Para iniciar la explicación vemos conveniente hacer referencia a las notaciones utilizadas en el proyecto y en nuestro léxico para mayor comprensión del lector. Cuando se hace referencia al término “partida” o “Play” (este último es en el código) se refiere al evento de repartir las fichas a los jugadores, ejecutar jugadas (jugar o pasar) en consecuencia del orden seleccionado y cuando se cumple una condición de finalización, se decide un ganador y con eso concluye la partida; por otra parte la referencia “torneo” o “Game” (este último es en el código) se refiere a un conjunto de partidas en las que los ganadores de cada una van acumulando puntos hasta llegar a una puntuación necesaria para completar el torneo y ganarlo. En nuestro diseño la diferencia entre ambas no está dada por una variación en el diseño visual para cambiar de una a otra, pero se permite modificar la cantidad de puntos a alcanzar por un jugador para ganar un torneo, por tanto una partida puede ser considerada como un torneo a 1 punto.

La clase central encargada de mover los hilos y dar órdenes en el proyecto es la clase Game<T, P>, donde T es el tipo de dato que será utilizado para otorgarle a la ficha, en caso de que dicho tipo T no tenga noción de valor (considerando que tienen valor el tipo int por ejemplo) las fichas no tendrían manera de compararse y saber que tan mayor o menor es una con respecto a otra, por esto se creó la clase Values<T> que cumple función de traductor (a la clase Values se le hará referencia como traductor en lo adelante del reporte, para más claridad en las explicaciones) a valores int, por ahora en el proyecto están implementadas la traducción de int a int (necesaria para mantener la estructura del constructor de las Fichas) y de char a int (considerando que los únicos elementos con traducción por el momento son las letras en mayúscula y cuyo valor depende del orden alfabético); el tipo P hace referencia a la estructura en la que serán almacenadas las fichas, tanto las que no sean repartidas a los jugadores (en caso de que no todas sean repartidas claro) como las que los jugadores decidan jugar en cada turno. Los tipos T y P mantienen su notación a lo largo del proyecto, o sea, siempre que se use la el tipo T se refiere al tipo de dato que contiene la ficha y de igual manera con el tipo P.

La clase contiene como propiedades:

```
8 references
public class Game<T, P>
{
    15 references
    public ITable<T, P> Table {get; }
    //Estructura en la que se almacenan las fichas
    20 references
    public IPlayer<T>[] Players {get; set; }
    //Array con los jugadores
    11 references
    public Values<T> Translator {get; set; }
    //Traductor de valores que puede tomar la ficha
    5 references
    public IOrder<T> Order {get; set; }
    //Generador de Orden
    5 references
    public IFinishGame<T> FinishGame {get; set; }
    //Caso de finalizacion de un juego y adicion de puntos
    4 references
    public IFinishPlay<T> FinishPlay {get; set; }
    //Caso de finalizacion de una partida
    4 references
    public IWinnerPlay<T> WinnerPlay {get; set; }
    //Define el ganador de la partida
    4 references
    public ISorter<T> Sorter {get; set; }
    //Repartidor de fichas a los jugadores
    3 references
    public ISorterPlayer<T> SorterPlayer {get; set; }
    //Repartidor de los jugadores en Players
    17 references
    public int Index {get; set; }
    //Indice del jugador que debe jugar
    5 references
    public int Passes {get; set; }
    //Cantidad de pases consecutivos
```

Cuando la interfaz gráfica recibe los datos que escoge el usuario, esta los envía al método Create de la clase MainWindow, el cual crea un Game con los valores escogidos de T y P, acto seguido le da valores al traductor y entra al método Form dentro del Game creado para darle valores a cada una de las interfaces que se muestran en las propiedades (Players, Order, FinishGame, FinishPlay, WinnerPlay, Sorter, SorterPlayer). Ya teniendo conformada la estructura absoluta de Game, se ejecuta el método Run de Game que sigue la idea siguiente:

-Si no hay fichas en ninguno de los elementos que deberían contenerlas, entonces debo crearlas.

-Si no hay fichas en la mesa (mesa se refiere a la estructura dentro de ITable que almacena las fichas jugadas por los jugadores) es porque nadie ha jugado, entonces debo repartir las fichas y llamar al método Move de Game, el cual sabiendo que está en el inicio de la partida le pide al jugador indicado una ficha.
-Si la mesa tiene fichas entonces llama al método Move, el cual le muestra la mesa al jugador para que este revise si puede jugar o no y en caso de que si, lo haga.
-Luego de realizada la jugada del turno revisa si se ganó la partida y en caso de que si, revisa si se ganó el torneo y restablece los valores iniciales para una nueva partida.

En otras palabras, el método Run cuando se ejecuta realiza el recorrido de una jugada, creando las condiciones pertinentes para esta en caso de no existir y revisando si con dicha jugada la partida finalizó. Por tanto cada partida y torneo se pueden representar como llamados al método Run con las propiedades de Game.

Siendo esta la idea de nuestro juego de Domino procederemos a explicar las variaciones:

-Elemento que contienen las Fichas:
La clase Values recibe un array de T, el cual se crea con métodos que hay en la clase MainWindow, además recibe un diccionario <T, int> que añade en Key los valores del array mencionado y le da valores ascendentes desde el 0, dicho de otra forma, a cada elemento del array de T se le hace corresponder su índice como valor. De esta manera para generar un nuevo tipo de dato para las fichas solo hay que crear un método que cree el array de ese tipo, en el orden que se le quiera dar los valores.

-Tipo de estructura:
La estructura utilizada muestra su diferencia en la interfaz gráfica; actualmente existen dos implementaciones, una con forma matricial para fichas mostrables organizadamente en la interfaz con forma matricial, para ver el domino de una manera más semejante a la real; la otra estructura contiene los elementos en una List<Ficha<T>> para tipo T más complicados de mostrar, como el caso de crear una clase nueva sin método ToString sobrescrito, cuando se usa esta estructura la interfaz gráfica solo muestra el historial de jugadas.

-Orden de los jugadores:

Se permite variar el orden de los jugadores, hay tres implementaciones actualmente, una con el orden usual siguiendo los índices de los jugadores en Players; otra en la que siempre juega el jugador con más puntos en su mano, en caso de que ese jugador pase, se revisa en el resto de jugadores quien es el que más puntos tiene, algoritmo que se repite si se repite la situación; la última implementación es en la que si un jugador pasa el sentido en que se juega cambia y los índices de Players empiezan a recorrerse de manera descendiente, variando entre ascendente y descendiente con cada pase.

-Adicionador de puntos:

El método encargado de adicionar puntos está en FinishGame, en principio pensábamos variar dentro de esta interface en jugar por puntos o por partidas, pero como la variación en el código hubiera sido mínima decidimos no hacerlo; lo que si se varió fue la adición de puntos, la cual puede ser clásica, o sea, aumentando los puntos del ganador en la suma de los puntos en la mano de cada perdedor; también puede ser por fichas, donde al ganador se le adiciona la cantidad de fichas en la mano de cada uno de los perdedores multiplicado por 5, en caso de que entre todos los perdedores no haya ninguna ficha, se le agregaran 5 puntos al ganador; y la implementación por pases, donde el ganador recibe 5 puntos por cada jugador que pasó después de que él jugara, también recibe 5 puntos solo por haber ganado la partida, esto porque podía pasar que ningún jugador se pasara y entonces el ganador no obtendría puntos, lo cual se veía injusto.

-Condiciones de terminar una partida:

Entre las variaciones de terminar una partida se encuentra la clásica, en la que la partida se acaba si la cantidad de pases es igual a la de jugadores o si algún jugador se quedó sin fichas; otra en la que se termina el juego si un jugador cualquiera pasa 3 veces seguidas o si algún jugador se queda sin fichas; y una última en la que el juego se acaba si la cantidad de pases es igual a la de jugadores o si algún jugador tiene menos de 10 puntos en su mano.

-Condiciones de ganar una partida:
Se puede ganar una partida de manera clásica, o sea, si hay algún jugador sin fichas es el ganador, sino gana el que tenga menos puntos; en otra variante gana el jugador con menos fichas sin importar los puntos; y en una tercera opción se revisa cuál es el jugador que más veces ha jugado antes de alguien que pasara y este es el ganador, sin importar si hay algún jugador sin fichas.

-Repartidor de fichas:
Teniendo las fichas creadas y almacenadas en una List<Ficha<T>>, el repartidor recibe dicha lista en que se almacenan, los jugadores para repartir las fichas y cuantas fichas debe repartir a cada jugador, en nuestro proyecto esto se puede hacer de 3 maneras, una en la que los jugadores reciben cualquier ficha de la lista hasta llenar la cantidad de fichas; otra en la que se reparte de igual manera pero luego los jugadores devuelven la mitad de esas fichas, las cuales son almacenadas en la lista y se reparten nuevamente al jugador hasta llenar la cantidad; y una última en la que todos los jugadores reciben 2 fichas en principio y el resto total de fichas a repartir (que sería cantidad de jugadores * (cantidad de fichas por jugador – 2)) se reparten a cualquier jugador sin pensar en cuantas fichas debería tener cada uno.

-Repartidor de jugador en Players:
Cuando el usuario inserta en la interfaz gráfica cuantos jugadores quiere de cada tipo, no puede decidir el orden exacto en que serán ubicados, pero tiene 2 maneras de escoger en nuestro sistema, una en la que los jugadores siguen el orden: Random, ThrowFat, Reserved, PassMaker, ejemplo:
Insertando los datos: 2, 1, 3, 1
Players: Random, Random, ThrowFat, Reserved, Reserved, Reserved, PassMaker
Sería la organización creada.
La otra manera recibe la misma información y crea Players de manera desorganizada, pues crea los jugadores de los tipos correctos pero los ubica en un índice cualquiera de Players.

Terminadas las variaciones, solo falta explicar la idea heurística tras la implementación de los jugadores, como ya se comentó hay 4 tipo diferentes de jugadores.

```
30 references
public interface IPlayer<T>
{
    53 references
    List<Ficha<T>> Hand {get; }
    //Para almacenar todas sus fichas
    14 references
    List<Ficha<T>> Possible {get; }
    //Luego de pasar por Search, aqui van las fichas que pueden ser jugadas en el turno del jugador
    17 references
    int[] Count {get; }
    //Array de tamaño igual a la cantidad de valores diferentes usados en la partida para heuristicas de algunos jugadores

    1 reference
    void Search(T left, T right);
    //Metodo para llenar Possible
    2 references
    Ficha<T> Play(Values<T> Translator);
    //Metodo para jugar sin saber los valores extremos de la mesa
    1 reference
    (Ficha<T>, T) PlayWT(T left, T right, Values<T> Translator);
    //Metodo para jugar sabiendo los valores extremos de la mesa
    1 reference
    void Load(Values<T> Translator);
    //Utilizado en heuristica con Count
    1 reference
    void Load(T left, T right, Values<T> Translator);
    //Utilizado en heuristica con Count
    1 reference
    void Clean();
    //Metodo para limpiar Count cuando se acaba la partida
    1 reference
    List<Ficha<T>> ReSort();
    //Metodo para devolver fichas en ReSorter
}
```

Random: Juega la ficha con índice 0 en Possible; no utiliza Load de ninguna manera particular; en ReSort devuelve la mitad de las fichas que le entregaron desde la posición 0 hasta la posición ubicada en el centro.

ThrowFat: Juega la ficha con mayor cantidad de puntos en Possible; no utiliza Load de ninguna manera particular; en ReSort devuelve la mitad de las fichas que le entregaron, estas fichas son las que tienen más puntos entre las que le repartieron.

Reserved: Jugador que siempre intenta tener fichas con todos los valores, utiliza Load (Translator) para tener contadores de cuantas fichas tiene con cada valor, lo realiza al recibir las fichas, luego revisa cual es la ficha de la que más repeticiones tiene en Possible y la juega, restando su repetición en Count; en ReSort devuelve la mitad de las fichas que le entregaron, devolviendo fichas ubicadas en posiciones cualquiera.

PassMaker: Jugador que siempre intenta jugar el valor que más veces ha pasado a los otros jugadores, utiliza Load (left, right, Translator) cada vez que algún jugador de la mesa se pasa para guardar dicha jugada, juega revisando si tiene fichas que hayan

pasado a los otros jugadores y devuelve la que más veces lo ha hecho, en caso de que nadie se haya pasado juega Random; en ReSort devuelve la mitad de las fichas que le entregaron, devolviendo fichas ubicadas en posiciones cualquiera.