Report

Indice general

Client

El cliente de la aplicación no es más que la interfaz gráfica del juego, está desarrollada con un **React, un framework de Javascript** para desarrollo web. Este está completamente separado del `Server` y estos se comunican mediante peticiones `http` intercambiándose informaciones entre ellos para poder crear un flujo en el juego.

Lo primero que vemos al correr nuestro juego es un menú de opciones para poder seleccionar el tipo de juego que queremos jugar, seleccionando por separado las diferentes variaciones de funcionalidades importantes del mismo.

En el menú se muestra un barra de progreso que se va llenando a medida que vas seleccionando una opción de cada variación del juego y en la parte inferior se va mostrando las opciones que vas escogiendo.

Una vez que seleccionas todas las variaciones deseadas aparecerá el botón de `play` para iniciar el juego.

Una vez iniciado el juego tendremos una interfaz dividida en dos secciones.

En la sección más grande tendremos el tablero del juego, el cual en dependencia del tipo de juego seleccionado mostrará las fichas jugadas de una forma o de otra. Estas fichas al salirse del tamaño de la pantalla se podrá hacer scroll para poder ver todas las esquinas del juego.

En la parte inferior tenemos la información del jugador actual y dos botones, **NEXT TURN** para dar paso a que juegue el otro jugador y el **RESET GAME** que es para reiniciar la partida con la misma configuración seleccionada. Una vez el juego concluya el botón **NEXT TURN** dirá **NEW GAME** el cual te permitirá seleccionar otra configuración para poder jugar una nueva partida con opciones diferentes.

Ahí también tenemos en el medio, las fichas del jugador actual que está jugando y a la izquierda tenemos una miniatura de una cara representando al jugador y el nombre del mismo con sus respectivos puntos hasta ese momento de la partida.

Si un jugador se pasa o se termina el juego, saldrá en el medio de la pantalla un cartel rojo mostrando **El jugador [nombre] se ha pasado!!**, y en caso de que el juego haya finalizado mostrará la lista de jugadores en el orden definido por la variación de "quien gana el juego" seleccionada al principio. Aquí un ejemplo

Y en la parte superior derecha tendremos dos botones: el **BACKGROUND** que es para poder cambiar la imagen del tablero, y el botón a su derecha con el ícono **"i"** que es para poder ver las opciones del juego seleccionadas al principio, que corresponden con la variación que se está jugando.

Un poco sobre el softweare

Nuestra aplicación al principio hace una petición `http` al servidor para que este nos dé la información sobre las opciones a mostrar al usuario. Al ser cargada se muestra dinámicamente mediante componentes que reciben la información, por lo que si se añaden más opciones este será capaz de adaptarse correctamente.

El usuario constará con pequeñas validaciones mientras selecciona las opciones, mostrando un mensaje de error si este comete alguno. También tiene un seguimiento en la parte inferior que se va generando dinámicamente y guardando las selecciones hechas para mejor feedback.

Todas estas opciones se guardan en un `Context` general y se va actualizando con cada selección y, al final de todas las opciones, este se enviará al servidor mediante otra petición http y donde el server devolverá la información inicial para mostrar en el juego.

Una vez dentro del mismo, al dar click en el botón de **NEXT TURN**, este hará una petición http al servidor pidiendo los datos de la nueva jugada, la cual será cargada y mostrada en pantalla.

Cuando termine la partida y se toca el botón de **NEW GAME** lo que hace la aplicación es borrar esos datos que tenía guardado en el `Context` y automáticamente cambia la vista para poder seleccionar una nueva variante del juego.

Si, por el contrario, se da clic en el botón de **RESET GAME** este lo que hace simplemente es realizar nuevamente la misma petición http que se hizo al principio para reiniciar el juego en nuestro server, manteniendo las mismas opciones del juego actual.

Server

El servidor del juego es una **API REST** desarrollada en C#, este tiene tres controladores que se comunican con el Cliente para mandar y recibir información y tiene también toda la lógica del juego, clases, implementaciones, etc.

En el `Server` tenemos los controladores de la API(`Controllers`), los modelos(`Models`) que son clases bases para el envío y recibimiento de los request del Cliente, y la parte de `Data` que contiene toda la lógica de la aplicación.

Controllers y Models

LoaderController: Este controlador es la primera petición que se realiza en el `Client` y es el que carga la clase `InterfaceOfOptions` que está dentro de los `Models` que tiene un conjunto de informaciones relacionadas con las diferentes variaciones ya desarrolladas del juego que se mostrarán en el cliente (id, nombre y descripción).

TypeGameController: Una vez que el usuario en la interfaz gráfica termina de seleccionar los diferentes aspectos para generar una variación del juego, esta información es mandada al `Server` y este `Controller` lo que hace es recibir esas opciones y usa las clases correspondientes para construir nuestro `Manager` (Clase que controla el flujo del juego). Una vez hecho esto, se inicializa el juego y se manda la información inicial de la partida de vuelta al cliente.

NextTurnController: Este `controller` se ejecuta cada vez que en el juego corresponde a un nuevo turno, en donde este se encarga de ejecutar los métodos necesarios del `Manager` para realizar la próxima jugada, y devuelve la información correspondiente de esa jugada que se realizó.

Data

La sección de la `Data` tiene una clase principal `Data` que tiene varios arrays de instancias de las variaciones que tenemos implementadas en el juego, así como varios métodos que `parsean` algunas informaciones del juego en una estructura específica para que sea más fácil usar esa información en el `Client` (usando JSON)

```
 // Datos correspondientes a la cantidad de jugadores posibles a seleccionar
public int[] countPlayers = new int[] {
    2,
    3,
    4,
    5,
    6,
    7,
    8,
    9,
    10
};
// Datos correspondientes al máximo número que se le podrá poner a una ficha
public int[] maxIdTokens = new int[] {
    3,
    4,
    5,
    6,
    7,
    8,
    9,
};
// Cantidad de fichas que se seleccionará por jugador
public int[] countTokens = new int[] {
    3,
    4,
    5,
    6,
    7,
    8,
    9,
    10,
    11,
    12,
    13,
```

```
    14,
    15,
    16,
    17,
    18,
    19,
    20,
};
// variaciones de los jugadores
public Player[] Players = new Player[] {
    new RandomPlayer(),
    new BotaGordaPlayer(),
    new HeuristicPlayer(),
};
// Variaciones del tablero
public IBoard[] Boards = new IBoard[] {
    new UnidimensionalBoard(),
    new MultidimensionalBorad(),
};
// Variaciones de como se calcula el valor de una ficha
public TokenValue[] TokensValue = new TokenValue[] {
    new SumOfFaces(),
    new SubOfFaces(),
    new RareProperties(),
};
// Variaciones de como se pueden colocar dos fichas en el tablero
public IMatch[] Matches = new IMatch[] {
    new EqualFace(),
    new RareEquivalence(),
};
// Variaciones de como se distribuye las fichas entre los jugadores
public IDistributeTokens[] DistributeTokens = new IDistributeTokens[] {
    new RandomDistribution(),
    new AllforOneDistribution(),
};
// Variaciones de como se finaliza el juego
public IFinishGame[] FinishGames = new IFinishGame[] {
    new AllPassFinish(),
    new APassFinish()
};
// variaciones de como se gana el juego
public IWinGame[] WinGames = new IWinGame[] {
    new FewPoints(),
    new ALotPoints()
};
// Variaciones de como se selecciona el próximo jugador
public INextPlayer[] NextPlayers = new INextPlayer[] {
    new OrderTurn(),
    new RandomTurn(),
    new InvertedTurn(),
    new NextPlayerLongana(),
```

```
};
```

Game

También tenemos una clase estática `Game` la cual tiene la instancia general del Manager y dos métodos que "parsean"(lo que hacen es cambiar los nombres de las pripiedades) para que cuando se convierta a formato JSON que no sea con los nombres por defecto que asigna C#

El primer método es :

```
public static List<FacesToken> TokenForJson( IEnumerable<Token> tokens );
```

el cuál te parsea una lista de fichas a este formato:

```
public class FacesToken {
    public int? Left {get; set;} // valor de la cara izquierda
    public int? Right {get; set;} // valor de la cara derecha
    public string? Direction {get; set;} // dirección de la ficha en el tablero
}
```

que en JSON se veria asi:

```
{
    "left": <value:int>,
    "right": <value:int>,
    "direccion": "<direction:string>"
}
```

El método `PlayersForJson` parsea una lista de players con su información:

```
public static List<ResPlayer> PlayersForJson( PlayerInfo[] players, Refery refery );
```

Que lo transforma a:

```
public class PlayerInfo {
    public int? Id {get; set;}
    public string? Name {get; set;}
    public int? Points {get; set;}
    public FacesToken[]? HandTokens {get; set;}
}
```

y el método `TokensInBoardJson` que recibe la la matriz que representa el tablero y la convierte en una matriz de tokens parseados

```
public static List<List<FacesToken>> TokensInBoardJson ( (Token, string)[,] Tokens );
```

Classes

Dentro de esta carpeta tenemos varias clases que representan algunas cosas generales del juego. Dentro de estas están:

Auxiliar Classes

En el archivo `AuxiliarClasses.cs` tenemos varias clases, que son clases auxiliares para tareas específicas, como es el caso de las que usamos para parsear la información de alguna colección, los estados del juego entre otras cosas.

En este archivo podemos ver varias clases como:

`StatusCurrentPlay` : Esta clase guarda algunas informaciones del estado del juego después de cada jugada realizada. Esta clase es usada por el manager para capturar la información de cada jugada, información que es pública a todas las clases, pero principalmente pensada para los que los jugadores la puedan usar para sus estrategias.

`PlayInfo` : Esta clase también guarda información de las jugadas, pero esta clase contiene más información que la otra porque es la información que se manda al frontend para que sea mostrada.

`PlayerInfo` : Esta clase contiene información básica sobre un jugador(cantidad de fichas, puntos, y sus ids). Esta clase es usada por el `refery` y se usa para cuando queremos retornar información de un jugador, pero no queremos que no se tenga acceso a sus métodos para evitar, entre otras cosas, que los jugadores que necesiten de información de otro jugador hagan trampa.

`ResPlayer` : Esta clase también guarda la información de un jugador, pero esta clase en específico es la que se usa como parseo a JSON para que sea enviada al frontend

`FacesToken` : Esta clase parsea la información básica de una ficha(valor de sus caras, y su dirección en el tablero, si esta allí) a JSON para que sea usada en el frontend.

`PublicInformation` : Esta clase contiene toda la información publica que se maneja durante el juego y que se usa para las distintas implementaciones que necesiten de esta información.

Como te habrás dado cuenta, hay clases que guardan los datos de las mismas cosas, pero una guarda más información que otra. Esto se podía guardar todo en una mima clase y solo asignarle valores a las cosas que creamos conveniente, pero esto implicaría que las demás propiedades sean nulas, dando paso a posibles errores a la hora de usar las clases, así como no tener una idea clara de cuáles SI son las propiedades que tienen valores y cuáles no. Es por eso que preferimos crear varias clases que se referieren a la misma cosa, pero guardan diferentes cantidades de información.

Manager

Esta clase es la encargada de controlar el flujo de la partida, y posee todas las características(reglas) seleccionadas por el usuario.

- `public void AssignDependencies( ... )` : Este método recibe las clases que representan las variaciones seleccionadas por el usuario y estas son guardadas como propiedades internas del Manager.
- `public void StartGame( ... )` : Este método ejecuta las primeras acciones para iniciar el juego, o sea, crea todas las fichas(mediante el método `BuildTokens` del Board) y las reparte entre los jugadores(Mediante el método `distributeTokens` que corresponde a una de las variaciones seleccionadas por el usuario).
- `public PlayInfo GamePlay()` : Este método realiza toda la jugada que corresponde, guarda las actualizaciones pertinentes en sus propiedades internas y devuelve un resumen de dicha jugada, todo esto mediante las variaciones seleccionadas por el usuario al inicio del juego. Mas específicamente, mediante el método `NextPlayer` correspondiente a dicha variación devuelve el id del jugador que le toca jugar. Este mediante el método `Play` del refery se realiza la jugada de dicho jugador y se controla que se halla hecho correctamente. Posteriormente se realiza el guardado se las informaciones que son públicas en sus propiedades y se retorna la información necesaria para que será representada en la Interfas Gráfica(esta información es la que representa la clase auxiliar `PlayInfo`).
- `public int SearchPlayerIndex(int Id)` : este método simplemente busca en la lista de los jugadores el índice en el arreglo al que le corresponde el jugador con dicho `Id` .

Refery

Esta clase tiene la tarea de controlar las jugadas de cada jugador(esta clase la implementamos como forma de evitar el surgimiento de jugadores que incumplieran las reglas preestablecidas de la partida), además posee las fichas de todos los jugadores y les ordena a los mismos elegir que ficha jugar, revisando si la elegida es válida, y finalmente colocándola en el tablero. Posee varios métodos entre los que se encuentran:

- `MakeTokens` : Recibe y asigna a sus propiedades internas los jugadores y las fichas que tienen respectivamente.
- `Play` : El método más importante y se encarga de que el jugador realice una jugada correcta, o sea, controla que los

jugadores no hagan trampas y devuelve la información de la jugada. Este manda a llamar el método `PlayToken` correspondiente al jugador y que devuelve la ficha que va a jugar. En Refery controla que la jugada sea válida y en caso correcto llama `PlaceToken` del board para que sea puesta en el tablero y, posteriormente se elimina esa ficha de la mano del jugador.

Luego tenemos otros métodos auxiliares como son `Hand`, `Count`, `Points`, `Player` y otros más que ayudan a realizar funcionalidades auxiliares específicas de ayuda a los otros métodos.

Interfaces

En este directorio están todas las interfaces que se están usando en el programa. Ahi está la interfaz `IManager` que representa al manager y algunas funcionalidades variables que se pueden implementar

Implementaciones específicas

En este directorio están todas las implementaciones de diferentes variaciones del juego. Estas implementaciones se explican aquí

Detalles del funcionamiento del Server

Una vez ejecutamos el server, en el archivo `Program.cs` se añade como una dependencia del juego a la clase `Manager`

```
builder.Services.AddSingleton<IManager, Manager>();
```

Esto es para que después, mediante una inyección de dependencias en los controladores poder usar esta clase. Se especifica que sea una dependencia `Singleton` para que una vez se cree la instancia de `Manager` que representa el juego, esta misma sea la que se utilice en los demás controladores.

Posteriormente, cuando la interfaz gráfica carge, se realizará la primera petición al server mediante el controlador `LoaderController`, el cual devolverá las opciones del juego que estaban previamente guardadas en la clase `InterfaceOfOptions`.

Después de que en el Cliente se seleccionen todos las variaciones del juego a utilizar este es mandado nuevamente al Server mediante el controlador `TypeGameController`. Es en este momento en el que se genera la primera y única instancia de manager. Este controlador lo que hace es asignarle al manager todas esas variaciones que se seleccionaron previamente, y mediante el método `StartGame` del manager se ejecuta la primera jugada y, la información que es devuelta por este método es enviada al Cliente. Este método lo que devuelve son los jugadores con sus respectivas fichas, el nombre de quien jugó, su Id y sus puntos. Esta información es la que se muestra en el Cliente.

Cuando en el cliente se ejecuta solicita la próxima jugada, esta petición se hace mediante el controlador `NextTurn Controller`, el cual simplemente ejecuta el método `GamePlay` del `Manager` y este método es quien ejecuta toda la jugada correspondiente y devuelve la información de dicha jugada(quien la realizó, si se pasó o no, etc).

Una vez termine la partida esta es reiniciada con las mismas opciones o es cambiada por otras nuevas, en cualquier caso siempre se ejecutará esta misma lógica.

Vista general de la Abstracciones

- `Player` Esta clase es abstracta y contiene métodos generales de los jugadores. Tiene dos métodos abstractos, la creación del clon del jugador y `Play Token` que devuelve el índice de la ficha a ser jugada. También tiene la propiedad `IDPlayer` con el id y el nombre del jugador

Dentro de la carpeta `Interfaces` están las interfaces principales del juego que describes las abstracciones de las funcionalidades del mismo. Cada interfaz representa una característica del juego que puede ser modificada siguiendo las reglas que la interfaz establece.

Dentro de las variaciones que se pueden realizar están:

- `Player`: Es una clase abstracta que tiene varias propiedades fijas y el método abstracto `PlayToken` que representa la estrategia del mismo
- `IBoard`: especifica los métodos necesarios para poder crear una variación del Tablero

- `IDistributeTokens` : especifica los métodos necesarios para poder crear una variación de como se reparten las fichas a cada jugador
- `IFinishGame` : especifica los métodos necesarios para poder crear una variación de como se termina una partida
- `IMatch` : especifica los métodos necesarios para poder crear una variación de como dos fichas se pueden jugar(que tengan una cara en común por ejemplo)
- `INextPlayer` : especifica los métodos necesarios para poder crear una variación de como se selecciona el próximo jugador a jugar.
- `ITokenValue` : especifica los métodos necesarios para poder crear una variación de como se calcula el valor de una ficha
- `IWinGame` : especifica los métodos necesarios para poder crear una variación de como se selecciona el/los ganadores del juego

Abstracciones específicas

Dentro de los aspectos específicos variables del juego tenemos:

Board

El board es la clase que se encarga de mantener y darle forma a las fichas que los jugadores han jugado, esta puede variar en dependencia del tipo de juego que se quiera jugar, por lo que se creó la interfaz `IBoard`, que modela los métodos básicos que tiene un tablero.

Dentro de las responsabilidades que tienen los tableros está la construcción de las fichas que se van a usar para jugar mediante el método:

```
public List<Token> BuildTokens(int MaxIdOfToken, TokenValue calcValue)
```

y además contiene otro método importante llamado:

```
public bool ValidPlay(Token token);
```

que usando una instancia de la clase IMatch valida si una ficha puede ser jugada en el tablero.

Tablero Unidimensional

Este es la primera variación del tablero y representa una mesa en donde los jugadores solo pueden jugar fichas por la esquina izquierda o derecha de la cola de fichas ya jugadas.

Dentro de los métodos de la clase están los definidos por la misma interfaz `IBoard` :

- `public List<Token> BuildTokens( int MaxIdOfToken, ITokenValue calcValue )` : Recibe el número máximo que puede tener una ficha y una forma de calcular el valor de la ficha y lo que hace es construir todas las fichas posibles de dos caras, desde el [0,0] hasta el [MaxIdOfToken, MaxIdOfToken].

- `public (Token, string)[,] TokensInBoard` : Devuelve una matriz [1, n] donde están colocadas todas las fichas de la partida, esto es simulando visualmente a un tablero. Como todas las fichas se colocan en horizontal, el alto de la matriz es 1, cuando se devuelva la matriz, las todas las posiciones de las fichas deben de ser válidas.

- `public void PlaceToken( Token token, int IdPlayer )` : Recibe una ficha y el id del jugador que la va a jugar y en caso de que la jugada sea válida la coloca en el tablero y la añade a una lista en donde guarda cada ficha con su jugador para mantener un histórico de jugadas. Este método se apoya de otro interno de la misma clase llamada `Play` que lo que hace es recibir la ficha y la posición donde se va a colocar la ficha y colocarla y, si intercambiar las caras de las fichas de ser necesario. Esto es para que todas las fichas estén debidamente organizadas y todas las caras coincidan correctamente.

Tablero Multidimensional

Este tablero es un poco diferente, ya que los jugadores pueden jugar sus fichas y estas pueden ser colocadas por los laterales de la pila de fichas ya jugadas o, si se jugó algún doble, entonces se podrán jugar por los cuatro lados de la ficha(las dos caras normales más por encima y por debajo), saliendo de este otra ramificación del tablero por donde se podrá jugar normalmente.

Esta implementación usa un diccionario para guardar las posiciones en la "matriz" ficticia de las fichas:

```
private Dictionary< Coord, InfoToken > board = new Dictionary<Coord, InfoToken> (); // Tablero de jueg
```

Este tablero usa dos clases internas, `Coord` y `InfoToken` en donde `Coord` es una clase interna con dos variables **X** y **Y** para guardar las coordenadas en el tablero de donde se jugó la ficha; y tenemos a `InfoToken` que contiene la ficha que se jugó y la dirección en la que se coloca (horizontal o vertical).

Dentro de los métodos de la clase están los definidos por la misma interfas `IBoard`:

- `public List<Token>BuildTokens(int MaxIdOfToken, ITokenValue calcValue)` : Crea todas las fichas posibles de dos caras desde el [0,0] hasta el [MaxIdOfToken, MaxIdOfToken], pero si son dobles, entonces crear la ficha pero con 4 caras por donde jugar, para poder jugar por las 4 direcciones.
- `public void PlaceToken(Token token, int IdPlayer)` : Este método toma la ficha a jugar y hace un recorrido por todas las fichas del tablero buscando que ficha tiene alguna posición libre para jugar y si las dos fichas cumplen las reglas necesarias para ser jugadas. Una vez que encuentra una ficha por donde jugar entonces comprueba la dirección de la ficha en el tablero(horizontal o vertical) y jugar por un lado o por otro en dependencia y, si la ficha es un doble, comprobar también si se puede jugar por arriba o por abajo de la misma. Aquí también se realiza el proceso de cambio de caras de ser necesario para que si en cualquier momento queremos renderizar la matriz las fichas estén organizadas y coincidan unas con otras.
- `public bool ValidPlay(Token token)` : Valida si la ficha dada puede ser jugada en el tablero, o sea, si existe alguna posición en el tablero por donde pueda ser jugada la ficha.
- `public Tuple<Token, int>[] OrderListOfTokensByPlayer` : Convierte el diccionario que teníamos con las fichas y los ids de los jugadores que realizaron esa jugada en un arreglo que es devuelto.
- `public (Token, string)[,] TokensInBoard` : Convierte el diccionario que representa el board en una matriz de Tokens con sus respectivas direcciones, donde en la posición [i,j] estará la ficha con coordenadas x,y en el diccionario. Este método se apoya de otros dos llamados `BordersBoard` y `NormalizeBoard` para modificar un poco el board. Esta normalización se debe a que cuando añadimos por la izquierda una nueva ficha a una ficha del tablero en la posición [0,0] por ejemplo, esta nueva ficha es colocada en la posición [-1,0] y esta no es una posición válida en una matriz de elementos de C#. Para solucionar esto lo que se hace es usar el método `BordersBoard` que calcula las esquinas del tablero basado en los índices de las posiciones(el índice más a la derecha, a la izquierda, arriba y abajo) y después con el método `NormalizeBoard` lo que hacemos es correr toda la matriz al cuadrante positivo de tal forma que la ficha más a la esquina derecha y abajo sea el [0,0]. Ya con el tablero de esta forma podemos crear la matriz y retornarla correctamente.

Indice

Estrategia de un jugador

Abstraido en una clase abtracta `Player` que representa la estrategia de cualquier jugador. Posee en la clase abstracta la asignación de su nombre y Id, y un método abstracto `PlayToken` que debe devolver la posición en el array que se le pasa de la ficha que desee jugar, si incumple este principio devolviendo un número de una posición inválida en el array, se considera un turno perdido.

```
public abstract int PlayToken( IBoard board, Token[] hand);
```

Random Player

Este jugador realiza las jugadas de manera random. Mediante el método `PlayToken` este recibe las fichas que le corresponden y selecciona una ficha válida de manera random que será la que jugará. En caso de no tener fichas válidas retorna `-1`.

Bota Gorda Player

Selecciona entre todas sus fichas válidas la de más valor, siempre basándose en la implementación de valor seleccionada en la

partida.

Este simplemente lo que hace es recorrer todas las fichas que tiene y quedarse con la ficha válida de mayor puntaje. Si no tiene jugadas válidas retorna `-1`.

Heuristic Player

Jugador basado en unas heurístas simples, siempre intenta salir con un doble, y jugar la ficha cuyas caras estén más repetidas en su mano.

Lo primero que hace el jugador es que mediante el método privado `Organize(...)` guarda en una lista la cantidad de fichas que tiene cada número hasta el máximo posible.

```
private void Organize(int maxidtoken, Token[] hand);
```

También usa otro método llamado `Double` que retorna el índice de todos los dobles que hay en su mano

```
private List<(Token,int)> Double(Token[] hand);
```

Luego si es la primera persona en jugar de todas, entonces mediante el método `Start(...)` realiza la primera jugada:

```
private int Start(Token[] hand);
```

Este método lo que hace es devolver el doble con el número mas alto que tenga, y en caso de que no tenga dobles entoces devuelve la ficha con las caras que tengan los números que más se repiten en su mano.

Si por el contrario simplemente le toca jugar, entoces jugará la ficha con los números que más se repitan en su mano. Si no tiene jugadas válidas entonces retorna `-1`.

Indice

Distribución de las fichas por los jugadores

Abstraido en una interfaz `IDistributeTokens` con un método `DistributeTokens` el cual debe ser implementado devolviendo la distribución de las fichas para la partida que el implementador desee.

```
List<Token>[] DistributeTokens(List<Token> tokens,int numberofplayers,int countTokens);
```

Random Distribution

Esta implementación simplemente reparte de manera random las fichas que le tocan a cada jugador.

Todas las fichas del mismo tipo

Reparte las fichas siguiendo la idea de dar tantas fichas con igual representación en una de sus caras como se pueda, cuando no se puedan dar más se completan aleatoriamente. Esto se implementó de la siguiente manera: Primero que todo se ordenaron los jugadores de manera aleatoria y seleccionado un número random en el rango de las representaciones de las fichas y escogiendo todas las fichas que tengan ese número en alguna de sus caras y dárselas al jugador(obviamente sin salirse de la cantidad de fichas máximas que puede tener un jugador) y, en caso de que le falten fichas se seleccionan las que faltan de manera aleatoria.

Este mismo procedimiento es seguido para todos los jugadores, siendo el primer jugador al que le repartieron las fichas el que más de un mismo tipo va a tener.

Indice

Valor de las fichas

Abstraido en una Interface `ITokenValue` con un método `Value` que recibe un `Token` y debe devolver un entero que represente el valor de esa ficha.

Suma de caras

El valor de una ficha esta dada por la suma absoluta del valor de sus caras.

Resta de caras

El valor viene dado por la diferencia absoluta de las caras de la ficha.

Cálculo raro y aleatorio

Auxiliandonos de un delegado creamos varias formas de calcular el valor de la ficha y aleatoriamente aplicamos una en la ficha dada.

Dentro de las formas de calcular el valor estan:

```
 private int a(int right, int left){
    return (int)Math.Abs(Math.Pow(right,3) - Math.Pow(left, 2));
}
private int b(int right, int left){
    return (2 + right + left)/2;
}
private int c(int right, int left){
    return (int)Math.Abs((Math.Cos(left)-Math.Sin(right))*10);
}
private int d(int right, int left){
    return Math.Abs((right + left)*(right-left));
}
private int e(int right, int left){
    return (int)((Math.Sqrt(right) + Math.Log2(left + 1))*5);
}
private int f(int right, int left){
    return 0;
}
private int g(int right, int left){
    return 1000;
}
private int h(int right, int left){
    return (int)Math.Abs(Math.PI*(Math.Pow(right,2)-Math.Sqrt(left+10)));
}
```

Entonces cuando se consulte el valor de una ficha, este seleccionará un método de estos de forma random y devolverá como valor de la ficha la evaluación de la expresión matemática que contiene el método.

Indice

Final de la partida

Abstraido en una interface `IFinishGame` con un método booleano `FinishGame` que recibiendo el estado del tablero y la

información de cada jugador debe decidir si el juego terminó.

```
bool FinishGame( IBoard board, IEnumerable<PlayerInfo> players );
```

La mayoria se pasan

El juego finaliza cuando alguien se pegue o la mayoría de ellos se hallan pasado al menos dos veces a lo largo de la partida

Todos se pasan

El juego termina cuando algún jugador se pega, o si ocurren la cantidad de jugadores en pases consecutivos(también cuenta si todos los jugadores se pasan)

Indice

Siguiente Jugador

Abstraido en una interfaz `INextPlayer` que contiene el método `nextPlayer` que devuelve el ID del jugador que le toca jugar en ese momento dado de la partida.

```
int NextPlayer( PlayerInfo[] players );
```

Orden orgánico

El orden es orgánico, fijo donde se preestablece un orden al iniciar la partida y ese se mantiene hasta el final de la partida. Por defecto, el orden establecido es el mismo en el que estaban los jugadores en el array la primera vez que se recibió.

Para hacer esto tenemos usamos una variabl curs donde cada vez que el método es llamado, este aumenta su valor (o se reinicia a cero si el valor es mayor que la cantidad de jugadores) y devuelve el jugador en esa posición.

Orden aleatorio

El siguiente jugador es seleccionado de forma aleatoria, por lo que cada vez que el método es llamado, este simplemente escoge un número random entre 0 y la cantidad de jugadores y devuelve el jugador que se encuentre en esa posición.

Invirtiendo el orden

El juego comienza con un orden establecido, pero si alguien se pasa este orden es invertido y el orden establecido por defecto es el orden en el que estaban los jugadores en el array la primera vez que se recibieron. Para seleccionar el próximo jugador lo que hace el método es guardar cada vez que es llamado el estado de la cantidad de fichas que tiene cada jugador, así cada vez que el método es llamado nuevamente se compara la cantidad de fichas actual con las del estado anterior y, si el jugador actual que jugó tiene las mismas fichas que en el estado anterior(quiere decir que se pasó y no realizó ninguna jugada) entonces el próximo jugador a jugar es el anterior a él, y en caso contrario(la cantidad de las fichas son diferentes por lo que si realizó la jugada) el próximo jugador es el que le sigue a él.

Todas las fichas

El mismo jugador repite su turno hasta que no le queden jugadas válidas por realizar. Este método usa la misma idea que la anterior; guarda los estados de la cantidad de fichas por jugadores y, mientras el jugador actual siempre tenga fichas diferentes, es porque tiene jugadas válidas y siempre será el próximo jugador él mismo, hasta que la cantidad de fichas sea la misma y el turno pase al siguiente jugador.

Indice

Ganador

Abstraido en una inteface `IWinGame` el cual debe devolver un `IEnumerable` con el orden en que quedan los jugadores al finalizar de cada jugada.

Esta interfas solo un método que devuelve la lista ordenanda de los jugadores según si criterio de orden para ganar el juego

```
IEnumerable<PlayerInfo> GetWinnersGame( IBoard board, IEnumerable<PlayerInfo> players );
```

Más puntos

Esta implementación establece que gana el jugador con mayor suma de los valores de las fichas que le quedan en la mano, pero si alguien se pega, este es el que gana.

Por lo tanto el método lo que devuelve es, dado la lista de los jugadores, esta es ordenado de tal forma que todo jugado tiene más puntos que cualquier jugado tal qu i < , excepto si alguien se pegó, en cuyo caso el mayor jugador ganador tendrá cero puntos.

Menos puntos

Esta implementación establece que gana el jugador con menor suma de los valores de las fichas que le quedan en la mano.

Por lo tanto el método lo que devuelve es, dado la lista de los jugadores, esta es ordenado de tal forma que todo jugado tiene menos puntos que cualquier jugado tal qu i < .

Indice

;

Match entre fichas

Abstraído en una interfa IMat cuya implementación debe devolver si dos fichas se pueden jugar una con otra.

Esta interfaz cuenta con tres métodos básicos para validar jugadas entre dos fichas.

```
public interface IMatch
{
    // Valida si dos fichas pueden ser jugadas juntas
    bool ValidateMatch( Token token1, Token token2 );
    // Valida si por la cara face1 de ficha1 se puede jugar la cara face2 de ficha 2
    bool ValidateMatch( Token token1, int face1, Token token2, int face2 );
    // Devuelve un array con las caras de las fichas que pueden ser jugadas
    int[] WhichFacePlay( Token token1, Token? token2 );
    // Retorna un clone
    IMatch Clone();
}
```

Caras iguales

Esta implementación se basa en que si dos fichas son jugables entre si entonces es porque tienen al menos una cara en común.

- `bool ValidateMatch( Token token1, Token token2 );` : Devuevle true si el `token1` y el `token2` tienen al menos una cara con valores iguales.
- `bool ValidateMatch( Token token1, int face1, Token token2, int face2 )` : Devuelve true si la cara `face1` de `token1` es igual a la cara `face2` de `token2` .

- `int[] WhichFacePlay( Token token1, Token? token2 )` : Retorna una lista con los valores de las caras que tienen en común las dos fichas.

Conexiones raras

La fichas se pueden jugar siguiendo ciertas reglas. En general, dos fichas son aptas para jugarse si:

1. El valor de la cara de una ficha es el número previo al de una cara de la otra ficha.
2. Si una cara tiene valor cero, entonces se puede jugar con cualquier otra ficha.
3. Si una cara de una ficha es múltiplo de una cara de la otra ficha.
4. No se ha jugado por esa cara

Indice☝