Second_Project_Dominoes

Proyecto de Programación II. Facultad de Matemática y Computación. Universidad de La Habana. Curso 2021.
Dominoes es una aplicación que permite jugar Dominoes contra una serie de "inteligencias" lo original de esta aplicación es que hemos tratado de crear código de manera que implementar una nueva variante de domino o una nueva inteligencia requiera de la menor cantidad de cambios posibles.
Es una aplicación de Consola, desarrollada con tecnología .NET Core 6.0 y en el lenguaje C#.
La aplicación está dividida en dos componentes fundamentales:
`VisualDominoes` es un aplicación de consola que renderiza la interfaz gráfica y sirve los resultados.
`DominoEngine` es una biblioteca de clases donde está implementada lo necesario para la lógica del juego.

Sobre el Juego

Manual de Usuario

El Juego comienza con una presentación luego deberá modelar el juego mediante una serie de elecciones
Primera Elección:
Debe seleccionar uno de los números de al lado para elegir la variante del domino (Ejemplo: 2 para la versión doble 9)

- 0 Doble 7
- 1 Doble 8
- 2 Doble 9
- 3 Doble 10

Segunda Elección:
Debe elegir la cantidad de jugadores
Tercera Elección:
La cantidad de fichas por jugador
Cuarta Elección:
Tipo de juego (Consta con tres variaciones de domino)

- 0 Classic Dominoes (Domino Clásico)
- 1 Pretty Boy (Domino de figuritas)
- 2 Sloten (Domino Robaito)

La Quinta Elección:
Seleccionar el tipo de player que jugara; cada vez que seleccione un jugador este le pedirá un identificador o nombre:

- 0 Human Player (Para que el usuario pueda jugar)
- 1 Ramdom Player (Jugador Ramdom)
- 2 Drop Fat Player (Botagordas)
- 3 Almost Clever Player (Casi Inteligente :))

Luego que haya seleccionado conformado el juego comenzara el juego
La parte visual se compone de dos partes una que muestra la mesa y la mano del jugador actual y otra que muestra la mano de todos los jugadores

Análisis de proyecto

El proyecto cuenta de una estructura ficha que esta de compuesta de dos valores, que podemos hacer variar, cuando variamos el valor podemos tener fichas de figuras, colores, números etc... Tiene una clase mesa que almacena las fichas que serán jugadas. Contiene una clase jugador con una lista de fichas, y una estrategia que se le pasa al constructor, esta determina como jugara este. Tenemos la clase reglas, que determinan un conjunto de utilidades. Ejemplo: Comprueba si es el turno de un jugador , si hay final, si existe un ganador, si hay empate, si una jugada es válida y genera las fichas. Las reglas del juego son modeladas a través de las clases que implementan condiciones de finalización y de ganador, que se les pasan en el constructor, lo cual nos permite hacer variar en parte como se desarrolla la partida. Las Clases que implementan GameLogic tienen métodos que permiten cambiar de jugador, a uno valido (Busca al jugador que le toca, si este no puede jugar, lo pasa ,cambia el turno y repite el proceso hasta encontrarlo), contiene un método que desarrolla el proceso del turno actual (Pedirle al jugador actual una jugada, añadir a la mesa, quitarla de la mano y pasar turno). Esto no se desarrolla de forma igual en todos los juegos, por eso lo consideramos como un aspecto a variar en la construcción de una nueva lógica de juego, en otras variantes como el robaito este método modela algunos cambios, como es la cuestión de que si el jugador no lleva, roba hasta que pueda jugar. Hay un método que dice si hay final lo cual es determinado por las reglas que sean pasadas a la lógica de juego, otro método que nos pareció necesario variar fue la forma de repartir las fichas. El juego se desarrolla repitiendo un ciclo. Buscar un jugador valido, preguntar si hay final, desarrollar lo que pasa en el turno actual y repetir hasta que se termina el juego luego tenemos un ganador o un empate.

Sobre la ingeniería de software

Para implementar la lógica de los juegos hemos implementado varias interfaces fundamentales:

```
davidalba, 2 months ago | 1 author (davidalba) | 8 references
public interface IEndCondition<TValue, T> where TValue : IValue<T>
{
    1 reference
    public bool IsFinal(List<Player<TValue, T>> players);
}
```

Quien implemente la interfaz deberá implementar un método que servirá de condición de finalización esto nos permitirá crear variaciones de las misma. Recibe una lista de jugadores y devuelve true si se cumple determinada condición

Ejemplo de Clase que implementa la interface:

```
davidalba, 2 months ago | 1 author (davidalba) | 3 references
public class IsLocked<TValue, T> : IEndCondition<TValue, T> where TValue : IValue<T>
{
    1 reference
    public bool IsFinal(List<Player<TValue, T>> players)
    {
        foreach (var item in players)
        {
            if (!item.Pass) return false;
        }
        return true;
    }
}
```

IsLocked implementa la condición de finalización de un juego trancado y esto pasa cuando todos los jugadores están pasados

```
davidalba, 2 months ago | 1 author (davidalba) | 9 references
public interface IWinCondition<TValue,T> where TValue : IValue<T>
{
    2 references
    public bool IsWinner(Player<TValue,T> player,List<Player<TValue,T>> players);
}
```

Quien implemente la interfaz deberá implementar un método que servirá de condición para determinar si un jugador es ganador y permitirá crear distintas formas en las que un jugador gana por ejemplo en un juego donde las fichas tiene caras contables como números una posible forma de ganar puede ser por cantidad de puntos el que más tenga gana en una variante donde las caras son figuras o colores una posible condición de ganar es el jugador que primero se quede sin fichas. El método Recibe un jugador y una lista de jugadores y devuelve true si se cumple determinada condición.

Ejemplo de Clase que implementa la interface:

```
davidalba, 2 months ago | 1 author (davidalba) | 1 reference
public class WinnerByChips<TValue, T> : IWinCondition<TValue, T> where TValue : IValue<T>
{
    2 references
    public bool IsWinner(Player<TValue, T> player, List<Player<TValue, T>> players)
    {
        return (player.NumChips == players.Min(x => x.NumChips));
    }
}
```

WinnerByChips implementa un criterio de ser ganador basado en la cantidad de fichas si el jugador que se le pasa es el que menor cantidad de fichas tiene entonces este puede ser un posible ganador

Nota: Las condiciones de ganador y de finalización se conjuntan para buscar el estado de finalización del juego. No implementamos simplemente una condición de finalización porque nos parecía atractiva la idea para mejoras y extensión del proyecto que el jugador usando condiciones de ganador pudiera "ganar" algo por determinadas acciones independientes al fin. Ejemplo ganar 5 puntos cada vez que el próximo jugador se pase esto es a criterio de la variante de domino que se desea implementar

```
public interface IRankable: IComparable<IRankable>
{
    12 references
    public int Rank();
}

public interface IValue<T> : IEquatable<IValue<T>>
{
    12 references
    public T? Value { get; }
}
```

Las interfaces anteriores, es necesario que sean implementadas por los valores de las fichas, con IRankable se le otorga valores, para que en algunas Variantes del juego se pueda determinar ganador, algunas estrategias puedan determinar que jugada es la mejor, e IValue obliga guardar un objeto genérico, y también obligándolo a que defina como se verifica si un IValue es enlazable con otro mediante la implementación de IEquatable

```
public interface IStrategy<TValue, T> where TValue : IValue<T>
{
    // Devuelve true si tiene una jugada Valida, y en el parametro de move se devuelve la jugada con la posisión
    1 reference
    public bool ValidMove(Player<TValue, T> player, Board<TValue, T> board, Rules<TValue, T> rules,
      out (Chip<TValue, T>,TValue) move);
}
```

En esta interfaz definimos que debe cumplir una estrategia. Con el método booleano ValidMove el controlador del juego sabe si tienes jugada valida o no, en caso de tenerla también sabría cual es.

```
public interface IGameLogic<TValue,T> where TValue:IValue<T>      davidalba, 2 months ago • "Mejoras de genericid
{
    // Turn guarda la cantidad de turnos desde el comienzo de la partida
    8 references
    int Turn { get; }
    // board es la mesa de donde se juegan las fichas
    9 references
    Board<TValue, T> board { get; }
    18 references
    List<Player<TValue, T>> Players { get; }
    // Rules guarda las reglas basicas del juego
    14 references
    Rules<TValue, T> Rules { get; }
    10 references
    List<Chip<TValue, T>> Chips { get; }
    // CurrentPlayer guarda la referencia del jugador que tiene que jugar en el turnto dado
    14 references
    Player<TValue, T>? CurrentPlayer { get; }

    // Este metodo actualiza a CurrentPlayer
    1 reference
    void ChangeValidCurrentPlayer();

    // Este metodo hace que el jugador que le toque juegue mientras pueda jugar
    1 reference
    void CurrentTurn();

    // Dvuelve True si el juego termó, false en el caso contrario
    1 reference
    bool EndGame();

    // Reparte las fichas a los jugadores
    1 reference
    void HandOutChips(int CountChip);
}
```

Las clases que implementan estas interfaces son las encargadas de definir como debe fluir el juego, el orden de jugadas, como se reparten las fichas, etc. Para hacer algunas de las verificaciones, por ejemplo, si ya el juego termino, o alguien ganó, la clase que implemente esta interfaz necesita de una Rule, clase que será explicada a continuación.

Clases que no implementan interfaces:

En algunos casos consideramos que las clases ya eran suficientemente genéricas, y no necesitaban sufrir cambios para variar la jugabilidad, Casos como Player, donde solo varia la estrategia, que se le define en el constructor. Parecido pasa con Rules, donde solo se almacenaban condiciones de victoria y de fin de juego, que podían definirse en un constructor haciendo uso de delegados, y la clase ficha, que solo variaba el valor de cara, por eso la interfaz para el valor y no para la ficha:

```
public class Chip<TValue,T> where TValue : IValue<T>
{
    16 references
    public TValue LinkR { get; }
    16 references
    public TValue LinkL { get; }
    1 reference
    public Chip(TValue linkR, TValue linkL)
    {
        this.LinkR = linkR;
        this.LinkL = linkL;
    }
}
```

Los valores se definen en el constructor.

Análisis de la parte visual

En la clase ControllerGame se construyen los distintos tipos de dominós en dependencia de las preferencias del usuario, haciendo uso del método Customitation, método que devuelve las preferencias del usuario para almacenarse en la variable Custom:

```
(int CountChip, int LinkedValues, int countPlayer, int maxNumChip, int ChipForPlayer, int GameType) Customs = InterPrints.Customitation();
//Aqui se construye el juego segun las respuestas del usuario
TypeGame typeGame = (TypeGame)Customs.GameType;
```

Luego haciendo uso del Switch Case, según la elección que haya hecho el usuario se construye el juego.

El método NewGame hace correr de forma coherente los métodos del IGameLogic que se haya implementado:

```
public class ControllerGame
{
    3 references
    TValuesContents Values = new();
    1 reference
    public void MakeGame()···
    // Metodo que construye las reglas
    3 references
    public (Rules<TValue,T>,List<Player<TValue,T>>) CustomPlayerAndRules<TValue, T>
    (int countPlayer,IWinCondition<TValue,T>[] winConditions, IEndCondition<TValue,T>[] finalConditions) where TValue : IValue<T>,IRankable
    3 references
    public void NewGame<TValue, T>(IGameLogic<TValue, T> Game, int numChipPlayer) where TValue : IValue<T>···

}
```

InterPrints es una static class con todos los métodos que requieran impresión en consola, incluyendo los de interacción con el usuario

```
public static class InterPrints
{
    //Imprime la parte frontal de juego(Primera portada)
    1 reference
    public static void Front()···
    //Imprime la barra de progreso que se muestra al inicio
    1 reference
    private static void BarProgress(int progreso, int total = 100) //Default 100···
    //Selector (Imprime devuelve e interactuá con la entrada de la selecciones del usuario)
    6 references
    public static int PrintSelect(ICollection<string> selected, string description,int min, int count)···
    //Menú para construir la lista de jugadores según la opción
    1 reference
    public static void AddPlayer<TValue, T>(List<Player<TValue, T>> players, int select, int order) where TValue : IValue<T>, IRankable···
    // Según la opción elegida permite tomar el numero de caras del domino
    1 reference
    public static int VersionChips(int select)···
    // Imprime la mesa
    1 reference
    public static void PrintTable<TValue, T>(Board<TValue, T> table) where TValue : IValue<T>···
    //Imprime las manos
    2 references
    public static void PrintHand<TValue, T>(List<Chip<TValue, T>> Hand) where TValue : IValue<T>···
    //Imprime las secuencia del juego
    1 reference
    public static void PrintGame<TValue, T>(IGameLogic<TValue, T> Game) where TValue : IValue<T>···

    1 reference
    public static (int CountChip, int LinkedValues, int countPlayer, int maxNumChip, int ChipForPlayer, int GameType) Customitation()···
    // Metodo que define como va a interactuar el usuario como jugador con la consola
    1 reference
    public static bool AskNextPlay<TValue, T>(Player<TValue, T> player, Board<TValue, T> board,
    Rules<TValue, T> rules, out (Chip<TValue, T>, TValue) value)where TValue : IValue<T>      You, 1 second ago • Uncommitted changes ···
}
```

Para imprimir con mayor facilidad, hacemos uso de varios Enum, que a su vez facilita el trabajo con el switch en ControllerGame:

```
// Opciones de la versión
7 references
enum VersionDomioes
{
    1 reference
    Doble7,
    1 reference
    Doble8,
    1 reference
    Doble9,
    1 reference
    Doble10,
}
// Tipo de palyer
7 references
enum TypePlayer
{
    1 reference
    HumanPlayer,
    1 reference
    RandomPlyer,
    1 reference
    BotaGorda,
    1 reference
    AlmostClever,

}
//Tipo de juego
6 references
enum TypeGame
{
    1 reference
    ClasicDominos,
    1 reference
    PrittyBoy,
    1 reference
    Stolen,
}
```

Posibles mejoras a deficiencias:

- Como posibles mejoras esta la optimización e implementación de más variantes de juegos.
- Reajustar el funcionamiento y aumentar la abstracción para implementar juegos más potentes.
- Implementar un jugador de domino más inteligente que los actuales.
- Implementar una interfaz gráfica.