# PROYECTO DE PROGRAMACIÓN II

## CIENCIAS DE LA COMPUTACIÓN

## 2do AÑO

## Curso 2022

Loitzel Ernesto Morales Santiesteban C-212
Fernando Valdés García C-212

# Clases básicas:

- **Token<T>**: Representa una ficha cuyas caras son de tipo T. Toda aparición de este parámetro de tipo en las clases y métodos definidos en este assembly hacen uso de este mismo tipo, el cual debe implementar la interface **IEvaluable**. Sobre esto se entrará en detalle más adelante.
  Esta clase almacena una colección de tipo T para representar los elementos en las caras de la ficha, al igual que una máscara booleana para saber cuales caras de la ficha se encuentran ocupadas (es decir, si esta ficha está conectada a otra en el tablero por una cara, dicha cara se marcará como ocupada). **Token** también contiene los getter **Outputs** y **FreeOutputs** para mostrar todas las caras de la ficha y solamente las disponibles respectivamente; y un método **HasOutput(T output)** que indica si la ficha tiene libre dicha cara. Finalmente el método **PlaceTokenOn(T output)** marca la cara introducida como ocupada. Si esta no existe o no está disponible, lanzará una excepción.
- **Board<T>**: Almacena las fichas colocadas en el tablero en una lista privada de Token. Para colocar una ficha nueva se utiliza el método **PlaceToken(Token<T> token)**. En caso de ser la primera ficha del juego, solo recibe la ficha y la inserta en la lista. De no ser así, también recibe un parámetro **T output** representando la cara por donde se deberá colocar esta ficha. Luego, recorre la lista de fichas colocadas hasta encontrar una que posea libre la misma cara que se pasó como parámetro, y entonces se marcan como ocupadas en ambas fichas una cara correspondiente a la introducida, a la vez que se agrega la nueva ficha a la lista. Este método lanza excepción en 3 casos: si se intenta llamar al **PlaceToken()** sin parámetro **output** más de una vez, si la ficha a colocar no tiene libre la cara enviada, y si no existe una ficha en el tablero con la cara **output** libre.
- **History<T>**: Esta clase almacena un historial de jugadas mediante una colección de **PlayData<T>**, donde se encapsula el nombre del jugador que movió, un arreglo conteniendo las caras que están disponibles en el tablero, la ficha jugada (null si se pasó), y la cara por donde se colocó la ficha (null si fue la primera ficha o si se pasó). La entrada de esta clase es el método **AddMove**, que recibe la información que se guardará en un nuevo **PlayData** que será agregado a la colección.
- **Player<T>**: Contiene el nombre, estrategia, y mano actual de un jugador. El getter **TokensInHand** devuelve un arreglo con una copia de las fichas en la mano (la copia es para evitar adulteración de las fichas desde la implementación de la estrategia). El método **AddToken(Token<T>)** agrega una ficha a la mano del jugador. Este es usado por el **GameManager** al repartir las fichas al inicio del juego, y por el **EffectsExecution** para hacer que el jugador robe fichas del pool de fichas sobrantes.
  El método más importante de esta clase es el méodo **Play()**, que hace que el jugador intente colocar una ficha. Recibe un arreglo con las caras disponibles en el tablero, un arreglo de **PlayData<T>** representando el historial de juego, y el **evaluator<T>** que se está usando en la partida. Este método requiere especial atención por temas de seguridad.

```
internal PlayData<T> Play(T[] BoardOutputs, PlayData<T>[] movesHistory, evaluator<T> evaluator)
    {
        Token<T>[] optionsToPlay = AvailableOptions(TokensInHand, BoardOutputs);

        if(optionsToPlay.Length == 0) return new PlayData<T>(this.Name, AvailableOutputs: BoardOutputs);

        var Move = playersStrategy(movesHistory, evaluator, optionsToPlay, BoardOutputs);
        if (!Move.Item1.HasOutput(Move.Item2)) {
            throw new ArgumentException("Wrong strategy token management. The returned token doesn't have
        }

        Token<T>? originalToken = null;
        foreach (var token in hand) {
            if (token.Equals(Move.Item1)) {

                hand.Remove(token);
                originalToken = token;
                break;
            }
        }
        return new PlayData<T>(this.Name, BoardOutputs, originalToken, Move.Item2);
    }
```

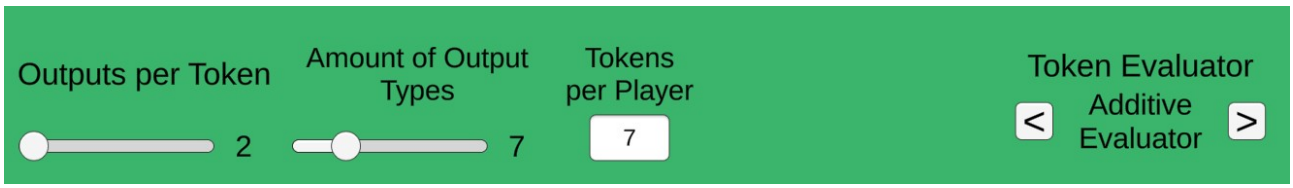
Primeramente se obtiene el arreglo **optionsToPlay**, el cual es una colección de las fichas en la mano del jugador que este puede colocar en el tablero. Si no puede colocar ninguna, el objeto **PlayData** devuelto tendrá la ficha y el output como null, señal de que el jugador se pasó. Luego se llama a la estrategia, la cual toma la decisión de qué ficha jugar y por cual cara, datos que se guardarán en **Move** en forma de tupla. Si la implementación de la estrategia estaba defectuosa y la cara que envió no se encontraba libre en la ficha enviada, entonces se lanzará una excepción. El siguiente paso será remover la ficha a jugar de la mano. Como a la estrategia se le pasó una copia de las fichas para evitar posibles adulteraciones, será necesario recorrer las fichas en **hand** (la colección con las fichas originales en la mano), y verificar cual ficha se corresponde a la enviada por la estrategia, para finalmente retornar los datos de la jugada.
**Nota:** el método **Equals**() de la clase **Token** fue sobreescrito de modo que dos fichas se considerarán iguales si tienen las mismas caras en iguales cantidades.

## Elementos configurables:

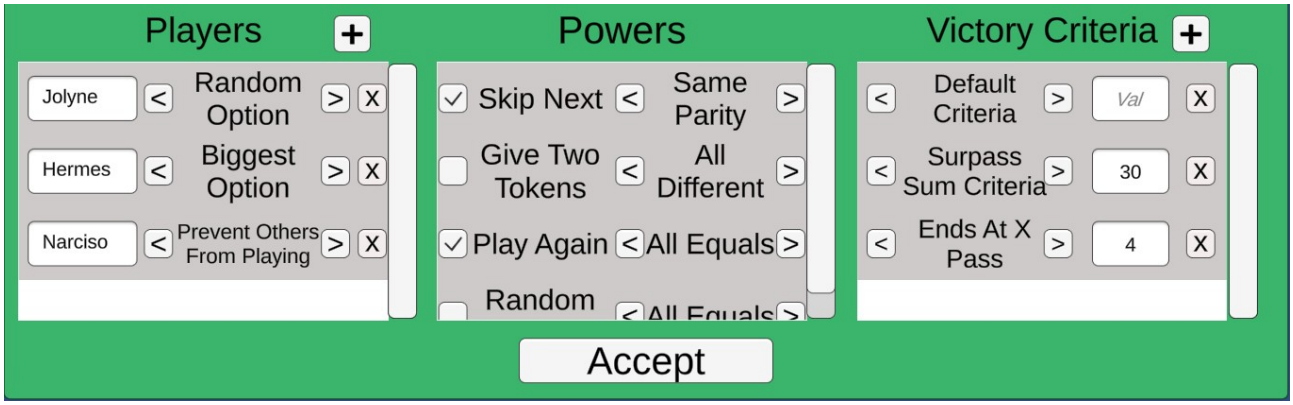
Existen varios conceptos del dominó que admiten múltiples variaciones en este proyecto. Es posible agregar nuevas variantes de algunos de estos conceptos, simplemente creando una nueva implementación del delegate o clase que corresponda al concepto deseado. Además, el usuario de la aplicación podrá seleccionar cualquier combinación de dichas implementaciones y simular una partida con esta. Los elementos configurables son los siguientes:

- **Tipo de caras de las fichas:** Representa el tipo de dato de las caras de las fichas para un juego. Este tipo debe implementar la interfaz **IEvaluable**, la cual exige implementar el método **int Value**(), que devolverá el valor entero de una cara. Tambien es necesario un método estático **Generate**(**int n**), que se encarga de generar una de cada posible cara del tipo escogido, hasta generar n caras distintas. Las implementaciones que existen actualmente son números (**Number**) y letras (**Letter**). Los valores de esta última se asignan según la versión hispana del juego de mesa Scrabble. **Letter** consta de control de excepciones, como evitar que se pasen caracteres no presentes en el diccionario para su construcción, o que se exijan mas tipos de ficha al método **Generate** que la cantidad de letras presentes en el diccionario.

```
private static readonly Dictionary<char, int> charValues = new Dictionary<char, int>{
    {'a', 1}, {'b', 3}, {'c', 3}, {'d', 2}, {'e', 1}, {'f', 4},
    {'g', 2}, {'h', 4}, {'i', 1}, {'j', 8}, {'k', 5}, {'l', 1},
    {'m', 3}, {'n', 1}, {'ñ', 20}, {'o', 1}, {'p', 3}, {'q', 10},
    {'r', 1}, {'s', 1}, {'t', 1}, {'u', 1}, {'v', 1}, {'w', 4},
    {'x', 8}, {'y', 4}, {'z', 10}
};
```




- **Caras por ficha:** La cantidad de caras que tendrá cada ficha en este juego.
- **Cantidad de tipos de caras:** Representa cuántas caras distintas existirán en el juego. Por ejemplo, para una variante de dominó con fichas desde el blanco (0) hasta el 6, el parámetro será 7, y para un juego del 0 al 9 será 10. Este valor es el **n** que se le pasa al método **Generate**() del tipo de cara escogido.
- **Fichas por jugador:** La cantidad de fichas que se repartirá a cada jugador al inicio del juego. Si la cantidad que se escogió no es suficiente, pues los parámetros anteriores no generan la cantidad de fichas necesaria, entonces el engine lanzará una excepción. Sin embargo la aplicación visual atrapa esta excepción y advierte al usuario que las fichas no son suficientes y debe reingresar el parámetro.
- **Evaluador de fichas:** Es la implementación de un **evaluator<T>** que se usará para calcular el valor total de una ficha. Actualmente existen dos implementaciones: **Additive Evaluator**, el cual se puede considerar el criterio estándar, ya que el valor total de una ficha será la suma de los valores de cada una de sus caras; y **Five Multiples Priority**, el cual le otorgará un valor mayor a la ficha si la suma de los valores de sus caras es un múltiplo de 5.




- **Jugadores:** En este panel es posible seleccionar la cantidad de jugadores que participarán, nombrarlos, y escoger las estrategias de cada uno. Si al enviar la información la cantidad de jugadores es 0, la aplicación advertirá al usuario que debe corregir la entrada.
- **Estrategias:** El campo junto al nombre de cada jugador permite elegir la estrategia de juego de este. Una estrategia es una implementación de un delegate **strategy<T>**, el cual recibe el historial de juego hasta el momento, el **evaluator** que

se esté usando, las fichas que el jugador puede colocar en este turno, y las caras disponibles en el tablero. Con esta información la estrategia tomará una decisión y devolverá una tupla con la ficha a jugar y la cara por donde deberá jugarse. Actualmente existen tres implementaciones de estrategias diferentes:

- **Random Option**: Escoge una ficha aleatoria entre las que el jugador tiene disponibles para colocar.
- **Biggest Option:** Utiliza el **evaluator** para escoger la ficha con mayor valor total entre las disponibles.
- **Prevent Others From Playing:** Primeramente, el método usa el **evaluator** para formar un arreglo de longitud igual a la cantidad de fichas que tenga el jugador disponibles, en el cual cada posición contiene el valor total de cada ficha. A dicho arreglo se le llamará la prioridad de cada ficha. Entonces, usando el historial, se buscan las salidas que hicieron que alguno de los jugadores se pasase. Luego se iterará por cada ficha disponible en la mano. Si alguna de estas hace que el tablero vuelva a tener una salida por la cual algún jugador se pasó, la prioridad de esta ficha se duplicará. Si por el contrario la ficha elimina una cara por la que alguien se pasó, no agrega nuevas caras por las que alguien se haya pasado, y no puede ser colocada de otra forma, su prioridad se reducirá a la mitad. Finalmente se elegirá la ficha con mayor prioridad para ser jugada. Además, cada vez que se le aumente la prioridad a una ficha, se incrementa también la prioridad de la cara por donde se colocaría. Una vez que se tenga la ficha a jugar, esta se enviará junto a la cara que contenga que sea de mayor prioridad.

- **Poderes:** Para entender este campo primero es necesario hablar de la clase **Power<T>** y el delegate **tokenFilter<T>**. Un filtro recibe una ficha con caras de tipo T, y devuelve true o false en dependencia de si la ficha cumple determinadas características. Los filtros existentes actualmente son: **Same Parity**, que indica si todos los valores de cada cara de la ficha tienen la misma paridad, **All Different**, que dice si todas las caras de la ficha son diferentes, y **All Equals** que indica si todas las caras son iguales.
  Ahora, la clase **Power** es una forma de encapsular un filtro y un delegate de tipo **Action<EffectsExecution<T>>**, también conocido como un efecto. Por otra parte, la clase **Powers<T>** contiene una colección de **Power**, y un método **GetEffects(Token<T> token)** que se llama desde el manager cada vez que se juega una ficha. Este método comprueba por cuales filtros pasa la ficha y devuelve un arreglo con los efectos correspondientes. Un efecto representa una acción que se realizará en el juego cuando una ficha con ciertas características es jugada. Cada efecto llama a algún método de la clase **EffectsExecution**, la cual actúa como intermediario entre los efectos y el manager. Los efectos implementados actualmente son:
  - **Skip Next:** Se saltará el turno del próximo jugador.
  - **Give Two Tokens:** Hace que el próximo jugador robe 2 fichas de las sobrantes.
  - **Play Again:** Permite al último jugador que movió jugar de nuevo.
  - **Random Turn:** El próximo turno será para un jugador aleatorio.
- **Criterios de victoria:** En este campo se seleccionan los criterios de finalización de juego que se deseen. Estos se definen a través de delegates del tipo **victoryCriteria<T>**, los cuales reciben el historial de juego, el evaluador, las manos de cada jugador, y algunos recibirán un valor que puede ser pasado en la aplicación como se ve en la imagen. Se pueden seleccionar varios criterios a la vez, y cuando uno, o varios en el mismo turno, cumplan con sus requisitos, estos devolverán los ganadores de la partida. Los criterios implementados actualmente son los siguientes:
  - **Default Criteria:** Es el criterio de finalización clásico. Cuando un jugador se queda sin fichas, este se considera ganador, y cuando no se pueden jugar más

fichas, el o los jugadores con la menor cantidad de valor en sus manos, serán considerados ganadores.
- **Surpass Sum Criteria:** Este criterio necesita recibir un número, el cual se puede pasar en el campo de entrada en la aplicación visual. Este criterio salta cuando un jugador haya colocado en la mesa una cantidad de valor mayor o igual que el número enviado, y se considerará ganador a este jugador. Si el juego se tranca y ningún jugador superó la suma, entonces se considerará que no hubo ganadores.
- **End At X Pass:** Este criterio también necesita un número. Se activará cuando se la cantidad de pases en el juego sea igual al número que se envió. Los ganadores serán aquellos que tengan la menor cantidad de valor en sus manos.

**Nota:** Cuando se agregue una nueva implementación de uno de los aspectos variables, es necesario ir a la clase **Implementations**, buscar el método que se corresponda al aspecto deseado, y, en el arreglo que este retorna, adicionar una instancia de una tupla de string y el tipo de dato de la implementación, con el nombre de esta y el método que corresponde a la implementación.

## La clase GameManager<T>

Esta clase es el punto de entrada de la biblioteca de clases. Es a través de esta que la aplicación visual se comunica con el engine. Esta guarda la información del estado del tablero, el historial de juego, y los jugadores en objetos **Board<T>**, **History<T>** y **Players<T>**[].

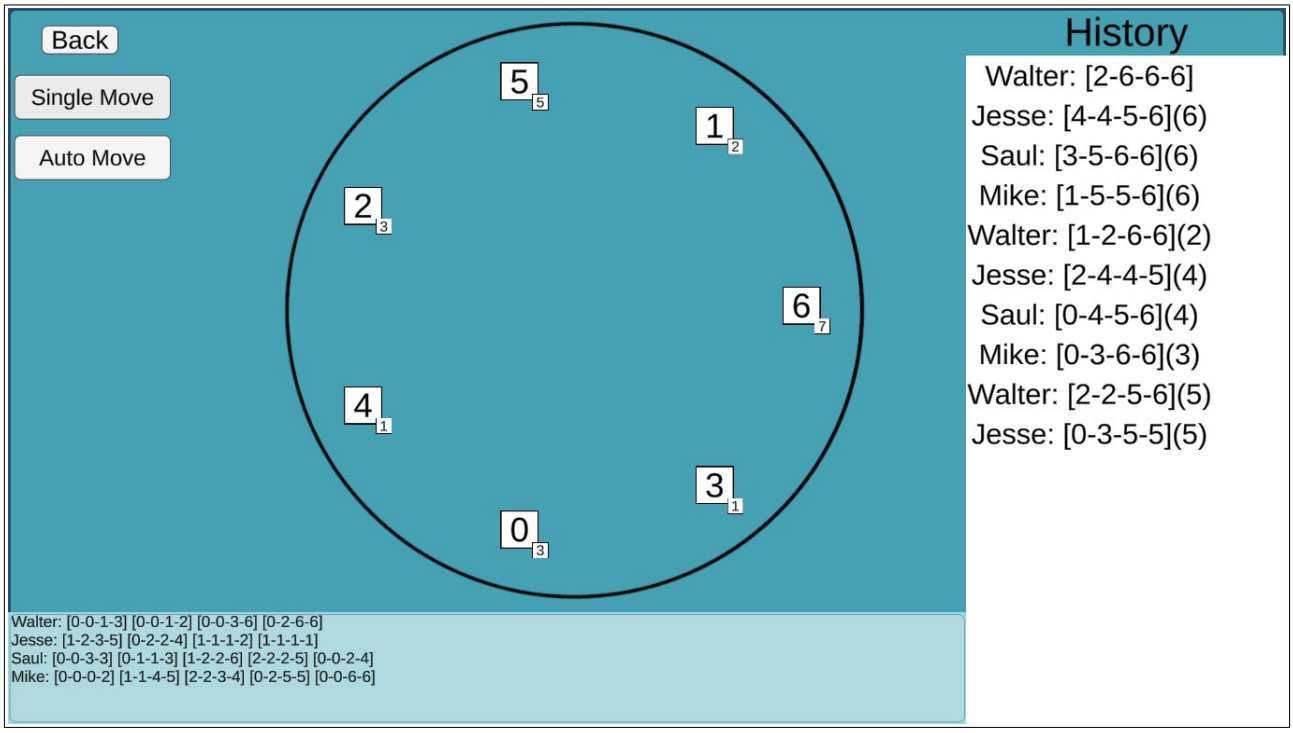
Parámetros del constructor:

- **strategy<T>**[] strategies: Un arreglo con los delegados que serán las estrategias de cada jugador. El tamaño del arreglo representa la cantidad de jugadores.
- **Generator<T>** generator: La función generadora correspondiente al tipo T.
- **int** tokenTypeAmount: La cantidad de tipos de salida a generar. Es el valor que se le pasará al **generator** para crear cada una de las caras.
- **int** tokensInHand: La cantidad de fichas que se repartirá a cada jugador al inicio del juego.
- **int** outputsAmount: La cantidad de caras de cada ficha.
- **string**[] playerNames: Un arreglo con el nombre de cada jugador. Debe tener la misma longitud que **strategies**.
- **evaluator<T>** evaluator: El evaluador de fichas que se usará en el juego.
- **CriteriaCollection<T>** victoryCheckerCollection: Una colección de criterios de finalización de juego.
- **Powers<T>** powers: El conjunto de poderes de fichas que se usarán en la partida.

Esta clase también tiene una referencia a un objeto del tipo **EffectsExecution<T>**, la cual recibe una instancia del **GameManager**, y posee métodos que permitirán que los **Effects** realizen cambios controlados en el juego. Estos métodos son **GiveToken**(**int n**), el cual quita n fichas del pool de fichas que no se repartieron (si quedan), y se las da al jugador que le toque jugar después; y el método **SetLastPlayerIndex**(**int index**), el cual permite cambiar el puntero que señala al último jugador en mover, para así poder definir quien es el próximo jugador en turno.
Al llamarse al constructor, los parámetros enviados se asignarán a los campos correspondientes, y luego se generarán todas las fichas que existirán en este modo de juego por medio de la clase **TokenGeneration**. Posteriormente se instanciarán los jugadores, cada uno con su estrategia correspondiente según los elementos en **strategies**. Luego se repartirá aleatoriamente a cada jugador, la cantidad de fichas que se haya solicitado. De no alcanzar las fichas para repartir a todos, se lanzará una excepción.

El método **MakeMove**() es lo que se llama desde la aplicación visual. Este método se encarga de realizar una jugada y devolver un **WinnerPlayData<T>** (clase derivada de **PlayData<T>**). Este objeto además de contener los datos de la jugada que se almacenan en un **PlayData<T>**, contiene además un arreglo con los nombres de los ganadores (null si aún no se ha alcanzado un estado de finalización del juego, y un arreglo vacío si el juego acabó sin ganadores). El orden de funcionamiento de este método es: primero hace jugar al jugador en turno, y toma la ficha y cara por donde desea jugar. Luego coloca esta ficha en el tablero si el jugador no se pasó, y además la inserta en el historial. Posteriormente ejecuta los efectos que haya disparado esta ficha, y se comprueba si se alcanzó un estado de finalización de juego.

## Pantalla de juego




La pantalla de juego cuenta con dos botones, Single Move, para mover una ficha a la vez, y Auto Move, para simular la partida automáticamente. En el área inferior se muestran las manos de cada jugador, y a la derecha está el historial. Cada entrada del historial significa que un jugador colocó la ficha indicada por la cara que se dice a la derecha. El círculo central se actualiza en cada jugada, y muestra cuantas caras libres de cada tipo existen en el tablero. Por ejemplo, en este turno del juego hay cinco caras con el número 5 libres, dos caras con el 1, siete caras con el 6, etc.

## Dificultades durante el desarrollo:

Debido a ciertas incompatibilidades entre el compilador de C# de Unity y .NET 6.0, al modificar el engine se deberá tener en cuenta lo siguiente:

- No usar interpolación de strings.
- No usar el operador .. para obtener un intervalo de un arreglo o string.
- No usar tuplas como parámetros out: **(int, string) tupla = (1, "a")**.
- Abstenerse de usar el namespace **System.Reflection**.

- Unity parece ser incompatible con la clase **Random**. Debido a esto fue necesario implementar la clase **RandomGen**, la cual contiene un método para generar un número aleatorio en un intervalo. El algoritmo empleado es un generador substractivo, y la fuente es: http://rosettacode.org/wiki/Subtractive_generator#C.23