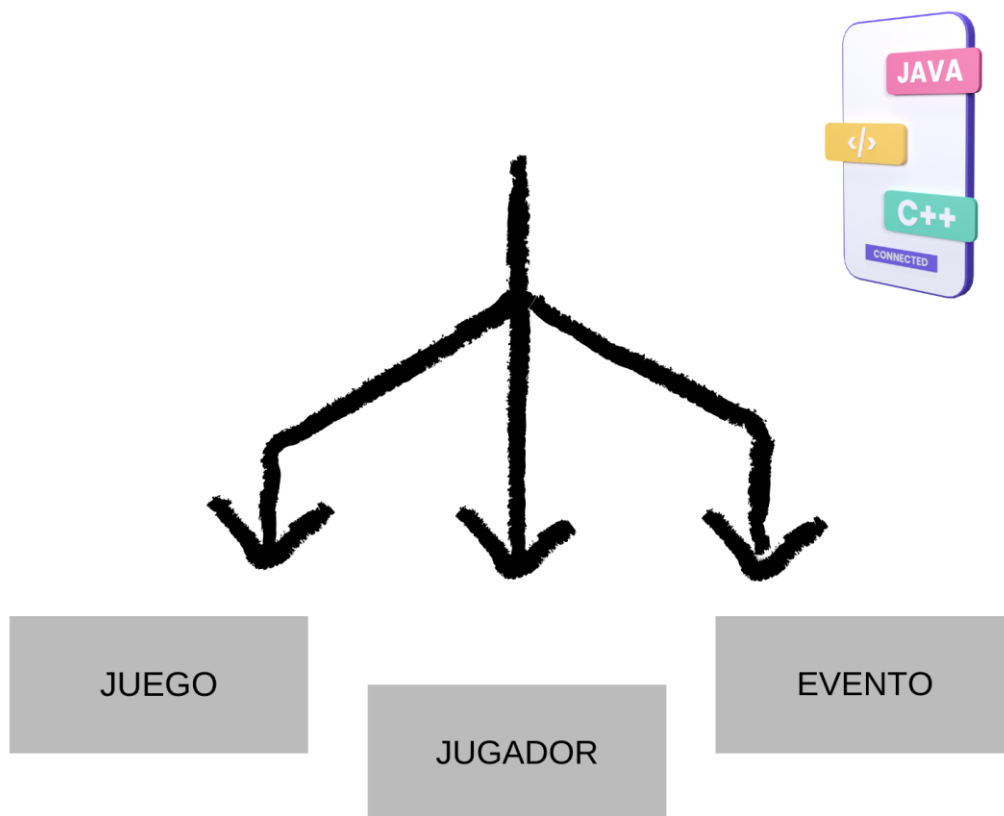


# UML JUEGOS (DIAGRAMA DE CLASES)



# INDICE

Presentación del Problema.....	Pág. 3
Clases Creadas.....	Pág. 3
Explicación de las Clases.....	Pág. 3-7
Conexión de las Clases.....	Pág. 7-11
Resumen del UML.....	Pág. 12
Como exportar a código.....	Pág. 12-14

## Presentación del problema:

En nuestro caso se nos pidió hacer un UML para una aplicación para organizar quedadas para jugar a juegos de mesa principalmente. Además, esta aplicación debe tener un apartado para vender juegos dentro de la misma, donde un jugador ponga en venta su juego y los demás puedan enviarle diferentes ofertas a este vendedor y este tendrá el control de aceptar o negar la oferta.

Debemos de saber que además un jugador puede ser administrador y es el que se encarga del mantenimiento de todo.

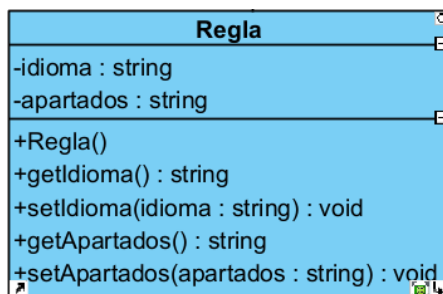
## Clases creadas:

Hemos decidido hacer la creación de 6 clases para poder manejar todos los errores y relaciones entre ellos posibles.

1. Juego -> Nos enfocamos en los juegos con todos sus datos y métodos
2. Regla -> Es una clase simple que lo que hace es complementar a Juego
3. Jugador -> Una de las clases principales en la cual se basa casi todo el funcionamiento de la aplicación.
4. Evento -> Otra clase principal, ya que es la principal función de la app, la de crear eventos para poder jugar con personas
5. Mercadillo -> Una clase que nos permite añadir una funcionalidad extra a la aplicación, para poder vender juegos dentro de la app
6. Venta -> Por último, esta clase que nos permite guardar todos los datos de la venta de un libro en el mercadillo.

## Explicación de las clases:

1º Regla: Para esta clase sencilla hemos decidido ponerle solo dos atributos, por un lado, el idioma de las reglas y por otro los apartados (ya que creemos que una regla en específico puede tener diferentes apartados dentro de ella), por otro lado, tenemos el constructor predefinido donde inicializamos los atributos y los getters y setters.



2º Evento: Para esta clase hemos puesto diferentes tipos de atributos, como puede ser el id del evento, el día, la hora, el sitio en donde se va a jugar, el organizador del evento, una lista de todos los jugadores y un boolean para saber si está lleno o no. Por otro lado, en los

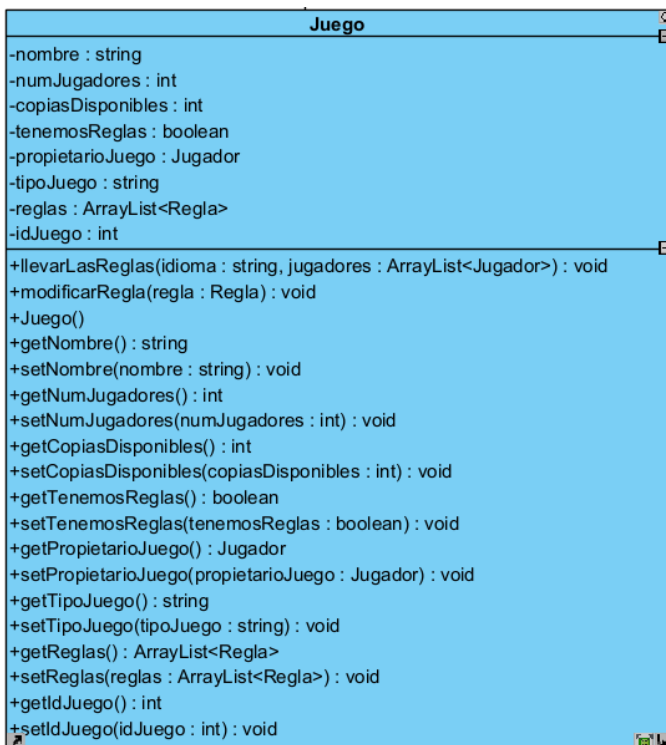
métodos tenemos a parte de los getters, setters y el constructor predefinido, varios métodos particulares como pueden ser tieneJuego que es un método que devuelve verdadero o falso basándose en si el organizador tiene o no al menos un juego, ya que se nos dice que solo pueden organizar eventos aquellos jugadores que tengan juegos, también tenemos un método para enviar una notificación a aquellos jugadores que tengan interés en juegos parecidos al que vamos a jugar en cierto evento, esto para que los jugadores puedan descubrir nuevos juegos y estar al día de los eventos relacionados con todos sus interés.

Evento
-idEvento : int -organizador : Jugador -juego : Juego -ubicacion : string -fecha : date -hora : double -jugadoresActuales : ArrayList<Jugador> -completoONo : boolean
+tieneJuego(jugador : Jugador) : boolean +enviarNotificacion(interesados : ArrayList<Jugador>, juego : Juego) : void +Evento() +getIdEvento() : int +setIdEvento(idEvento : int) : void +getOrganizador() : Jugador +setOrganizador(organizador : Jugador) : void +getJuego() : Juego +setJuego(juego : Juego) : void +getUbicacion() : string +setUbicacion(ubicacion : string) : void +getFecha() : date +setFecha(fecha : date) : void +getHora() : double +setHora(hora : double) : void +getJugadoresActuales() : ArrayList<Jugador> +setJugadoresActuales(jugadoresActuales : ArrayList<Jugador>) : void +getCompletoONo() : boolean +setCompletoONo(completoONo : boolean) : void

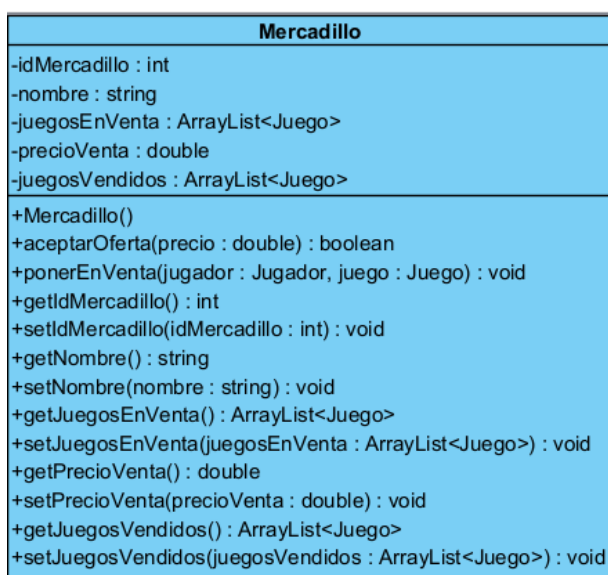
3º Jugador: En cuanto a esta clase tenemos atributos varios como el nombre, apellidos (información principal) y luego tenemos dos listas, una con todos los juegos de los cuales es propietario dicho jugador y otra con los intereses que esté tiene (cardgame, war..) y un boolean para saber si es administrador o no. En métodos, como en los anteriores tenemos los getters, setters y constructor predefinido y en este caso tenemos más métodos específicos, como puede ser un método para poder recibir las notificaciones que nos mandan los eventos a aquellos jugadores interesados en esos tipos de juegos, uno para crear un nuevo evento con toda la información necesaria, un método mediante el cual confirmamos nuestra asistencia a un evento en específico y por último, un método que nos permite enviar una oferta a un juego que esté disponible en el mercadillo. Nosotros también tenemos otro método el cuál es un boolean que se compenetra con una variable para saber si el jugador es administrador, en caso de que sea este jugador tiene unas tareas de mantenimiento de juegos, jugadores... En caso de que no, no pasa nada, el jugador sería normal y no tendría tareas extras

Jugador
-nombre : string -apellidos : string -telefono : int -gmail : string -interesesJuegos : ArrayList<Juego> -juegosQueTiene : ArrayList<Juego> -idJugador : int -esAdministrador : boolean
+enviarOferta(Juego : juego, double : oferta) : void +confirmarAsistenciaEvento(Evento : event) : void +crearEvento(creador : Jugador, juego : Juego, ubicacion : string, fecha : date, hora : double) : Evento +recibirNotificacion(event : Evento) : void +Jugador() +getNombre() : string +setNombre(nombre : string) : void +getApellidos() : string +setApellidos(apellidos : string) : void +getTelefono() : int +setTelefono(telefono : int) : void +getGmail() : string +setGmail(gmail : string) : void +getInteresesJuegos() : ArrayList<Juego> +setInteresesJuegos(interesesJuegos : ArrayList<Juego>) : void +getJuegosQueTiene() : ArrayList<Juego> +setJuegosQueTiene(juegosQueTiene : ArrayList<Juego>) : void +getIdJugador() : int +setIdJugador(idJugador : int) : void +getEsAdministrador() : boolean +setEsAdministrador(esAdministrador : boolean) : void

4º Juego: Analizando la clase de Juego de la cuál disponemos, tenemos algunas variables como puede ser el nombre, el número de jugadores, las reglas, el identificador, si tenemos reglas propias o no, el tipo de juego, etc. Por otro lado, tenemos los métodos en donde disponemos de el constructor predefinido, los getters y setters y luego hemos añadido dos métodos particulares, que son “llevarLasReglas” mediante el cual establecemos que jugadores se van a llevar las reglas y el idioma de las reglas que se van a llevar.



5º Mercadillo: En cuanto al mercadillo, como en otras clases, tenemos el identificador, el nombre, una lista con los juegos que tenemos en venta, otra lista con los juegos que ya se han vendido. Además, disponemos de los getters y setters, del constructor predefinido y de otros métodos como pueden ser el método para aceptar una oferta en donde enviamos el precio y otro para poner en venta un juego, donde enviamos el juego que queremos vender y el jugador que lo vende.

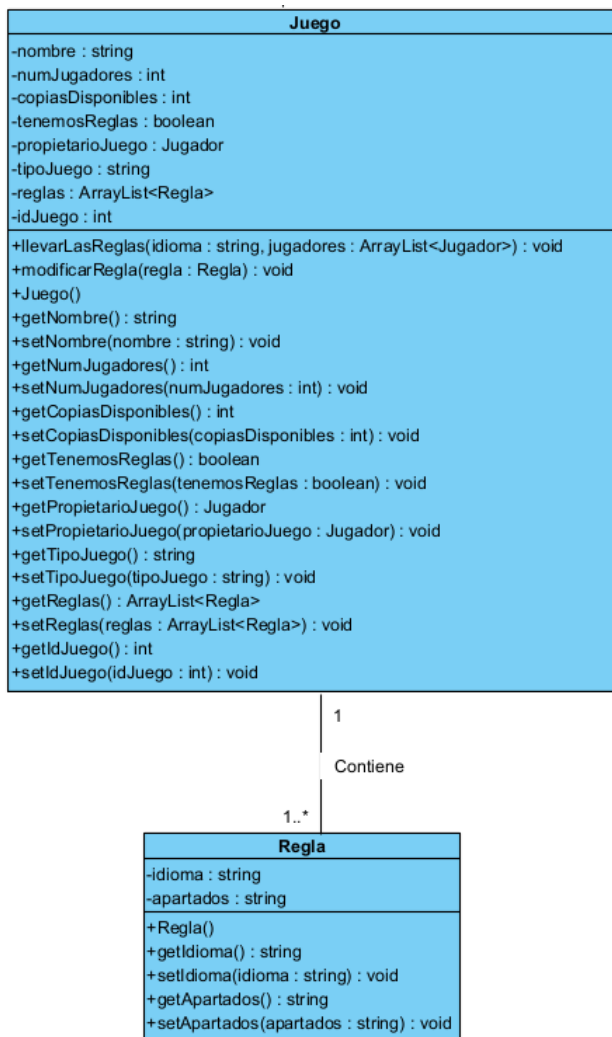


6º Venta: Para terminar un poco todas las clases explicadas por encima, tenemos la clase Venta en donde vamos a guardar más que nada la mayoría de datos de cuando un juego se vende, como puede ser el identificador de la venta, el juego, el jugador que lo vende, el jugador que lo compra, una lista de todas las ofertas que se han enviado a ese juego, etc. Por otro lado, en métodos tenemos en este caso solo los getters y setters y el constructor predefinido.

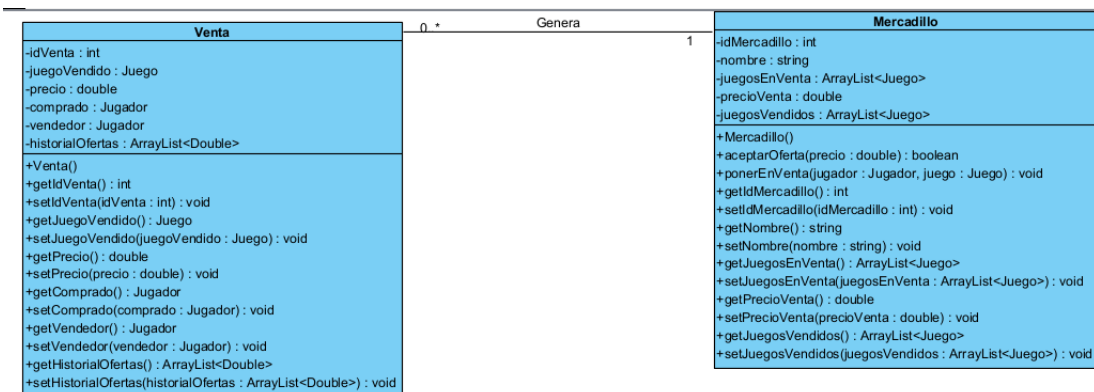
Venta
-idVenta : int -juegoVendido : Juego -precio : double -comprado : Jugador -vendedor : Jugador -historialOfertas : ArrayList<Double>
+Venta() +getIdVenta() : int +setIdVenta(idVenta : int) : void +getJuegoVendido() : Juego +setJuegoVendido(juegoVendido : Juego) : void +getPrecio() : double +setPrecio(precio : double) : void +getComprado() : Jugador +setComprado(comprado : Jugador) : void +getVendedor() : Jugador +setVendedor(vendedor : Jugador) : void +getHistorialOfertas() : ArrayList<Double> +setHistorialOfertas(historialOfertas : ArrayList<Double>) : void

### Conexiones de las clases:

-Primero que todo, tenemos la relación entre reglas y Juego, está relación es una asociación en donde la fuerte es Juego y la multiplicidad es así, ya que nosotros estimamos que un Juego puede tener de una a varias Reglas, mientras que una Regla, aunque sea parecida a la de otro juego, es única y exclusivamente para un Juego.

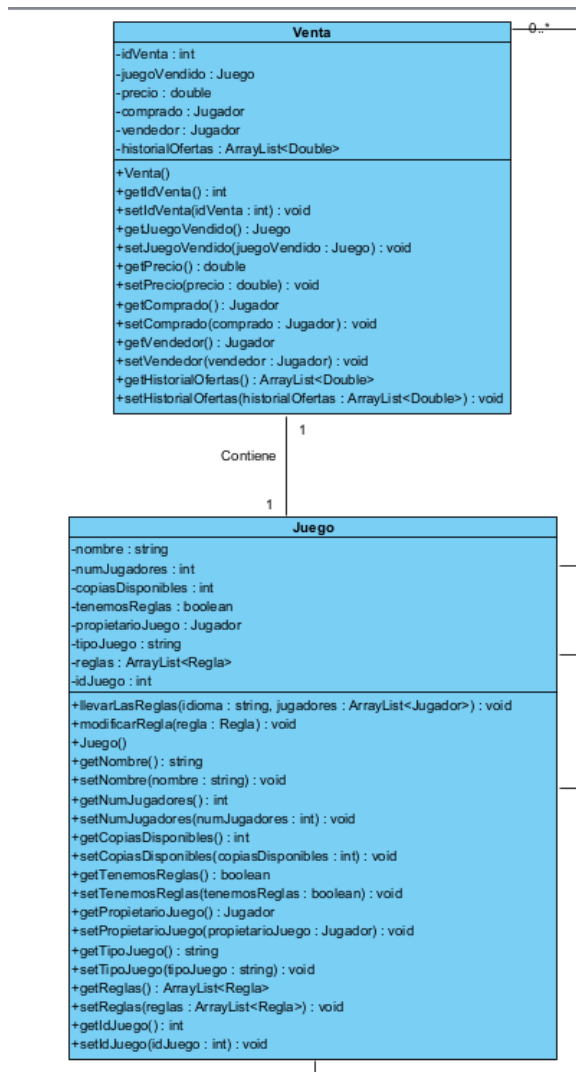


-Posteriormente, tenemos la relación entre Venta y Mercadillo, en donde tenemos otra relación de asociación en donde la fuerte es en Mercadillo y tenemos esa multiplicidad ya que, solo puede haber un mercadillo, pero para ese mercadillo puede haber de 0 a varias ventas, ya que nosotros estimamos que puede ser que aún no se ha vendido ningún juego.





-Siguiendo con Venta, la tenemos también relacionada con Juego en una relación de asociación en donde la clase fuerte es Juego, hablando de la multiplicidad tenemos en ambos lados un 1 ya que una venta es exclusivamente para un juego y viceversa.

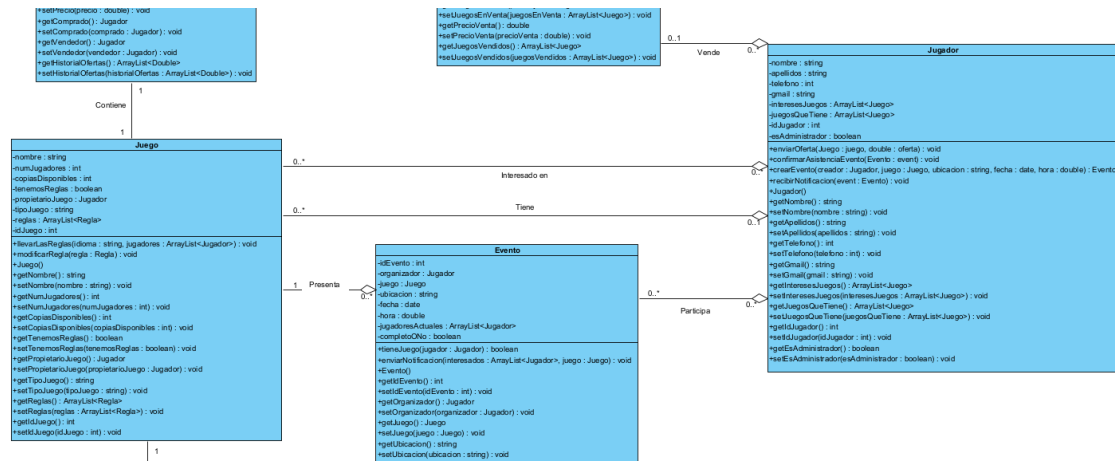


-Aquí tenemos una especie de mezcla, en donde podemos ver como se relacionan Juego, Jugador y Evento:

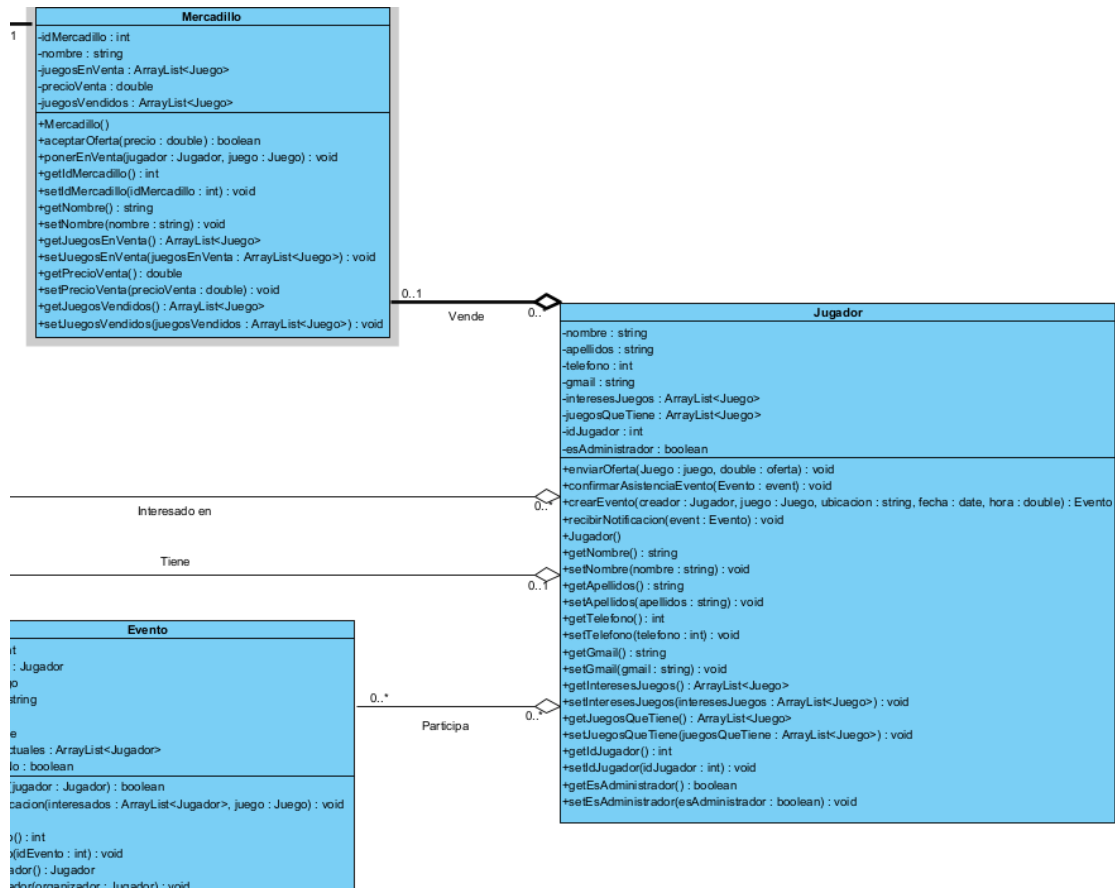
1. Empezamos por Juego y Jugador: Jugador y Juego están relacionados con una relación de agregación (Tiene) en donde la clase fuerte es Jugador, en cuanto a la multiplicidad tenemos que un juego puede ser de nadie o de un jugador (ya que estipulamos que son juegos únicos con sus identificadores únicos), por otro lado, un jugador puede tener ningún a varios juegos.
2. Empezamos por Juego y Jugador: Jugador y Juego están relacionados con una relación de agregación (Interesado en) en donde podemos ver que de ningún a varios jugadores pueden estar interesado en un juego en especial, y que un jugador puede estar interesado en ningún o varios juegos.
3. En cuanto a Evento con Juego, tenemos otra relación de agregación donde la clase fuerte es Evento y vemos que un juego puede estar presente en varios eventos a lo

largo del tiempo, pero para un evento solo podemos tener un juego (hemos estipulado que en los eventos solo se pueda jugar uno y no a varios).

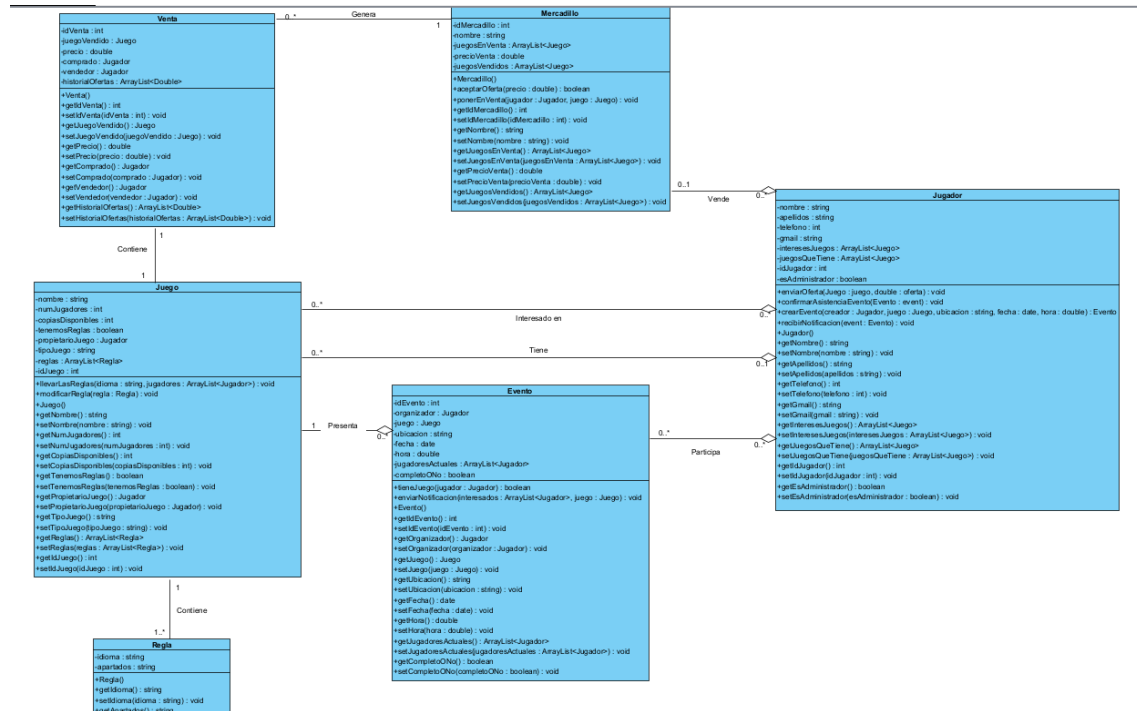
- Para terminar, está parte, tenemos la relación de Evento con Jugador, en donde tenemos una relación de agregación con la fuerte como jugador y en cuanto a la multiplicidad, tenemos que un evento puede tener de ningún a varios jugadores apuntados (ya que puede que nadie se haya apuntado aún) y un jugador puede estar apuntado a ningún o varios eventos a la vez.



-Por último, tenemos la relación entre Mercadillo y Jugador: cómo podemos ver en la imagen de abajo es una relación de agregación en donde la clase fuerte es jugador y vemos que, al solo haber un mercadillo, un jugador solo puede estar asociado a un mercadillo, donde por el otro lado el mercadillo puede tener de ningún a varios jugadores.



El UML completo sería como podemos ver en la siguiente imagen:



## Resumen del UML:

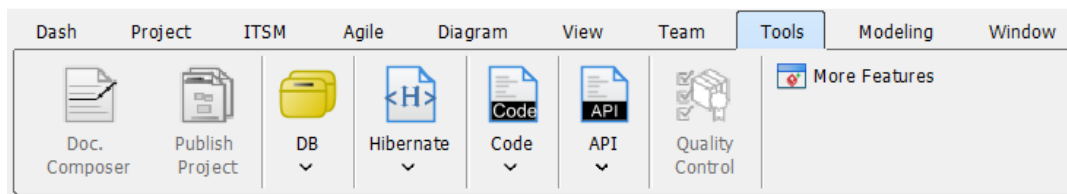
Desde nuestro punto de vista, sí que cubrimos todas las necesidades que nos exigió nuestro jefe a la hora de desarrollar la app para juegos de mesa. Pensamos esto, ya que un jugador puede tener o estar interesado en un juego a la vez, los juegos tienen reglas básicas, y un evento está relacionado tanto con juego como con jugador, ya que ambas cosas son necesarias para que exista el evento.

Un jugador también puede vender un juego en el mercadillo y una vez que lo venda se guarda como un ticket (Venta) con varios datos acerca de esa venta, el jugador también puede estar interesado en comprar un juego y por eso puede enviar ofertas a un juego X.

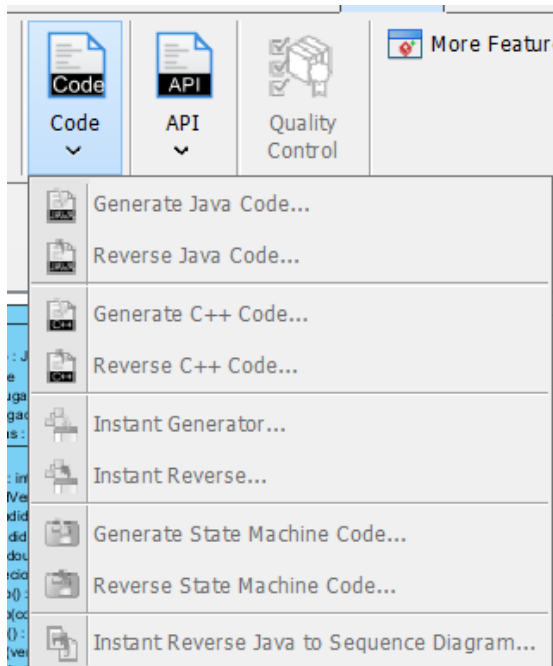
## Exportar Clases a Código:

Una vez terminado todo nuestro esquema UML de la App para juegos de mesa, podemos hacer, gracias a Visual Paradigm el código automático de estas clases y sus relaciones ¿Como haríamos esto? Primero que todo debemos descargarnos la versión de prueba del pro.

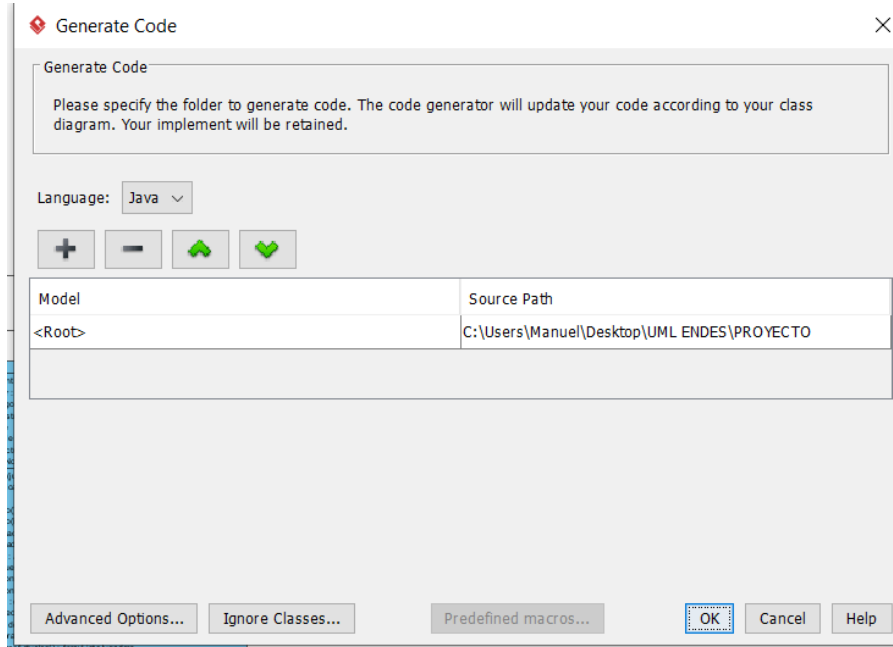
Una vez descargada, cargamos el proyecto y hemos de irnos a la parte donde pone tools:



Una vez en esta pestaña, deberemos de tocar sobre donde pone "Code":

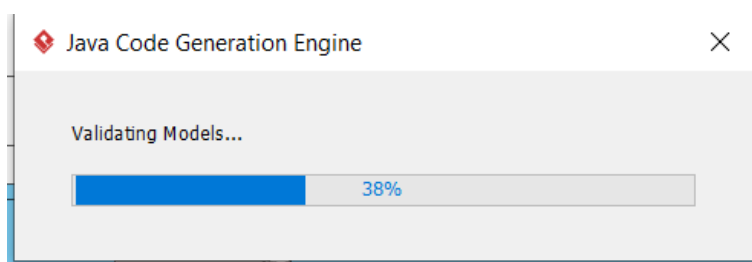


Una vez hayamos dado a Code, deberemos de dar al botón de Generate Java Code y nos abrirá una nueva pestaña:



















En la imagen de arriba, deberemos de seleccionar el lenguaje que nosotros queramos generar el código (en nuestro caso, será en Java), y luego un sitio en donde vamos a guardar las clases, en mi caso, en la carpeta del proyecto del trabajo, una vez hecho, le damos a “Ok”.

Nos aparecerá un cartel parecido al de abajo:



Una vez termine de cargar, sí no, nos sale ningún error en las clases, nos iremos a la dirección donde dijimos que queríamos guardarlo y le entraremos y cómo podemos ver, las clases se han creado correctamente y si entramos tienen todos sus atributos y métodos correspondientes.

Nombre	Fecha de modificación	Tipo	Tamaño
 .PROYECTOENDES.vpp.lck	05/05/2024 23:43	Archivo LCK	0 KB
 Evento	05/05/2024 23:47	Archivo de origen ...	3 KB
 Juego	05/05/2024 23:47	Archivo de origen ...	3 KB
 Jugador	05/05/2024 23:47	Archivo de origen ...	3 KB
 Mercadillo	05/05/2024 23:47	Archivo de origen ...	2 KB
 PROYECTOENDES	05/05/2024 23:47	Visual Paradigm Pr...	908 KB
 PROYECTOENDES.vpp.bak_052d	25/04/2024 10:32	Archivo BAK_052D	260 KB
 PROYECTOENDES.vpp.bak_053d	25/04/2024 18:06	Archivo BAK_053D	180 KB
 PROYECTOENDES.vpp.bak_054d	25/04/2024 20:46	Archivo BAK_054D	180 KB
 PROYECTOENDES.vpp.bak_055d	25/04/2024 20:57	Archivo BAK_055D	180 KB
 PROYECTOENDES.vpp.bak_056d	25/04/2024 21:02	Archivo BAK_056D	180 KB
 PROYECTOENDES.vpp.bak_057d	25/04/2024 21:15	Archivo BAK_057D	212 KB
 PROYECTOENDES.vpp.bak_058d	25/04/2024 21:21	Archivo BAK_058D	196 KB
 PROYECTOENDES.vpp.bak_059d	26/04/2024 22:18	Archivo BAK_059D	440 KB
 PROYECTOENDES.vpp.bak_060d	27/04/2024 10:55	Archivo BAK_060D	276 KB
 PROYECTOENDES.vpp.bak_061d	27/04/2024 10:55	Archivo BAK_061D	216 KB