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 Engine que cuenta con los siguientes archivos que se listan a continuación:

- Auxiliares.cs
- CondicionesdeParada.cs
- CondicionesdeVictoria.cs
- Creadores.cs
- Estrategias.cs
- Evaluadores.cs
- Ficha.cs
- Juego.cs
- JuegosPorDefecto.cs
- Jugadas.cs
- Jugador.cs
- Mesa.cs
- Ordenador.cs
- Parejas.cs
- Scoring.cs
- Reflection.cs
- Registro.cs
- Repartidores.cs
- Ronda.cs
- Torneo.cs
- Validadores.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 Auxiliares.cs y Reflection.cs.

Dentro de Auxiliares tenemos 2 clases estáticas, una genérica y otra no, en la genérica tenemos los siguientes métodos.

Selector: devuelve un objeto de tipo T y recibe un string subtitulo y una lista 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.

En la Clase no genérica tenemos:

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

YesOrNo: Apoyándose en **Selector** pregunta al usuario por una pregunta de sí o no, devolviendo verdadero si sí y falso si no.

CloneList: Clona una lista de fichas elemento a elemento para evitar que las implementaciones de ciertas clases o interfaces puedan interferir con el correcto funcionamiento del juego.

ClonePlayers: Procedimiento análogo a **CloneList** pero con una lista de jugadores.

NumberSelector: Permite al usuario elegir con las flechas direccionales un número que puede representar determinada configuración. Ej. Doble máximo que admite el juego, horizontal de 1 a 1 y vertical de 10 en 10.

Dentro de Reflection.cs se encuentra la clase estática del mismo nombre que cuenta con los siguientes métodos:

DefalultInstanceCreator: este método recibe un Type y devuelve una nueva instancia de ese Type mediante reflection.

ListCreator: Recibe una lista de Type y devuelve una lista con instancias de cada Type que contenga el parámetro de entrada.

TypesListCreator: Crea una lista de Type mediante el ensamblado que contiene a un cierto tipo T o sus derivados.

Para el modelado del juego de Dominó hemos creado diversos objetos que encierran conceptos importantes para el Dominó y los juegos de mesa en general, estos son:

Ficha: Clase concreta que tiene 3 propiedades, PartIzq, PartDer y Valor 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**. Además cuenta con un método Turn que rota la ficha 180 grados.

```
// el concepto de ficha
38 references
public class Ficha
{
    // tiene parte izquierda y parte derecha además de un valor dado por el evaluador
    2 references
    public Ficha(int Izq, int Dch, int valor)
    {
        this.PartIzq = Izq;
        this.PartDch = Dch;
        this.Valor = valor;
    }

    // parte izquierda
    19 references
    public int PartIzq { get; private set; }

    // parte derecha
    19 references
    public int PartDch { get; private set; }

    // valor de la ficha
    7 references
    public int Valor { get; private set; }

    // rota la ficha
    2 references
    public void Turn()
    {
        int Temp = PartIzq;
        this.PartIzq = this.PartDch;
        this.PartDch = Temp;
    }
```

Jugada: Clase concreta que encapsula el concepto de jugada, tiene la ficha que se desea jugar, el lado de la mesa donde se desea jugar y si está rotada o no

```
// estructura que representa una jugada
46 references
public class Jugada{

    // cual ficha se quiere jugar
    26 references
    public Ficha Ficha {get;}

    // la posición donde se quiere jugar
    11 references
    public bool Posición {get;}

    // si la ficha está rotada o no
    9 references
    public bool Girada {get;}

    13 references
    public Jugada (Ficha ficha, bool posición, bool girada)
    {
        Ficha = ficha;
        Posición = posición;
        Girada = girada;
    }
}
```

Jugador: En la clase concreta **Player** se encuentra encapsulado el concepto de un jugador de juegos de mesa, Tiene una lista de fichas llamada mano, una propiedad interna que es el valor actual de la misma, y recibe en su constructor el nombre, y la instancia de estrategia que va a utilizar. Tiene un método Play que llena la lista de posibles y retorna la jugada que selecciona su estrategia, además de un método PrintHand para mostrar su mano en consola.

```
73 references
public class Player
{
    2 references
    public Player(string nombre, IStrategy estrategia)
    {
        this.Nombre = nombre;
        this.Mano = new List<Ficha>();
        this.Estrategia = estrategia;
    }

    // mano del jugador
    16 references
    public List<Ficha> Mano { get; private set;}

    12 references
    public int Puntos { get { return puntos(); } }

    // nombre del jugador
    13 references
    public string Nombre { get; }

    6 references
    public IStrategy Estrategia { get; }

    // retorna la jugada que se va a hacer
    1 reference
    public Jugada Play(Board mesa, List<IValid> reglas, Register registro, List<Pareja> parejas)
    {
        List<Jugada> posibles = Auxiliar.LlenarPosibles(Mano, reglas, mesa);
        return this.Estrategia.Jugada(new List<Ficha>(this.Mano), posibles, mesa, registro, new List<Pareja>(parejas));
    }
```

Mesa: Estructura que representa el tablero de juego, tiene una lista de jugadas que se actualiza con cada jugada hecha, cuenta con una propiedad Valor 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 y un Clone que copia la mesa para evitar cambios inesperados.

```
// clase que define a la mesa
29 references
public class Board
{
    // lista de fichas que la representa
    10 references
    private List<Ficha> table { get; set; }

    // valor actual de la mesa, acorde al valor interno de cada ficha
    0 references
    public int value { get { return GetValue(); } }

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

    5 references
    public Ficha First { get { return table.First(); } }

    5 references
    public Ficha Last { get { return table.Last(); } }

    // constructor
    3 references
    public Board()
    {
        table = new List<Ficha>();
    }
```

Pareja: Concepto de pareja que consiste en agrupar 2 jugadores, Player1 y Player2, para permitir a las estrategias variar acorde a los movimientos de la pareja.

```
// entidad de pareja
25 references
public class Pareja
{
    // la suma de los puntos de ambos jugadores
    0 references
    public int puntos { get { return Player1.Puntos + Player2.Puntos; } }

    // recibe 2 jugadores
    1 reference
    public Pareja(Player p1, Player p2)
    {
        this.Player1 = p1;
        this.Player2 = p2;
    }

    8 references
    public Player Player1 { get;}
    8 references
    public Player Player2 { get;}

}
```

Evento: 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.

```
// eventos por los cuales está conformado el registro,
8 references
public class Evento
{
    // nombre del jugador que jugó la ficha
    3 references
    public string Jugador { get; private set; }

    // cual fue su jugada
    7 references
    public Ficha Jugada { get; private set; }

    // en que lado de la mesa jugó
    4 references
    public bool Posicion { get; private set; }

    2 references
    public Evento(string nombre, Ficha jugada, bool posicion)
    {
        Jugador = nombre;
        Jugada = jugada;
        Posicion = posicion;
    }
```

Register: Guarda los eventos que conforman el juego en un listado de eventos. Permite imprimir el último evento mediante PrintActual, además de un método PrintRegister para imprimir el registro entero y un Clone para hacer copias del registro.

```
// entidad encargada de llevar un registro del juego actual
15 references
public class Register
{
    // el turno en el que se jugó la ficha
    4 references
    private int Turno { get; set; }

    // cuantos turnos se han jugado
    3 references
    public int Count { get { return Registro.Count; } }

    // permite indexar por el turno
    4 references
    public Evento this[int i]
    {
        get { return Registro[i]; }
    }

    // almacena los eventos
    9 references
    private List<Evento> Registro { get; set; }

    2 references
    public Register()
    {
        Registro = new List<Evento>();
        Turno = 0;
    }
```

Aspectos Personalizables del Juego:

CondicionesDeParada: Verifica si se debe detener el juego si se cumple alguna de las condiciones implementadas, actualmente están estas implementaciones:

- PegueStop: Detiene el juego si alguien se queda sin fichas para jugar
- TranqueStop: Detiene el juego si nadie puede jugar
- EqualsStop: Detiene el juego si en algún momento las manos de todos los jugadores valen igual

CondicionesDeVictoria: Si se cumple alguna de las condiciones de victoria implementadas devuelve una lista de jugadores vencedores, estas son las implementaciones actuales:

- PegueWin: gana el que se pegó.
- HigherWin: hereda de la clase abstracta PointWin y le otorga la victoria al jugador que más puntos tenga en la mano al finalizar el juego.
- LowerWin: hereda de PointWin y le otorga la victoria al jugador que menos puntos tiene en la mano al momento de finalizar el juego.

Estrategias: se encargan de elegir una jugada óptima para el juego, hasta el momento se tienen implementadas:

- HumanST: El usuario elige su jugada, tiene un método booleano Ambiguo, que devuelve verdadero si la ficha se puede jugar por ambos lados de la mesa. Además cuenta con un método AskPos que pregunta al usuario por cual posición jugar la ficha si es ambigua.
- BotagordaST: Elige la jugada que lo deje con la menor cantidad de puntos en la mano, el clásico Botagorda.
- RandomST: Elige la jugada aleatoriamente entre las posibles de acuerdo a las reglas.
- BotaDoblesST: Elige la jugada priorizando eliminar las fichas dobles.
- DrunkST: Además de las jugadas posibles adiciona jugadas y elige aleatoriamente pudiendo elegir jugadas mal.
- CleverST: Reconstruye la mesa tratando de elegir jugadas tal que otros jugadores se hayan pasado y juega por la que más se han pasado.
- ParejasST: Analiza las jugadas de la pareja y trata de no matarle la jugada.

Evaluador: Formas de darle valor a las fichas, implementaciones hasta ahora:

- DefaultEvl: le pone a las fichas el valor por defecto del dominó tradicional, la suma del valor de sus partes izquierda y derecha.
- DoubleEvl: evalúa solo las fichas dobles, su valor será el valor de una de las caras.

- EvenEvl: evalúa solo las fichas pares, su valor será la mitad de la suma de sus partes.
- OddEvl: evalúa solo las fichas impares, su valor será el máximo entre sus partes.

JuegosPorDefecto: Configuraciones predeterminadas de juegos de Dominó clásicos como doble 6 o doble 9.

- Doble6: Configuración predeterminada para la variante doble 6.
- Doble9: Configuración predeterminada para la variante doble 9.

Ordenador: Representa formas de ordenar los jugadores para que jueguen en ese orden:

- RandomOrd: Ordena los jugadores de forma aleatoria.
- ParejasOrd: 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 ordena.
- ValueOrd: Ordena los jugadores de manera descendente por el Valor actual de sus manos.

Repartidores: Se encarga de repartir las fichas a las manos de los jugadores:

- RandomRep: reparte aleatoriamente, lo más cercano posible a un juego tradicional.
- HigherRep: organiza las fichas disponibles de mayor a menor y reparte las mayores aleatoriamente entre los jugadores.

Score: Formas de puntuar a los jugadores en el torneo:

- WinP: otorga puntos por ganar un juego.
- PointP: otorga puntos de acuerdo al valor de las manos de los jugadores que no ganaron.

Validadores: Reglas por las que se rige el juego actual, representa si una determinada jugada es valida

- ClassicValid: puedes jugar si las partes opuestas de las fichas coinciden, las reglas clásicas del dominó.
- MaximValid: puedes jugar si la ficha elegida tiene mayor valor que la puesta en mesa.
- UpperValid: puedes jugar si la ficha elegida en la cara coincidente es mayor que la cara coincidente la mesa en su respectiva posición.

Las responsabilidades de crear los juegos y torneos en sus distintas modalidades se encuentran en Creadores.cs, procedemos a exponer su contenido:

PlayerCreator: es una clase estática encargada de la creación de la lista 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 una lista de Jugadores que será la empleada por el Juego o el Torneo.

GameCreator: es una clase estática que usando Reflection llena las distintas listas de las configuraciones que tendrá el juego, 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 juego con las configuraciones seleccionadas.

TournamentCreator: es 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 creando el Torneo con las configuraciones seleccionadas.

JUEGO:

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

- Lista de jugadores
- Máximo doble que tienen las fichas
- Cantidad de fichas en la mano de los jugadores
- Lista de Reglas
- Repartidor de fichas
- Lista de Ordenadores de Turnos
- Cantidad de rondas para para cambiar de ordenador
- Lista de Evaluadores de fichas
- Lista de Condiciones de Parada
- Lista de Condiciones de Victoria
- Lista de Parejas

Métodos que implementa:

FillBox: llena el almacén de fichas (la caja de dominó) de acuerdo con el evaluador, de manera tradicional, Ej. Si es doble 9 se llena desde el [0,0] hasta el [9,9].

RunGame: El juego constituye un conjunto de Rondas donde se aplicará en método Run de la clase Ronda, la cual encapsula un turno de cada jugador que está presente en el juego.

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.

PairWin: Añade la pareja del que ganó a la lista de jugadores.

Swift: Después de una determinada cantidad de Rondas cambia de Ordenador de Turnos y reordena los jugadores.

RONDA:

Clase estática que se encarga de manipular un turno de cada jugador.

Métodos que implementa:

Pásate: Recibe el 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.

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

Run: Método booleano que recorre la lista de jugadores aplicando las respectivas jugadas de estos, si alguna condición de parada se cumple entonces devuelve verdadero para alertar al juego de que se deben aplicar las condiciones de victoria, actualiza el registro con cada jugada e imprime el estado actual de la mesa y la jugada elegida por el jugador.

TORNEO:

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

- Puntos que se deben tener para ganar el torneo
- Cantidad de partidas que se deben jugar
- Lista de jugadores
- Forma de premiar a los jugadores ganadores de un juego
- Tabla de posiciones con cada jugador y sus puntos en un Diccionario<string,int>

Métodos de torneo:

Win: Revisa la tabla de posiciones y devuelve verdadero si se cumplió con la condición de finalización del torneo.

GameOver: 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.

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

Run: llama al GameCreator para que el usuario elija como se va a jugar esa partida, al finalizar esta premia a los jugadores e imprime la tabla de posiciones al final de cada juego.

Funcionamiento 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 o TournamentCreator para iniciar una partida o un torneo respectivamente, 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

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