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 más 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 ganadores en la partida]
    public List<PlayerStats> punctuation [Puntuación de los jugadores]
    public bool ItsAFinishGame [True si se es el último estado de esta partida]
    public List<GamePlayerHand<IToken>> Hands [La mano de los  jugadores]
    public IBoard board [Tablero en estado 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.

Luego de haber explicado las interfaces y la lógica de nuestro proyeto, pasaremos a explicar las clases usadas para nuestra solución.

Relacionadas con el torneo:

Clase Championship

Esta clase implementa la interfaz IChampionship.

Tiene como propiedades dos eventos: el evento CanContinue el cual pregunta al final de cada partida si se puede continuar jugando; y el evento status el cual es el evento encargado de enviar la información acerca del torneo, desde poner una ficha hasta el final del mismo. También como propiedades tiene: un entero CountOfGames que guardará la cantidad de partidas del torneo; un juez de tipo IChampionJudge; una lista de juegos de tipo IGame que guardará todas las partidas jugadas en el torneo; un PlayersCoach que es el encargado de organizar los jugadores a nivel de torneo; un bool HaveAWinner que va a ser true si existe un ganador en el torneo, y falso si no; una lista de juegos de tipo IGame llamada FinishedGames que va a contener los juegos finalizados; un stack de GameStatus llamado GameStatus que va a contener todos los estados de las partidas; una lista de jugadores llamada Winner que contendrá los ganadores a nivel de torneo; un bool ItsChampionOver que será true si el campeonato cumple la condición de finalización, y falso si no; una lista de PlayerStats que va a contener las puntuaciones de los jugadores; y una lista de jugadores llamada AllPlayers que contendrá todos los jugadores que han jugado en el torneo.

El método Run() es el encargado de jugar las partidas. Por cada una de las partidas a jugar, se llama al método PlayAGame de la clase Game, el cual es el encargado de jugar la partida; luego se agrega este game a la lista de juegos finalizados del juez; luego se le pregunta al juez si se puede seguir jugando, si se puede entonces se da la orden NextPlay y si no se puede entonces se invoca el método ChampionOver().

El método GameOver() se encarga de hacerle push al último estado de la partida en el stack de GameStatus y de agregar el juego a la lista de juegos finalizados.

El método CreateAChampionStatus() devuelve un nuevo ChampionStatus del torneo.

El método PrintGames() recibe un gamestatus como parametro. En este metodo, se crea un championStatus llamando al método CreateAChampionStatus(), luego a este ChampionStatus se le agrega el gameStatus recibido y luego se invoca el evento status.

El método ControlPlayers() es el encargado de preguntar al juez que jugadores son los que pueden jugar la partida.

El método PlayerStatistics() es el encargado de crear la lista de PlayerStats, y de actualizar las puntuaciones de cada jugador en cada partida jugada. Esta lista nueva que crea cada vez que se llama el método es guardada en la propiedad PlayerStats.

El método ChampionOver() hace que la variable ItsChampionOver sea true, indicando así que el torneo acabó. Luego llama al método PlayerStatistics() ya explicado anteriormente. Luego se crea un nuevo ChampionStatus mediante el método CreateAChampionStatus() y este ChampionStatus se invoca mediante el evento status.

El método ChampionWinners() devuelve la lista de ganadores del campeonato según el criterio del juez.

Clase Utils:

Esta clase tiene un solo metodo genérico y es el encargado de devolver una lista de tipos de las clases que hereden de una clase o implementen una interfaz. Esta clase es usada mucho en la clase ChampionCreate para crear el torneo mediante reflection.

Clase CreateChampionship:

Esta clase es la encargada de crear el campeonato teniendo en cuenta las elecciones del usuario.

Como propiedad tiene un observador que es el encargado de escribir en la interfaz visual y de recibir la información del usuario.

El método ChooseInt() devuelve un int en el rango dado por el usuario.

El método CreateAChampion() devuelve una instancia de Championship

Luego se le pide al usuario que elija las características de su campeonato tales como: tipo de juez a nivel de torneo, criterio de parada del campeonato, crirerio para ganar un campeonato, criterio para ser descalificado de un campeonato, criterio de parada de un juego, como se le calcula el score a una ficha, que tipo de jugada es valida, que tipo de juez se desea tener a nivel de juego, con que tipo de ficha se desea jugar, el tipo de jugador, la estrategia de dicho jugador.

Todas las elecciones se realizan mediante el método GetChoice() y el uso de reflection.

Todas las clases deben tener un campo estático llamado Description, este contendrá la descripción de la clase para que el usuario sepa en que consiste dicha clase.

El método GetChoice() recibe como parametro un array de Type, que contiene los tipos de las opciones que tiene el usuario. Se crea un array de string que contiene las descripciones de las opciones, y se le da a elegir al usuario cual opción desea. Este método devuelve un entero que representa el índice de la opción elegida; y mediante reflection se crea la instancia de la opción elegida por el usuario.

Clase CalculateChampionScore

Esta clase tiene como objetivo calcular el score de un jugador en el campeonato. Tiene como propiedades: diccionario players que establecerá una relación entre los id de los jugadores y una lista de sus puntuaciones por juego; un diccionario score que guardará la relación entre el id del jugador y su puntuación en el torneo; una lista de enteros playersId que guardará los id de los jugadores.

En su método constructor recibe como parámetro la lista de ids, la cual guarda en playersId y luego invoca al método Run().

El método Run() se encarga de inicializar los diccionarios players y scores con cada uno de los playersId.

El metodo AddPlayerScore() se encarga de añadir al diccionario players el score de un jugador en un juego. Si el id del jugador no aparece en la lista, se agregará al diccionario con un total de 0 puntos. Luego se llama al metodo CalculateScore().

El método CalculateScore() es el encargado de actualizar el diccionario Scores con la puntuación total del jugador en el torneo.

El método GetScores() devuelve el score correspondiente al id del jugador.

El método GetPlayerScore() devuelve una lista de los scores correspondientes al id del jugador por cada juego.

El método LessPlayerScore() es el encargado de disminuir el score del jugador seleccionado.

Relacionadas con el juego:

Clase Board.

Esta clase implementa la interfaz IBoard.

La responsabilidad de la clase es tener todo lo relacionado con el tablero.

Como propiedades tiene: una lista de fichas llamada temp,en la cual guardará las fichas del tablero ; una lista pública llamada board que devolverá una clonación de la lista temp; un IToken llamado First que devolverá la primera ficha del tablero; un IToken llamado Last que devolverá la última ficha del tablero. En su método constructor se encarga de crear la lista temp.

El método GetTokens() es el encargado de devolver la lista temp clonada.

El método AddTokenToBoard recibe dos parámetros, uno es la ficha a jugar y el otro es un entero que indica por cual lado se jugará; este método funciona de la siguiente manera: si la lista temp no tiene fichas aún significa que el tablero está vacío, entonces la ficha recibida como parámetro se añade a la lista temp, de lo contrario se revisa el entero recibido como parametro y la ficha se añade o se inserta a la lista temp.

Esta clase cuenta con dos métodos de clonación, uno en el que recibe una lista de fichas y otro en el que no. La clonación que no recibe parámetros simplemente devuelve una nueva instancia de Board. La clonación que recibe una lista de ficha como parámetros devuelve la misma instancia de Board pero con la lista temp igualada a la lista recibida como parámetro.

Clase ChooseSideWrapped:

Esta clase implementa la interfaz IChooseSideWrapped. Su función es decidir dónde se puede jugar una ficha o no.

Tiene como propiedades: un entero index, este entero contiene el índice por el cual se jugará la ficha; un bool ChaChoose que será true si el jugador tiene opciones para elegir donde colocar la ficha; una lista de enteros WhereCanMatch que contendrá los índices por donde se puede jugar la ficha.

El método AddSide() recibe un entero de parámetro, y su función es añadir ese entero a la lista WhereCanMatch.

El método Run() verifica si el jugador tiene opciones para elegir donde colocar la ficha o no.

Clase ChooseStrategyWrapped:

Esta clase implementa la interfaz IChooseStrategyWrapped . Su función es seleccionar la posición de una ficha en el tablero.

Tiene como propiedades: un bool CanMatch que representa si puede poner la ficha en el tablero o no; un bool FirstPlay que indica si el la primera ficha en el tablero o no; un IBoard que representa el tablero; un IToken que representa la ficha que se quiere jugar; una lista de ChooseSideWrapped llamada side que contendrá todas las posibles posiciones en las que se puede jugar.

El método ControlSide() recibe un entero de parametro que representa el indice por el que el jugador desea jugar la ficha. En este método se controla que el lugar a poner la ficha sea válido, y si lo es se devuelve un ChooseSideWrapped con ese índice.

El método AddSide() recibe un ChooseSideWrapped como parámetro, y su función es agregar ese parámetro a la lista side . Además, si la variable CanChoose de este ChooseSideWrapped es true, entonces a la variable CanMatch se le asigna el valor de true.

Clase Game:

Esta clase implementa la interfaz IGame y representa una partida de dominó.

Tiene como propiedades: un evento llamado GameStatus que es el encargado de las acciones del juego; el evento CanContinue el cual es el encargado de determinar si se puede seguir jugando o no; un IBoard llamado board el cual representa el tablero actual del juego; una lista de Iplayers llamada GamePlayers que contendrá los jugadores de la partida pero como clonación; una lista protegida de Iplayer llamada players que contendrá las instancias originales de los jugadores de la partida; un entero MaxDoble que representa el doble máximo a jugar; un IjudgeGame el cual será el juez a nivel de partida; una lista de GamePlayerHand que va a contener las manos de los jugadores; una lista de PlayerStats que va a contener las estadísticas de los jugadores; un TokensManager que va a ser el administrador de las fichas.

El método GetPlayersList() es el encargado de devolver la lista GamePlayers como clonación de la lista players.

El método PlayerScores() devuelve la respuesta del método PlayerScores() dentro del juez, dado que el juez es el encargado de saber cómo se calculan las puntuaciones.

El método Winner() devuelve la respuesta del método Winner() dentro del juez, dado que es el juez del juego el encargado de saber cuál es la condición para ganar.

El método AssignTokens() es el encargado de repartir las fichas. Por cada uno de los jugadores del juego, se le asigna una lista de Itokens la cual la decide el método GetTokens() del TokensManager.

El método MatchHandAndPlayer() devuelve una lista de tupla donde se enlaza el id del jugador y su mano de fichas.

El método PullAPlayerAndHand() recibe un jugador como parámetro y devuelve la mano de ese jugador.

El método PlayAGame() es el encargado de jugar una partida. Lo primero que se hace es repartir las fichas. Luego mientras el juez determine que el juego no ha acabado se hace lo siguiente: se crea una variable de tipo IwatchPlayer, se le pide al jugador la ficha que va a jugar mediante el método Turno(), se crea una variable de tipo IChooseStrategyWrapped para determinar por que lado del tablero se debe poner la ficha, si el jugador lleva entonces se añade la ficha al tablero. Luego espera la confirmación del evento CanContinue para saber si se puede continuar jugando. Una vez el juez haya determinado que el juego finalizó, se llama al método GameEndStatus().

El método PlayerScore() devuelve una variable de tipo double que contiene la puntuación del jugador en esta partida. El double que devuelve es la respuesta del método PlayerScore() del juez.

El método EndGameStatus() es el encargado de enviar toda la información de la partida una vez esta finaliza. Dentro de este método se actualizan las puntuaciones de los jugadores y se crea la lista con los ganadores de a partida llamando al método Winner().

El método Turno() es el encargado de devolver la ficha que el jugador va a jugar llamando al método BestPlay() del jugador.

Relacionados con el juez:

Clase Judge:

Esta clase implementa la interfaz IjudgeGame. Es el encargado de manejar el juego y verificar que todo sea correcto. Es la clase que tiene toda la información sobre las características del juego.

Tiene como propiedades: un IstopGame que representa la condición de parada del juego, un IgetScore que representa como calcular la puntuación de una ficha, un IwinCondition que representa la condición de ganada, un IvalidPlay que representa que tipo de jugada es válida en el juego, un IPlayer que representa el jugador del cual es el turno actual, una lista de IChooseStrategyWrapped que va a representar todas las posibles posiciones por donde jugar la ficha, un diccionario playerscores que va a relacionar el índice del jugador con su puntuación, un CalculatePlayerScore que va a representar como se calcula la puntuación de un jugador.

El método RunWatchPlayer() es el encargado de crear una instancia de WatchPlayer .

El método PlayerMeetsStopCriteria() devuelve true o false si el jugador cumple con la condición de parada. Esto lo hace llamando al método de igual nombre dentro de la variable de tipo IstopGame.

El método ValidPlay() devuelve un IChooseStrategyWrapped que contiene la ficha y por dónde es posible jugarla.

El método AddPlayerScore() es el encargado de añadir al jugador recibido como parámetro al diccionario que contiene su puntuación.

El método EndGame() es un bool que va a determinar si el juego debe parar o no. Primero revisa si algún jugador cumple con la condición de parada, de ser así devuelve true y termina el juego. Luego revisa que el juego no este trancado verificando que algún jugador pueda jugar alguna ficha.

El método PlayerScore() devuelve el IPlayerScore de un jugador.

El método PlayerScores() devuelve un listado de los IplayerScores de todos los jugadores del juego.

El método CheckHand() comprueba si la mano del jugador contiene la ficha que el jugador desea jugar.

El método AddTokenToBoard() es el encargado de jugar una ficha y añadirla al tablero.Si se determina que el jugador está haciendo trampa entonces se penaliza quitándole puntos. Si se desea jugar al principio del tablero, se inserta en el tablero, y si se desea jugar al final se añade a la lista.

Los métodos PlayFront() y PlayBack() primero revisan si se necesita virar la ficha y luego llaman al método AddTokenToBoard().

El método AddPlayerScoreStatus() es el encargado de añadir el valor de la ficha al score del jugador en cada jugada.

El método Winner() devuelve una lista de los ganadores del partido. Esto lo hace llamando al método Winner() de su IwinCondition.

El método ItsThePlayerHand() comprueba que una lista de fichas recibida es la mano del jugador en específico.

Clase CorruptionJudge:

Esta clase hereda de Judge e implementa la interfaz IjudgeGame.

El método MakeCorruption() devuelve true si el juez decide ser corrupto y falso si no. Tiene un 50% de probabilidad de cometer corrupción.

El método LessPlayerScore() tiene el objetivo de disminuir el score de un jugador si este comete fraude.

El método AddTokenToBoard() cambia un poco. Primero se comprueba si el jugador es corrupto y si la ficha que va a jugar es válida. Si se cumple que MakeCorruption() devuelve true, el jugador es corrupto y la ficha a jugar no es válida entonces: se le hace la oferta al jugador de disminuir su score en un tercio, si este acepta entonces se juega la ficha elegida. Si no se cumple entonces se llama al método base.

Clase ChampionJudge:

Esta clase implementa IchampionJudge y es el encargado de saber todas las características de un torneo.

Tiene como propiedades: un IstopGame que representa la condición de parada del torneo, un IwinCondition que representa la condición necesaria para ganar el torneo, un IvalidPlay que representará si un jugador puede jugar un juego, un IgetScore que representará cómo calcular el score de un jugador en el torneo, una variable de tipo CalculateChampionScore que será la encargada de calcular y guardar los

scores del torneo, una lista de juegos llamada FinishedGames que contendrá todos los juegos ya completados y una lista de enteros playersId que contendrá los id de los jugadores del torneo.

El método Run() se encarga de inicializar la lista de ids y crear el CalculateChampionScore.

El método EndGame() representa si el torneo cumple con la condición de parada. Esto lo hace llamando al método MeetsCriteria del IstopGame.

El método AddFinishGame() se encarga de agregar al listado de juegos completados el juego que reciba como parámetro y de actualizar las puntuaciones de los jugadores.

El método ValidPlay() decide si un jugador puede continuar jugando. Esto lo hace retornando el resultado del método ValidPlay que se encuentra en la implementación de la interfaz IvalidPlay.

El método Winners() devuelve la lista de ganadores del torneo. Esto lo hace retornando la respuesta del método Winner encontrado en la implementación de la interfaz IwinCondition.

Clase CorruptionChampionJudge:

Esta clase hereda de ChampionJudge e implementa IchampionJudge.

El método MakeCorruption() devuelve true si e juez decide ser corrupto y falso si no. Tiene un 50% de probabilidad de cometer corrupcion.

El método EndGame(): si el juez decide ser corrupto entonces devuelve lo contrario al método EndGame() de la base, si no es corrupto devuelve el mismo resultado que el método base.

El método ValidPlay(): si el juez decide ser corrupto, el jugador es de tipo corrupto y el jugador es descalificado del torneo; el juez le hace una oferta de disminuir su puntuación a la mitad para poder seguir jugando. Si no, se llama al método base de ChampionJudge.

Relacionados con la interfaz gráfica:

Clase Observer:

Esta clase es la encargada de imprimir en la interfaz visual y de recibir el input del usuario.

Tiene una propiedad que es un int Speed que contendrá la velocidad con la que se imprimirá todo en pantalla.

En su método IntResponses() recibe: un string type que va a contener la pregunta que se le hará al usuario y un array de string con las opciones que el usuario puede elegir. Si las opciones es null, entonces el usuario debe elegir un número y se llama al método GetIntNumber(). Si no, se le muestran las opciones en la interfaz gráfica y el usuario escoge su opcion.

El método GetIntNumber() se encarga de devolver un entero que el usuario elija.

El método BoolResponses() se encarga de devolver true o false. Esto lo hace auxiliándose del método IntResponses().

El método SettingSpeed() se encarga de devolver la cantidad de milisegundos que se va a demorar en mostrar todo en pantalla y esto lo hace auxiliándose de IntResponses().

El método PrintStart() muestra en pantalla un pequeño mensaje de bienvenida.

El método PrintChampionStatus() recibe un ChampionStatus, y en dependencia de ese ChampionStatus es lo que se escribirá en pantalla. Si es un juego que está en desarrollo se llama al método PrintGameChange(); si es un juego que ha terminado se llama al método PrintFinishGame() y si es que el torneo terminó se llama al método PrintFinishChampion().

El método PrintGameChange() se encarga de escribir en la interfaz gráfica la mano de los jugadores y del tablero.

El método PrintFinishGame() se encarga de escribir en la interfaz gráfica como ha terminado el tablero, cada jugador con su puntuación y los ganadores de la partida.

El método PrintFinishChampion() se encarga de escribir en pantalla los ganadores del torneo.

Clase ChampionStatus:

Esta clase es la que contiene el estado actual del campeonato y por la cual se guía el observer para saber que mostrar en pantalla. Esta clase es la que sabe si el torneo acabó, o si hay un juego en proceso, o si se terminó una partida.

Clase GameStatus:

Esta es la clase que conoce el estado del juego. El observer usa esta clase para saber que imprimir en la pantalla.

El método SetActualPlayer() devuelve el jugador actual.

El método SetActualPlayerHand() devuelve la mano del jugador actual.

Relacionadas con el jugador.

Implementaciones de IPlayer

Clase Player:

Esta clase implementa la interfaz Iplayer.

Tiene como propiedades: una lista de fichas que representa su mano; un entero id que será un id unico asociado al jugador; una IplayerStrategy que representa la estrategia del jugador en cierta jugada y una lista de IplayerStrategy que contiene todas las estrategias elegidas para un jugador.

El método AddStrategy() recibe una estrategia como parámetro y la agrega a la lista de estrategias.

El método AddHand() recibe una lista de Tokens y la iguala a su mano.

El método PossiblePlays() devuelve una lista de todas las fichas que se pueden jugar en dependencia de lo que el juez determine sea válido.

El método BestPlay() devuelve la ficha a jugar. Este método se encarga de evaluar todas las fichas devueltas por el método PossiblePlays() y se queda con la ficha mejor evaluada. Esta evaluación la hace en dependencia de su estrategia.

El método ChooseStrategy() devuelve una estrategia random de la lista de estrategias.

El método ChooseSide() retorna la respuesta del ChooseSide() de la estrategia.

Clase CorruptionPlayer:

Esta clase hereda de la clase Player e implementa la interfaz Iplayer y la interfaz ICorruptible.

El método BestPlay() devuelve la ficha mejor evaluada de toda su mano, sin importar que sea válida o no.

El método Corrupt() devuelve un bool y este método es el encargado de decidir si aceptar una oferta o no de parte del juez.

Clase GamePlayerHand:

Su funcionalidad es tener toda la información relacionada con las fichas de un jugador.

Tiene como propiedades: PlayerID (el cual contiene el id del jugador al cual le pertenece la instancia de la clase), el bool Iplayed (devuelve true si jugó y falso si no), un hashset handprivate que contendrá la mano del jugador, un stack de fichas llamado FichasJugadas que contendrá en orden las fichas jugadas por el jugador.

El método ContainsToken() tiene la función de devolver true si el jugador tiene la ficha en su mano y falso si no.

El método AddLastPlay() se encarga de agregar la ficha jugada al stack FichasJugadas y la remueve de la mano del jugador.

Clase PlayerScore.

Esta clase implementa la interfaz IPlayerScore y es la encargada de aumentar o disminuir la puntuación de un jugador. Sus propiedades son PlayerID y Score. En su constructor recibe un entero correspondiente al ID del jugador al que le corresponde el PlayerScore. El metodo de clonación devuelve una nueva instancia de PlayerScore con el mismo id y el mismo score que la instancia actual.

Clase CalculatePlayerScore

Esta clase es la encargada de calcular el score del jugador. Esta clase cuenta con un único metodo que recibe como parámetros un IPlayerScore que representa el score del jugador, un GamePlayerHand que representa la información respecto a la mano del jugador, un double que son los puntos que se van a añadir o restar de la puntuación del jugador y un bool que será true si el valor se score será añadido y falso si será restado. En nuestra implementación del método, el score recibido será multiplicado por 1.2 si el jugador no ha jugado ninguna ficha; sino, se mantendrá el mismo score.

Clase PlayersCoach:

Esta clase es la encargada de administrar los jugadores.

Como propiedades tiene: una lista de jugadores AllPlayers que contiene todos los jugadores a nivel de torneo, una lista que contiene listas de jugadores donde cada elemento de la lista son los jugadores que participaron en un juego en común, una lista que contiene lista de jugadores llamada LastPlayerPlays que va a contener la lista de los players que han jugado los juegos ya terminados.

En su método AddPlayers() recibe una lista de jugadores y añade esta lista a su propiedad players.

En el método GetNextPlayers() devuelve la lista de jugadores que van a jugar en el siguiente juego.

Clase PlayerStats:

Esta clase se ocupa de guardar toda la información respecto a la puntuación de un jugador.

Tiene como propiedades: un Iplayer que es al jugador que le corresponde la instancia de PlayerStats y un double nombrado puntuación que va a contener la puntuación del jugador la cual es inicializada en -1.

El método AddPunctuation() funciona de la siguiente manera: recibe una puntuación de parámetro y si la propiedad es menor que 0 (inicializada en -1) entonces la propiedad de puntuación es igualada a la puntuación recibida en el parámetro.

Clase WatchPlayer:

Esta clase es la que recibe un jugador para conocer las características de juego sin tener acceso al juez.

Esta clase tiene como propiedades un IgetScore, un IstopGame, un IvalidPlay, un IwinCondition y un IBoard.

Implementaciones de IplayerStrategy:

Clase RandomStrategy:

Juega una ficha random y elige un lado para jugar la ficha random

Clase BGStrategy:

Juega la ficha con más puntos y elige el lado para jugar la ficha tal que se quede en la mesa la parte de la ficha con menor puntuación.

Clase SemiSmart:

Juega una ficha teniendo en cuenta aspectos como si es doble, o si tiene mas fichas de ese cierto numero; y elige el lado para jugar la ficha tal que se quede en la mesa la parte de la ficha con menor puntuación.

Relacionados con las fichas:

Implementaciones de Itokenizable:

Clase NormalInt:

Esta clase implementa la interfaz Itokenizable y representa los enteros para poderse usar como parte de las fichas de dominó.

Clase EnergyGenerator:

Esta clase implementa la interfaz Itokenizable.

Como esta clase funciona es que cada instancia tiene un minPotencia y un maxPotencia; y el componentValue (que es lo importante en un Itokenizable) se calcula como un número random entre el minPotencia y el maxPotencia de la instancia multiplicado por 0, -1 o 1.

Implementaciones de IToken:

Clase Token:

Esta clase implementa la interfaz IToken y representa una ficha.

Tiene como propiedades: un Itokenizable Part1 y un Itokenizable Part2 que representan las dos partes de una ficha. Esto significa que cualquier objeto puede ser parte de una ficha mientras implemente la interfaz Itokenizable.

El método SwapToken se encarga de "virar" una ficha intercambiando sus partes.

Implementaciones de Igenerator:

Clase Fichas Enteros:

Esta clase implementa Igenerator.

Es la clase encargada de generar las fichas de un domino común.

Clase Fichas Termoelectricas:

Esta clase implementa Igenerator.

Es la clase encargada de generar las fichas correspondientes a la termoeléctrica. Funciona de tal manera que los Itokenizable que componen una ficha son generados de forma aleatoria.

Clase TokensManager:

Esta clase implementa la interfaz ItokensManager, y su función es repartir las fichas de forma random.

Implementaciones de IWinCondition:

Clase Abstracta WinCondition:

Esta clase implementa la interfaz IwinCondition.

En su método Winner devuleve una lista de jugadores que contiene los ganadores del juego. Primeramente crea una array de enteros que contendrá los scores de los jugadores. Luego le pregunta al método FinalWinner por la lista de ganadores del juego.

El método FinalWinner es el método que van a implementar las clases que hereden de WinCondition.

Clase MinScore:

Hereda de WinCondition.

Selecciona como ganadores a los jugadores con el menor score.

Clase MaxScore:

Hereda de WinCondition.

Selecciona como ganadores a los jugadores con el mayor score.

Clase MiddleScore:

Hereda de WinCondition.

Selecciona como ganadores al jugador cuyo score caiga en el medio.

Clase WinChampion:

Esta clase implementa IwinCondition.

Es la encargada de determinar los ganadores del torneo.

Tiene como propiedades: un double Percent que representa el porciento de juegos ganados que debe tener un jugador para ser ganador; una lista de jugadores; un IgetScore para conocer como calcular el score del torneo; un diccionario que le asignará una lista de scores a cada jugador en el campeonato.

El método Run() se encarga de agregar al diccionario todos los scores de los jugadores.

El método Winner() se encarga de devolver en una lista los ganadores del campeonato. Esto lo hace de la siguiente manera: identifica los ganadores de cada juego y se le va agregando un punto a cada jugador. Luego se ordenan los jugadores en correspondencia al Wplayer_Comparer. Luego se revisa si el score del jugador es mayor que 0, y si es así se considera ganador del campeonato y se agrega a la lista que será devuelta.

Implementaciones de IValidPlay:

Clase Abstracta ValidPlayClass:

Esta clase implementa la interfaz IvalidPlay. Su función es determinar si una jugada es válida.

El método ValidPlay devuelve un objeto de tipo IChooseStrategyWrapped que va a contener si la ficha puede ser jugada por la parte delantera del tablera o la parte trasera del tablero, o ambas.

Los métodos ValidPlayFront() y ValidPlayBack() devuelven un objeto de tipo ChooseSideWrapped que va a contener la ficha en la posición que puede ser jugada.

El método abstracto Match() es el que las clases que hereden de esta clase deberan implementar.

El método FirstPlay() devuelve true si el tablero esta vacío y falso si no.

Clase ClassicPlay:

Hereda de ValidPlayClass.

Una jugada es válida si las partes de las fichas a enlazar son iguales

Clase BiggerValidPlay:

Hereda de ValidPlayClass.

Una jugada es válida si las partes de las ficha a jugar es mayor que la parte de la ficha en la mesa.

Clase SmallerValidPlay:

Hereda de ValidPlayClass.

Una jugada es válida si las partes de las ficha a jugar es menor que la parte de la ficha en la mesa.

Clase ValidChampionPorcientPerdidas:

Implementa IvalidPlay.

Su función es decidir si un jugador es descalificado o no en dependencia de la cantidad de juegos que ha perdido.

Clase ValidChampionPerdidasConsecutivas:

Implementa IvalidPlay.

Su función es decidir si un jugador es descalificado o no en dependencia de la cantidad de juegos que ha perdido de forma consecutiva.

Implementaciones de IstopGame:

Clase Classic:

Si el jugador se queda sin fichas

Clase CertainScore:

Si el score del jugador en el momento es igual a un score específico.

Clase StopChampionPerPoints:

El torneo acaba cuando un jugador acumule cierta cantidad de puntos.

Clase StopChampionPerHaveAWinner:

El torneo acaba cuando exista un ganador

Implementaciones de IgetScore:

Clase ClassicScore:

El valor de la ficha es la suma de sus partes

Clase Double:

Si la ficha es doble, su valor es el doble

Clase ScoreChampionNormal:

Esta implementación es usada para calcular el score en el torneo. La forma en que lo hace es: recorre cada una de las puntuaciones del jugador por juego, si es una puntuación positiva entonces esta se añade al score total, si es negativa entonces se aumenta un contador; luego si hubo alguna puntuación negativa, la puntuación total se divide entre la cantidad de veces que hubo puntos negativos y esa es la puntuacion del jugador en el torneo.