por ejemplo llenar la lista con las diferentes reglas que se van a aplicar. Para su funcionamiento se apoya en los métodos **Selector** y **YesOrNo**, de este último hablaremos luego.

Auxiliar.cs:

PossiblePlays: Mediante la mano de un jugador, la mesa y las reglas que aplica el juego actual, se encarga de llenar un IEnumerable de jugadas posibles, que luego serán pasadas al jugador para que su estrategia determine la jugada que le convenga.

YesOrNo: Apoyándose en **Selector** hace al usuario una pregunta de sí o no, devolviendo verdadero en caso afirmativo y falso en caso negativo.

NumberSelector: Permite al usuario elegir con las flechas direccionales un número que puede representar determinada configuración. Ej.: Doble máximo que admite el juego. La flechas de dirección horizontales varían el número de 1 en 1, mientras que las verticales lo hacen de 10 en 10, con Enter se devuelve el número actual, que será la decisión final del usuario.

Reflection.cs:

DefaultInstanceCreator: Este método recibe un Type y devuelve una nueva instancia de ese Type mediante reflection.

CollectionCreator: Recibe un IEnumerable de Type y devuelve un IEnumerable con instancias de cada Type que contenga el parámetro de entrada.

TypesCollectionCreator: Crea un IEnumerable de Type mediante el ensamblado que contiene a un cierto tipo T o sus derivados.

Extensors:

En esta carpeta tenemos dos clases de métodos extensores. Debido a la existencia de dos interfaces distintas que agrupan objetos que son clonables de distintas formas, fue necesario hacer esta división:

IEnumerableExtensors.cs: Aquí tenemos al método extensor de IEnumerables con objetos del tipo ICloneable (Interfaz que explicaremos más adelante). Este método **Clone** devuelve un nuevo IEnumerable con clones de todo el contenido del IEnumerable original.

A pesar de tener solo un método actualmente, la idea es que esta clase contenga más métodos extensores de IEnumerable en un futuro.

SimpleCloneableExtensors.cs: Esta vez tenemos dos métodos extensores:

El primero es un método extensor de la propia interfaz ISimpleClone, Se encarga de devolver un clon del objeto utilizando el método **DefaultInstanceCreator** que explicamos en el apartado de Reflection.cs.

El segundo es el método extensor de IEnumerables con objetos del tipo ISimpleClone (Que también explicaremos más adelante). La función de este método es similar al del anterior archivo cs, solo que los objetos del IEnumerable son clonados utilizando el método extensor de ISimpleClone explicado anteriormente.

CloneableItems:

Aquí se encuentran los objetos que implementan la interfaz ICloneable:

- Board
- Couple
- Event
- Move
- Piece
- Player
- Register

Board: Estructura que representa el tablero de juego, tiene una lista de Piece que se actualiza con cada jugada hecha, cuenta con una propiedad Value que es la suma total de los valores de las fichas puestas hasta el momento, además tiene métodos **Add** y **PreAdd** para incluir en la mesa por derecha y por izquierda respectivamente.

```
30 references
public class Board : ICloneable<Board>
{
    10 references
    private IList<Piece> Table { get; set; }

    174 references
    public int Value { get { return GetValue(); } }

    5 references
    public int Count { get { return Table.Count; } }

    5 references
    public Piece LeftPiece { get { return Table.First(); } }

    5 references
    public Piece RightPiece { get { return Table.Last(); } }

    3 references
    public Board()
    {
        this.Table = new List<Piece>();
    }
```

Couple: Concepto simple de pareja que consiste en agrupar 2 jugadores, Player1 y Player2, Los cuales recibe en su constructor.

```
26 references
public class Couple : ICloneable<Couple>
{

    2 references
    public Couple(Player p1, Player p2)
    {
        this.Player1 = p1;
        this.Player2 = p2;
    }

    8 references
    public Player Player1 { get; }

    8 references
    public Player Player2 { get; }
```

Event: Guarda una jugada por el nombre del jugador, la ficha y la posición en la que se jugó. Además de un método **PrintEvent** que imprime el evento en consola. Los Event son las entradas del objeto Register.

```
11 references
public class Event : ICloneable<Event>
{
    6 references
    public string PlayerName { get; private set; }

    9 references
    public Piece PlayedPiece { get; private set; }

    6 references
    public Move.Position Position { get; private set; }

    4 references
    public Event(string name, Piece piece, Move.Position position)
    {
        this.PlayerName = name;
        this.PlayedPiece = piece;
        this.Position = position;
    }
```

Move: Clase concreta que encapsula el concepto de jugada, tiene la ficha que se desea jugar, el lado de la mesa donde se desea jugar (Para lo cual se apoya en el Enum Position) y si está rotada o no.

```
81 references
public class Move : ICloneable<Move>
{
    31 references
    public enum Position
    {
        14 references
        left,
        10 references
        right
    }

    30 references
    public Piece Piece {get;}

    13 references
    public Position PiecePosition {get;}

    11 references
    public bool IsTurned {get;}

    15 references
    public Move (Piece piece, Position position, bool isTurned)
    {
        this.Piece = piece;
        this.PiecePosition = position;
        this.IsTurned = isTurned;
    }
```

Piece: Clase concreta que tiene 3 propiedades, LeftSide, RightSide y PieceValue que representan el valor de la parte izquierda y la parte derecha respectivamente, así como el valor propio de la ficha como objeto, el cual estará determinado por los evaluadores de fichas (Clases que implementan la interfaz IEvaluator). Además cuenta con un método **Turn** que rota la ficha 180 grados.

```
36 references
public class Piece : ICloneable<Piece>
{
    2 references
    public Piece(int left, int right, int value)
    {
        this.LeftSide = left;
        this.RightSide = right;
        this.PieceValue = value;
    }

    20 references
    public int LeftSide { get; private set; }

    20 references
    public int RightSide { get; private set; }

    11 references
    public int PieceValue { get; private set; }
```

Player: En la clase concreta **Player** se encuentra encapsulado el concepto de un jugador de juegos de mesa, Tiene una lista de fichas llamada Hand, una propiedad interna que es el valor actual de la misma (O sea la suma de los valores de todas sus fichas), y recibe en su constructor el nombre, y la instancia de estrategia que va a utilizar. Tiene un método **PlayerChoice** que llena un IEnumerable de jugadas posibles utilizando el método auxiliar **PossiblePlays** y retorna la jugada que selecciona su estrategia, además de un método **PrintHand** para mostrar su mano en consola.

```
54 references
public class Player : ICloneable<Player>
{
    2 references
    public Player(string name, IStrategy strategy)
    {
        this.Name = name;
        this.Hand = new List<Piece>();
        this.Strategy = strategy;
    }

    21 references
    public IList<Piece> Hand { get; private set; }

    10 references
    public int CurrentPoints { get { return Points(); } }

    21 references
    public string Name { get; }

    4 references
    public IStrategy Strategy { get; }
```

Register: Guarda los Event que conforman el juego en una lista de eventos. Permite imprimir el último evento mediante **PrintActual**, además de un método **PrintRegister** para imprimir el registro entero.

```
17 references
public class Register : ICloneable<Register>
{
    4 references
    private int Turn { get; set; }

    4 references
    public int Count { get { return GameRegister.Count; } }

    5 references
    public Event this[int i]
    {
        get { return GameRegister[i]; }
    }

    9 references
    private IList<Event> GameRegister { get; set; }

    2 references
    public Register()
    {
        GameRegister = new List<Event>();
        Turn = 0;
    }
```

```
Resumen del Juego:
------------------

1:Player 1 Jugó la ficha [2|2]
2:Player 2 Jugó la ficha [1|2] a la izquierda
3:Player 3 Jugó la ficha [2|6] a la derecha
4:Player 1 Jugó la ficha [3|1] a la izquierda
5:Player 2 Jugó la ficha [6|6] a la derecha
6:Player 3 Se pasó
7:Player 1 Jugó la ficha [3|3] a la izquierda
8:Player 2 Jugó la ficha [5|3] a la izquierda
9:Player 3 Jugó la ficha [5|5] a la izquierda
10:Player 1 Jugó la ficha [6|3] a la derecha
11:Player 2 Jugó la ficha [1|5] a la izquierda
12:Player 3 Se pasó
13:Player 1 Jugó la ficha [3|2] a la derecha
14:Player 2 Jugó la ficha [1|1] a la izquierda
15:Player 3 Jugó la ficha [2|5] a la derecha
16:Player 1 Jugó la ficha [5|6] a la derecha
17:Player 2 Jugó la ficha [6|4] a la derecha
18:Player 3 Jugó la ficha [4|5] a la derecha
19:Player 1 Se pasó
20:Player 2 Jugó la ficha [0|1] a la izquierda
Fin de la partida.

Pulse Enter
```

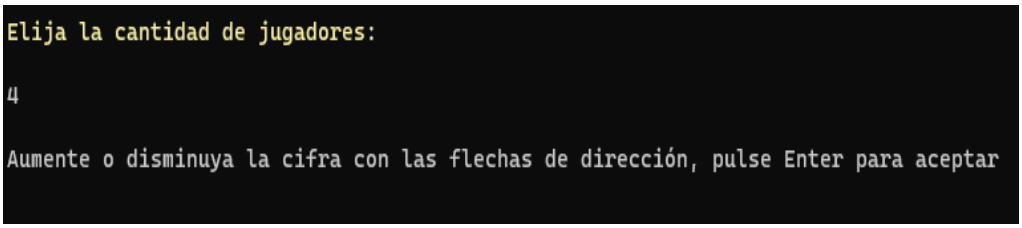
Creators:

Aquí están los encargados de interactuar con el usuario cuando este está configurando sus juegos o torneos:

- PlayerCreator.cs
- GameCreator.cs
- TournamentCreator.cs

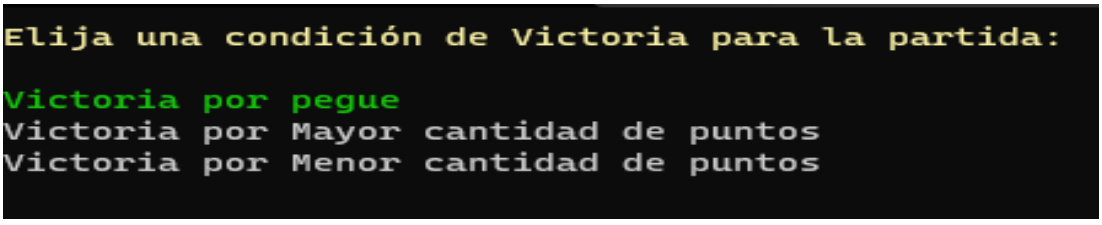
PlayerCreator.cs:

Posee una clase estática encargada de la creación del IEnumerable de jugadores. Tiene un método **Players** que apoyándose en los métodos auxiliares y los de Reflection anteriormente explicados y mediante las elecciones del usuario crea un IEnumerable de Jugadores que será la empleada por el Juego o el Torneo.



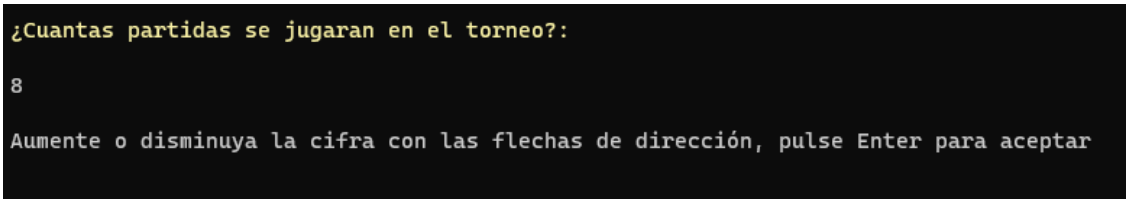
GameCreator.cs:

Posee una clase estática que usando Reflection llena las distintas listas de las configuraciones que tendrá el juego (Para lo cual utiliza el método auxiliar **ConfigSelector**), pregunta al usuario por el máximo doble que las fichas aceptan y la cantidad de fichas en la mano de cada jugador, y si no puede desarrollarse el juego fuerza a reelegir los valores. Al final del proceso devuelve un nuevo Game (Lo explicaremos más adelante) con las configuraciones seleccionadas.



TournamentCreator.cs:

Posee una clase estática que se encarga de la creación del Torneo, pregunta al usuario por las configuraciones básicas de este, así como forma de otorgar puntos a los ganadores y termina devolviendo el Tournament (Lo explicaremos más adelante) con las configuraciones seleccionadas.



Interfaces:

Como su nombre lo indica, aquí se encuentran todos los contratos o interfaces del programa. Las implementaciones concretas de la mayoría de estas interfaces están en la carpeta Implementations. Las interfaces implementadas actualmente son:

- ICloneable
- IDealer
- IDefaultGame
- IEvaluator
- IScore
- ISimpleClone
- IStopCondition
- IStrategy
- ITurnOrderer
- IValid
- IWinCondition

Explicaremos las funcionalidades básicas de cada una y así como las clases que implementan cada una de estas interfaces.

ICloneable:

Engloban el concepto de objetos que se pueden clonar, donde cada uno tiene una manera específica distinta de hacerlo. Contiene al método **Clone** que es el encargado de cumplir esta función.

Las clases que implementan esta interfaz son las que explicamos anteriormente en la carpeta CloneableItems.

IDealer:

Se encargan de repartir las fichas a las manos de los jugadores.

Clases que la implementan:

- AlphabeticalDealer: Reparte las fichas a los jugadores por el orden alfabético de sus nombres, siguiendo el mismo orden en que estaban las fichas en la caja cuando la recibió.
- HigherDealer: Organiza las fichas disponibles de mayor a menor y reparte las mayores aleatoriamente entre los jugadores.
- RandomDealer: Reparte aleatoriamente, lo más cercano posible a un juego tradicional.

Estas implementaciones están en la carpeta Dealers de Implementations.

IDeafultGame:

Configuraciones predeterminadas de juegos de Dominó clásicos como doble 6 o doble 9. Por supuesto pueden ser implementados otros juegos.

Clases que la implementan:

- ClassicGames (Clase Abstracta): Posee valores fijos de todas las características de un juego clásico, excepto el doble máximo y la cantidad de fichas por jugador, lo cual tendrá que ser implementado por sus herederos los cuales momentáneamente son:

  - Double6: Configuración del clásico doble 6.

  - Double9: Configuración del clásico doble9.

Estas implementaciones están en la carpeta DefaultGames de Implementations. Los herederos de ClassicGames están en la carpeta ClassicGamesImplementations.

IEvaluator:

Formas de darle valor a las fichas

Clases que la implementan:

- DefaultEvaluator: Le pone a las fichas el valor por defecto del dominó tradicional, la suma del valor de sus partes izquierda y derecha.
- DoubleEvaluator: Evalúa solo las fichas dobles, su valor será el valor de una de las caras.
- EvenEvaluator: Evalúa solo las fichas pares, su valor será la mitad de la suma de sus partes.
- OddEvaluator: Evalúa solo las fichas impares, su valor será el máximo entre sus partes.

Estas implementaciones están en la carpeta Evaluators de Implementations.

IScore:

Formas de puntuar a los jugadores en un torneo:

Clases que la implementan:

- PointScore: Otorga puntos a los ganadores de acuerdo al valor de las manos de los jugadores que no ganaron.
- WinScore: Otorga un único punto por ganar un juego, de esta manera podrían interpretarse los puntos de los jugadores como partidas ganadas.

Estas implementaciones están en la carpeta Scores de Implementations.

ISimpleClone:

Engloba el concepto de objetos que pueden clonarse, similar a ICloneable pero esta vez los objetos se clonan de manera única, es decir, la implementación de su método clone es igual en cada caso, para eso se creó un método extensor de esta interfaz que ya explicamos anteriormente. La idea de esta interfaz surge ante la necesidad de clonar implementaciones de algunas interfaces, por lo que todas las clases que implementan esta interfaz son otras interfaces.

Clases que la implementan:

- IStopCondition
- IStrategy
- IValid

Todas estas las explicaremos más adelante y están en la carpeta Interfaces de Core.

IStopCondition:

Son las encargadas de detener el juego cuando se cumpla determinada condición.

Clases que la implementan:

- BlockStop: Detiene el juego si nadie puede jugar.
- EmptyHandStop: Detiene el juego si alguien se queda sin fichas para jugar, o sea, si se pega.
- EqualHandsStop: Detiene el juego si en algún momento las manos de todos los jugadores valen igual.

Estas implementaciones están en la carpeta StopConditions de Implementations.

IStrategy:

Se encargan de elegir una jugada óptima para el jugador en el juego, basándose en determinados criterios y patrones.

Clases que la implementan:

- CleverStrategy: Reconstruye la mesa tratando de elegir jugadas tal que otros jugadores se hayan pasado y juega por la que más se han pasado.
- CouplesStrategy: Analiza las jugadas de la pareja y trata de no matarle la jugada.
- DoublesStrategy: Elige la jugada priorizando eliminar las fichas dobles
- DrunkStrategy: Además de las jugadas posibles adiciona jugadas inventadas por el (Incorrectas obviamente) y elige aleatoriamente pudiendo elegir jugadas mal, producto de los efectos del Alcohol.
- GreedyStrategy: Elige la jugada que lo deje con la menor cantidad de puntos en la mano, el clásico Botagorda.
- HumanStrategy: Es la estrategia que permite jugar al usuario, interactúa con él pidiéndole una siguiente jugada, y pregunta por especificaciones de la posición de la ficha o si rotarla, de ser necesario.
- RandomStrategy: Elige la jugada aleatoriamente entre las posibles de acuerdo a las reglas.

Estas implementaciones están en la carpeta Strategies de Implementations.

ITurnOrderer:

Representa formas de ordenar los jugadores para que jueguen en un orden determinado.

Clases que la implementan:

- CouplesOrderer: Ordena los jugadores por parejas de forma tradicional primero todos los jugadores primarios y luego sus parejas, si no se está jugando por parejas no realiza ninguna acción.
- RandomOrderer: Ordena los jugadores de forma aleatoria.
- ValueOrderer: Ordena los jugadores de manera descendente por el Valor actual de sus manos.

Estas implementaciones están en la carpeta TurnOrderers de Implementations.

IValid:

Reglas por las que se rige el juego actual, representa si una jugada dada es válida.

Clases que la implementan:

- ClassicValid: Puedes jugar si las partes opuestas de las fichas coinciden. Son las reglas clásicas del dominó.
- HigherExtremeValid: Puedes jugar si el valor de la respectiva cara de la ficha que se va a jugar es mayor que el de la cara que coincide con ella de la respectiva ficha en la mesa.
- HigherValueValid: Puedes jugar si la ficha elegida tiene mayor valor que la puesta en mesa.

Estas implementaciones están en la carpeta Valids de Implementations.

IWinCondition:

Las condiciones de victoria devuelven un IEnumerable de Jugadores que cumplen con determinada condición, los cuales son los ganadores de la partida.

Clases que la implementan:

- EmptyHandWin: Ganan los jugadores que se hayan quedado sin fichas en la mano.
- HigherPointsWin: Le otorga la victoria al jugador que más puntos tenga en la mano al finalizar el juego.
- LowerPointsWin: Le otorga la victoria al jugador que menos puntos tiene en la mano al finalizar el juego.

Estas implementaciones están en la carpeta WinConditions de Implementations.

Como se le da al usuario la posibilidad de elegir varias condiciones de victoria procuramos que condiciones de victoria opuestas (Como HigherPointsWin y LowerPointsWin) no entren en conflicto y no generen juegos no válidos. Para esto hicimos que las condiciones de victoria funcionen por prioridad, la cual está dada por el orden en que el usuario las selecciona, es decir, siguiendo dicho orden, la lista de ganadores final será la primera lista no vacía que devuelvan las condiciones de victoria, por lo tanto, se revisa una segunda condición de victoria si y solo si la lista que devolvió la condición anterior está vacía. Si todas las condiciones devuelven listas vacías el juego no tiene ganador.

Por ejemplo en el caso de HigherPointsWin y LowerPointsWin, al ser casos particulares de condiciones de victoria que no devuelven listas vacías, solo será válida para el juego la que haya seleccionado primero el usuario.

Toda esta funcionalidad pertenece a la clase Game, que explicaremos más adelante.

Hasta ahora hemos explicado funcionalidades y objetos por separado, solo una pocas se entrelazan y se utilizan entre ellas, por lo que aún no hemos hablado de conceptos de primera importancia como un Juego o un Torneo.

Esos conceptos se encuentran en la carpeta que nos falta por explicar, por lo cual procedemos a explicar estos conceptos a continuación.

GameManagers:

Como dijimos anteriormente, aquí están los encargados de reunir todas las funcionalidades anteriores y crear el juego como tal. Las clases encargadas de esto son:

- Game
- Round
- Tournament

Las explicaremos de menor a mayor complejidad.

Round:

Clase estática que se encarga de Llevar a cabo un turno de cada jugador, o sea, una ronda completa. Esto lo hace a partir de un método booleano **Run** que recibe como parámetro:

- Colección de jugadores (IEnumerable de Player)
- Conjunto de reglas del juego (IEnumerable de IValid)
- Conjunto de condiciones de parada (IEnumerable de IStopCondition)
- La mesa de Juego (Board)
- El registro del juego (Register)
- Colección de parejas en juego (IEnumerable de Couple)

La idea de que este método sea booleano es que devuelva **true** en el momento que se cumpla una de las condiciones de parada, o **false** si al término de una ronda se debe seguir jugando.

Resumiendo la funcionalidad de este método podemos decir que recorre la Colección de jugadores aplicando las respectivas jugadas de estos, si alguna condición de parada se cumple entonces devuelve **true** para alertar al juego de que se deben aplicar las condiciones de victoria (Esto lo explicaremos más adelante), actualiza el registro con cada jugada e imprime el estado actual de la mesa y la jugada elegida por el jugador. Todo esto lo hace apoyándose en métodos auxiliares de la propia clase como:

SkipTurn: Recibe un jugador, las reglas y la mesa y devuelve falso si el jugador no puede jugar ninguna de sus fichas.

Play: Recibe un jugador, la mesa y la jugada elegida por este y la coloca en la mesa, luego la remueve de la mano del jugador.

CheckValid: Recorre las reglas y analiza si alguna de ellas valida la jugada, y si es así devuelve verdadero.

Game:

Recibe en su constructor una serie de aspectos que en conjunto conforman al juego los cuales son:

- Colección de jugadores (IEnumerable de Player)
- Un Int que representa el doble máximo que tendrán las fichas.
- Un Int que representa la cantidad de fichas de fichas que tendrá cada jugador al repartir.
- Conjunto de reglas del juego (IEnumerable de IValid)
- Un IDealer, que repartirá las fichas guiándose por los Int anteriormente mencionados
- Una colección de ordenadores de turno (IEnumerable de ITurnOrderer)
- Un int que representa cada cuántos turnos se reordenarán los jugadores. (Si es 0 permanecerán en el mismo orden todo el juego).
- Colección de evaluadores de fichas (IEnumerable de IEvaluator).
- Conjunto de condiciones de parada (IEnumerable de IStopCondition)
- Colección de condiciones de victoria (IEnumerable de IWinCondition)
- Colección de parejas en juego (IEnumerable de Couple)

Métodos que implementa:

FillBox: Llena el almacén de fichas (la caja de dominó), de manera tradicional, Ej. Si es doble 9 se llena desde el [0,0] hasta el [9,9]. Otorga además los valores a la fichas de acuerdo a la colección de IEvaluator.

RunGame: En esencia, un juego es un ciclo de rondas, por tanto en este método se hacen iteraciones del método **Run** de Round, pasándole todo los parámetros necesarios, hasta que esta devuelva true, en ese momento se revisa la colección de IWinCondition y se declaran a los ganadores del juego, si es que los hubo.

GameOver: Imprime un breve resumen del estado del juego al finalizar este, informando sobre los jugadores con las fichas que quedaron en su mano y el valor de esta, el estado final de la mesa y como quedó el registro.

CoupleWin: Añade la pareja del que ganó a la lista de ganadores (Winners), en caso de no estar, y por supuesto si es que estamos en un juego por parejas.

Swift: Después de una determinada cantidad de Rondas cambia de ITurnOrderer y reordena los jugadores. En caso de que solo se haya escogido uno, simplemente se vuelve a llamar a su método **Organize**.

Tournament:

Permite jugar en la modalidad de torneo, entre sus propiedades están:

- Puntos que se deben tener para ganar el torneo (Int)
- Cantidad de partidas que se deben jugar (Int)
- Colección de jugadores (IEnumerable de Player)
- Forma de puntuar a los jugadores ganadores de un juego (IScore)
- Tabla de posiciones con cada jugador y sus puntos (Dictionary<string,int>)

Un torneo es una secuencia de partidas donde los ganadores puntúan y al final resulta como ganador absoluto el que más puntos tenga. Un torneo se detiene al alcanzar un número de partidas previamente elegido o cuando uno de los jugadores alcanza un número de puntos, también elegido previamente.

Métodos que implementa:

Run: Ciclo donde cada iteración crea y ejecuta un juego con la colección de jugadores recibida en el constructor. Puntúa a los jugadores e imprime la tabla de posiciones al final de cada juego. Detiene el torneo si se alcanza el límite de partidas o si el método **Win** retorna verdadero.

Win: Revisa la tabla de posiciones y devuelve verdadero si algún jugador alcanzó el límite de puntos.

TournamentOver: Organiza la tabla de posiciones de mayor a menor por los puntos obtenidos y crea una lista con los ganadores del torneo. Luego imprime los ganadores y la tabla de posiciones.

PrintScoreboard: Ordena la tabla de posiciones y la imprime en consola.

Funcionamiento General del Proyecto:

Una vez ejecutada la aplicación de consola se le da al usuario la opción de escoger entre nueva partida, nuevo torneo o salir, el usuario recorre este menú con las flechas direccionales hacia arriba y hacia abajo, en dependencia de su elección se llama a los métodos GameCreator.**Game** o TournamentCreator.**Tournament** para crear una partida o un torneo respectivamente, y ejecutarlos luego. Si su elección es salir se cierra el programa.

Bibliografía:

- Documentación oficial de Microsoft sobre Reflection en C# , https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection

- Documentación oficial de Microsoft sobre comentarios enfocados en documentación en C#, https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/documentation-comments

- Documentación oficial de Microsoft sobre Linq en C#, https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq

- Documentación oficial de Microsoft sobre namespaces e C#, https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/namespaces

- Empezar a Programar. Un enfoque multiparadigma con C#, Profesor Miguel Katrib, Editorial UH.