Juego de Dominó

Karen Dianelis Cantero López C211

Francisco Vicente Suárez Bellón C212

Con este proyecto, proponemos una solución al problema planteado, el cual era modelar un juego de dominó que tuviera la capacidad de ser cambiado fácilmente y poderle añadir nuevas implementaciones sin tener que cambiar nada del código.

Para este proyecto, hemos considerado cuales aspectos pueden ser cambiados de un juego de dominó, y en base a eso hemos hecho las siguientes abstracciones:

A nivel de juego:

1. Que tipo de jugada es válida:
    a. Clásico (solo si los numeros son iguales)
    b. Solo si el número a jugar es mayor al número ya en la mesa
    c. Solo si el número a jugar es menor al número ya en la mesa
2. Manera de darle una puntuación a una ficha:
    a. Clásico. El valor de la ficha es la suma de sus partes
    b. Doble. Si una ficha es doble, su valor se duplica
3. Diferentes tipos de jueces:
    a. Juez Honesto
    b. Juez Corrupto
4. Las estrategias que los jugadores utilizan para jugar:
    a. Estrategia Random
    b. Estrategia BotaGorda
    c. Estrategia de un jugador semi-inteligente
5. Las condiciones para ganar un juego:
    a. Gana el jugador que tenga menos puntos
    b. Gana el jugador que tenga mas puntos
    c. Gana el jugador que tenga la media de puntos
6. Las condiciones para que un juego termine:
    a. Clásico. Cuando alguien se pegue o se tranque el juego
    b. Se acaba cuando un jugador tenga un score específico
7. El tipo de jugador:
    a. Jugador Normal

b. Jugador Corrupto/Tramposo

A nivel de Torneo:

8. Tipo de Ficha:
    a. Fichas de enteros
    b. Fichas de termoeléctrica
9. Las condiciones de parada de un torneo:
    a. Cuando un jugador acumule x cantidad de puntos
    b. Cuando haya una cantidad x de ganadores
10. Las condiciones para descalificar a un jugador del torneo:
    a. Si ha perdido mas de un x por ciento del total de juegos
    b. Si el jugador ha perdido una x cantidad de veces consecutivas
11. Tipo del juez a nivel de torneo:
    a. Juez Honesto
    b. Juez Corrupto

Lógica del Juego

Existe un objeto, llamado el observer, el cual es el encargado de escribir todo lo necesario en la parte gráfica, y de recibir el input del usuario.

A modo de resúmen, el flujo del proyecto es el siguiente:

1. Lo primero que ocurre es que el usuario elige como crear su juego. Esto se hace desde la consola, mediante dicho observer.
2. Se crean las instancias de las características del juego mediante el uso de reflection, lo cual permite que al realizar otra implementación de una interfaz, esta se reconozca automáticamente.
3. Se crea el Torneo, el cual llama a su método Run(), dentro del cual se recorre por su lista de Games ya creados y se llama al método PlayGame() dentro de la instancia del juego correspondiente.
4. En cada juego:
    a. Se recorre por cada jugador y se le pide la ficha a jugar.
    b. El jugador devuelve, mediante su método “BestPlay()”, la ficha a jugar
    c. El juez rectifica que la ficha puede ser jugada, y si es así la agrega a la mesa.
    d. Hace lo mismo con cada jugador hasta que el método EndGame() devuelva true, en cuyo caso se cumplió la condición de paradas del juego seleccionada.
5. Una vez termine un juego, el Torneo rectifica que se pueda seguir jugando. Esto lo hace al verificar si se cumple la condición de parada seleccionada para el torneo.
6. Si se acaba el juego, mediante el observer, se imprimen en pantalla todas las estadísticas de los jugadores.
7. De lo contrario, se vuelve a llamar al metodo PlayGame() del próximo juego.

A continuación pasaremos a explicar en más detalle las interfaces, qué hacen y cómo funcionan.

Una vez se hayan elegido todas las opciones de juego, y todas las instancias estén creadas, se crea el objeto de Torneo, el cual es una implementación de la interfaz:

```
public interface  IChampionship<TStatus>
```

Como concepto, el torneo consiste en un conjunto de partidas independientes entre sí, con ciertas reglas como la de finalización de este, el/los ganadores y la validación de que jugadores deben continuar a la siguiente partida a los cuales delega esas funcionalidades dado que el comportamiento de este es solo de ejecutar las partidas.

El torneo contiene un método Run, el cual se encarga de inicializar el método inicializador de cada juego individual. Cada juego implementa la interfaz IGame<TStatus>

```
public void Run()
```

Además debe dar paso a otra clase responsable de portar la información con la entidad gráfica bajo los eventos: “Status”. El torneo invoca a una clase encargada de portar la información y enviar dicha información hacia la parte visual. Esta clase auxiliar puede ser utilizada en varios ámbitos graficos.

La implementación dada es:

```
public class ChampionStatus

        public Stack<GameStatus> FinishGame

        public List<PlayerStats> PlayerStats

        public bool HaveAWinner

        public List<IPlayer> Winners

        public bool ItsAnGameStatus

        public bool ItsAFinishGame

        public GameStatus gameStatus

        public bool FinishChampion
```

Su función principal será de portar el conocimientosobre el último estado de la partida en ejecución.

El evento CanContinue: se encarga únicamente de esperar confirmación de cualquier ente exterior sobre si se puede o no continuar dicho torneo. Para ello se determinó que se debe implementar un Enum Orders

```
public virtual event Predicate<Orders> CanContinue;

public virtual event Action<TStatus> status;
```

Explicación de las clases auxiliares:

En términos globales, se tiene un conjunto de jugadores a los cuales la responsabilidad de contenerlos y decicidir si se configuró para jugar en dicha partida o no, es delegada a la clase PlayersCoach la cual solo determina si un jugador desea o no jugar en dicha partida.

```
public class PlayersCoach
```

Para Interpretar las reglas del Torneo se auxilia de la implementación de la interfaz

```
public interface IChampionJudge<Tstatus>
```

Nota: El tipo Tstatus es determinado por el del tipo que implementa la Interfaz IGame<TStatus>

La funcion de este juez de torneo es determinar si es válido continuar y quién debe continuar; para ello se auxilia de las reglas del torneo (este juez es el encargado de la interpretación de las mismas).Estas reglas son:

- Condición de finalización del torneo
- Ganador o ganadores del torneo
- Si es válido o no continuar jugando por parte de un jugador
- Cómo se calcula el score de dicho jugador a nivel de torneo

Sus metodos fundamentales son:

`public virtual void Run(List<IPlayer> players)` *[El cual debe de invocarse antes de comenzar el torneo para dar a conocer a este los posibles jugadores]*

`public virtual bool EndGame(List<IGame<GameStatus>> game)` *[El cual, basándose en los criterios o reglas del juego determina si debe o no finalizar el torneo] [Por defecto, el torneo finalizará a lo sumo cuando se hayan jugado todas las partidas]*

`public virtual bool ValidPlay(IPlayer player)` *[En ella se determina si un jugador puede o no continuar jugando el torneo]*

`public virtual List<IPlayer> Winners()` [*Devuelve una lista con el/los ganador/es del torneo]*

Una Partida:

Como concepto, una partida debe contener todos los elementos que aseguren el funcionamiento de una partida de dominó típica:

```
public interface IGame<TStatus> : ICloneable<IGame<TStatus>>{

    event Action<TStatus>? GameStatus;

    event Predicate<Orders> CanContinue;

    IBoard? board { get; }

    List<IPlayer>? GamePlayers { get; }

    double PlayerScore(IPlayer player);

    IGame<TStatus> Clone();
```

```
        TStatus PlayAGame(IBoard board, List<IPlayer> players);

        List<IPlayerScore> PlayerScores();

        List<IPlayer> Winner();  }
```

Esta interfaz contiene la escencia de una partida. La implentación utilizada promueve el desacople de las funcionalidades de esta.

La clase implementada debe dar a conocer que tipo de implementación de TStatus es necesaria.

Bajo la implementación dada está la clase game estatus (Sera explicada mas adelante cuando se explique la interfaz gráfica).

Para comenzar una partida debe invorcarse el siguiente metodo, el cual recibe el tablero a jugar y los jugadores de dicha partida.

```
        TStatus PlayAGame(IBoard board, List<IPlayer> players)
```

Jugadores:

Los jugadores deben Implementar la interfaz IPlayer:

```
    public interface IPlayer : ICloneable<IPlayer>, IEquatable<IPlayer>, IEqualityComparer<IPlayer>,
IDescriptible, IEquatable<int>{

        List<IToken> hand { get; }

        int Id { get; }

        int TotalScore { get; set; }

        IPlayerStrategy strategy { get; }

        List<IPlayerStrategy> strategias { get; }

        void AddHand(List<IToken> Tokens);

        void AddStrategy(IPlayerStrategy strategy);

        IToken BestPlay(IWatchPlayer watchPlayer);

        int ChooseSide(IChooseStrategyWrapped choose, IWatchPlayer watchPlayer);

        IPlayer Clone();

        bool Equals(IPlayer? other);

        bool Equals(int otherId);

        int GetHashCode(IPlayer obj);

        string ToString();  }
```

Un jugador, como concepto abstracto debe:

- tener una o más estrategias de juego
- contener una mano de fichas para evaluar su mejor jugada
- devolver una ficha y dar la posición donde esta debe ponerse (para ello se le entrega un IWatchPlayer watchPlayer el cual debe contener la información de la partida como las reglas y el tablero)

Estrategia de jugadores:

```
public interface IPlayerStrategy : IDescriptible {

  int Evaluate(IToken itoken, List<IToken> hand, IWatchPlayer watch);

  int ChooseSide(IChooseStrategyWrapped choose, IWatchPlayer watch);   }
```

Esta estrategia se encarga de recibir una ficha, y evaluarla con respecto a las reglas de la partida. Luego le asigna una puntuación a esta ficha (mientras mayor sea la puntuación, mejor es la ficha). La estrategia tambien se encarga de decidir por que lado del tablero se debe jugar la ficha.

Juez:

El juez de la partida debe ser quien intérprete las reglas, añada fichas al tablero y evalúe si el juego llegó a su finalización.

```
public interface IJudgeGame {

  bool AddTokenToBoard(IPlayer player, GamePlayerHand<IToken> hand, IToken token, IBoard board, int side);

  bool EndGame(List<(IPlayer, List<IToken>)> players, IBoard board);

  double PlayerScore(IPlayer player);

  List<IPlayerScore> PlayersScores();

  IWatchPlayer RunWatchPlayer(IBoard board);

  IChooseStrategyWrapped ValidPlay(IPlayer player, IBoard board, IToken token);

  List<IPlayer> Winner(List<(IPlayer player, List<IToken> hand)> players);

}
```

Dado que es el conocedor del conjunto de reglas que se ejecutan en el juego en determinado momento, sus métodos deben de ser invocados por la clase game únicamente y debe ser ella quien solo tenga acceso a estos. Las responsabilidades del juez son:

- Decir si el juego llegó a las condiciones de finalización, lo cual se lo pregunta a IStopCondition.
- Decir es válida una acción como poner una ficha en el tablero, lo cual se lo pregunta a IValidPlay
- El juez tambien debe entregar un IWatchPlayer al jugador en su turno. Este objeto se crea para que el jugador tenga acceso a las reglas del juego, pero no tenga acceso a los metodos propios del juez.
- El juez debe agregar las fichas en el tablero en caso de ser posible y devolver true si fue realizado con éxito, para esto necesita conocer: el jugador que quiere realizar dicha accion, el tablero en su estado actual , el token que se desea jugar y la dirección del mismo.

- Devolver una lista con una implementacion de IplayerScore donde devuelva el score de estos en la partida
- Debe dar a conocer el score en double de un jugador en especifico, el cual lo calcula en dependencia de la interfaz IGetScore.
- Determinar el/los ganadores auxiliándose de IwinCondition.

Manager de las fichas:

```
public interface ItokensManager    {

    List<IToken> Elements { get; }

    IEqualityComparer<IToken> equalityComparer { get; }

    IComparer<IToken> Comparer { get; }

    List<IToken> GetTokens();

    bool ItsDouble(IToken itoken);   }
```

Esta clase contiene la responsabilidad de conocer como se debe repartir las fichas de los jugadores y como se comparan dos fichas.

La mano del jugador: dada la posibilidad de una implementación errónea de un jugador o juez puede verse afectado el comportamiento de la mano del jugador por ello es un concepto desacoplado.

```
public class GamePlayerHand<TToken>
```

La clase es genérica dado que puede ser usada en cualquier tipo de implementación de una ficha incluso de una externa de IToken

Tablero:

Un tablero debe de poder brindar la información para conocer la primera y última ficha, además de poder agregar fichas a este y poder ver todas las fichas que contiene.

```
public interface IBoard : ICloneable<IBoard> {

    List<IToken> board { get; }

    IToken First { get; }

    IToken Last { get; }

    void AddTokenToBoard(IToken itoken, int side);

    IBoard Clone();

    IBoard Clone(List<IToken> CopyTokens);

    string ToString();

}
```

Se recomienda que cuando se brinda la propiedad board sea por clonación de los elementos del mismo, para que el usuario que reciba el objeto board, le pueda cambiar una ficha por dentro, pero como es una copia, el verdadero board no se ve afectado.

Tokens:

Como concepto genérico, las fichas son cualquier elemento que tengan un comportamiento bajo la interfaz ITokenizable, se conforman por dos Partes.

```
public interface IToken {
    ITokenizable Part1 { get; }
    ITokenizable Part2 { get; }
    IToken Clone();
    bool ItsDouble();
    void SwapToken();
    string ToString();
}
```

Un Token, que implementa Itoken va a constar de dos partes, estas partes son de tipo Itokenizable. Esto trae la ventaja de que, si se quisiera, se pudiera tener una ficha tuviera diferentes tipos en cada cara, siempre que ese tipo implemente ITokenizable

```
public interface ITokenizable : IComparable<ITokenizable>, IEquatable<ITokenizable>, IDescriptible{
    string Paint();
    double ComponentValue { get; }
}
```

Generador de fichas

Como el comportamiento de generación de una ficha es autónomo a esta, se necesita de una implementación de un generador de fichas.

```
public interface IGenerator : IDescriptible{
    public List<IToken> CreateTokens(int maxDouble);
}
```

Este generador, conociendo el máximo doble solamente, debe poder crear una lista de tokens bajo sus propios criterios.

Interfaces de Respaldo

A concepto general, desde un torneo hasta una partida, debe de usarse una serie de reglas las cuales deben de tener un intérprete (en esta implementación, el intérprete es el juez)

- Condición de ganada IWinCondition:

Dado un criterio y una forma de tener una puntuación de otro objeto no necesariamente distinto, debe poder devolver una lista de los jugadores que han ganado bajo cierto criterio (una partida o un torneo).

```
public interface IWinCondition<TCriterio, TToken> : IDescriptible{

    List<IPlayer> Winner(List<TCriterio> criterios, IGetScore<TToken> howtogetscore);

}
```

- Condición de validez

Debe determinar si: 1. Una jugada es valida dentro de un juego o 2. Es valido que un jugador juegue un juego. (en dependencia de como se use la interfaz)

```
public interface IValidPlay<TGame, TPlayer, TCriterio> : IDescriptible {

        TCriterio ValidPlay(TGame game, TPlayer player);

}
```

- Condición de parada

Debe poder dictaminar si la partida o torneo ha llegado a una condición de finalización

```
public interface IStopGame<TCriterio, TToken> : IDescriptible{

        bool MeetsCriteria(TCriterio criterio, IGetScore<TToken> howtogetscore);

}
```

- Puntuación

Dado un criterio, sea una ficha o una jugador, debe poder devolver un double el cual sea la puntuación de este

```
public interface IGetScore<TToken> : IDescriptible{

    double Score(TToken item);

}
```

Interfaz Gráfica

Si se desea enlazar la implementación de backend dada a otro entorno visual debe de conocer el siguiente reporte.

Para ello debe tener un encargado de satisfacer los constructores de dichas clases ensamblándose primeramente las partidas y posteriormente las condiciones del constructor del torneo.

Una vez que se ha creado, debe suscribirse a los eventos de la clase torneo ChampionStatus y CanChoose.

El Champion Status es un contenedor de todo los que sucede en el torneo. Se enviará uno cada vez que ocurra un movimiento del dominó clásico en lo interno del torneo.

GameStatus:

```
public class ChampionStatus
        public Stack<GameStatus> FinishGame
        public List<PlayerStats> PlayerStats
        public bool HaveAWinner
        public List<IPlayer> Winners
        public bool ItsAnGameStatus
        public bool ItsAFinishGame
        public GameStatus gameStatus
        public bool FinishChampion
```

Contiene una pila con todos los estados de las partidas , donde el primer elemento de la pila es el estado mas reciente del juego y una lista de PlayerStats.

PlayerStats:

```
public class PlayerStats : IEquatable<PlayerStats>
        public IPlayer player
        public double puntuation [contiene el jugador y su puntuación a nivel de torneo]
        bool ItsAnGameStatus:
        bool ItsAFinishGame
        GameStatus gameStatus
        bool FinishChampion
```

Evento CanChoose:

Este evento espera una respuesta booleana para continuar la partida o torneo .

GameStatus:

```
public class GameStatus{
        public List<IPlayer> winners [Contiene en caso de haber los ganadores de la partida]
        public List<PlayerStats> punctuation [Puntuación de los jugadores]
        public bool ItsAFinishGame [True si se es el ultimo estado de esta partida]
        public List<GamePlayerHand<IToken>> Hands [La mano de los jugadores actual]
        public IBoard board [Tablero en estrado actual]
        public IPlayer actualPlayer [El jugador del turno que se esta jugando]
          public GamePlayerHand<IToken> PlayerActualHand [Mano del jugador en el turno actual]
}
```

Nota: En nuestra solución todos los valores se pasan clonados de una clase a otra.