MULTIJUGADOR EN RED

ÍNDICE

1. INTRODUCCIÓN

Para incorporar a nuestro proyecto la parte de multijugador en red, hemos seguido la arquitectura Cliente-Servidor.

La arquitectura Cliente-Servidor es un modelo de diseño de software en el que las tareas se reparten entre el servidor y el cliente. En nuestro caso, hemos considerado el sistema de la siguiente forma:

El servidor es un proveedor de servicios y atiende a las demandas de los clientes, por tanto en él reside el modelo de nuestro proyecto (la lógica del juego).

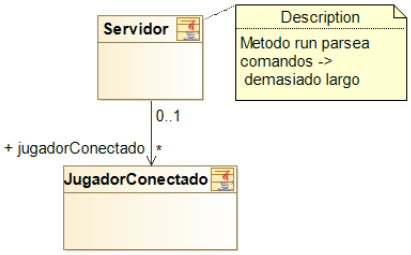
El cliente es un demandante de servicios, en él reside la vista (la interfaz del juego).

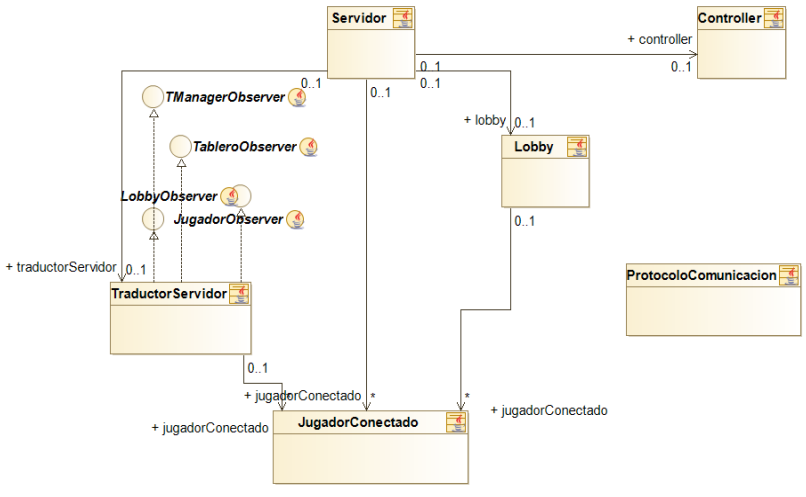
Para comunicar información entre el cliente y el servidor hemos considerado el uso de sockets.

2. CLASES/INTERFACES AÑADIDAS

i. Servidor

Version 1.0. En esta primera versión se consideró únicamente crear una clase servidor y JugadorConectado. Pero, debido a la complejidad del proyecto y el numero de comandos distintos que se pueden pasar por el socket, inmediatamente se amplió a la Version 2.0.





Version 2.0. Surgida como una ampliación de la primera, se crearon las clases Lobby, LobbyObserver, Server, Protocolo Comunicación y TraductorServidor. Con respecto al diseño, se recurrió al mismo **patrón observador** para notificar los cambios del Lobby a la vista y que se actualice autómaticamente. Ya que hay que enviar estas notificaciones al cliente, el TraductorServidor se encargaba de implementar las interfaces de observadores del modelo y enviaba por el socket la información. Este mismo traductor también disponía de un método para traducir los mensajes que llegaban

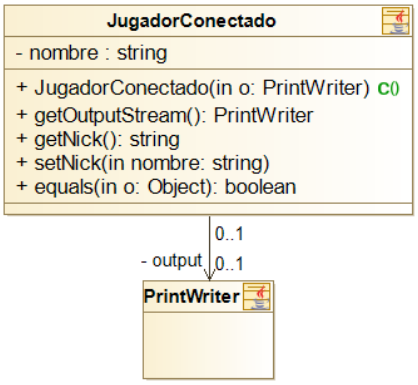
por el socket al servidor. Como se puede apreciar este traductor estaba acumulando una gran variedad de distintas funcionalidades y era una clase demasiado grande.

Última versión: Surgió con la necesidad de facilitar este parseo de mensajes que llegan por el socket. Nos dimos cuenta de que esto con una estructura jerárquica que utilice el **patrón comando** se resolvía muy fácilmente, ya que delega el parseo en las propias clases de los comandos y te devuelve esa información encapsulada en un objeto. Entonces, creamos los CommandLobby ya que los objectos que necesitaban para ejecutarse eran radicalmente distintos a los Command que ya utilizábamos. Entonces nos encontramos con un problema y era que teníamos dos tipos de comandos a ejecutar y otro tipo de acciones que dependían del estado en que se encontraba el cliente(Lobby,Game,Server).

Así, aplicando el **patrón estado,** creamos una estructura que se encargar de ejecutar las acciones correspondientes dependiendo del estado en que se encuentra el servidor.

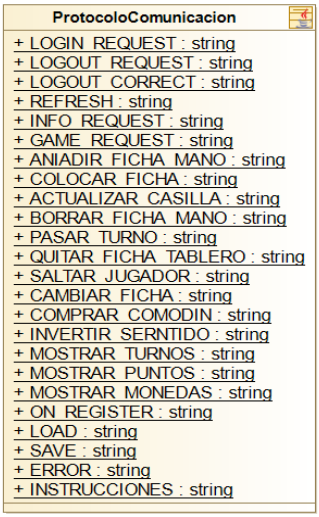
JUGADORCONECTADO

Esta clase almacena en sus atributos el nick del jugador y la salida del socket del cliente. Su constructor tiene un único argumento, la salida. Tiene métodos setter (para establecer el nick) y getter (de la salida y del nick). También sobreescribe el método equals para que dos jugadores conectados sean iguales si lo son sus nicks con el fin de evitar jugadores con el mismo Nick.



PROTOCOLOCOMUNICACION

Esta clase es de tipo **final**. Contiene una amplia lista de ordenes/peticiones **constantes** que se intercambian entre el cliente y el servidor para hacer más fácil la comunicación entre ambas, ya que los traductores utilizarán constantes para descifrar el tipo de mensaje, y así podremos evitar ciertos errores al parsearlos.

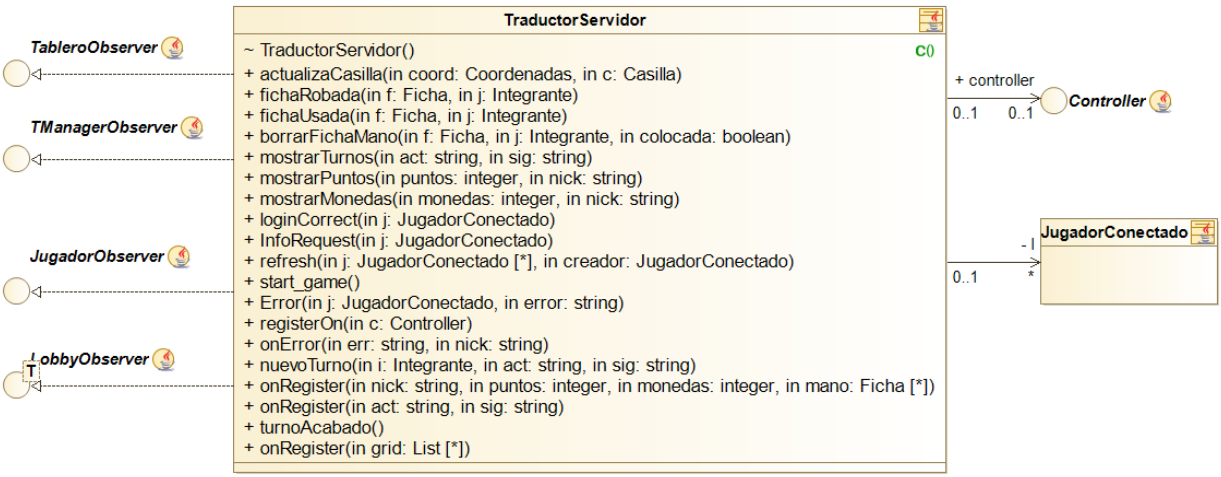


TRADUCTORSERVIDOR

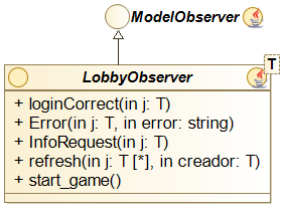
Esta clase es una clase especializada en enviar los mensajes correspondientes a los clientes.

Entre sus atributos tenemos una lista de jugadores conectados y el controller. El Controller lo utilizamos para que se registre a su lista de observadores.

Para enviar mensajes al cliente, el traductor implementa las interfaces TableroObserver, TManagerObserver, JugadorObserver y LobbyObserver<JugadorConectado>. Para ello crea un objecto CommandClient con la información correspondiente al tipo dinámico del comando y lo envía por el socket.



LOBBYOBSERVER

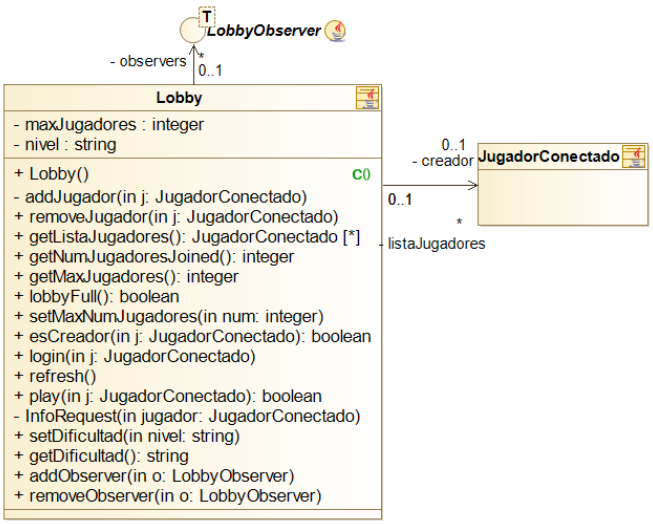


Esta interfaz forma parte de los observadores del patrón **Observador** que se encuentra en nuestra aplicación, por tanto implementa ModelObserver. En este caso se refiere a los observadores del **Lobby**. Así, tiene los métodos necesarios para que el Lobby pueda informar de sus cambios al cliente (a través del traductor). Se trata de una clase Genérica, esto tiene el sentido de que en la parte del servidor, la T corresponderá a un JugadorConectado y en el cliente String.

LOBBY

Como quiere notificar sus cambios a la vista, implementa Observable<LobbyObserver<JugadoConectado>>.

Entre sus **atributos** tenemos: el creador del lobby (jugador que se ha conectado primero al lobby); una lista de jugadores conectados al lobby (esta lista puede ser distinta a la de jugadores conectados al servidor, puesto que puede haber jugadores conectados al servidor y no al lobby); lista de LobbyObserver (para informar de los cambios al cliente); el número máximo de jugadores que pueden acceder al lobby (aunque se inicializa en un principio para evitar errores, posteriormente su valor es cambiado por el creador del lobby) y por último el nivel de dificultad de las IAs (introducido también por el creador).



Su **constructora** inicializa las listas y el número máximo de jugadores a 1, para evitar que se añadan jugadores al lobby antes de que el creador haya elegido el nivel y el máximo de jugadores.

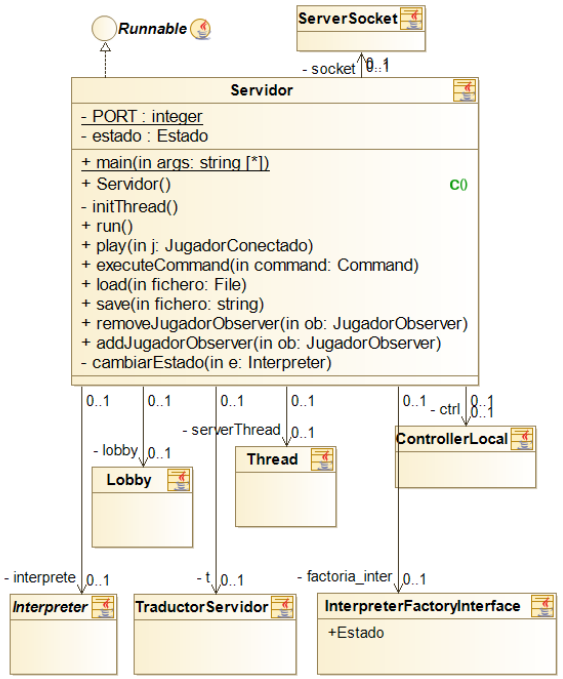
Esta clase contiene métodos getter (la lista de jugadores, número de jugadores y el nivel), setter (el número de jugadores, el nivel); métodos para añadir jugadores a la lista (si el lobby todavía no tiene ningún jugador, entonces ese jugador se establece como creador, si el lobby estuviera lleno informarle de ello y si no lo estuviera, de que su login ha sido correcto); para eliminar jugadores a la lista (teniendo en cuenta que si se elimina al creador, el puesto pasa al siguiente en la lista).

Los métodos más interesantes son el de **login** en el lobby (si el jugador ya está le informa de error, si no, le añade a la lista), **refresh** (informa a los clientes de la nueva lista de jugadores conectados al lobby), y **play** (que checkea que el jugador que quiere comenzar el juego sea el creador del lobby e informa de si pudieran comenzar o no el juego).

SERVIDOR

Hace de servidor de nuestra aplicación. Entre sus atributos encontramos el **socket** (serverSocket), que es el que se encarga de aceptar las conexiones de los clientes, el ServerState para conocer su estado, una instancia del Interpreter que variará dependiendo del estado y el modelo.

Esta clase implementa la interfaz **Runnable**, de esta manera sobreescribe el método **run**(). Este método se va a encargar de la conexión con los clientes.



El proceso siempre es el mismo, espero a que un cliente contacte con el servidor, cuando lo hace, me guardo su Socket. Entonces, creo una nueva hebra para atender sus peticiones durante toda la partida (de esta manera puede atender a varios clientes a la vez). En esta nueva hebra, a partir del socket del cliente me guardo su entrada y su salida. Así, creo una instancia de JugadorConectado al que le paso la salida del cliente para procesos posteriores en que nos hará falta.

A continuación, mientras que el cliente está conectado, leo de su entrada cuando me envíe un mensaje y si este mensaje no es nulo, compruebo con el estado, que tengo el intérprete correspondiente a la orden que se me pasa. Y entonces llamo al execute del interprete con el mensaje para que los descodifique y realice las acciones pertinentes. Si el mensaje fuera nulo, el cliente se desconectó por algún error y por tanto yo también lo desconecto del servidor.

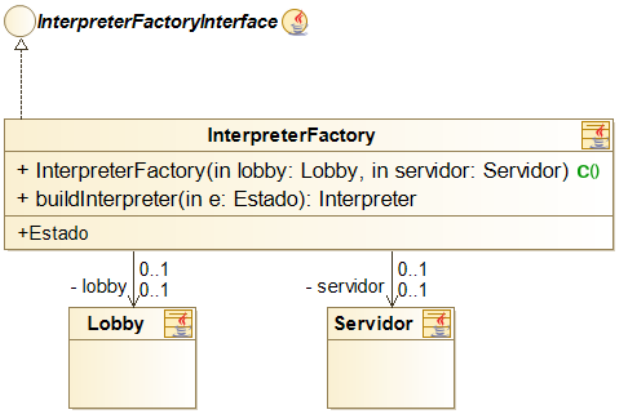
FACTORIA DE INTERPRETES

Con respecto a su diseño, se trata de **un patrón Factory Method.**

El uso de este patrón, nos permite crear objetos de tipo Interprete ocultado los detalles del creación del objeto al Servidor. La factoría relega la responsabilidad de elegir el tipo a instanciar a su método de creación (buildInterpreter en este caso). Lo vimos conveniente ya que la clase dinámica de Interprete puede cambiar en tiempo de ejecución.

Por un lado encontramos la interfaz InterpreterFactoryInterface : Proporciona una interfaz para la creación de intérpretes.

Por otro, la InterpreterFactory: Implementa InterpreterFactoryInterface. Es una clase que se encargara de crear el intérprete correspondiente dependiendo del estado del servidor en su método buildInterpreter. Esta clase encapsula la lógica para construir cada uno de los intérpretes estado.



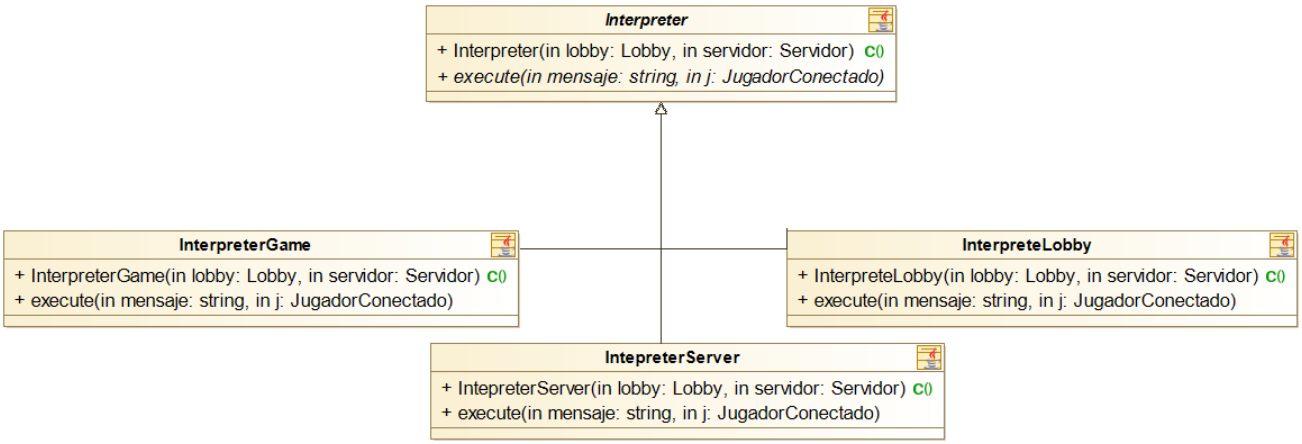
INTERPRETER

Sirviéndonos del **patrón Estado,** vamos a poder parsear y ejecutar los mensajes que llegan por el socket de manera distinta dependiendo del estado en que se encuentre el servidor. Cada estado va a ser representado por una clase y cada una de ellas implementa su método execute de maneras distintas.

Clase abstracta utilizada para parsear los comandos que llegan por el socket. Utilizando **el patrón estado,** de ella heredan 3 tipos de interpreter:

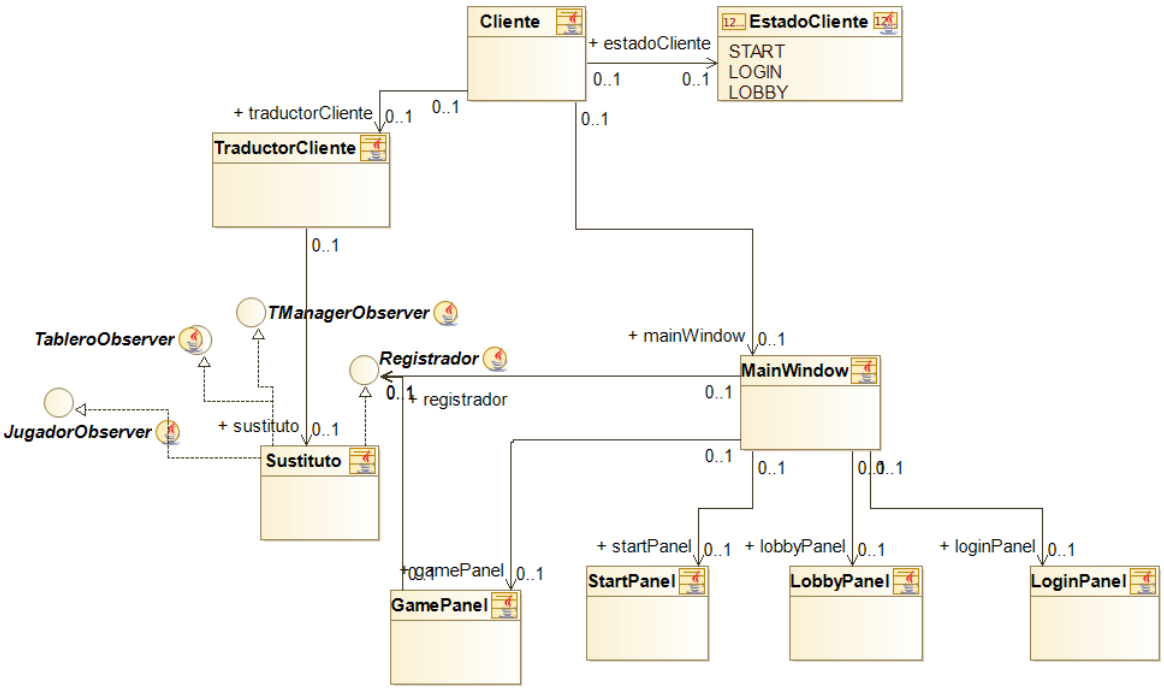
- InterpreterGame
- InterpreterServer
- InterpreterLobby

Tiene un método estático, tal que dado un ServerState devuelve la instancia correspondiente del



ii. Cliente

Versión 1.0: En la primera version del cliente, hicimos la reestructuración del Controller, creando las clases registrador (actual Controller) que la implementaba el Controller del modelo y la clase Sustituto. También teníamos un traductor para interpretar las ordenes que nos llegaban por el socket. En la siguiente imagen se puede apreciar la estructura.



Última versión: Igual que habíamos hecho en el servidor, quisimos eliminar la clase que parseara los mensajes y hacerlo con un **patrón comando** y así abstraemos la acción de interpretar del cliente. En esta parte no nos hace falta la fabrica puesto que no recibimos distintos comandos dependiendo del algún estado, son todos CommandClient.

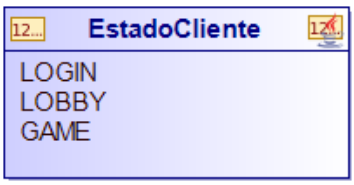
Una vez que hicimos esto, creamos un controlador para el cliente. Así, creamos también la interfaz Controller para registrarse como observador y poder ejecutar comandos (el antiguo registrador). Esta interfaz es la reciben como parámetro los distintos paneles en lugar del controlador y la implementan el controllerLocal y el controllerCliente.

La clase sustituto igualmente era inmensa y tenía muchísimas funciones distintas. De esta manera creamos una clase TableroClient, TManagerClient, JugadorClient y LobbyClient, cada una implementa la interfaz correspondiente del modelo. Son el sustituto del modelo en el cliente. Por tanto, se las pasa como modelo al constructor del controllerCliente y será en el encargado de interactuar con ellas cuando alguna vista quiera actualizar el modelo, o desde el servidor se quiera ejecutar un comando. En toda esta parte también se puede apreciar el patrón MVC.

ESTADOCLIENTE

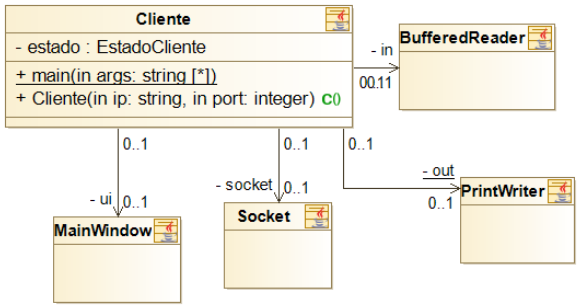
Enumerado para conocer el estado del cliente.

LOGIN-> El jugador se está logueando
LOBBY-> El jugador se encuentra en el lobby
GAME -> El jugador está en una partida



CLIENTE

Esta clase es el **cliente** de nuestra aplicación.



Entre sus **atributos** encontramos el socket del cliente, su salida (PrintWriter), su entrada (BufferedReader), su estado (EstadoCliente) y su GUI.

Su **constructora** inicializa su estado al estado inicial (LOGIN), inicializa su socket con el puerto y la ip del servidor (localhost), su entrada y salida.

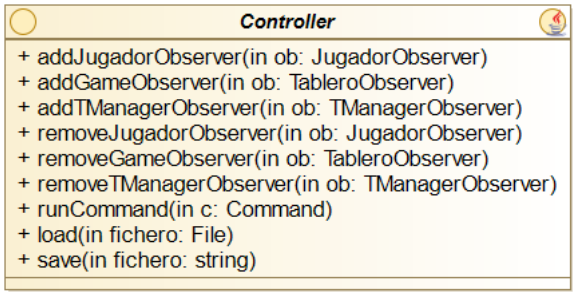
Este también crea el controllerCliente y a continuación se lo pasa a la constructora de la interfaz del cliente (la MainWindow).

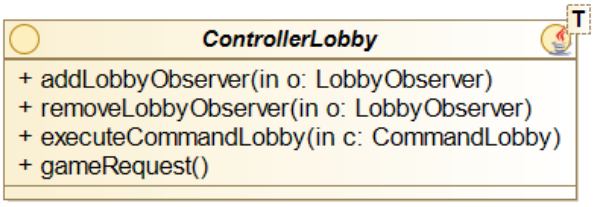
A continuación, while (true) lee de su entrada el mensaje que le haya enviado el servidor y lo manda a traducir al traductor.

CONTROLLERS

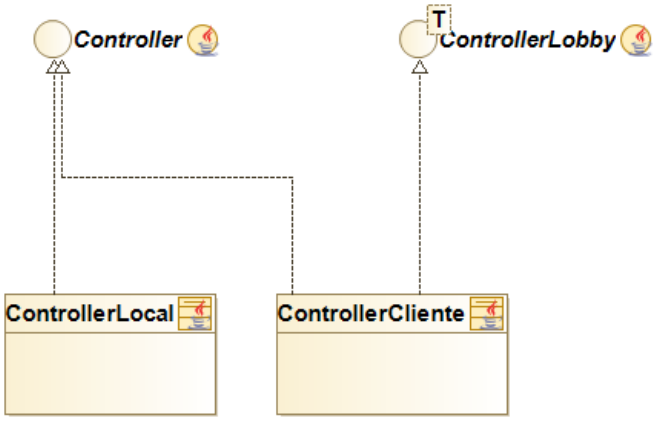
El hecho de crear la interfaz Controller surgió de que en la vista lo utilizabamos para ejecutar comandos. Ante la falta de un controlador en la parte del cliente, se pensó en crear uno en esta zona de la aplicación. Primero creamos el sustituto, que implementaba esta interfaz pero luego, se eliminó y se reestructuraron los controladores, creando el ControlleLocal y el ControllerCliente.

A continuación encontramos las interfaces que contienen los métodos que utiliza la vista para comunicarle información al modelo.

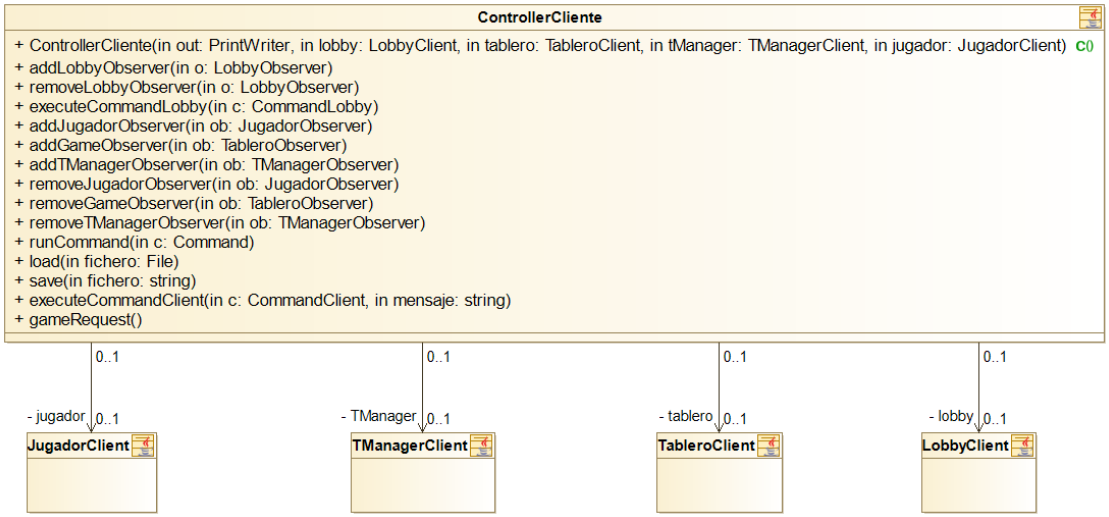




Finalmente disponemos de dos interfaces de controlador y dos clases controladores.



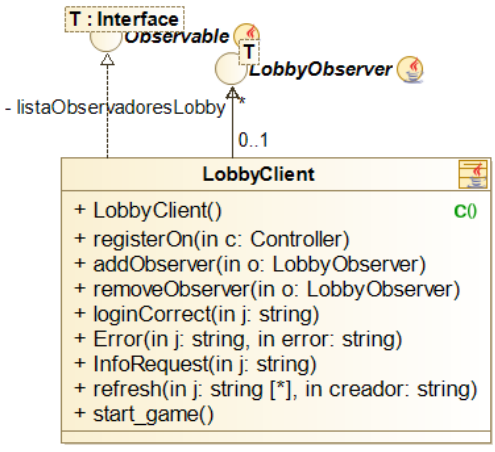
El ControllerLocal sigue siendo igual que el antiguo Controller. El Controller Cliente implementa por una parte la interfaz controller y por otra controllerLobby, este último le da acceso a poder executar comandos del Cliente.



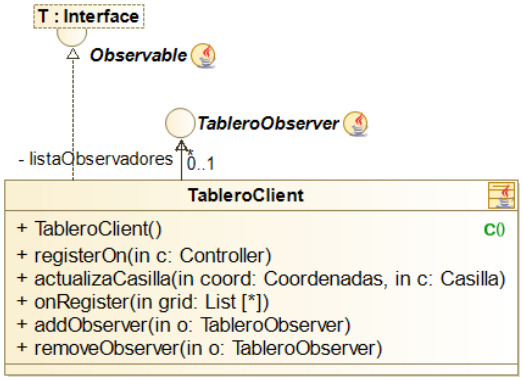
LOBBYCLIENT

Esta clase implementa LobbyObserver y Observable<LobbyObserver<String>>.

Contiene una lista de los observables de él y por tanto del Lobby. Cuando los comandos quieren ejecutar una acción sobre él, este les pasa la misma información a sus observadores.



TABLEROCLIENT



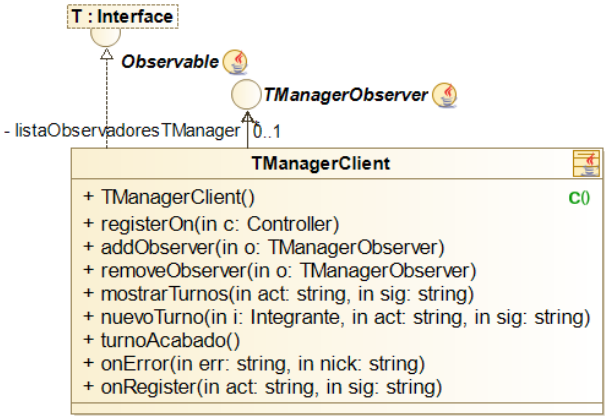
Esta clase implementa TableroObserver y Observable <TableroObserver>.

Contiene una lista de los observables de él y por tanto del tablero. Cuando los comandos quieren ejecutar una acción sobre él, este les pasa la misma información a sus observadores.

TMANAGERCLIENT

Esta clase implementa TManagerObserver y Observable <TManagerObserver >.

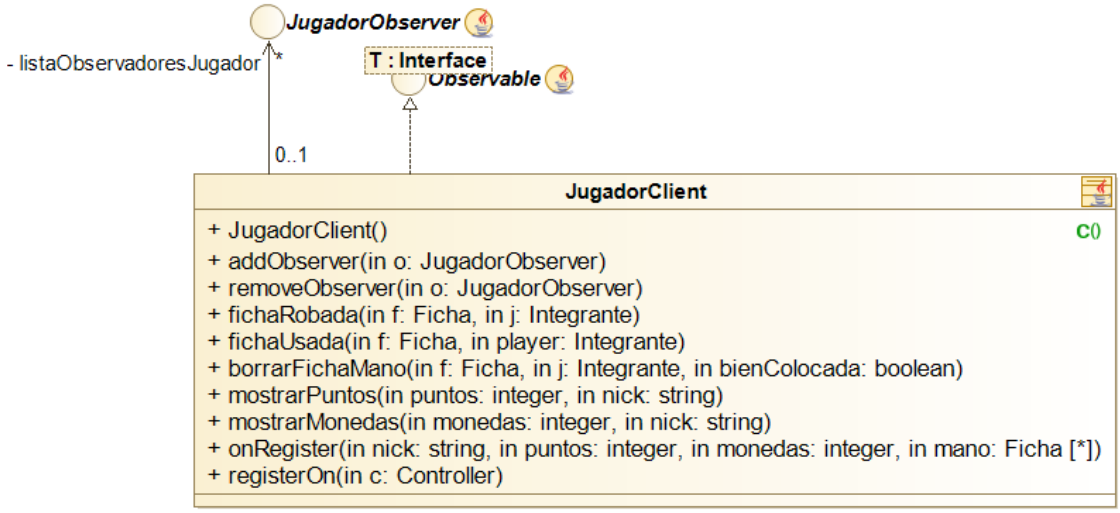
Contiene una lista de los observables de él y por tanto del Administrador de turnos. Cuando los comandos quieren ejecutar una acción sobre él, este les pasa la misma información a sus observadores.



JUGADORCLIENT

Esta clase implementa JugadorObserver y Observable < JugadorObserver .

Contiene una lista de los observables de él y por tanto de un jugador. Cuando los comandos quieren ejecutar una acción sobre él, este les pasa la misma información a sus observadores.



iii. Vista

Para la vista, se han añadido los siguientes paneles:

1. LoginPanel: Panel de registro. En este panel se le solicita al usuario su nick y se le notifica de algún error de registro.
2. LobbyPanel: Panel del lobby. En este panel se muestra el creador del lobby y la lista de todos los jugadores conectados al lobby. Además, al creador del lobby se le pregunta cuál es el número máximo de jugadores del lobby y la dificultad de las IAs si las hubiera.

Se modificaron los demás paneles y clases de la GUI para que ahora recibieran por parámetro no el ControllerLocal (antiguo Controller), si no el Controller (interfaz que implementa tanto el ControllerLocal como el ControllerCliente).

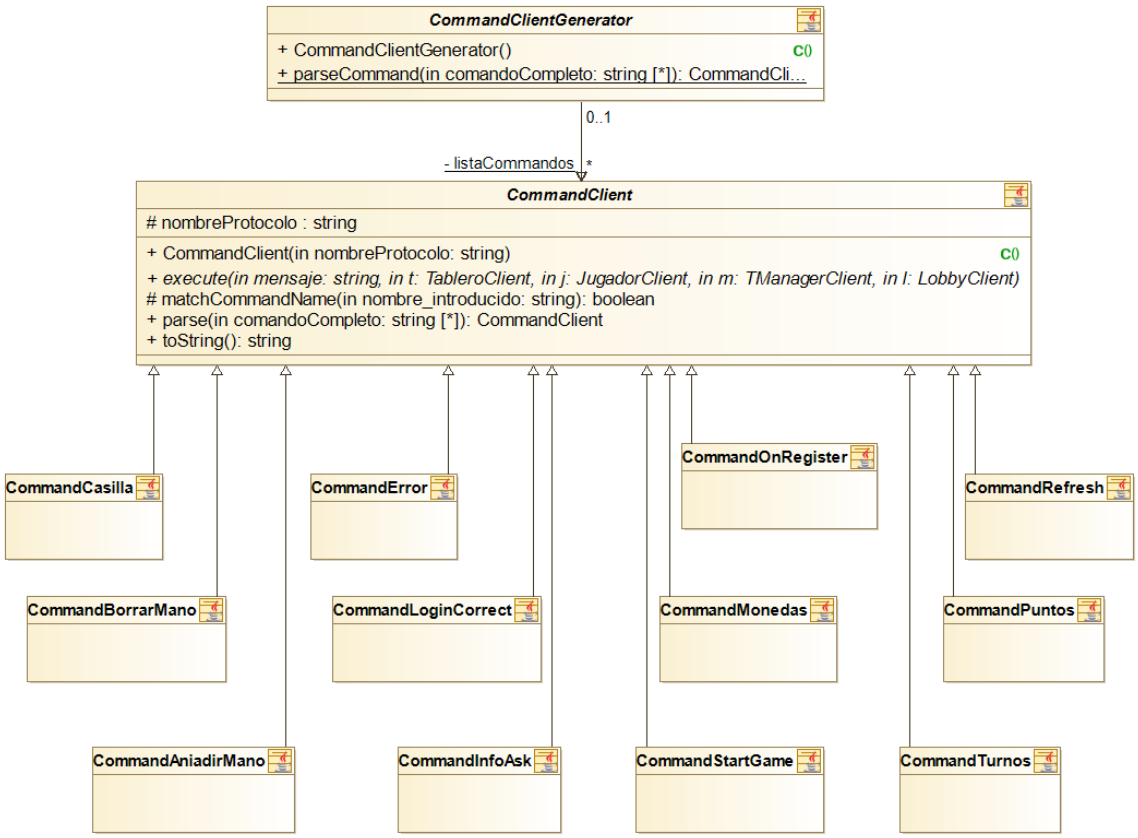
iv. CommandClient

Estructura que utiliza el **patron Commando** para facilitar la interpretación de los mensajes que viajan por el socket, en este caso del servidor al cliente. Este tipo de patrón nos proporciona que nuestro código sea muy limpio y fácilmente ampliable, pues para añadir un comando nuevo, solo se debe crear la clase correspondiente que herede de la clase abstracta CommandClient. Además hace los comandos fácilmente reutilizables en otras situaciones, en nuestro caso, utilizables desde el servidor y el cliente sin variaciones.

Contamos con una clase CommandClientGenerator, gracias a la cual, dado un mensaje, crea el objecto comando correspondiente. Podríamos hablar aquí de un patrón

de creación similar a **FactoryMethod** en funcionalidad, aunque sin la interfaz característica.

Esta estructura de comandos cuenta con 12 comandos distintos a ejecutar sobre el Cliente. Su método execute recibe como parámetros el talblero, jugador, TManager y Lobby del cliente.



v. CommandLobby

Estructura que utiliza el **patron Commando** para facilitar la interpretación de los mensajes que viajan por el socket, en este caso de la vista al servidor.

Contamos con una clase CommandLobbyGenerator, gracias a la cual, dado un mensaje, crea el objecto comando correspondiente.

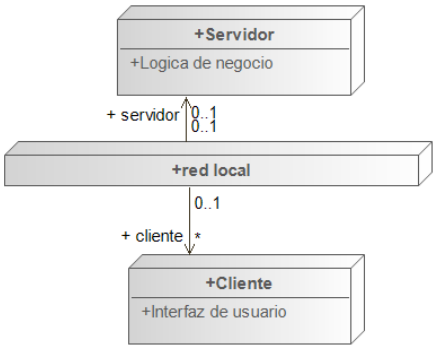
Esta estructura de comandos cuenta con 3 comandos distintos a ejecutar sobre el Lobby. Por tanto, en su método execute recibe al Lobby con parámetro.

3. CLASES/INTERFACES ADAPTADAS

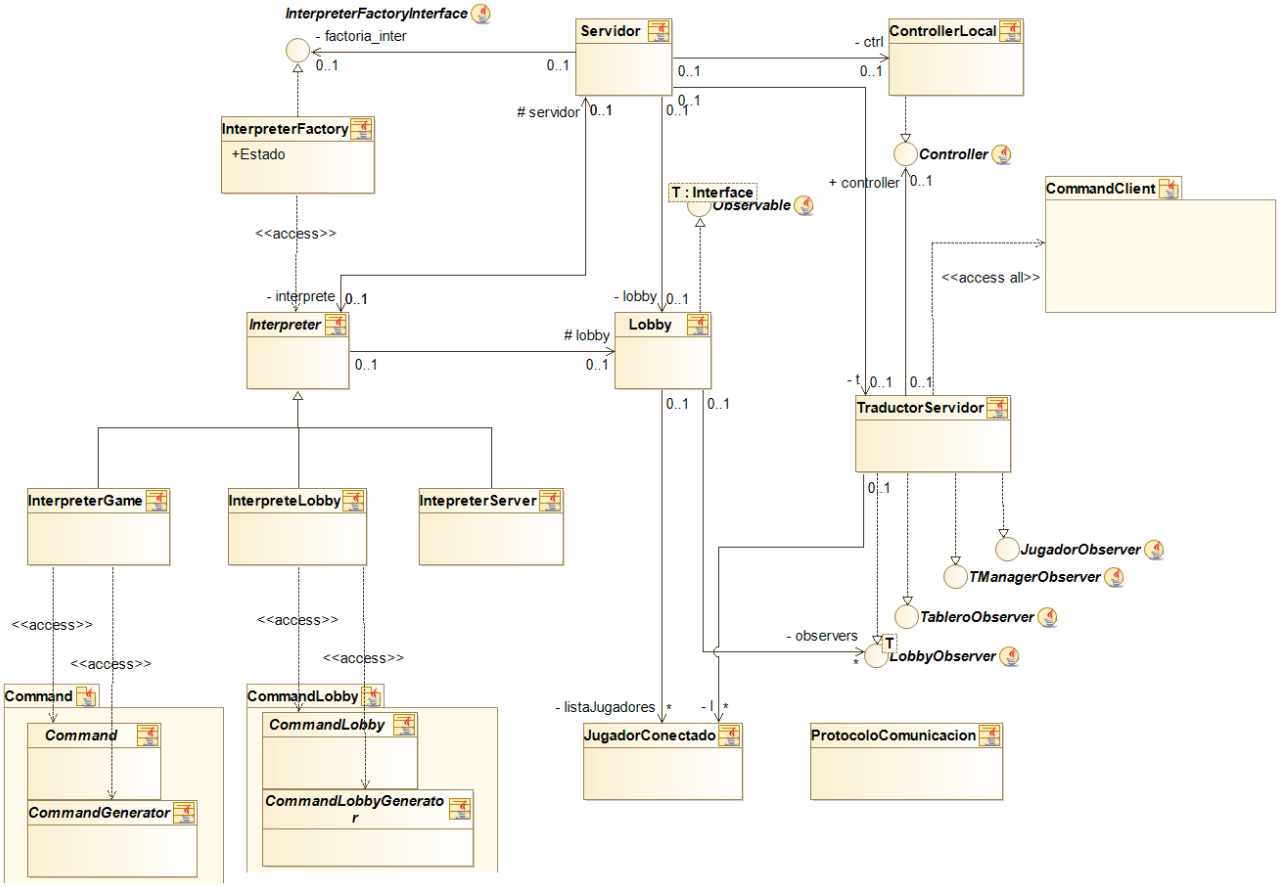
1. Se adaptó el Controller y se pasó a llamar ControllerClient para que implementara la interfaz Controller.
2. Se adaptaron los comandos para que tuvieran un toString y así poder enviarlos a través del socket.
3. Se adaptó la MainWindow para que pudiera cambiar de panel con el método setVista (a este método le llega el estado del cliente y según este estado cambia al panel correspondiente).
4. Se adaptó el controlador para que no iniciara la lista de jugadores en el constructor si no que se le pase como argumento el modelo y por tanto esta lista no se inicializará en el controlador sino en el modelo.

4. DIAGRAMAS

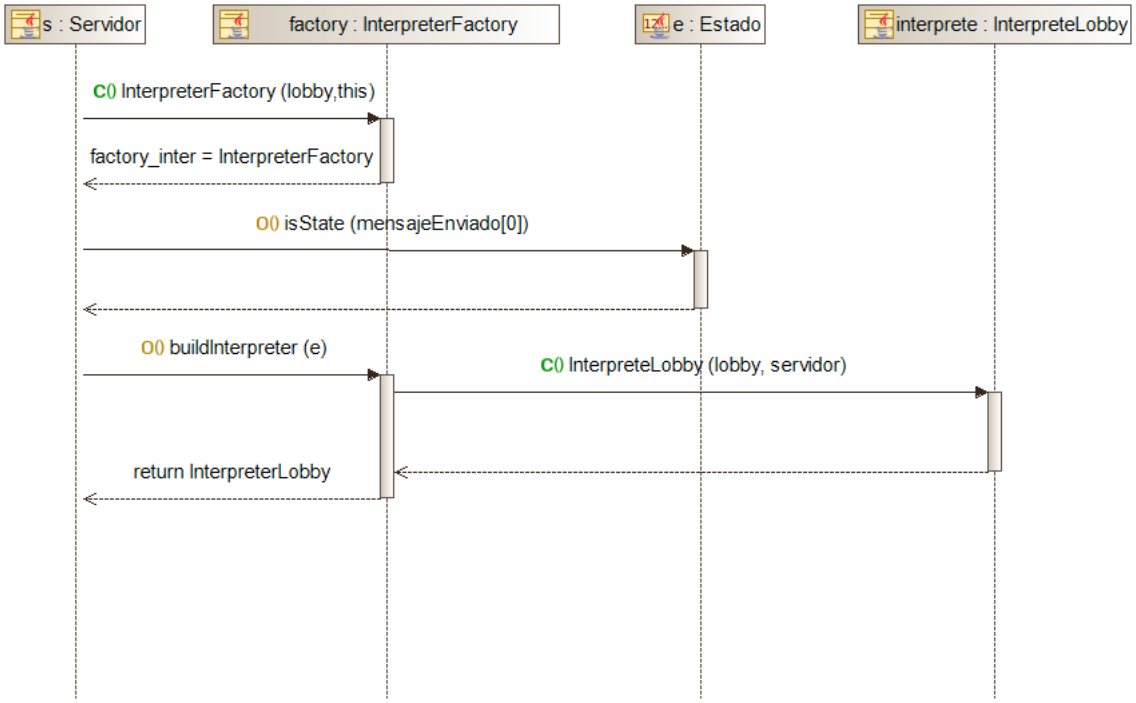
i. DIAGRAMA DE DESPLIEGUE



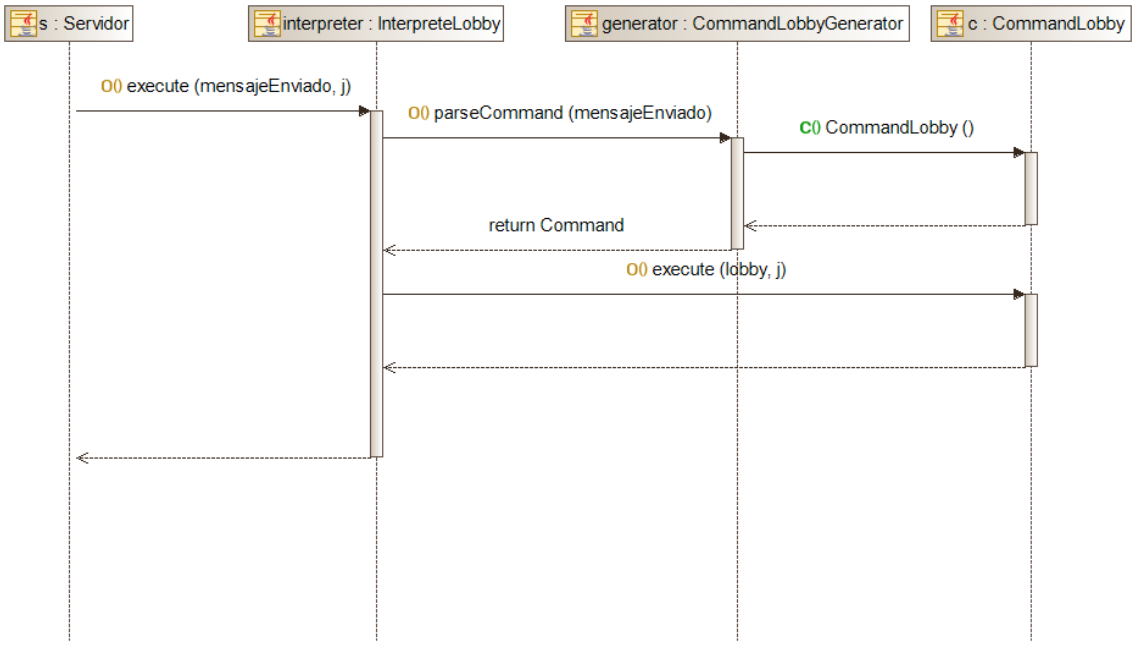
ii. DIAGRAMA DE CLASES DEL SERVIDOR



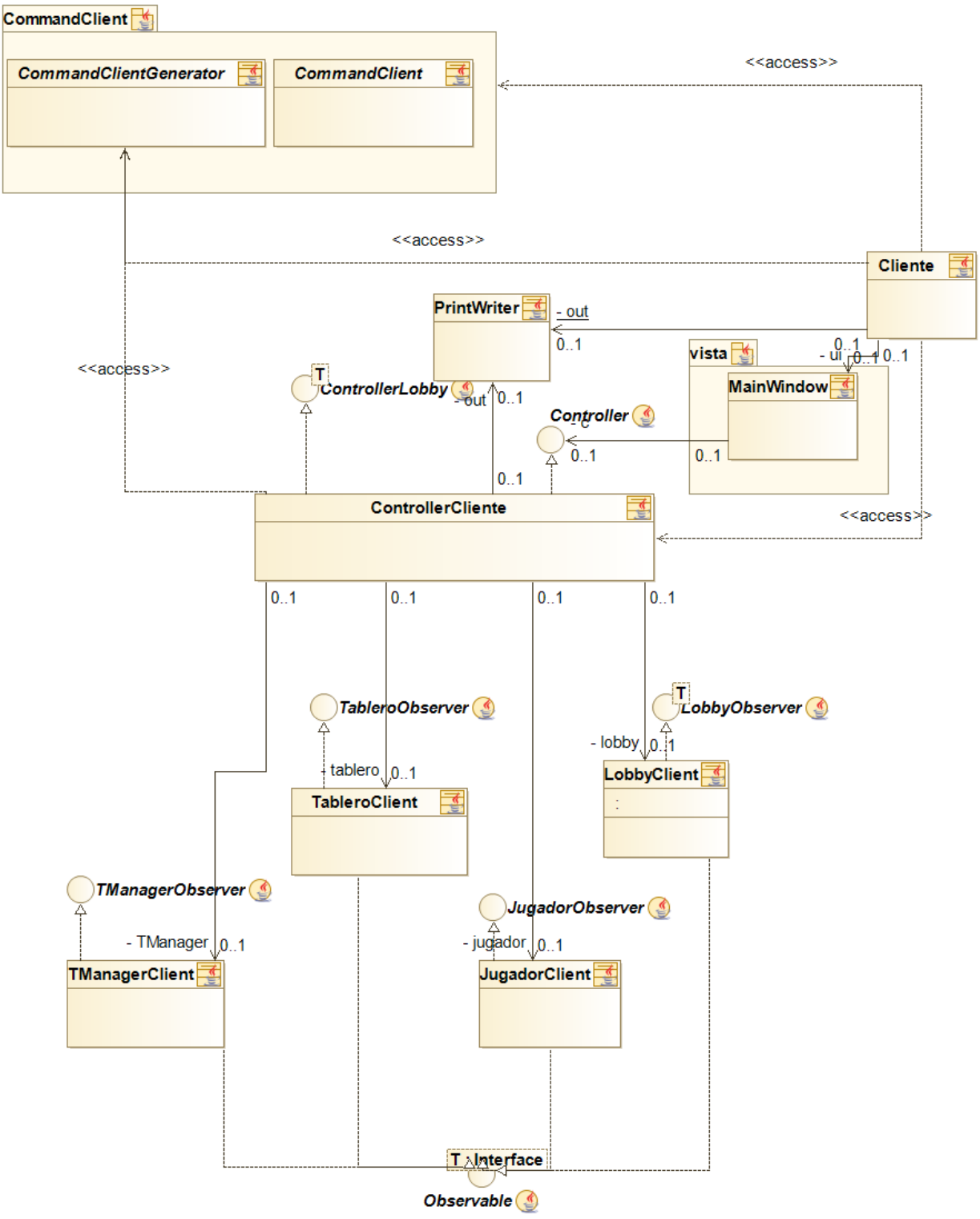
iii. CREACIÓN DE UN INTÉRPRETE CONCRETO



iv. EJECUCIÓN DE UN MENSAJE



v. DIAGRAMA DE CLASES DEL CLIENTE


CommandClient
CommandClientGenerator
CommandClient
<<access>>
<<access>>
Cliente
PrintWriter
- out
0..1
vista
- ui
0..1
0..1
0..1
MainWindow
<<access>>
T
ControllerLobby
- out
0..1
Controller
c
0..1
0..1
0..1
ControllerCliente
<<access>>
0..1
0..1
0..1
0..1
TableroObserver
T
LobbyObserver
- lobby
0..1
- tablero
0..1
TableroClient
LobbyClient
TManagerObserver
JugadorObserver
- TManager
0..1
- jugador
0..1
TManagerClient
JugadorClient
T
Interface
Observable