Domino

Sobre los datos del juego

Los modelos y texturas del juego son archivos que no se modifican una vez que se incorporan al proyecto que asumí no necesitarían del control de versiones (aka. git), además de ocupar mucho espacio, pero son necesarios para la ejecución del proyecto. A tal efecto los subiré comprimidos como parte de una "release". Opcionalmente se puede bajar aquí. Descomprima y póngalo en 'data/models' (data está en el directorio raíz, donde está también el .sln)

Para ejecutarlo

Abra una terminal en la carpeta principal (donde esta Domino.sln) y escriba

- Para ejecutar el backend directamente en la consola

```
dotnet run --project backend
```

- Para ejecutar el frontend gráfico

```
dotnet run --project frontend/application [<Regla>]
```

  Si [<Regla>] es un parámetro opcional para ejecutar directamente un conjunto de reglas sin tener que pasar por la ventana principal

BackEnd

Este proyecto está concebido como una plataforma para probar la jugabilidad de juegos de domino donde hayan sido modificadas tantas reglas como ha querido el usuario. Luego su objetivo principal es lograr la representación de los más alocados juegos de domino con el mínimo esfuerzo y la menor cantidad de código nuevo posible cada ocasión. Un ramal secundario de esta intención sería la evaluación de estrategias para jugar al dominó y aunque no abordemos estrategias demasiado complejas, si creo que esta creada una jerarquía que permitiría expresar estrategias de una complejidad bastante elevada y comparar su comportamiento. Como parte del proyecto contamos con una interfaz visual que hace mucho más disfrutable el proceso de la simulación, permitiendo incluso tomar parte a jugadores humanos, por lo cual este proyecto también pudiera ser usado con fines lúdicos. El BackEnd asume la representación del juego de dominó dentro del software, y está completamente separado del FrontEnd, comunicándose con él únicamente a través de archivos txt donde siguiendo un protocolo más menos simple recibe las configuraciones de la partida a simular, para luego simularla en una aplicación de consola. El BackEnd ni siquiera decide quién es el ganador o no. Sencillamente simula la partida hasta su final, manteniendo contacto con el FrontEnd a través de una aplicación de consola para tomar las jugadas de los jugadores humanos así como para ir toda la información que puede interesar mostrar en pantalla. O sea, el BackEnd esta diseñado para simular un solo juego de domino cada vez hasta su final, y posee una jerarquía diseñada exclusivamente para la representación de juegos de domino con las variaciones más bizarras

Filosofía

Primeramente definamos que es un juego de domino a los propósitos de este proyecto: Un juego de domino es aquel jugado con fichas de varias caras, todas con valores enteros, de las cuales cada jugador posee un conjunto, y la esencia del juego gira alrededor de colocarlas una a una sobre la mesa, usando una y **SOLO UNA** de las cabezas de la ficha como enganche con las fichas que ya se han colocado en la mesa anteriormente. Luego, al diseñar una partida de domino, el esquema por el que se rige el desarrollo de cada jugada de la partida debió ser definido, funcionando de la siguiente forma. Cada turno cuenta de las siguientes etapas:

1. Si se debe repartir fichas a todos los jugadores se reparte cuantas veces sea requerido
2. Si el jugador en turno tiene derecho a refrescar su mano, lo hace cuantas veces se le permita
3. Se determina si el juego acabo
4. El jugador en turno realiza una jugada, esta es la parte central del turno

Repitiendo esta sencilla secuencia de cuatro pasos es posible simular el transcurso de partidas de domino con características muy diferentes unas de las otras.

Criterios

Cuando se reparten fichas, existe un criterio en virtud del cual se toman fichas, ya sea el azar, o que el jugador tome fichas que le dejen la mano lo más completa posible, o que su mano sume una cifra determinada. Cuando un jugador refresca su mano puede hacerlo de acuerdo a diferentes criterios: robando fichas de manera aleatoria, descartando una y robando dos que sumen lo mismo que la que descarto, descartando fichas hasta quedarse en una cantidad determinada, o cualquier otra forma que pueda a usted ocurrírsele Cuando un jugador coloca una ficha existe un criterio para valorar si esta colocación es correcta: la cabeza usada vale lo mismo que la cara de la mesa usada, o la cabeza usada es el sucesor modular de la cara de la mesa, o el doble modular. Criterios para validar la acción de colocar una ficha pueden ser: la paridad de la jugada anterior o de si acaba de ocurrir un pase o de cualquier otra razón que usted se invente. He citado todos estos ejemplos para llegar a la esencia de lo que consideramos un criterio en la filosofía del diseño del proyecto. Un criterio es una forma de realizar una acción determinada. Por ejemplo, `IMoverTurno` es una interfaz que representa una acción, la de determinar a quien toca el siguiente turno. Dado que hay numerosas formas (criterios) de determinar a quien toca el siguiente turno, cada una puede ser expresada como una implementación diferente de la interfaz `IMoverTurno` , siendo entonces cada una de estas implementaciones un criterio distinto, pero del mismo tipo. Las diferentes etapas en que separamos la Lógica de un turno descritas anteriormente, cada una necesita a la hora de realizar cualquier acción, de que se le facilite un criterio bajo el cual realizarla. La magia de la extensibilidad de este proyecto radica, más allá de abstraer todas las acciones, en el hecho de no fijar un mismo criterio para la realización de una las acciones durante toda la partida. Si usted solamente abstrajera la acción representada por `IMoverTurno` por ejemplo, pero no creara una estructura capaz de brindar diferentes criterios `IMoverTurno` en el partido, sino que dotara de solamente de uno de estos criterios al juego, entonces, para cada combinación diferente del uso de los mismos criterios usted debería crear una implementación distinta de la interfaz `IMoverTurno` . Nosotros por el contrario separamos el criterio del momento de su uso, o sea, abstrajimos la toma de la decisión del criterio a usar en cada momento del funcionamiento del criterio en sí. Siendo así, quedan completamente separados la descripción de un criterio para realizar una determinada acción, y la máquina que va dando criterios al juego para que este los use. Así, podemos implementar las infinitas combinaciones en cuanto a momento de uso que tienen un grupo de criterios del mismo tipo, sin necesidad de tirar código nuevo cada vez que queramos alterar las razones por las que un criterio es escogido sobre otro, cómo?. Para ilustrarlo pongamos un ejemplo: Llegado el momento de validar una jugada, el juego le solicita al cuerpo de reglas que este le facilite un criterio `IValidador` y una vez lo tiene, evalúa como válida o no a una jugada determinada. Como ocurrió la selección del criterio que se usó? Existe una máquina, que al ser inicializada recibió una serie de criterios IValidadores, y una serie de combinaciones de predicados que no son más que las instrucciones acerca de cuando usar uno u otro de los criterios entregados inicialmente. Esta máquina, toma la información relevante del juego, evalúa los predicados, y de la combinación de predicados que se cumplen, devuelve el `IValidador` que al principio le fue indicado devolver al cumplirse esta combinación en específico. O sea, si hubiéramos inicializado la maquina con una lista de Criterios de otro tipo y otro juego de predicados, ella se limitaría a ver cuales predicados se cumplen y cuales no y devolver el criterio que fue instruida a devolver dada la situación actual en el momento de su creación. Siendo así, en vez de tirar código nuevo, para alterar la forma de escoger el criterio a emplear en una situación determinada basta entregar parámetros diferentes a la maquina en el momento de su implementación. Estos valores pueden ser entregados fácilmente a través de un txt, así que con solo cambiar números en un archivo txt, estaremos modificando tanto como queramos las reglas de nuestro juego de domino. Cuando único habría que tirar código nuevo, seria en el momento en que se nos ocurre una forma (criterio) completamente nueva de realizar una acción, en cuyo caso bastara implementar una interfaz para llevarla a cabo. Pasando a detalles de implementación

DaCriterio<T>

La Máquina anteriormente descrita no es más que el objeto `DaCriterio<T>` . Al ser creada, una instancia de esta clase toma una serie de criterios del mismo tipo `T` y una serie de predicados. Luego, lee de un txt cuales combinaciones de predicados dan lugar al uso de cada criterio. Simple y esencial, cada acción del juego, puede ser ejecutada de acuerdo a una diversidad de criterios y más aún, estos pueden cambiar a lo largo del juego y usarse diferentes criterios en diferentes situaciones. Esta máquina estandariza el proceso de seleccionar que criterio usar para una acción determinada cualquiera que sea la acción a lo largo del juego. Así: Un `DaCriterio<bool>` devuelve true para algunas combinaciones de predicado y false para otras, ideal para determinar si el juego acabo o no. Un `DaCriterio<IEmparejador>` devuelve para cada situación distinta del juego (expresada como combinación de predicados), un criterio para determinar si una ficha se puede encadenar a otra o no. Un `DaCriterio<IValidador>` funciona análogamente pero entregando un criterio para validar. El análisis de la clase Reglas_del_Juego nos permite apreciar que cada faceta de la partida donde el criterio a usar dependa de la situación es modelable de la misma manera, a través de un objeto `DaCriterio<T>` , donde `T` es el tipo del criterio a escoger. El hecho además de expresar todas las facetas de importancia que están sujetas a cambios de la misma manera, nos conduce a la posibilidad de tener un mecanismo estándar, común a todas para expresar los cambios que deseamos. En este caso, a través de txt podemos expresar todos los cambios que deseamos realizar a las reglas del juego, con la posible excepción de

que se necesite crear una forma(criterio) completamente nueva de realizar una acción y que no sea expresable como combinación de las formas anteriores. En tal caso para expresar el cambio bastara satisfacer una interfaz and et voila!. `DaCriterio<T>` es así el ladrillo con el que se construye el BackEnd de este proyecto, no importa si estas construyendo una pared de carga o una chimenea, lo harás de manera sencilla con estos ladrillos que además, son estándares, lo cual facilita muchísimo la traducción de un juego sumamente enrevesado de la cabeza del usuario, a la formalidad de los algoritmos computacionales

A Grandes Trazos

Sobre el Intercambio de Fichas

El proceso de intercambiar fichas puede ocurrir de dos maneras, una en la que el jugador en cuestión descarta o roba las fichas necesarias hasta quedarse con un número determinado en la mano, y otra en la que el jugador en cuestión roba o descarta fichas con el objetivo de terminar la operación con un balance determinado en cuanto a cantidad de fichas. Ambos modos de proceder son implementados por herederos de la clase `Cambiador`. En esencia la clase `Cambiador` se encarga de regular el proceso de intercambio de fichas de un jugador con las fichas fueras de juego. Tiene un campo para un Criterio de Intercambio, que expresa el criterio a usar para reemplazar fichas que el entregue por otras (por ejemplo Al azar, o reemplazar una ficha con la suma de otras dos), y además tiene campos para las restricciones numéricas dígase el balance final de la operación o la cantidad de fichas con las que debe terminar el jugador. En un `Cambiador` hay dos tipos de campos, uno referido a un ICriterio_de_Intercambio que se encargara de brindarnos el método que reemplaza unas fichas por otras, mientras que a través de los campos numéricos podemos regular la cantidad de fichas que toman parte en el intercambio sin necesidad de escribir código nuevo. O sea, a través de la Composición podemos lograr la expresión de intercambios de ficha de cualquier tipo. El proceso de un jugador refrescar su mano puede ser expresado completamente usando solamente un `Cambiador`. Por otra parte el proceso de repartir fichas es más complejo, pues puede implicar que los jugadores primeramente entreguen todos una cantidad de fichas y luego roben, o algo similar. Por tal razón, el proceso de Repartir está dividido en dos partes, una referida al descarte obligatorio de fichas por cada jugador y la otra referida al robo de fichas. Ambas partes son descritas por Cambiadores de uno u otro tipo, pero Cambiadores en fin. Siendo así, la operación de Repartir fichas es descrita perfectamente por una *tupla* de Cambiadores. El objeto `MoverFichas` del cual hablaremos más adelante en la clase `Reglas_del_Juego`, estará compuesto por dos objetos de tipo DaCriterio, uno que dará un `cambiador` a usar para Refrescar fichas o un valor null en dependencia de si se debe refrescar fichas en la situación actual del juego o no, y otro que devolverá una *tupla* de Cambiadores o una *tupla* null, en dependencia de si es momento de Repartir o no, y tal *tupla* sería la usada para describir el proceso de Repartir fichas a llevar a cabo. Complejo, quizás explicado demasiado vagamente, pero funciona y es tremendamente extensible

Clase Juego

La clase `Juego` es casi que el portal del BackEnd hacia afuera. Su método `IEnumerable<Action> Jugar`, es el método que es llamado para simular una a una las jugadas del juego e imprimiendo en consola. El método jugar un `IEnumerable` lazy, lo que nos permites ir tan lejos o tan pausado en la simulación como queramos. La otra propiedad visible de la clase `Juego` son las puntuaciones, que van aparte para que puedan ser mostradas una vez que el `IEnumerable` lazy `Jugar` haya concluido

Clase Estado

La clase `Estado` contiene la información del juego que sería visible a todos los que participan en él. Dígase, las caras de la mesa activas, una lista con todas las acciones de juego que se han realizado, las reglas que se están aplicando actualmente para intercambiar o repartir fichas, expresadas a través de Cambiadores. Así en general la clase `Estado` posee la información que en una partida de domino real pero que sería visible por todos. Siendo así, las fichas en la mano de los jugadores no forman parte de `Estado`, luego, muchas veces a lo largo de la implementación, cuando se va a describir el estado del juego se pasa un objeto de tipo `Estado` junto a la mano del Jugador en turno, cuando es pertinente pasar tal información. Otra peculiaridad de `Estado` la constituye el hecho de que al poseer métodos para actualizarse al ser realizada una determinada acción, debemos proteger a la instancia que lleva la información sobre el transcurso del juego de ser modificada por clases de menor jerarquía. Por tal razón, pusimos a Juego un Constructor que devuelve un `Estado` que representa una situación idéntica, pero en una nueva instancia, de manera que la instancia original que lleva la evolución del juego permanezca invariable a las clases de menor jerarquía como Jugador u otras de las que hacen uso de Estado, que son muy numerosas, por lo cual la no protección de `Estado` podría llevar a su uso indebido intencional o no

Clase `Reglas_del_Juego`

La clase `Reglas_del_Juego`, está conformada por objetos de tipo `DaCriterio` que se encargan de dar al Juego, o a las

clases de jerarquía inferior que así lo necesiten, los criterios a usar en la ejecución de cada proceso del Juego. A continuación una descripción de sus campos:

- Algunos campos informativos, como **cabezas_por_ficha**, **data_tope** o fichas_por_mano, que reflejan exactamente lo que su nombre refiere. Estos campos son iniciados al principio del juego y por supuesto luego no pueden ser alterados
- Un `ICreador`, que no es otra cosa sino la interfaz que se encarga de crear las fichas que se usaran en el juego, esta modelado con una interfaz, a pesar de que solo hayamos implementado el creador usual, porque quizás exista otra forma de concebir las fichas del juego de domino que no sea la Usual, pero que aún mantenga la estructura de arreglo de enteros
- Un `IPuntuador`, y no un DaCriterio tipo `IPuntuador` porque consideramos que realmente no tiene sentido que a mitad de juego cambie la forma de puntuar las fichas
- `Finisher`, `Emparejador`, `Validador`, `MoverTurno`. Estos son campos, donde un objeto de tipo DaCriterio administra para cada situación del juego el criterio a emplear para determinar si el juego acabo o no, si una jugada es correcta o a quien toca el siguiente turno. Las interfaces `IEmparejador`, `IValidador` e `IMoverTurno` son la forma en que los criterios para realizar cada acción son expresados
- Un objeto de tipo `MoverFichas`, que administra el criterio a emplear en cada momento para refrescar fichas y para repartir fichas

Estos campos en su mayoría no son visibles, pero sus funcionalidades son accesibles a través de métodos portal que posee la clase `Reglas_del_Juego`

Su funcionamiento

El funcionamiento del método `Jugar` es lo realmente relevante aquí. Como había dicho en un principio, para representar un Juego de Domino la filosofía a seguir fue representarlo como una sucesión de turnos, donde, cada turno consistía de 4 etapas

1. Repartir fichas de ser necesario
2. Refrescar la mano del jugador en turno de ser posible
3. Determinar si el juego acabo o no
4. El jugador en turno realiza una jugada

Así el método `IEnumerable Jugar`, realiza estas cuatro etapas. En las primeras dos etapas, mas allá de la información que muestra en consola para cumplir el protocolo de intercambio de información, se vale de la clase `Banquero` para realizar la repartición de fichas y refrescar la mano del jugador en turno. `Banquero` es uno de los campos de Juego

Clase `Banquero`

La clase `Banquero` abstrae el proceso de administrar las fichas, desde inmediatamente después de su creación, hasta controlar las manos de cada jugador y las fichas que se mantienen fuera. El método más grueso de `Banquero` es DarMano. Este es un método interno suyo, pero me referiré a él por su importancia. El método DarMano, devuelve una acción de Tipo Intercambio, que es posteriormente informada al Estado. El método DarMano, en esencia simula el proceso en el cual un jugador realiza un intercambio de fichas de algún tipo con las fichas afuera. Este método polifacético funciona tanto para Repartir como para Refrescar, cumplimos así el principio DRY, de tratar de tener de agrupar todo el código similar. En esencia DarMano, toma un objeto de tipo Cambiador con las reglas del intercambio de fichas a realizar y recibe una referencia del jugador que realizara el intercambio, y actúa como intermediario entre el jugador, su mano y las fichas de fuera a las que puede acceder en esta operación. Siendo así, el método DarMano es el corazón de todo intercambio de fichas en el juego, con las características que posea. Sobre la filosofía de como quedan expresados los cambios de fichas hablare más adelante. La clase `Banquero` administra además las manos de cada jugador, pero no es conveniente que `Banquero` sea accesible por clases ajenas a Juego, ya que estoy podría llevar a malentendidos o incluso desarrollo malicioso, que pretenda, llamando a los métodos Repartir y Refrescar modificar el transcurso correcto del juego. Por tal motivo, la instancia de `Banquero` que administra el juego solo es accesible por el Juego que lo contiene como campo. Para el uso del resto de las clases, que no tienen derecho a hacer uso de los métodos Repartir y Refrescar, está la clase envoltorio `Portal_del_Banquero`, que brinda acceso a las manos de los jugadores a las clases autorizadas para ello. `Banquero` tiene además un método para mantenerse actualizado a medida que ocurren las jugadas, y tal método es llamado por la instancia de tipo Juego a la que pertenece, que lo va alimentando de información. Luego de haber pasado la fase de los intercambios de ficha, y de determinar a través del objeto de tipo `DaCriterio<bool>` Finisher si el juego acabo o no, llega el momento de que el jugador realice la jugada debida. Es llamado el jugador a través del indexer de la clase `Organizador` y se le pasan los parámetros que describen el Estado del juego, a la espera de que decida y retorno la jugada que realizara

Clase `Organizador`

La clase `Organizador` es la encargada de mantener separado del resto de la implementación a las instancias de los jugadores, de esta manera no se puede hacer uso indebido de ellos. Solo pueden ser requeridos por la clase `Juego` y por la clase `Banquero`. La clase `Organizador` posee además otras informaciones relativas a la organización del juego, tomando importancia al determinar el orden de los jugadores, a través de un `IOrdenador` que le es pasado al principio, que después juzgaran los criterios `IMoverTurno` para decidir a quien toca el siguiente turno

Clase `Jugador`

La clase `Jugador` abstrae del juego la acción de decidir una jugada. En esencia funciona bastante como en la vida real. Lo que se espera de un jugador es precisamente lo que hace, analizar la situación del juego y realizar una jugada, o descartar una cantidad determinada de fichas de ser necesario. Tal y como a un jugador de la vida real, al jugador lo dotamos de una referencia a las reglas del juego, porque toda persona (o casi toda supongo) debe conocer las reglas de un juego para poder jugar. Si bien como dije inicialmente constituye un ramal secundario y no es el objetivo directo de este proyecto realizar jugadores demasiado complejos, creo que hemos realizado una jerarquía que resiste el primer escrutinio en cuanto a cuan profundo pueden llegar los jugadores. `JugadorHumano`. Satisfaciendo la interfaz de un jugador, la clase `JugadorHumano` busca recibir del Usuario las indicaciones de qué jugar. Tiene su propio epígrafe en el protocolo de comunicación del BackEnd y el FrontEnd a través de la Consola. Incluso si no llegamos a implementarlo en el FrontEnd, el BackEnd está listo para acoplarse a cualquiera que implemente la sencilla interfaz

Clase `JugadorVirtual`

Todos los jugadores de dominó real juegan igual: revisan todas las jugadas posibles y valoran cual de ellas es la más conveniente. Por esta razón damos una implementación al método `Jugar` en la clase `JugadorVirtual` y lo que abstraemos y dejamos a la implementación es el hecho de valorar cuan buena es una jugada, a través de un método protected llamado Valorar. Además, dotamos al jugador virtual de un campo tipo `IDescartador`, y a pesar de que solo hicimos la implementación del descartador usual, quisimos dejar abierta a través de la clase `IDescartador` de lograr, a través de la composición la combinación de jugadores de un tipo determinado con diferentes tipos de descartadores. No nos lanzamos más allá con el tema de las estrategias para descartar expresadas a través de `IDescartador` es porque como afirmamos, no es nuestro objetivo explorar la conveniencia de estrategias de dominó y mucho menos con la gran cantidad de variantes distintas de juegos de domino que podemos expresar y donde dudo que una estrategia de Descartar sea generalmente mejor que el resto. En nuestra jerarquía, la diferencia entre los jugadores radica en las diferentes formas en que otorgan una valoración a una jugada dado un Estado del juego. Así tenemos por ejemplo el `JugadroRandom` que otorga una valoración aleatoria a las jugadas, o el `Botagorda`, que usando las reglas determina la puntuación de cada ficha y juega la que más alto puntúe. Pero por supuesto, para jugadores más complejos requerimos procesamientos más complejos. Decidimos diseñar una jerarquía que permitiera almacenar información al `Jugador`, y que este luego la use en el proceso de valorar cada jugada. Esta jerarquía está encabezada por la clase `Memoria`, que mantiene un registro de las jugadas realizadas y lo va actualizando. Sobre la base del análisis de ese registro se pueden crear memorias que procesen todo tipo de informaciones. En nuestro caso solo alcanzamos a diseñar una `Memoria_de_Pases`, que guarda la información de cuales jugadores se han pasado a cada ficha. Diseñamos un jugador que la usa, el `JugadorPasador`, que da más puntuación a una jugada si contiene caras que alguno de sus rivales no lleva, y menos si no la lleva su compañero, así como triplica la puntuación de las jugadas que reducen el número de cabezas distintas en la mesa. Intentamos pero por cuestiones de tiempo se quedó en pañales seguir extendiendo la jerarquía de memorias para crear lo que llamamos `Full_Memoria` que almacenara cuantas veces un jugador ha subido, matado o dejado correr una data determinada, cuantas fichas ha tirado de cada número, cuantas fichas tenía en la mano a cada momento. Fue cuestión de tiempo, realmente, la jerarquía de memoria permite llevar a buen puerto tal intención de proponérnoslo, y luego, el usar tal información en el método `Valorar`, nos llevaría a una estrategia de juego de domino bien cercana a la usada en la realidad. Además implementamos un `JugadorSeguro`, que solo tira aquello de lo que tiene más para evitar pasarse. Por ultimo aclarar que todos los jugadores virtuales tiene un método para Apertura distinto del método para jugar de manera usual y que es el llamado cuando al jugador toca abrir el partido. Además, es también abstracto el método `ValorarDatas`, que nunca implementamos pero puede ser usado para dotar al jugador de un método para calificar por su importancia a cada una de las datas. O sea, no pudimos implementarlas tanto como hubiéramos creído, pero creemos que la jerarquía necesaria para implementar estrategias de domino bastante complejas está ahí. Las acciones de juego la representamos con una jerarquía pequeña que creemos suficiente a los propósitos del dominó:

Acciones

Describimos tres tipos de acciones de juego, que se repartan piezas, que un jugador refresque su mano o que un jugador realice una jugada (la cual incluye pasarse). Así la clase Action abstracta tiene tres herederos, `Intercambio`, `Jugada` y `Repartición`. `Repartición` solo contiene el método **ToString()**. Por otra parte tanto `Jugada` como `Intercambio` tienen en común el campo **autor**, con el nombre del jugador en llevar a cabo la jugada. `Jugada` tiene información relativa a la jugada de domino, dígase la ficha colocada, la cabeza de la ficha usada para su colocación, la cara de la mesa por la que la

ficha es colocada y la cantidad de fichas que quedan en la mano al jugador que la coloco. Por su parte `Intercambio` sencillamente tiene la cantidad de fichas tomadas y devueltas por el jugador

Predicados

La clase `Predicado` simula al Predicado de Lógica. Tiene únicamente un método Evaluar que toma el Estado del juego y la mano del jugador en turno y en base a ellos devuelve verdadero o falso. En base a la combinación de predicados se sustenta la toma d decisiones sobre los criterios a aplicar en cada momento del juego y pudiera ser necesario crear nuevos predicados (lo cual es muy sencillo pero implica tirar código) de no existir entre los predicados ya preparados alguno que el usuario desee emplear para expresar una situación determinada del juego. No obstante las implementaciones de predicados que realizamos fueron bastante amplias, y como se verá cuando describamos los juegos que es posible jugar, cubren una cantidad de situaciones probables muy cómodas permitiendo hacer a este proyecto sumamente extensible

Interfaces

No listaré todas las interfaces, son muchas. En todo caso generalmente sus nombres y los nombres de sus métodos las hacen bastante explicitas. Las interfaces se encargan de moldear la forma que debe tener un criterio que se use para llevar a cabo determinada acción. Implementamos numerosas veces cada interfaz y dotamos al juego de muchas opciones. No obstante por supuesto no somos adivinos y no podemos implementar cada criterio distinto para determinar si dos fichas son encadenables o la forma de determinar a quien toca el siguiente turno. Por tal razón, más allá de la expresividad que aquí hemos alcanzado, para implementar criterios nuevos, el usuario solo tendrá que hacer la implementación de una interfaz. Lo otro, a lo mejor salta por ahí un `NotImplementedException` . No obstante, creemos que la estructura para dotar al juego de una flexibilidad mayúscula con la mayor comodidad posible esta escrita.

Sobre el Protocolo

El Protocolo, con ligeros cambios, aparece tal cual descrito en los archivos adjuntos. Sencillamente el juego al cargarse lo hace a partir de archivos txt que previamente deben haber sido rellenados por el FrontEnd. En esos archivos se sigue un patrón más menos similar para cargar usando unas pocas líneas a cada uno de los DaCriterio que expresan las reglas del juego. De igual manera hay archivos para la cantidad de datas. Una información más detallada sobre como configurar el juego aparece en cada uno de esos archivos, por si se desea desacoplar el FrontEnd actual y enganchar otro. La última parte en ser añadida del protocolo de cargado fue la parte correspondiente a cargar los jugadores. En el archivo Jugadores.txt también está toda la información sobre como se debe realizar el rellenado del archivo. Existe una carpeta llamada *Configurado*, que es donde el implementador del FrontEnd deberá reescribir los txt si desea expresar un juego configurado y diseñado especialmente por el usuario. De otra forma, el implementador del FrontEnd solo deberá pasar por consola el nombre de la carpeta donde se encuentran los txt del juego a jugar y este se ejecutara.

Implementaciones

Para probar el poder de extensibilidad de este proyecto, hemos implementado algunos juegos desde el más sencillo hasta versiones creadas por nosotros que despliegan una elevada complejidad. Y lo hemos hecho sin modificar ninguna línea de código en específico para ellos, sino tan solo empleando un cuerpo de predicados y de criterios para realizar las diferentes acciones de juego. Las diferencias entre cada una de las siguientes implementaciones radican únicamente en el cambio de los valores numéricos de los txt.

- ### Usual

  Un juego de siete fichas de lo más común.

- ### Diez Fichas

  Un juego de diez fichas con la peculiaridad de que al repartir, la mano de cada jugador suma exactamente 90

- ### Burrito

  Juego tradicional donde si un jugador no lleva fichas roba hasta finalmente poder jugar. El juego termina cuando alguien se pega o no quedan fichas fuera y se tranca el partido

- ### Longaniza

  Juego tradicional donde cada jugador coloca tantas fichas como desee hasta que no pueda colocar más y entonces pasa el turno

- **Melcocha**

  En este juego, al repartir fichas se garantiza que cada jugador comience la partida con una mano lo más variada posible. Mientras cada jugador tenga más de tres fichas, si usted no lleva tiene derecho a tirar un doble por cualquier cabeza sin importar que no encadene. Si usted no lleva y al menos el líder del juego tiene más de tres fichas entonces usted roba fichas de manera similar al burrito hasta que lleve

- **Camarón**

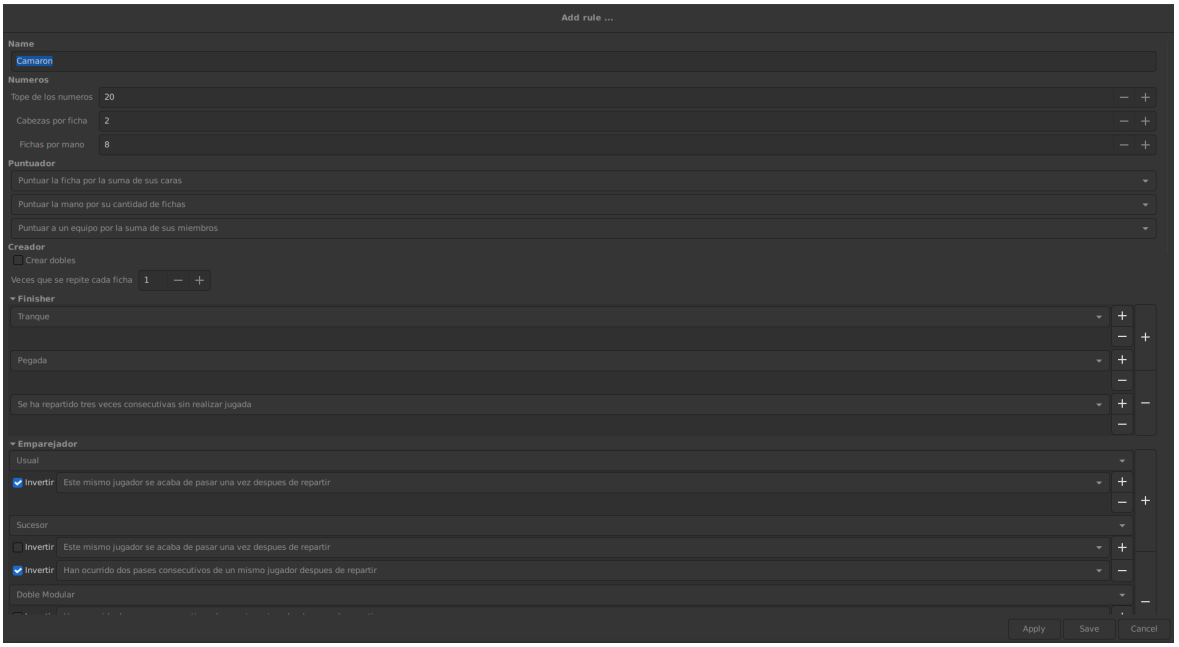
  Juego fanfarrón que creamos para mostrar la amplitud de posibilidades de nuestro proyecto. Una vez que el juego ha comenzado, cada jugador debe colocar una ficha tal que la paridad de su puntuación sea diferente a la paridad de la puntuación de la jugada anterior. Si el jugador no lleva, entonces roba una ficha y tiene la oportunidad de volver a colocar, pero en este caso además de satisfacer la condición de la paridad, la ficha que coloque debe ser colocada usando una cabeza tal que valga lo que el sucesor modular de la cara de la mesa por donde la coloque. Si aun así el jugador no lleva el jugador descarta una ficha de su elección y le son devueltas dos fichas tales que sumen lo mismo que la ficha que el devolvió. Ahora tiene la oportunidad de por fin realizar su jugada, pero además de la restricción de la paridad debe satisfacer que la cabeza por la que coloque la ficha debe ser el doble modular de la cara de la mesa por donde la coloque. Si aun así, el jugador en turno no puede realizar jugada, entonces todos los jugadores devuelven sus manos al montón y cada jugador toma tantas fichas como robo. Si durante el turno de un mismo jugador este proceso de repartir ocurre tres veces sin que pueda realizar jugada, entonces el juego se declara terminado, al igual que se declara terminado si un jugador se pega

Los puntuadores en estos juegos tiene disimiles características, Algunos puntúan a un equipo por la suma de sus puntuaciones individuales y otros por la mejor o peor de estas. Algunos puntúan a un jugador por la suma de sus fichas, otros por sus fichas más altas y otros por su cantidad de fichas.

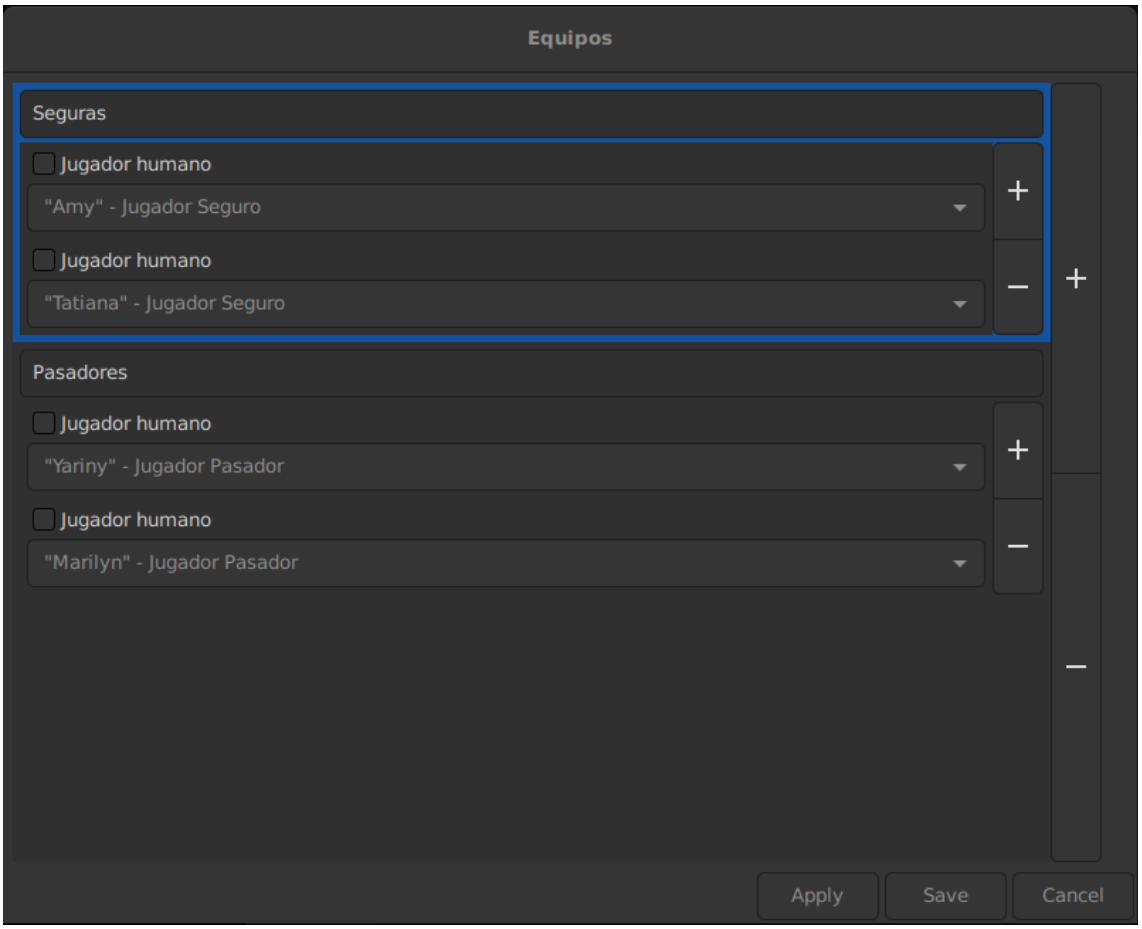
Frontend

Estructura interna

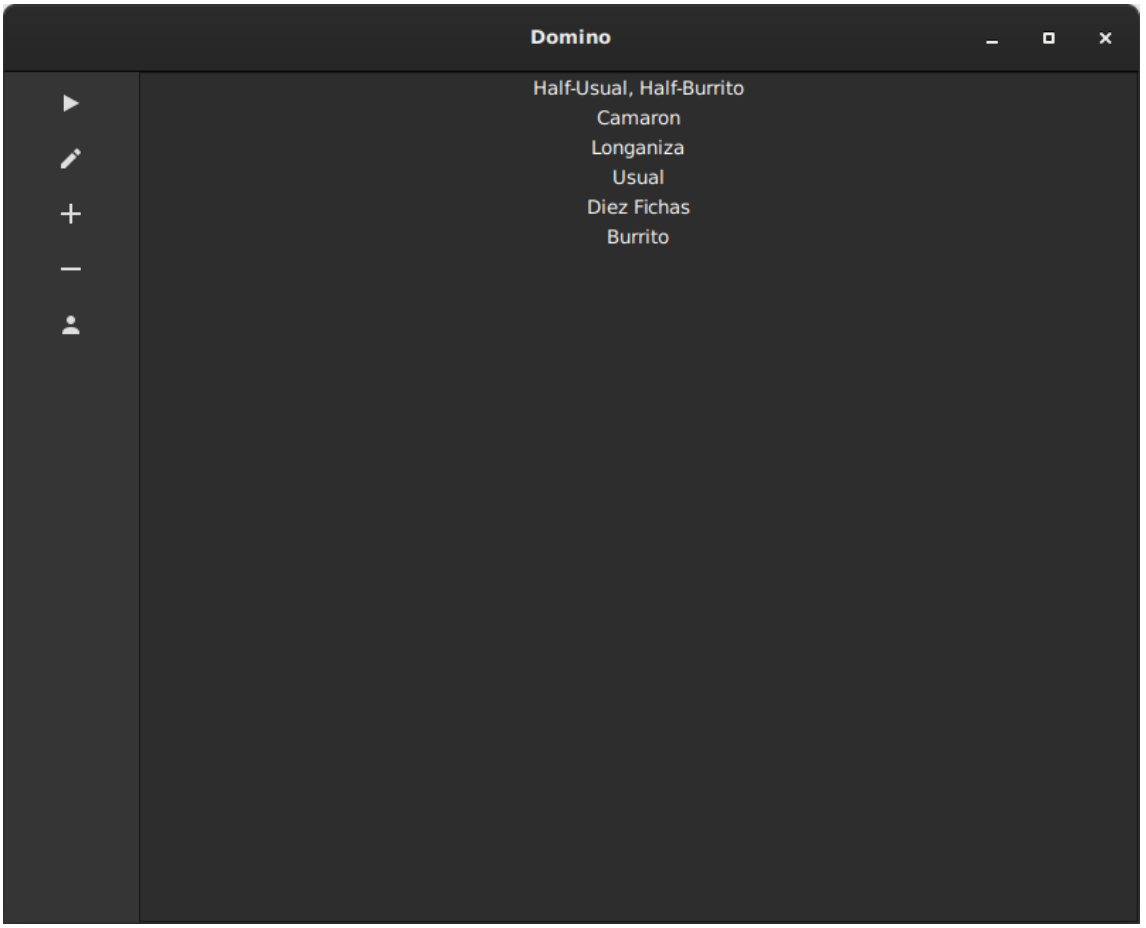
- `Frontend` : Aquí están todo lo que tiene el **frontend** (la parte gráfica) del proyecto
  - `Application` : punto de entrada a la aplicación. Implementa `Patch.Application` que a su vez implementa `Gtk.Application` que es necesaria para trabajar con **Gtk**.
  - `IListBoxRow` : representa un objecto que puede adicionarse a un objeto `ListBox`
    - `Rows`
      - `Rows.emparejador` : clase asociada al archivo "Emparejador.txt" de las reglas del juego
      - `Rows.finisher` : clase asociada al archivo "Finisher.txt" de las reglas del juego
      - `Rows.moverturno` : clase asociada al archivo "MoverTurno.txt" de las reglas del juego
      - `Rows.refrescador` : clase asociada al archivo "Refrescador.txt" de las reglas del juego
      - `Rows.repartidor` : clase asociada al archivo "Repartidor.txt" de las reglas del juego
      - `Rows.validador` : clase asociada al archivo "Validador.txt" de las reglas del juego
  - `ListBox` : widget que se usa para representar variadas opciones de configuración a lo largo del proyecto (de la parte gráfica)
  - `Message` : clase auxiliar usada para lanzar diálogos estilo "message box" (cuadro de diálogo)
  - `RuleDialog` : clase que encapsula el controlador de un regla. Desde este pueden editarse de forma más fácil las reglas del dominó



- `TeamDialog` : clase que encapsula el controlador de los equipos. Desde este pueden editarse de forma más fácil la disposición de los equipos y los diferentes jugadores



- `Window` : ventana principal desde la cual se accede al juego. Desde esta se puede acceder a las opciones de configuración de los equipos, las reglas y se pueden iniciar las partidas



- `Frontend.Patch`
  - class Application: Clase auxiliar para sortear un bug en la biblioteca **GioSharp** que causa un error interno cuando se lanza un evento de tipo **GLib.Application.Opened** (el marshaler recibe un puntero a un array de punteros a objetos nativos de type GObject y lo interpreta como un puntero a un objeto, mi clase sobrescribe es comportamiento)

```
public class OpenedArgs : GLib.SignalArgs
{
  public GLib.IFile Files
  { /* ~~^~~ Aquí debería ser GLib.IFile[] */
    get
    {
      return GLib.FileAdapter.GetObject (Args [0] as GLib.Object);
    }
  }

  public int NFiles
  {
    get
    {
      return (int) Args [1];
    }
  }

  public string Hint
  {
    get
    {
      return (string) Args [2];
    }
  }
}
```

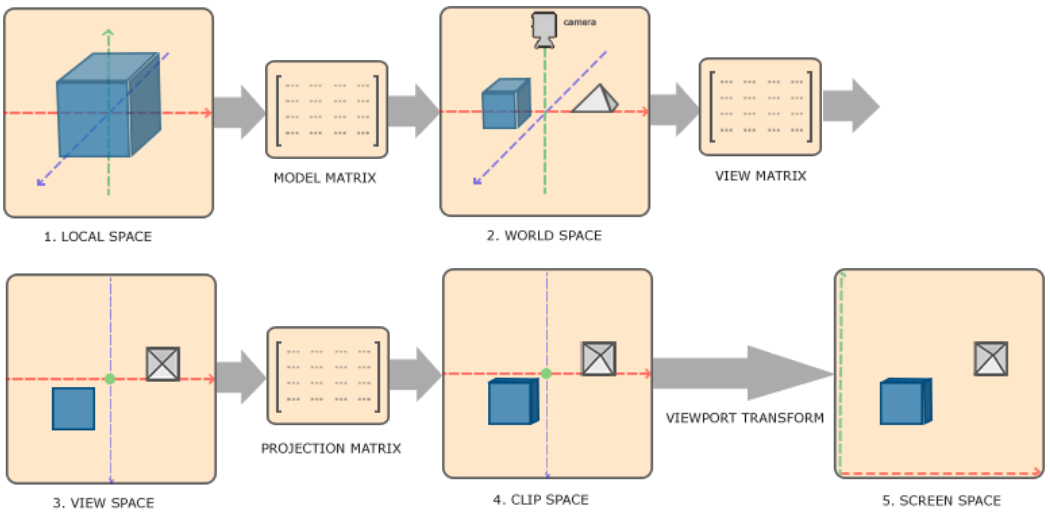
- `Frontend.Game` : Aquí esta el código del juego como tal
  - `Backend` : Esta clase es el controlador lógico (del **backend**). Mantiene una cola (concurrente) para guardar los eventos del juego según son interpretados. Internamente ejecuta el backend en un proceso aparte y toma de la salida estándar todo lo necesario para representar una partida de dominó, lanzando eventos que describen a

cada uno según el controlador gráfico lo requiera.

- `Window` : Es el controlador gráfico. Usa una ventana proporcionada por la biblioteca **Gtk** para crear una contexto de **OpenGL** y representar el juego usando el motor gráfico `Frontend.Engine`

- `Frontend.Game.Objects` : Colección de objetos gráficos
  - `AtrilObject` : Representa las fichas del jugador de turno. Actualmente en desarrollo
  - `PieceBoard` : Representa el tablero sobre el cual se muestran las fichas. Ademas implementa la lógica para posicionar correctamente las fichas según se juegan, y desde luego mantiene una lista interna de objetos `PieceObject` que representan las fichas (solo están soportadas las fichas con dos caras)
  - `PieceModel` : Encapsula el modulo y las textures de una ficha concreta. También se encarga de crear la texture correspondiente a su número procesalmente (usa la biblioteca **Cairo**, parte del paquete de **GtkSharp** para generarlas), por lo que el valor de las caras de una ficha puede ser totalmente arbitraria
  - `PieceObject` : implementa una sencilla lógica de lista (referencias next, prev, etc.) que se utiliza en la clase `PieceBoard` para dibujar cada una; también facilita el trabajo ya que simula las conexiones de las fichas en un tablero real

- `Frontend.Engine` : Aquí es donde más trabajo puse en comparación con el resto del proyecto, ya que es de muy bajo nivel (aquí hay trabajo con vertices, vectores y matrices de transformaciones)
  - `Gl` : Representa un contexto del motor gráfico. Asume que se ejecuta de forma exclusiva en un contexto de **OpenGL**, aunque no asume control directo sobre el renderizado (deja libertad al usuario para renderizar cada cuadro cuando quiera y configurar el `Viewport` según convenga). Como tal, maneja la iniciación del contexto de **OpenGL** (usa la biblioteca **OpenTK** para el acceso de "alto nivel" a la API de **OpenGL**). Todo lo relacionado con el motor gráfico se accede por aquí
    - `Gl.Camera` : encapsula las matrices de transformación y las abstracciones necesaria para manipularlas. En concreto guarda las matrices de proyección y de cámara, porque la matriz del modelo las proporciona cada `Engine.Object` . El proceso empieza con las coordenadas en espacio local (local space), estas se multiplican por la matriz de model, que transforma en espacio global; se multiplica por la matriz de cámara (world space), que las transforma en espacio de cámara (view space); se multiplica por la matriz de proyección que las transforma en recortado (clipping space); al final **OpenGL** internamente las transforma a espacio de pantalla (screen space) usando la configuración del viewport



    - `Gl.Dds` : implementa un cargador para el formato DDS (DirectDraw Surface) que es un formato de imagen comprimido que puede ser usado directamente por **OpenGL**, o sea, no necesita descomprimirse en datas de pixeles para que pueda usarse en una textura
    - `Gl.DirLight` : Encapsula los datos de una luz direccional sin posición, algo así como una luz ambiental que tiene una dirección definida; por ejemplo la luz del Sol incide sobre todas partes, pero desde una dirección definida
    - `Gl.IBindable` : Representa un objeto de **OpenGL** que debe "bindearse" para usarlo desde un shader. Actualmente solo lo implementan las clases `Gl.Ssbo` y `Gl.Ubo` y no considero que en el futuro lo usen más clases
    - `Gl.IDrawable` : Representa un objeto que puede dibujarse
    - `Gl.ILocalizable` : Representa un objeto que tiene una posición definida en espacio global (world space)
    - `Gl.IPackable` : Representa un objeto que puede empaquetar sus datos en un flujo de bits. Usado por las

las implementaciones de `Gl.Light` para empaquetarse en un *SSBO*

- `Gl.IRotable` : Representa un objeto al que se puede rotar usando una vector como guía
- `Gl.IScalable` : Representa un objeto al que puede escalarse
- `Gl.Light` : clase abstracta base para las luces que pueden adicionarse en una escena. Como tal almacena los valor cromáticos de la luz
- `Gl.Loader` : clase estática que se encarga de cargar en tiempo de ejecución la API de **OpenGL**. Es necesaria porque **OpenGL** es solamente un standard y cada fabricante de tarjetas gráficas crea su propia implementación (**OpenGL** es algo así como una interfaz que cada productor implementa) y que difieren mucho entre cada computadora. Esta clase usa la API especifica para cada plataforma para cargar la biblioteca correspondiente y obtener las funciones (y extensiones) que cada cual implementa
- `Gl.Material` : encapsula un material, esto es, un concepto abstracto de material. Cada material tiene mapas de iluminación (comúnmente las más reconocibles como texturas), de normales, la brillantez, la rugosidad, etc.
- `Gl.Model` : clase base que representa un modelo. Un modelo es un conjunto de "meshes", que a su vez son un conjunto de vertices, y los datos asociados a estos
- `Gl.Pencil` : esta clase implementa en concepto abstracto de lápiz. En este motor gráfico el lápiz es lo que se utiliza para dibujar las primitivas gráficas (triángulos) de cada modelo. Principalmente ayuda al rendimiento porque disminuye la cantidad de cambios de contexto necesarios para dibujar una escena. En **OpenGL** existen unos objetos llamados *array de vertices* (vertex array object) o VAO, que representan la estructura de una buffer de vertices; cambiar de VAO es generalmente costoso en tiempo, por lo que lo optimo es disponer de un VAO que se utiliza para dibujar toda la escena

  ```
  /* Pencil layout */
  layout (location = 0) in vec3 vPos;
  layout (location = 1) in vec3 vNormal;
  layout (location = 2) in vec3 vTexCoords;
  layout (location = 3) in vec3 vTangent;
  layout (location = 4) in vec3 vBitangent;

  /* ~~^~~ Esto es un VAO como puede usarse desde un shader */
  /* Cada "location" es un atributo que puede activarse en  */
  /* el VAO y corresponde a una sección de un elemento del  */
  /* array de vertices                                      */
  ```

- `Gl.PointLight` : clase que guarda los datos de una luz omnidireccional con una posición concreta (como una farola)
- `Gl.Program` : clase que encapsula un programa de shaders. Cada programa se compone de varios shaders, que juntos se envían a la GPU del ordenador y se ejecutan cada vez que se dibuja algo
- `Gl.SingleModel` : implementa un modelo básico
- `Gl.SpotLight` : clase que guarda los datos de una luz direccional con una posición concreta (como una linterna)
- `Gl.Ssbo` : encapsula un SSBO (Shader Storage Buffer Object), que es un buffer independiente del programa (`Gl.Program`) que puede accederse desde un shader. Posee la cualidad de ser re-dimensionable y modificable (desde el shader), por lo cual puede usarse como una `IList<T>` para almacenar objectos que implementen la interfaz `Gl.IPackable` y puedan ser accesibles desde el código del shader. Actualmente `Gl` usa tres de estos para almacenar los tres tipos de luces que están implementadas
- `Gl.Ubo` : encapsula un UBO (Uniform buffer object), que es una buffer independiente del programa (`Gl.Program`) que puede accederse desde un shader. Es similar al SSBO, pero es de solo lectura (desde el shader) y no se puede re-dimensionar una vez creado, pero es más rápido. Actualmente `Gl` usa uno para almacenar las matrices de transformación
- `Object` : es la base para los objetos representables con el motor gráfico
- `Skybox` : es una objecto especial que representa un "skybox", que no es mas que un modelo muy grande que contiene toda la escena y provoca la ilusión de un espacio abierto