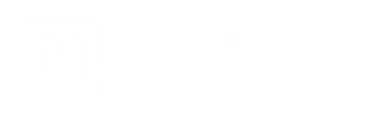


|  |
| --- |
| Creando un ajedrez  en línea  Proyecto de Fin de Grado |
| Javier Ausejo Fernández-Andes Curso 2018-19/2019-20  IES Comercio  Desarrollo de Aplicaciones Multiplataforma  Tutor: David Pérez |



# ÍNDICE

[1. INTRODUCCIÓN. 3](#_Toc38380648)

[2. OBJETIVOS. 4](#_Toc38380649)

[3. FASES DEL PROYECTO. 5](#_Toc38380650)

[4. AMPLIACIÓN Y POSIBLES MEJORAS. 19](#_Toc38380651)

[5. CONCLUSIÓN. 19](#_Toc38380652)

[6. BIBLIOGRAFÍA. 19](#_Toc38380653)

# INTRODUCCIÓN.

Desde un principio, en este proyecto se tiene la pretensión de crear un **ajedrez en red desde cero** **desarrollado en Java** que sea, desde luego, óptimo y agradable para los clientes que hagan uso de esta aplicación. A lo largo de este documento, iremos viendo cómo hemos encarado el proyecto en sus diferentes fases, desde su nacimiento hasta la creación de una primera versión en la que se incluirán diferentes elementos en virtud de las tres premisas u objetivos de trabajo que a continuación voy a explicar.

**En primer lugar**, como podemos intuir, se tratará de una **aplicación cliente-servidor** y, por tanto, debemos definir la relación que existirá entre estos: el servidor puede comprender muchas partidas simultáneamente, actuando como fuente de energía de estas, y encargándose de emparejar a los diferentes jugadores estableciéndose, además, como nexo en un canal de comunicaciones que permitirá a los clientes de una misma partida estar en contacto entre ellos.En resumen, como premisa básica, tenemos que **el servidor hará de intermediario de todas las partidas que comprenda, encargándose de que haya comunicación entre los clientes de una misma partida**.

**En segundo lugar**, vamos a definir el concepto de **comunicación entre y para clientes**. Para ello, previamente, hay que contextualizar y hablar de la **interfaz gráfica de usuario**. La GUI contará, principalmente, con cuatro elementos:

1. Un panel que comprenderá los elementos oportunos para que el usuario indique su nickname en la partida, así como una etiqueta indicando su estado de conexión respecto al servidor y una serie de botones para conectarse y desconectarse.
2. Un panel que hará de tablero de ajedrez.
3. Un panel de registro de los movimientos de la partida que informa sobre el transcurso de esta.
4. Un panel de chat en la que los jugadores en cuestión podrán entablar comunicación entre ellos.

Una vez definidos los elementos básicos de la interfaz, decimos que la comunicación entre y para clientes, **es toda aquella interacción del usuario con la GUI en la que se precise intervención del servidor**, como lo puede ser la conexión/desconexión del usuario, el envío/recepción de mensajes del chat o el movimiento de figuras del tablero de ajedrez.

Por último, **en tercer lugar**, como consecuencia de lo explicado en estos dos puntos, se define que trabajaremos con **sockets TCP** enviando y recibiendo objetos de tipo **ObjectOutputStream y ObjectInputStream**. Se elige este tipo de datos porque, a mi juicio y como ya veremos más adelante, **a través de la serialización de objetos se nos permitirá crear clases a medida y hacer que la comunicación entre y para clientes sea más limpia y ordenada**.

# OBJETIVOS.

Por mediación del tutor, decidimos trabajar este proyecto aplicando la **metodología ágil SCRUM**, cuya lógica responde a una gestión del trabajo en base a una división racional del proyecto en varios sprints. En cada uno de estos sprints, se deben establecer una serie de objetivos lógicos a cumplir en un determinado periodo de tiempo.

En este caso, **dividimos el proyecto en** **cinco sprints**, dando un margen de dos semanas de trabajo a cada uno de ellos. Una vez finalizado este periodo de tiempo, se concerta con el tutor una reunión en la que se revisa el trabajo llevado a cabo durante estas dos semanas, proponiendo posibles mejoras o cambios de planteamiento con el objetivo final de que el proyecto sea lo más eficiente posible.

Una vez definida la mecánica de trabajo a seguir, podemos explicar los diferentes sprints o fases que atraviesa el desarrollo del proyecto:

* **Sprint 1: Organización del proyecto y documentación de las herramientas de trabajo.**

Se trata de una fase de estudio en la que se introducen las herramientas de trabajo que vamos a utilizar. La idea de este sprint es **determinar las tecnologías que terminaremos utilizando**, con una respuesta clara que justifique el porqué las elegimos, procurando **articular una** **organización del proyecto lo más acertada posible**.

* **Sprint 2: Diseño de la interfaz de usuario.**

Si queremos que la aplicación sea funcional, una parte importante es que esta se deje ver y sea atractiva visualmente. Durante esta etapa, se repiensa la GUI en torno a los elementos básicos explicados en la introducción y, finalmente, **tratamos de crear una interfaz intuitiva que haga la aplicación lo más sencilla posible**.

* **Sprint 3: Diseño del sistema de emparejamiento y del chat.**

Este sprint supone la primera toma de contacto con la lógica interna del programa, más allá de la propuesta estética que se lleva a cabo en el anterior sprint. La premisa básica es **definir el sistema de emparejamiento del servidor**, con la vista puesta en que comprenda **varias partidas simultáneamente**. Para comprobar su correcto funcionamiento, deberemos programar también el **chat**, lo que implica concretar la **estructura de los mensajes** y el envío y recepción de estos por parte del servidor y de los clientes.

# FASES DEL PROYECTO.

* 1. Sprint 1: Organización del proyecto y documentación de las herramientas de trabajo.

**Tareas a cumplir:**

* Estudio de los sockets TCP y sus posibilidades.
* Esbozo preliminar de *Mensaje.java*, útil en clave de comunicación.

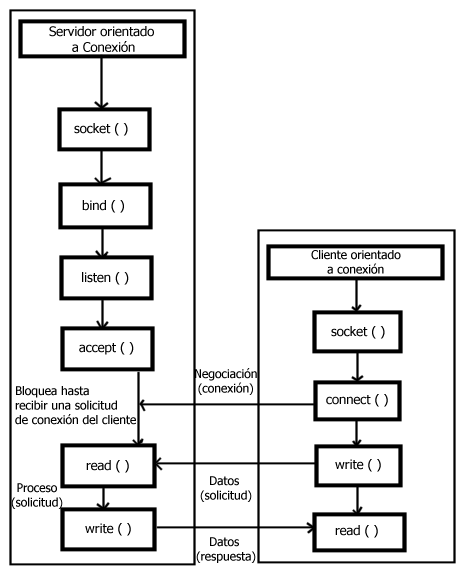
**Desarrollo:**

Tal y como ya adelantamos en la introducción, este proyecto se basa en el uso de **sockets**, que, sabemos, son elementos que actúan como extremo en un proceso de comunicación por red entre diferentes aplicaciones o servicios, permitiendo así que un mensaje pueda circular de un punto a otro de la red.

Como vemos, se trata de un **proceso de** **comunicación por red**, y, por tanto, concluimos por entender que **todo socket debe tener asociado una IP** (dirección en la que se está ejecutando el servidor), **y un puerto** a través del cual la aplicación que se está ejecutando en el servidor puede escuchar e interpretar los mensajes que lleguen, y, en definitiva, comunicarse con los clientes.

Una vez comprendidos los elementos básicos de un socket, determinamos que **trabajaremos sobre un socket de tipo TCP**, lo que nos proporciona una **comunicación cliente-servidor bidireccional** **orientada a la conexión** que garantiza la total entrega de los datos en el orden en que fueron enviados hasta el momento en el que alguno de los dos se desconecte.

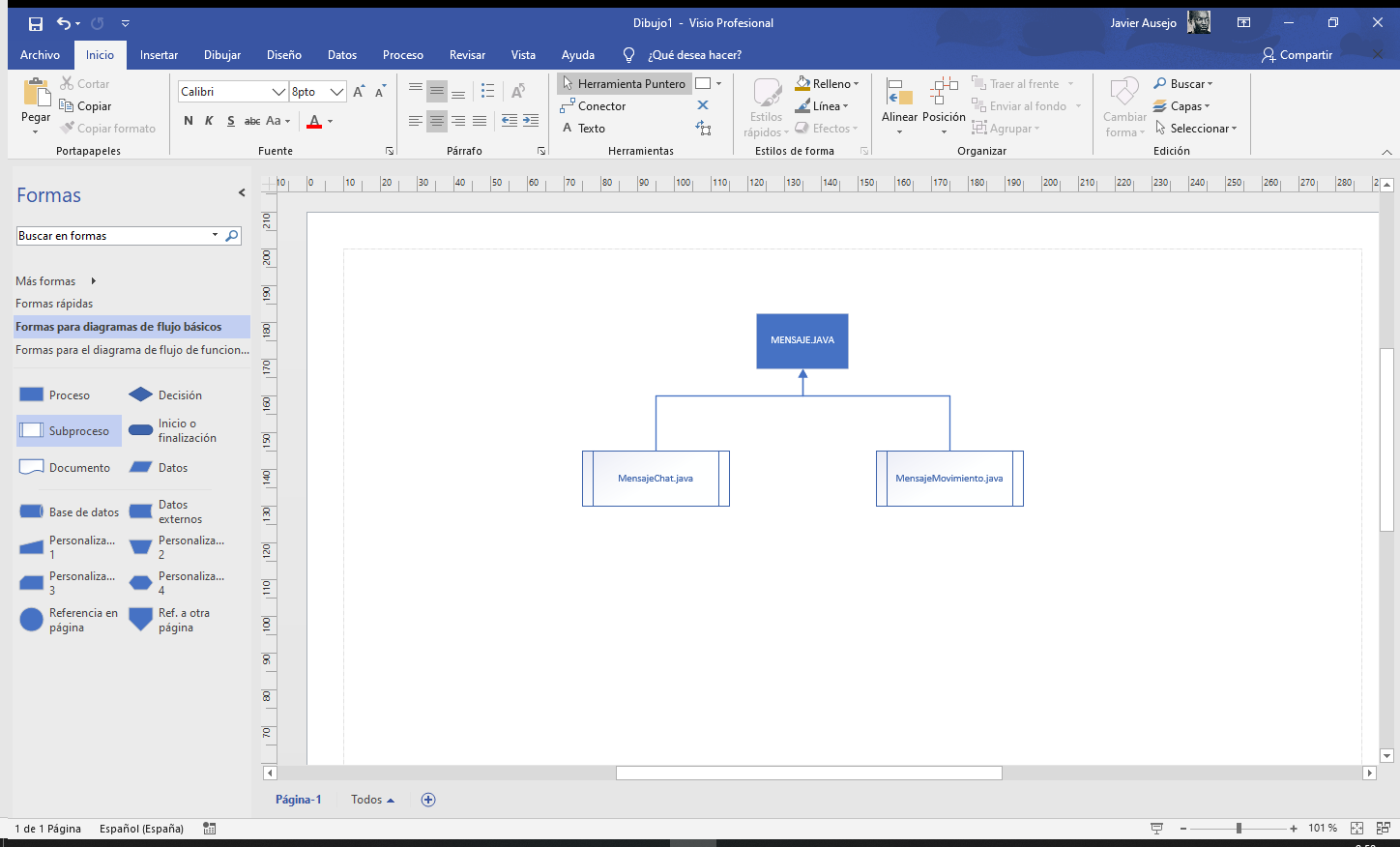
Los sockets TCP funcionan, resumidamente, de la siguiente manera:

* **Inicio del servidor**, que se encuentra permanentemente escuchando peticiones de clientes.
* **Petición de conexión del cliente** a la dirección IP del servidor a través de su puerto.
* Respuesta de **petición aceptada** por parte del servidor.
* Apertura de un **stream de datos** entre cliente y servidor, que permite la comunicación de un flujo secuencial de datos en el que ambos tienen permisos de lectura y escritura hasta que se cierre la conexión.

Contextualizado ya lo que es un socket TCP, podemos proceder a introducir **de qué manera se enviarán los datos entre cliente y servidor**. Dado que este stream que nace de la conexión del cliente al servidor soporta diferentes tipos de datos (bytes, tipos de datos primitivos etc.), **tenemos la posibilidad de crear una clase Java que instancie objetos *Mensaje.java* hechos a nuestra medida y enviarlos por red gracias a la serialización**, lo que nos facilitaría la interpretación de dichos objetos haciéndola más limpia y ordenada.

Este proceso de serialización de objetos no es más que la conversión de un objeto a bytes, que luego se podrán recuperar y volver a convertir en un objeto de dicha clase. Para ello, debemos hacer uso de las clases **ObjectOutputStream** y **ObjectInputStream**, que son las únicas que nos permiten enviar y recibir objetos en sockets TCP a través de los métodos ***readObject()*** *y* ***writeObject()****.*

A estas alturas de la práctica, definimos que la clase *Mensaje.java* será una interfaz de la que, por lo menos, colgarán dos clases nuevas: ***MensajeChat.java*** y ***MensajeMovimiento.java***. Ambas implementarán ***Serializable***.



Este *Mensaje.java* se trata de un boceto preliminar y muy primario que nos ayuda a ubicarnos y a entender cómo funcionará la comunicación entre el servidor y el cliente. La idea final es que *Mensaje.java* instancie objetos para, como hemos dicho antes, poder interpretar mensajes como lo que son: mensajes de chat o movimientos de figura. Por tanto, dado que utilizaremos la serialización de objetos, en sus correspondientes sprints veremos cómo evoluciona *Mensaje.java* y cómo a cada mensaje le podremos asociar un objeto de tipo ***Jugador.java* o *Casilla.java*,** con el **objetivo de trabajar en un sistema de emparejamiento más eficiente que luego permita a los jugadores de una misma partida hablar con el rival y mover las figuras**.

* 1. Sprint 2: Diseño de la interfaz de usuario.

**Tareas a cumplir:**

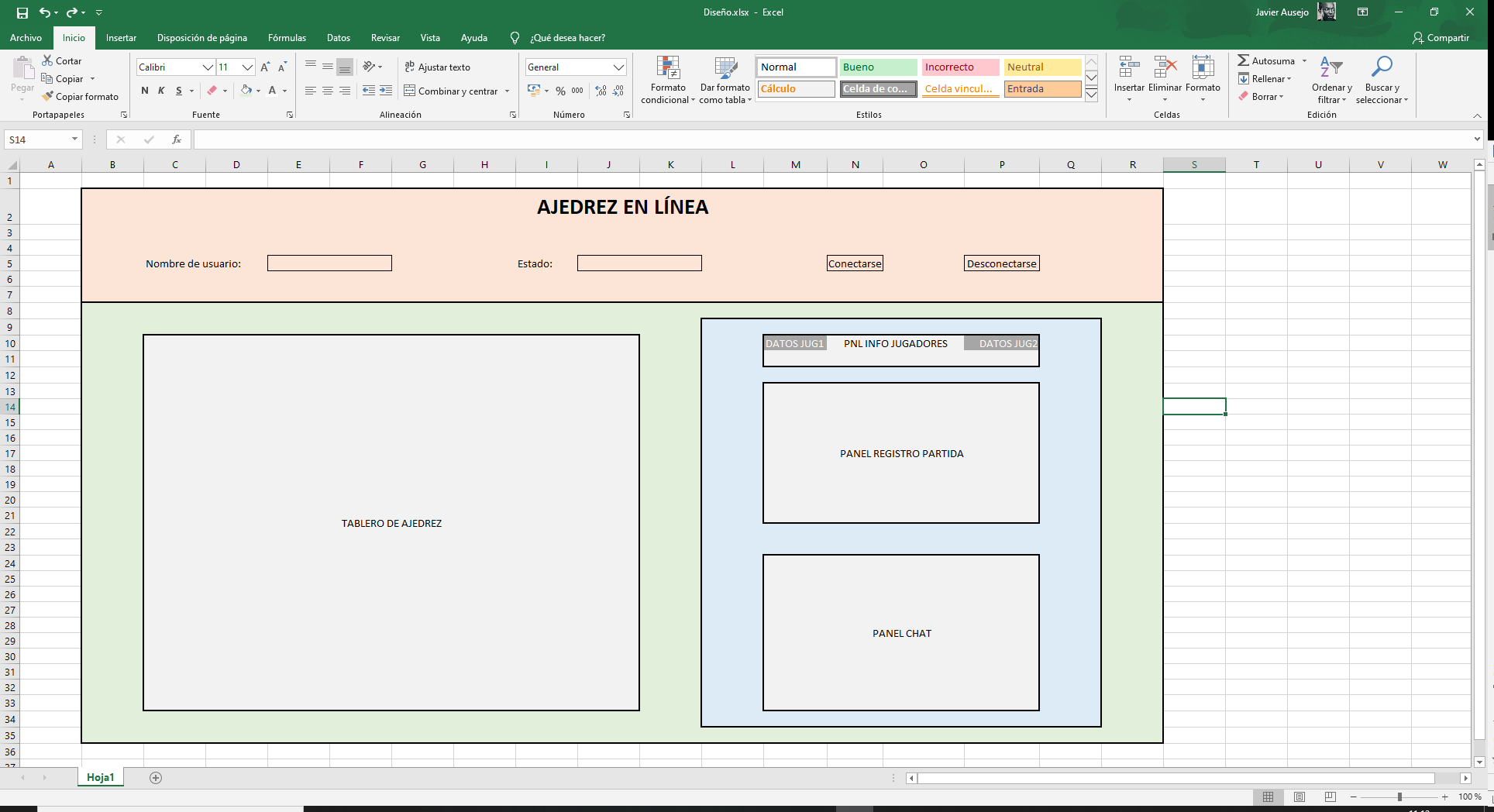
* Diseño prototipo de la GUI de la aplicación.
* Programación preliminar de *Tablero.java* y *Casilla.java.*
* Introducción a la clase *Figura.java*.

**Desarrollo del sprint:**

**La interfaz de usuario es la cara visible de la aplicación** y lo que puede determinar, en virtud de lo intuitiva y accesible que sea, si el usuario hace uso de este programa o se busca otro que se adapte mejor a sus necesidades. Por tanto, además de la importancia que supone que la aplicación haga lo que se le presupone que tiene que hacer, no es menos notable que cuente con una GUI **atractiva** y, en medida de lo posible, **autoexplicativa** para que el usuario final no tenga que comerse la cabeza ni perder tiempo pensando en cómo empezar a usar el programa.

Dado que en el plano estético se trata de una aplicación sencilla (solo contará con una “pantalla”), debemos cuidar que la disposición de los elementos quede ordenada y compensada.

**¿Con qué elementos contará nuestra aplicación?** Como las imágenes tienen más fuerza que las palabras, se presenta una imagen/croquis a modo de resumen con el diseño inicial:



La imagen muestra, principalmente, **dos paneles**: uno color naranja y uno de color verde, que representarían las **zonas de parámetros conexión y de área de juego**, respectivamente, que estarán disponibles (o no) en función de nuestro estado de conexión respecto al servidor.

En el **panel de parámetros de conexión** identificamos:

* Un JLabel con “**Nombre de usuario**” y un JTextField en el que se indicaría.
* Un JLabel con “**Estado**” y, a su derecha, otro que indicaría nuestro **estado de conexión** respecto al servidor (conectado, desconectado, a la espera de partida) con un color de fondo que nos ayudara a identificarlo.
* **Dos JButtons para conectarse y conectarse del servidor**.

Por otro lado, en el panel verde, que representa el **área de juego**, encontramos más miga que en el anterior, pues presenta:

* Un **tablero de ajedrez** que nace de *Tablero.java*, del que ahora hablaremos en profundidad.
* Un panel azul, que ocupa la parte derecha del panel del área de juego. Este panel azul, a su vez, se subdivide en tres paneles:
  + Un JPanel con **información de los jugadores en la partida**, en el que se mostrarían sus nicknames, información acerca de los turnos y el tiempo restante para la ejecución de un movimiento.
  + Un JPanel que incluirá un JList con las **incidencias de la partida**: movimientos, si el movimiento se trata de jaque o de jaque mate, tiempo de ejecución, si se ha comido una figura etc.
  + Un JPanel que funcionará a modo de **chat,** a través del cual los jugadores podrán entablar conversación durante el transcurso de la partida, es decir, una vez hayan sido emparejados por el servidor.

En definitiva, **en el panel de parámetros de conexión encontraríamos un título para la aplicación, así como diferentes elementos para identificarnos y hacer peticiones de conexión/desconexión al servidor**, mientras que **en el panel de área de juego encontramos los elementos de juego** necesarios que serán accesibles y que cobran sentido una vez hayamos encontrado rival contra el que jugar.

Una vez explicados superficialmente los conceptos presentes en nuestra interfaz gráfica, trataremos de explicar la **lógica que hay detrás de *Tablero.java***.

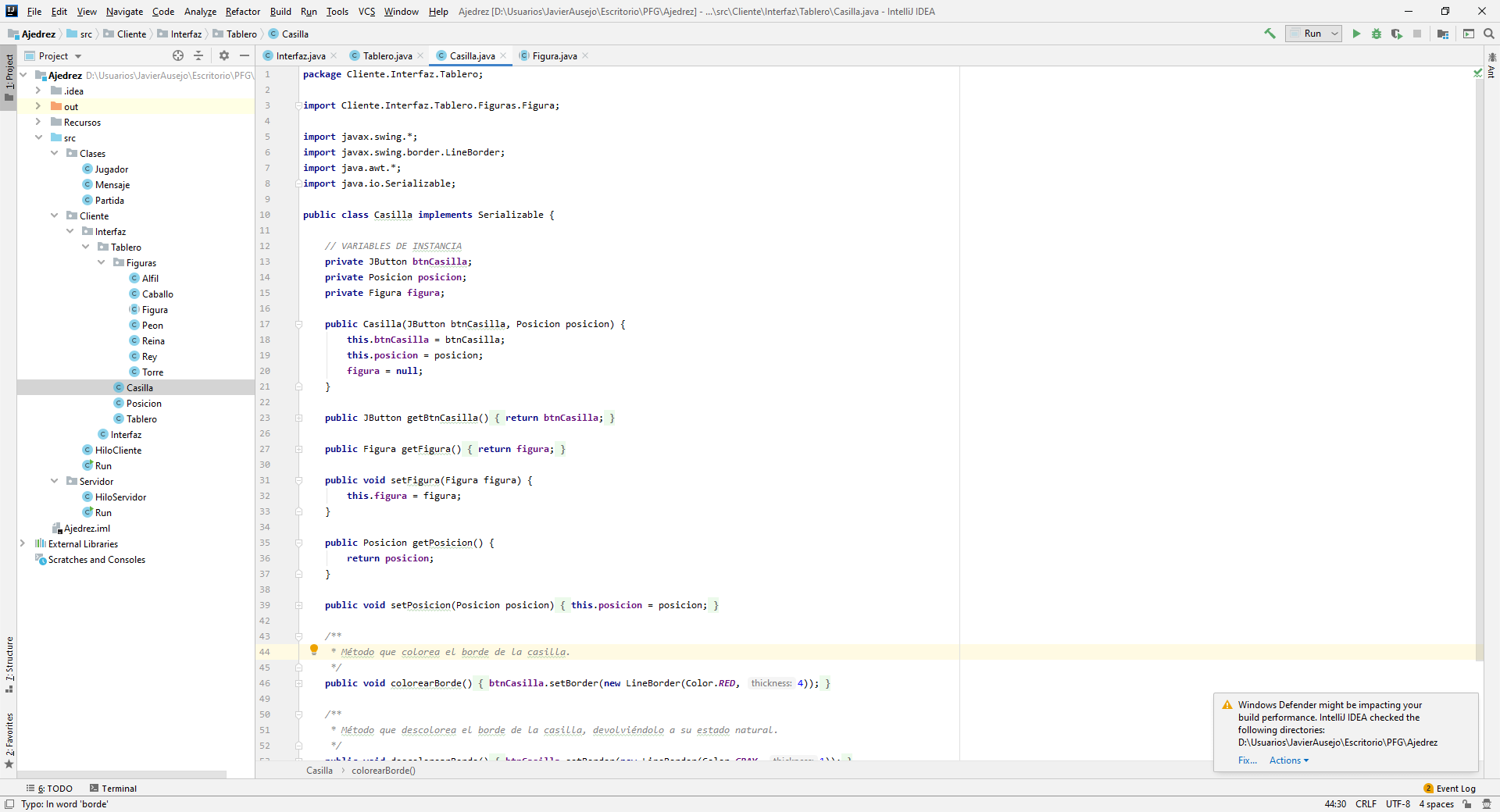
Desde un primer momento, pensé en abstraer el tablero del resto del diseño y programación de la GUI porque sabía que iba a ser el corazón de la aplicación y, por tanto, un elemento lo demasiado grande e importante como para no contar con una clase propia.

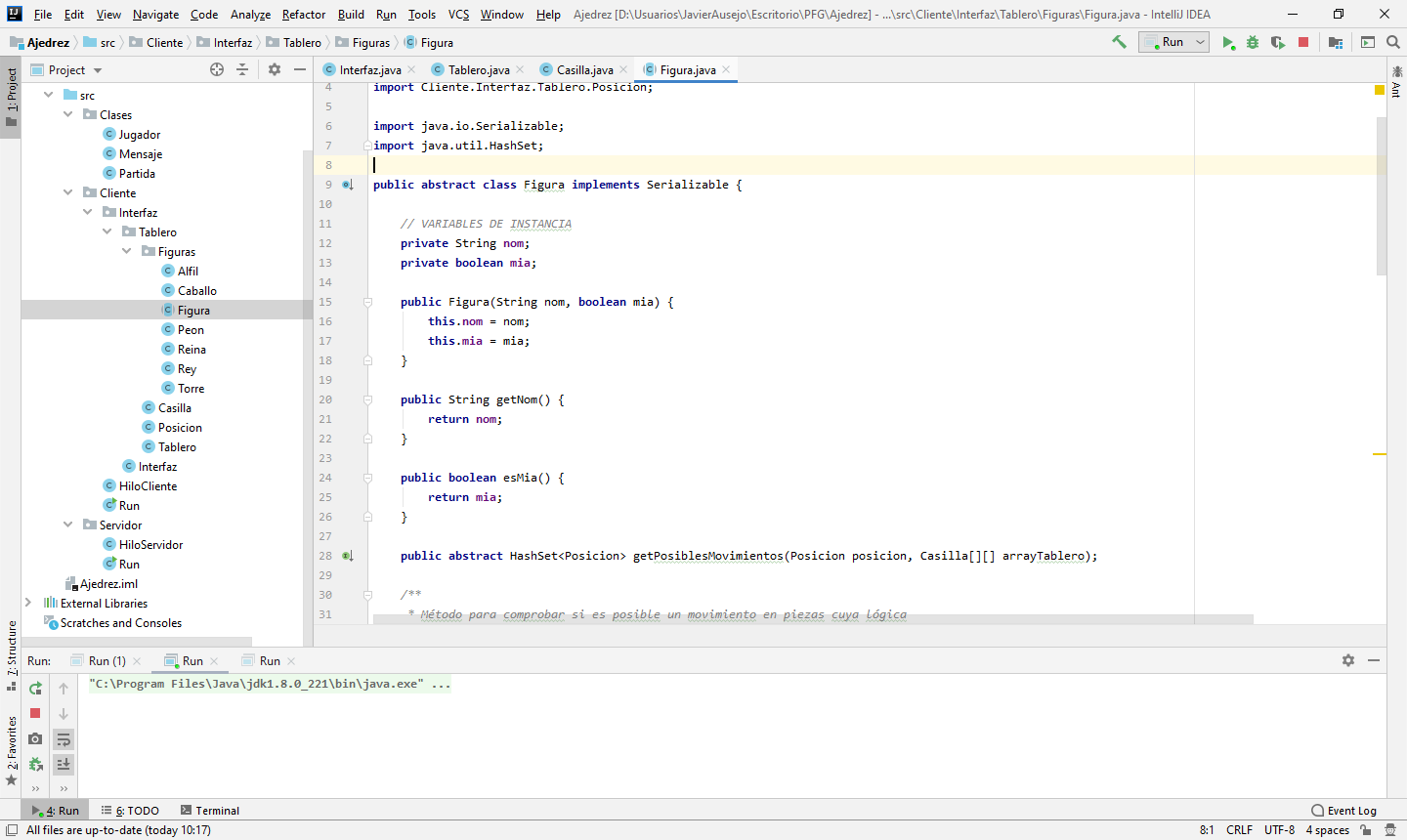
El tablero es un nuevo JPanel que contará con diferentes elementos y, mirando al futuro inmediato de este proyecto, deberá contar con las variables de instancia necesarias para poder enviar mensajes al servidor, identificar si el turno de la partida es el de su jugador o el del rival, conocer si ese jugador juega con figuras negras o blancas etc. Además de esto, naturalmente, el tablero es un conjunto de casillas ordenadas que pueden estar ocupadas por una serie de figuras.

En definitiva, extraigo la conclusión de que es necesario crear dos nuevas clases: ***Casilla.java*** y ***Figura.java*,** y que el tablero de ajedrez, en este punto de la práctica, se tratará de un JPanel que deberá implementar ***ActionListener*** y que estará formado por un array bidimensional de casillas que estarán ocupadas por instancias de la clase *Figura.java*. Ambas clases pueden formar parte de los mensajes a intercambiar con el servidor, por lo que deben implementar *Serializable*.

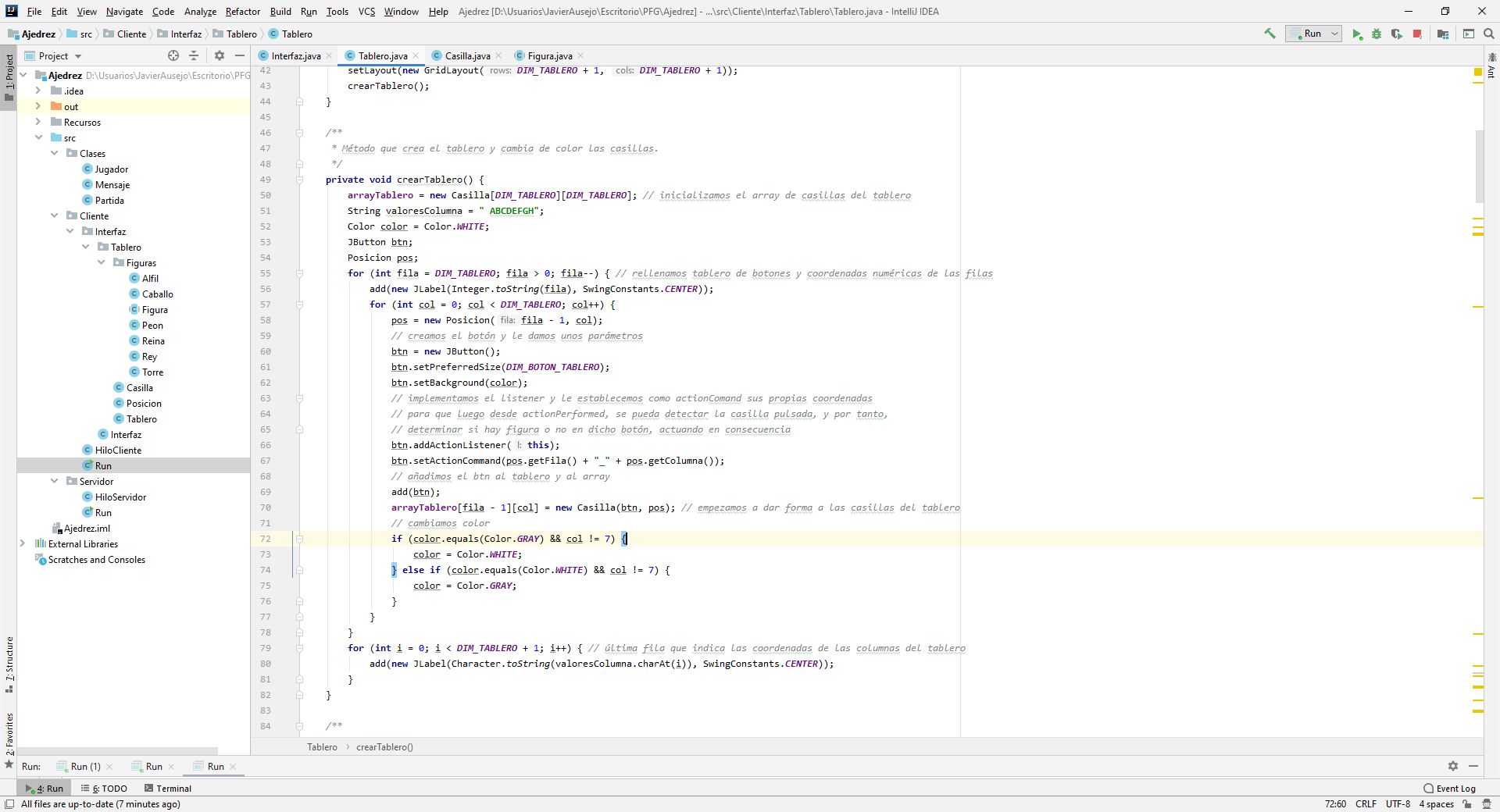
Sabemos que un tablero de ajedrez cuenta con diferentes tipos de figuras que no funcionan de la misma manera, por lo que haremos de *Figura.java* una clase abstracta de la que beberán las clases particulares del peón, del alfil, del caballo, del rey, de la reina y de la torre. También sabemos que cada una de estas figuras deberá comprender un método para obtener sus posibles movimientos en virtud del estado del tablero y de la posición que ocupa, es por ello que incluiremos el método abstracto ***getPosiblesMovimientos()*** colgando de *Figura.java*.

Las clases *Casilla.java* y *Figura.java* se corresponderán, por el momento, con los siguientes esquemas (incluyendo sus correspondientes *getters* y *setters*):





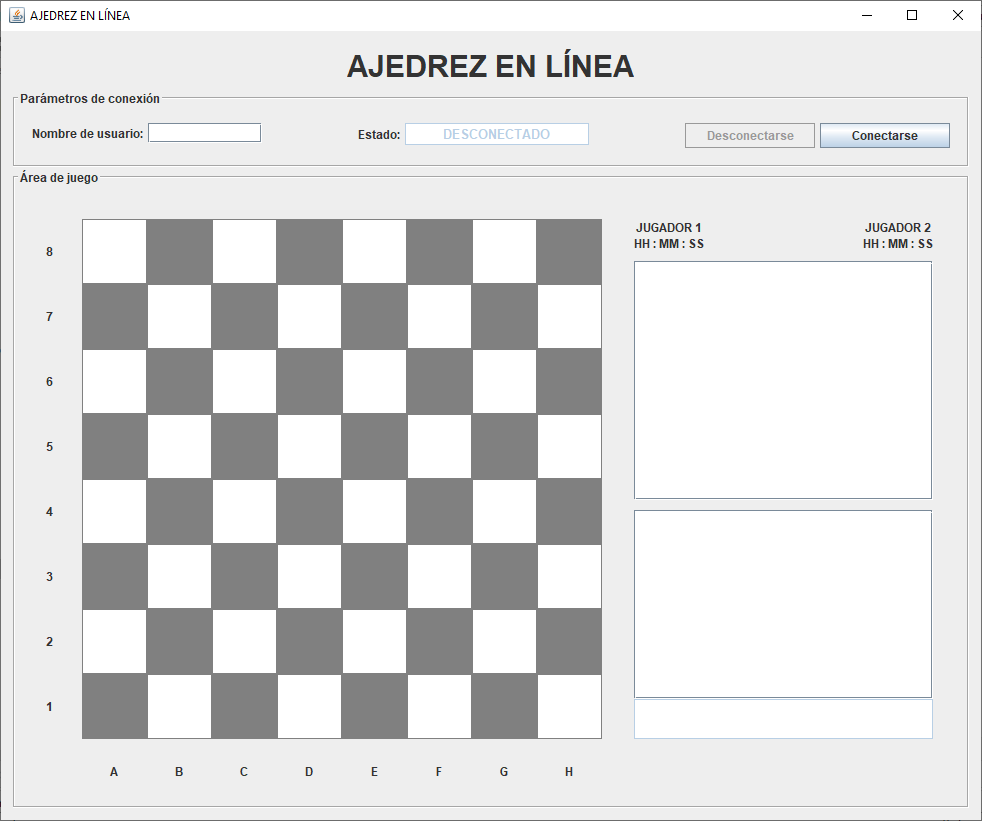
Explicado esto, podemos volver a *Tablero.java* y comprender el método que crea el array bidimensional que lo conforma.



Como leemos en el código, creamos un tablero 8x8 en el que los botones tienen un tamaño predeterminado y van alternando el color, dándole la forma de un tablero de ajedrez. Cada vez que creamos una casilla, le añadimos un botón, una posición y una figura ***null***. ¿Por qué una figura *null*? Dado que un jugador al momento de iniciar la aplicación no sabe si va a controlar las figuras blancas o negras, creo que es más adecuado delegar esto al sistema de emparejamiento y añadir las figuras en el momento en el que se encuentre rival.

También es importante señalar que, gracias al ***setActionCommand***, podremos conocer las coordenadas de la casilla pulsada y actuar en consecuencia.

La forma final que cobra nuestra interfaz de usuario es la siguiente:



* 1. Sprint 3: Diseño del sistema de emparejamiento y del chat.

**Tareas a cumplir:**

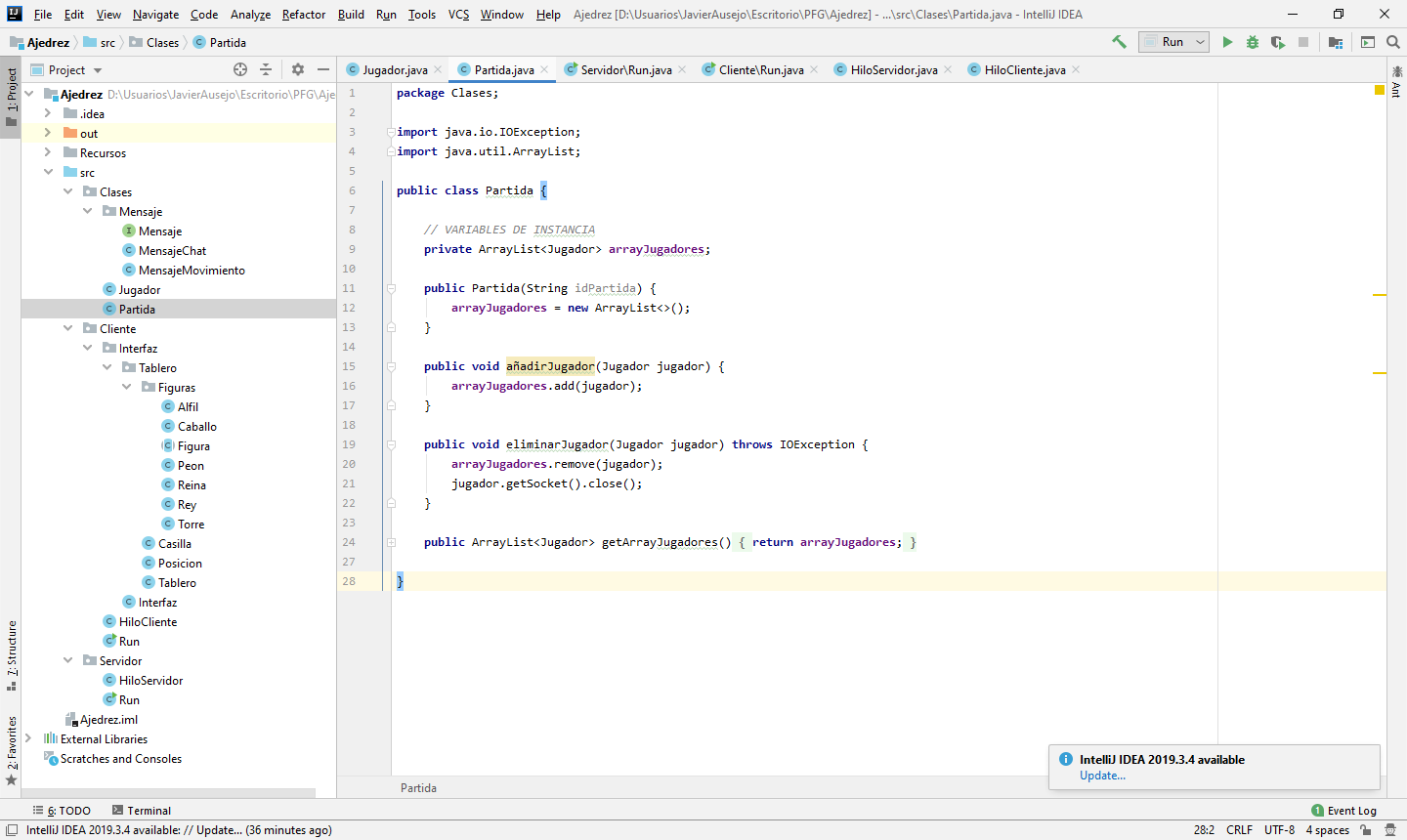
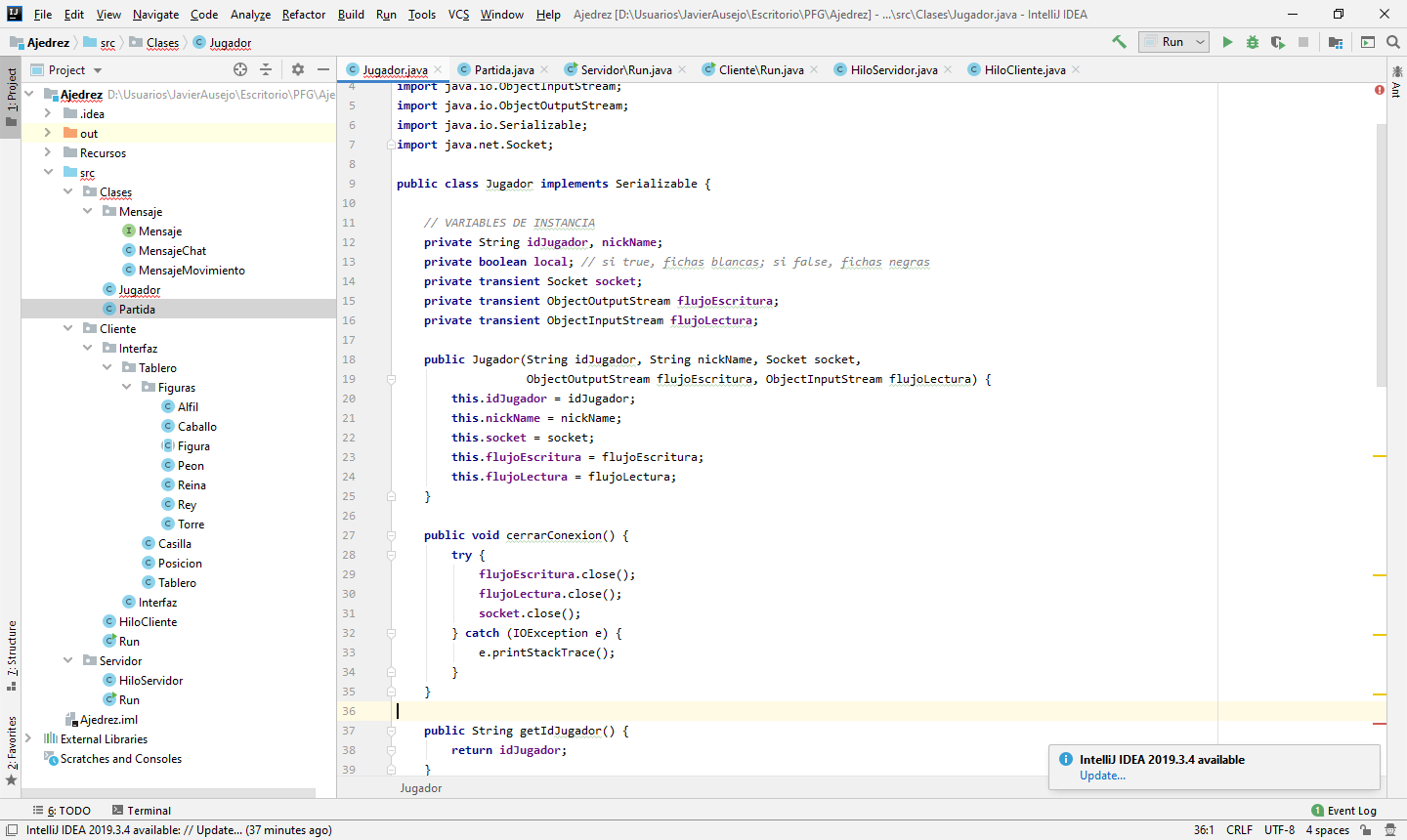
* Justificar la existencia de *Jugador.java* y *Partida.java*.
* Programación del sistema de emparejamiento.
* Programación del servicio chat de la aplicación.

**Desarrollo:**

Entendemos que **nuestro ajedrez se tratará de una aplicación multijugador que podrá ejecutar varias partidas de manera simultánea**. Es por ello que, para facilitar la posterior programación del sistema de emparejamiento, creamos dos clases nuevas acordes a esta lógica: ***Jugador.java*** y ***Partida.java***.

Un jugador se conectará al servidor y se definirá como tal cuando realice una petición de conexión/entrada a este y esta sea aceptada. En este punto, en el que la petición se ha convertido en un nuevo jugador, el servidor deberá actuar en consecuencia buscándole una partida en la que haya hueco disponible, o creando una y dejándole a la espera de la conexión de un rival.

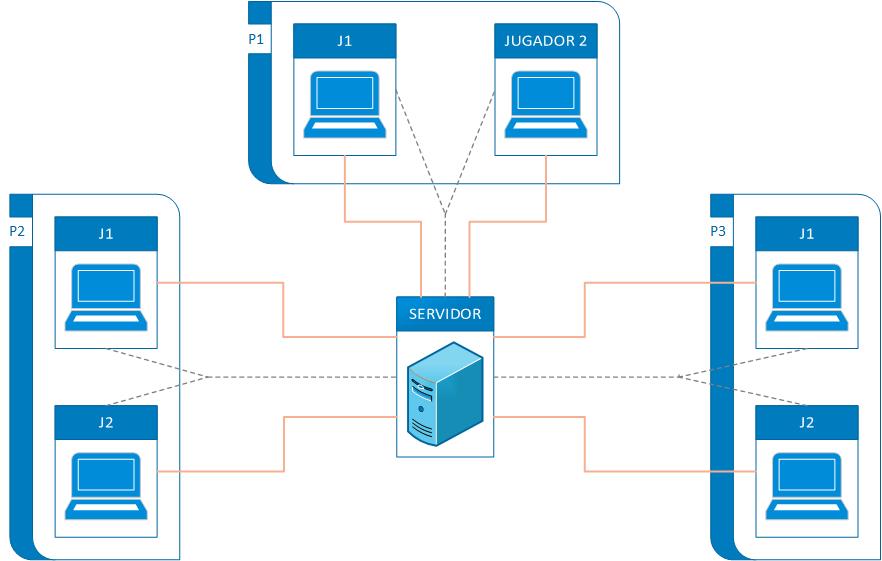
Una vez explicado esto, comprendemos que **nuestro sistema de emparejamiento integrará un array de instancias de *Partida.java* en la que cada una de ellas deberá tener asignada un par de jugadores/clientes, que se agruparán en torno a un array de objetos *Jugador.java***.

(\**Partida.java* y *Jugador.java*, cada uno con sus correspondientes *getters* y *setters*, aunque no se vean.)

Como vemos en las imágenes de los códigos, **cada una de las partidas y de los jugadores tendrá un identificador**. El identificador de la partida será único y secuencial a medida que se vayan creando nuevas, mientras que el de los jugadores será único dentro de la partida vinculada que les toque jugar.

Además, vemos que cada instancia de ***Jugador.java* comprende una serie de elementos de red** (“*socket”, “flujoEscritura”* y “*flujoLectura”*), que son necesarios para cerrar la conexión entre el cliente y el servidor, si se dieran las circunstancias. También vemos que estas variables están **precedidas del argumento “*transient”*.** Esto es porque la serialización de objetos solo nos permite la transmisión de datos de tipo primitivo; lo que hace “*transient*” es poner estos valores a *null* en el momento que estos se transmiten por red, evitando así que salte algún tipo de excepción.

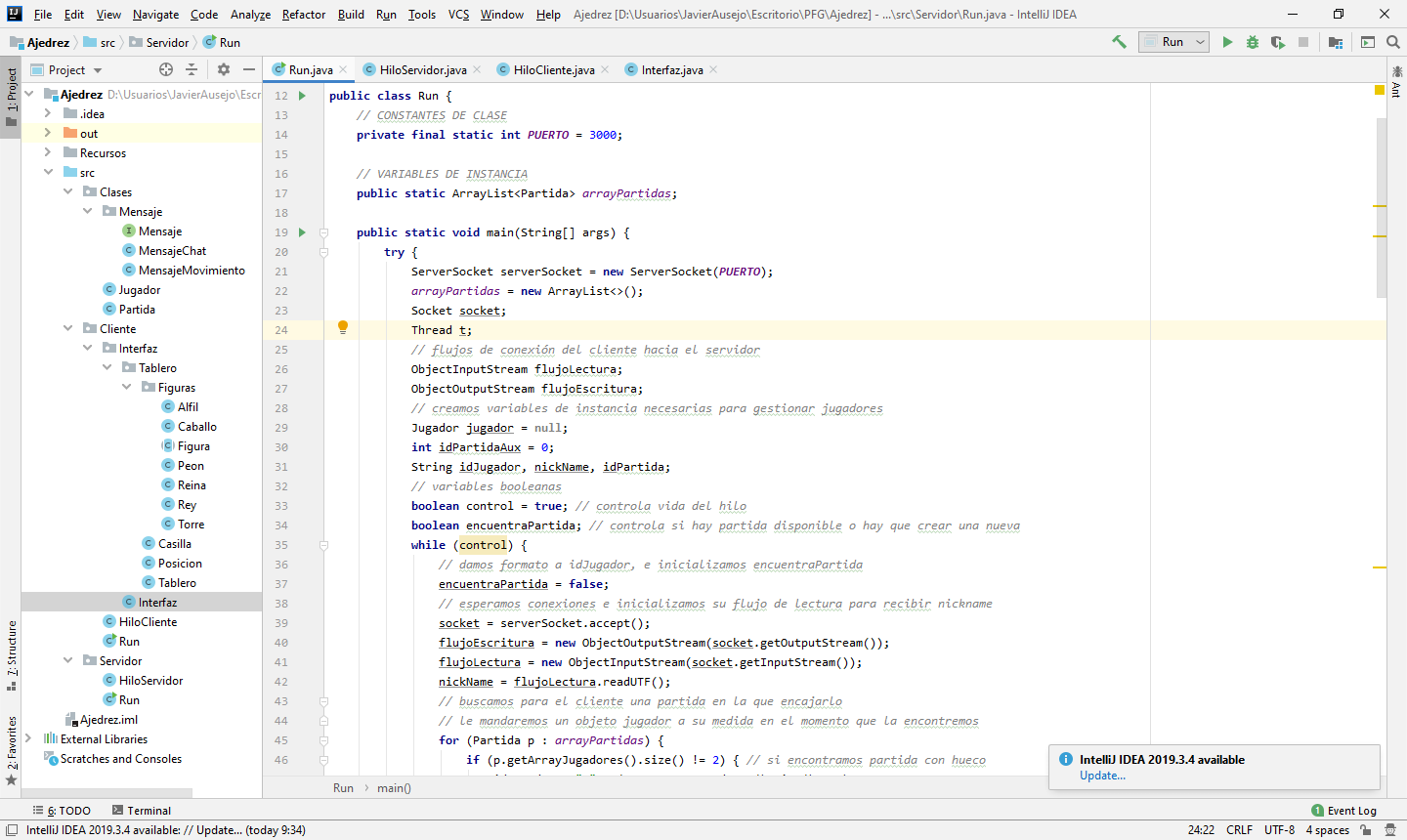
Por tanto, a golpe de imagen, tenemos que el sistema de emparejamiento agrupará los jugadores en diferentes partidas de la siguiente manera:



Entrando en la parte más densa del sprint, trataremos de explicar el código que rodea la puesta en marcha del servidor y la conexión de los clientes. Por parte del servidor, señalar que **utilizaremos el puerto 3000 para iniciar la escucha de los clientes/conexiones**.

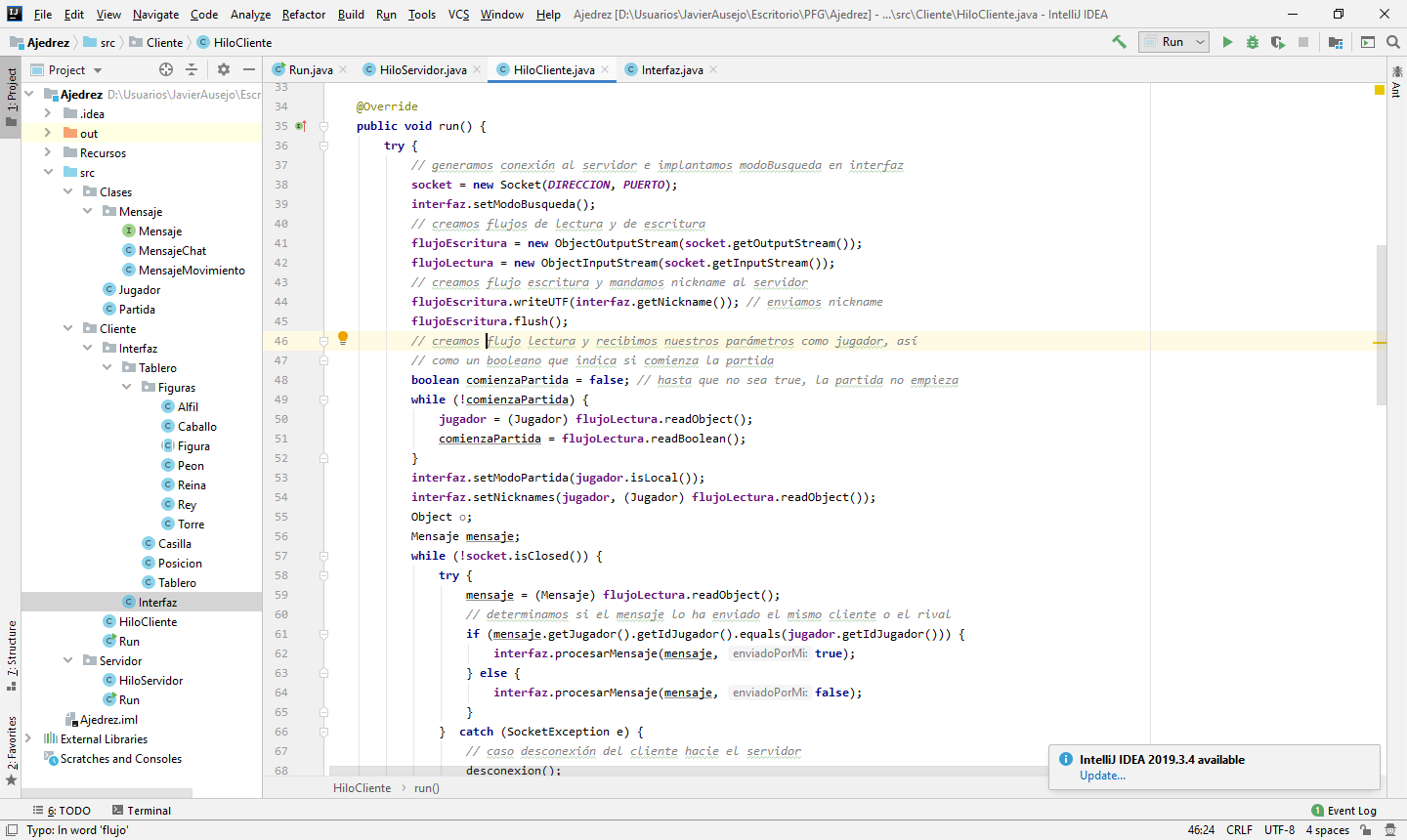
Entramos a simular el proceso de conexión:

1. Ejecutamos el *Run.java* del servidor, quedándose a la **espera de conexiones**.

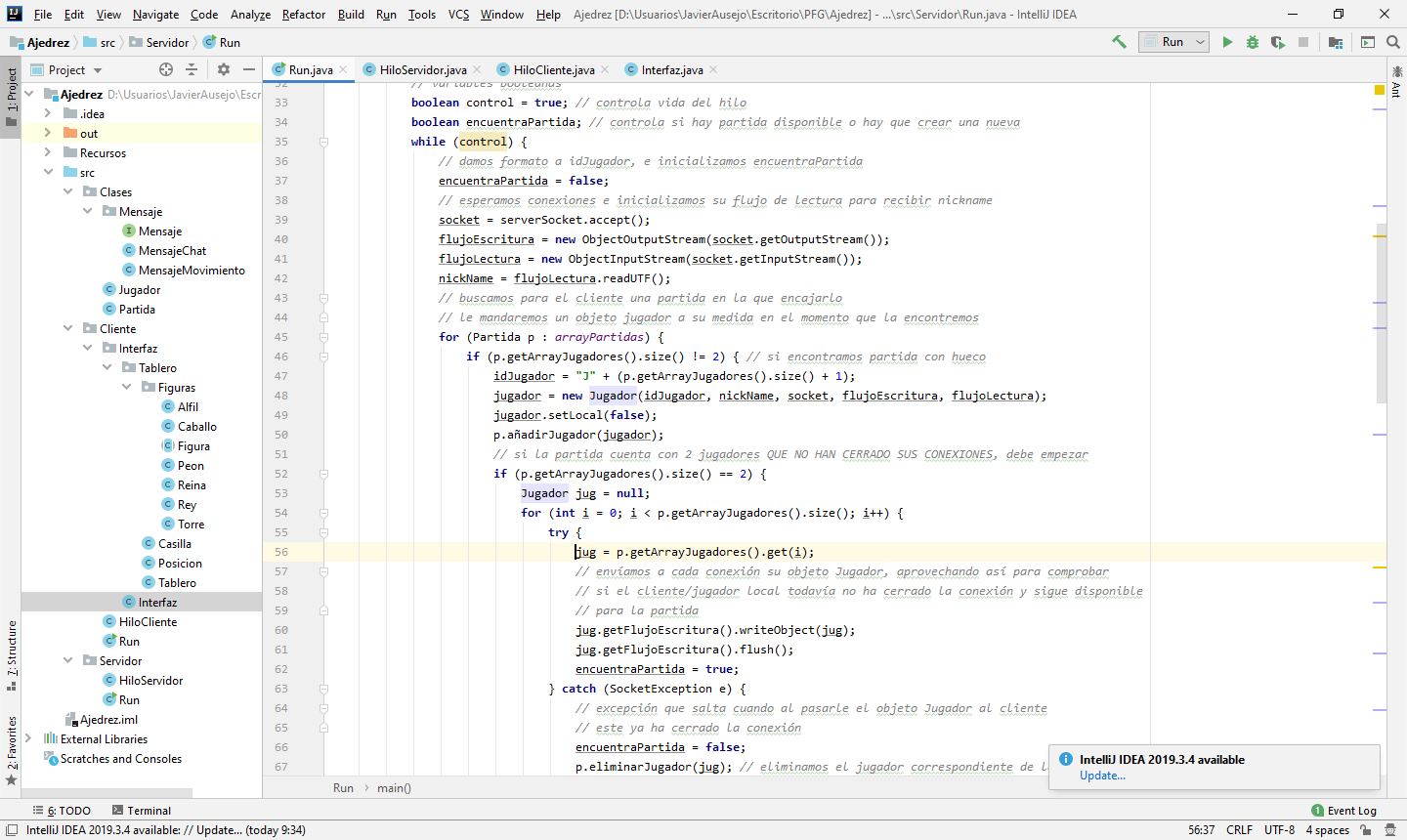


1. Iniciamos el *Run.java* del cliente, abriéndose una interfaz de usuario en la que deberemos incluir un nombre de usuario y clicar sobre el botón conectar.

Se ejecutará el método *run()* de *HiloCliente.java*, en el que se controlan las posibles excepciones de error que puedan salir al tratar de conectarse. En este método, **el cliente se conecta al servidor y le envía el nickname escrito en la interfaz**.

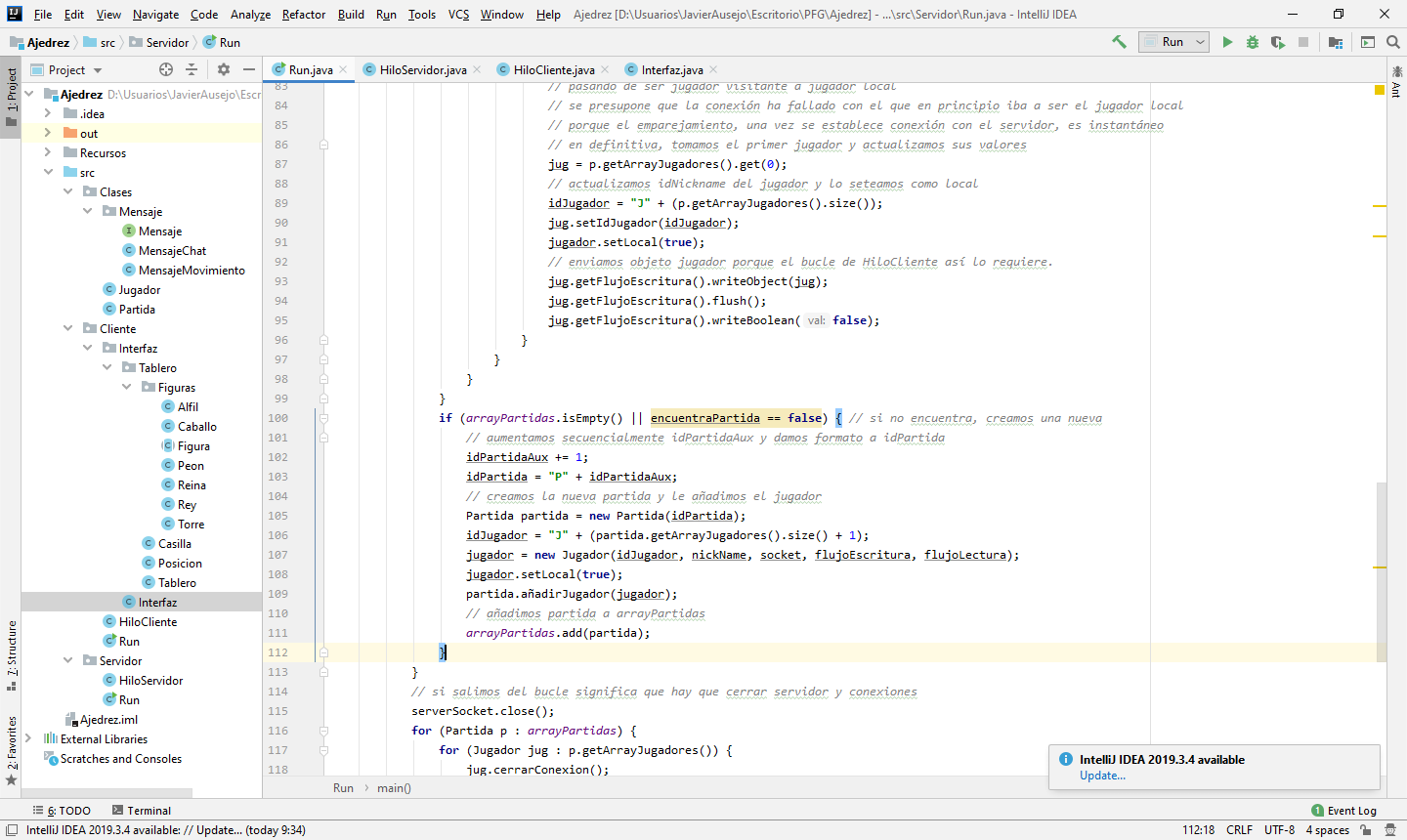


1. El **servidor recibe el nickname del cliente en cuestión**, generando previamente el correspondiente flujo de lectura y flujo de escritura de la conexión pertinente.

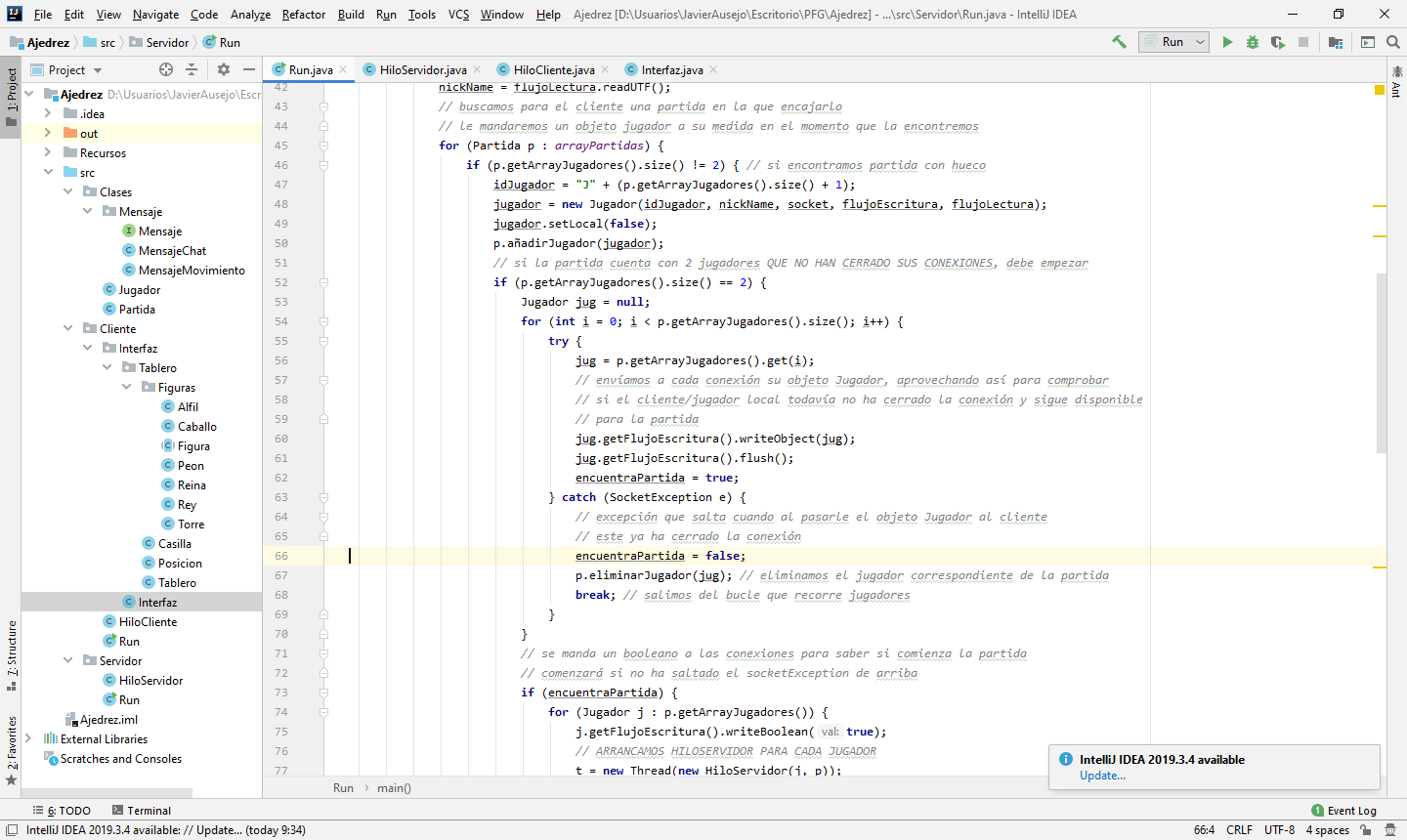


**En este punto comprendemos dos posibilidades: que el servidor esté recién arrancado y por tanto no haya ninguna partida creada a la que vincular al jugador, o bien que ya existan una serie de partidas y haya que rebuscar en ellas para saber si esta conexión tiene cabida en alguna**.

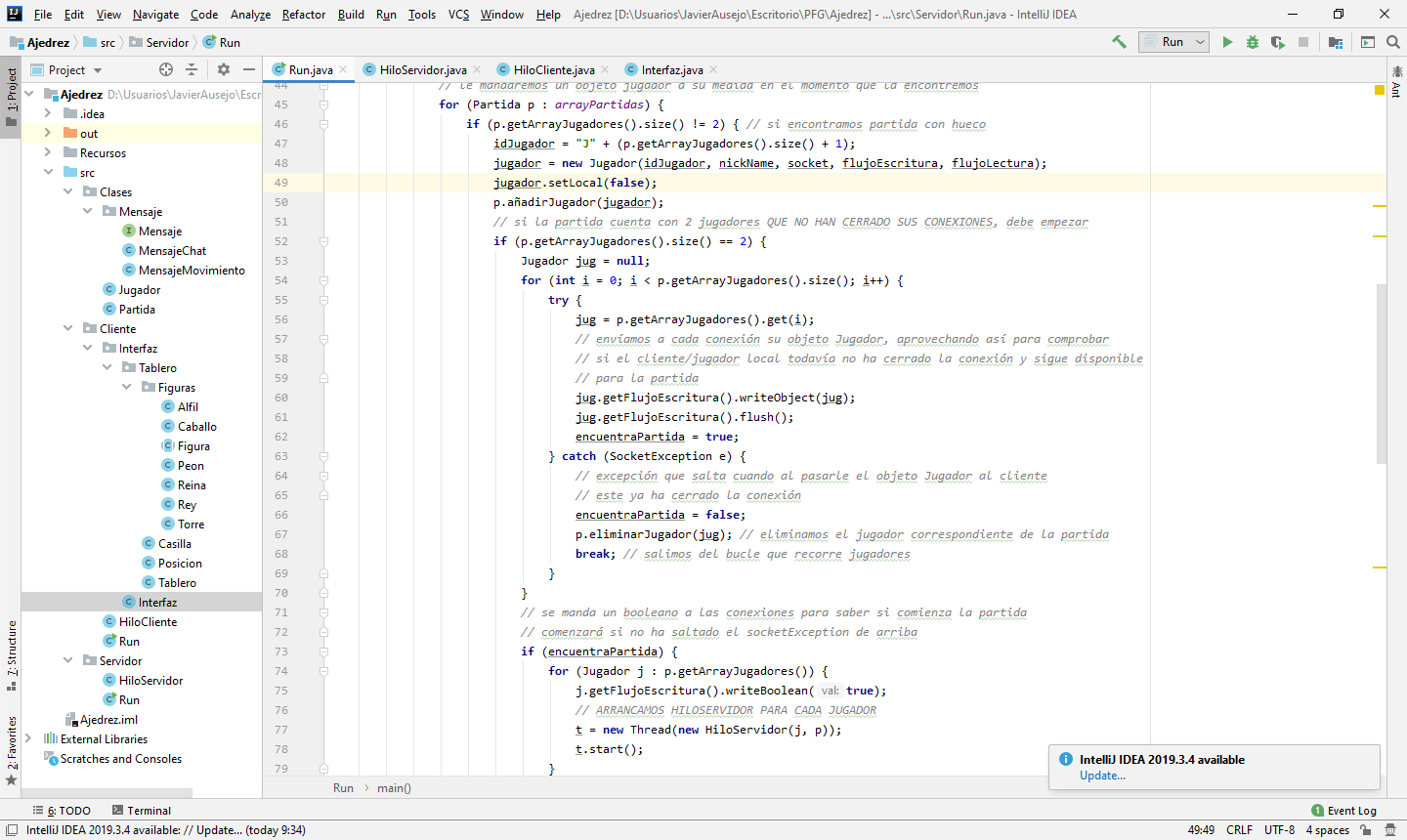
Si no existe ninguna partida, o en la búsqueda de partidas ya existentes el jugador no tiene cabida porque ya hay 2 clientes en cada una de ellas, se ejecuta esta parte del código en la que se crea una nueva partida y se añade la conexión entrante con el identificador “J1” y como local, es decir, como poseedor de las figuras blancas.



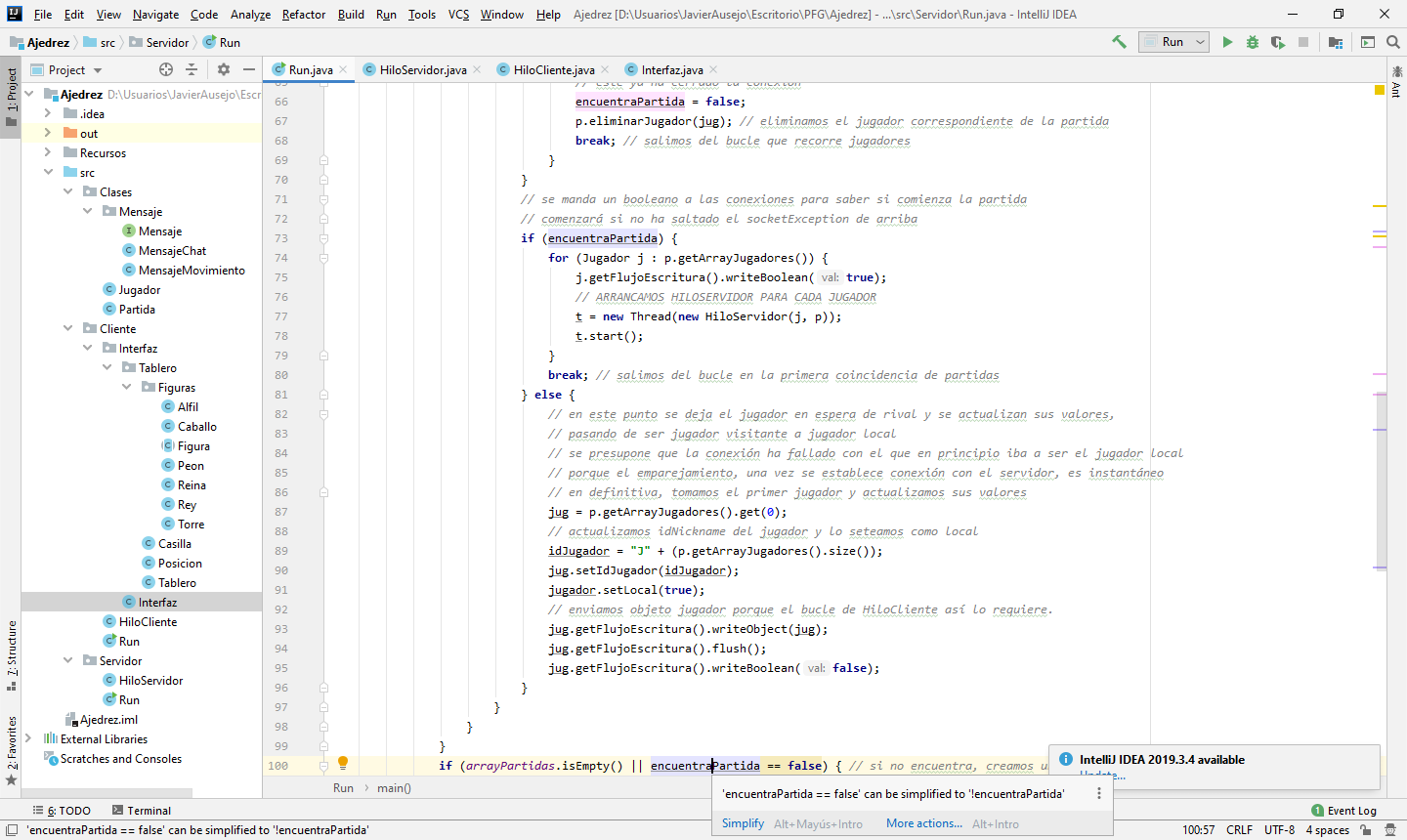
Si, por el contrario, se encuentra una partida con un hueco libre para la conexión entrante, se presupone que esta conexión tiene que ser la segunda en la partida y se le asigna el identificador “J2” y las figuras negras.



A continuación, el camino vuelve a dividirse en dos posibilidades puesto que **puede ser que “J1”, en el proceso de espera de conexión de un rival, se haya desconectado del servidor**. Por ello, antes de dar comienzo a la partida, **a cada jugador de la partida se le debe mandar una instancia *Jugador.java* con sus datos representativos para corroborar que las conexiones de ambos clientes siguen en pie**. Si saltara alguna excepción, la variable booleana “*encuentraPartida*” se pondría a *false*.

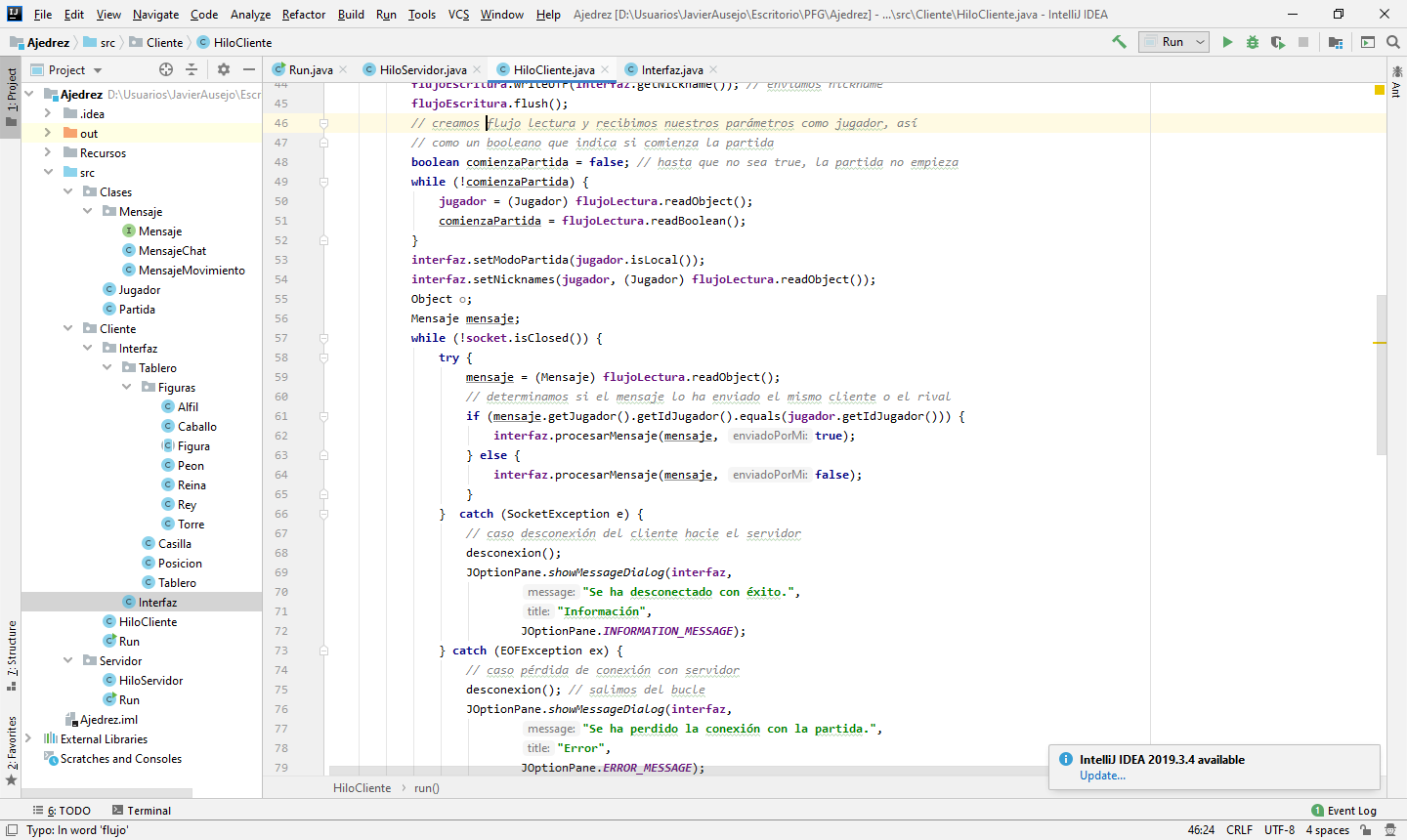


Si la conexión de “J1” siguiera disponible, es decir, si “*encuentraPartida”* estuviera a *true*, a cada uno de los jugadores de la partida correspondiente se les mandaría por red un booleano de valor verdadero para darles a entender que la partida debe comenzar. Si “*encuentraPartida*” está a *false*, el jugador “J2”, que controlaba figuras negras, pasaría a ser “J1” con figuras blancas y se le mandaría un booleano a valor falso para darle a entender que su emparejamiento todavía está pendiente de encontrar un rival.



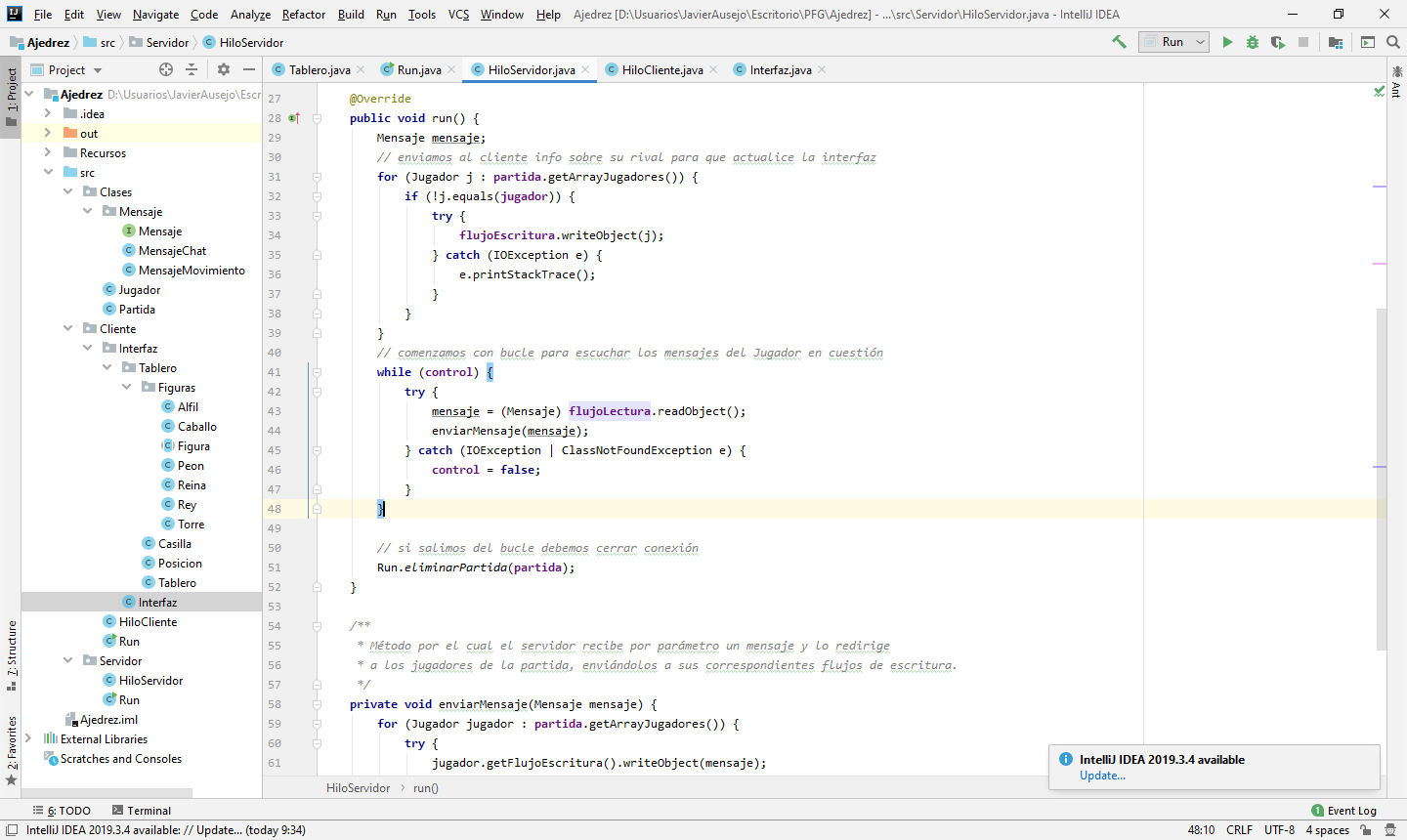
1. El **cliente**, ajeno a la lógica de emparejamiento del servidor, está programado de tal manera que queda **bloqueado en un bloque *while()* hasta que no recibe por red un booleano de valor verdadero**, lo que significaría el pistoletazo de salida de la partida habilitando y deshabilitando los elementos del la GUI correspondientes para empezar la partida.

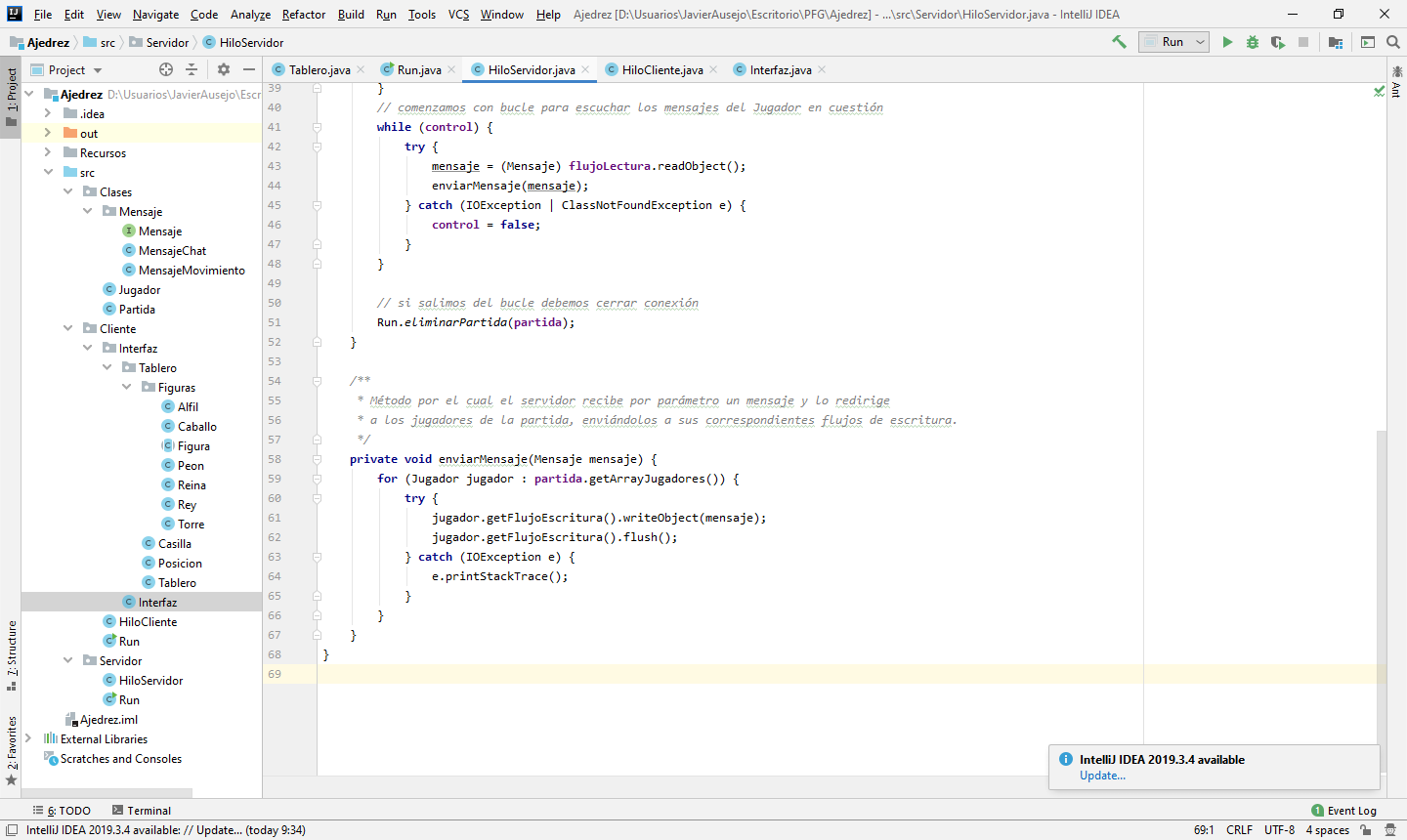
Como vemos, **se recibe una instancia *Jugador.java* que determina nuestro identificador y si jugamos con figuras blancas o negras**, lo que hace que nuestra interfaz cargue a nuestra disposición un color u otro.



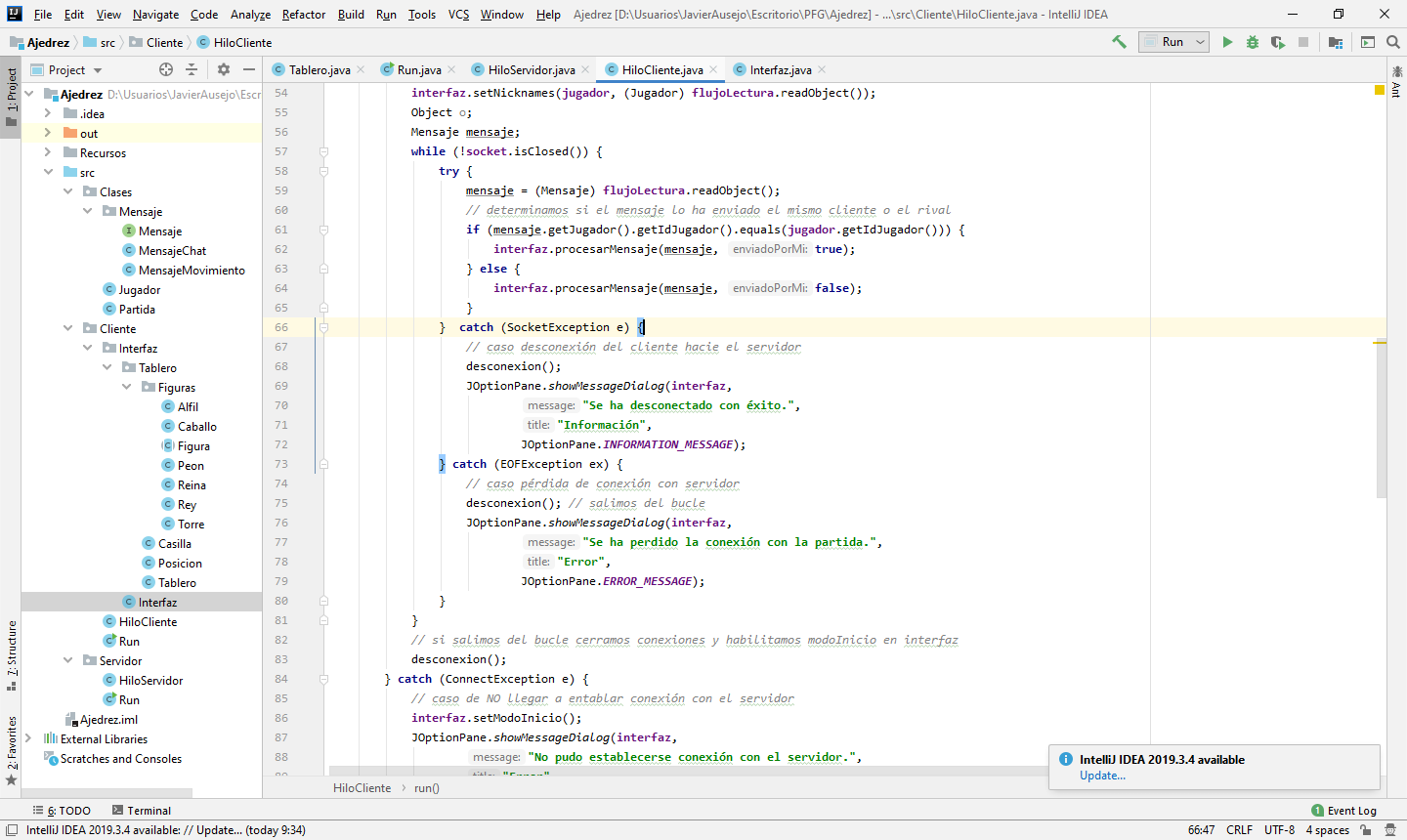
1. Una vez vinculados dos jugadores en una misma partida, **por parte del servidor ejecutamos el método *run()* de *HiloServidor.java* para cada uno de los jugadores de la partida**. Se encarga de recibir instancias de *Mensaje.java* de los clientes de una partida para, posteriormente, reenviar dichos objetos a cada uno de los jugadores de la partida.

En el momento que saltara una excepción en la recepción de los mensajes de red, eliminaríamos la partida desconectando a los clientes oportunos del servidor.





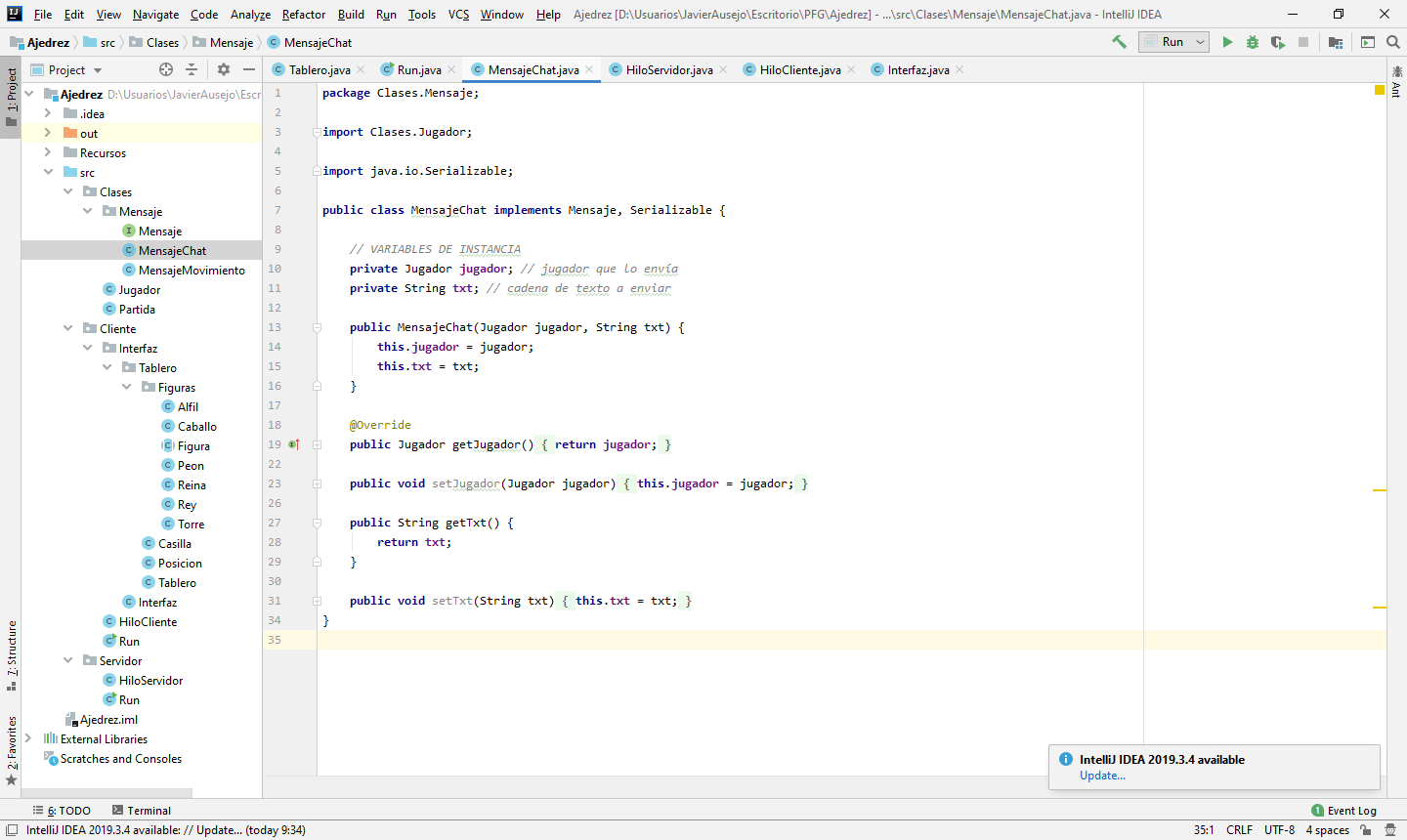
1. Finalmente, **por parte del cliente, la conexión queda atrapada en un bucle *while()* mientras el *socket* esté disponible**. Para poder identificar si el mensaje que se recibe del servidor lo hemos mandado nosotros o lo ha mandado el rival, nos apoyamos en el identificador que asignamos a cada uno de los jugadores de la partida.

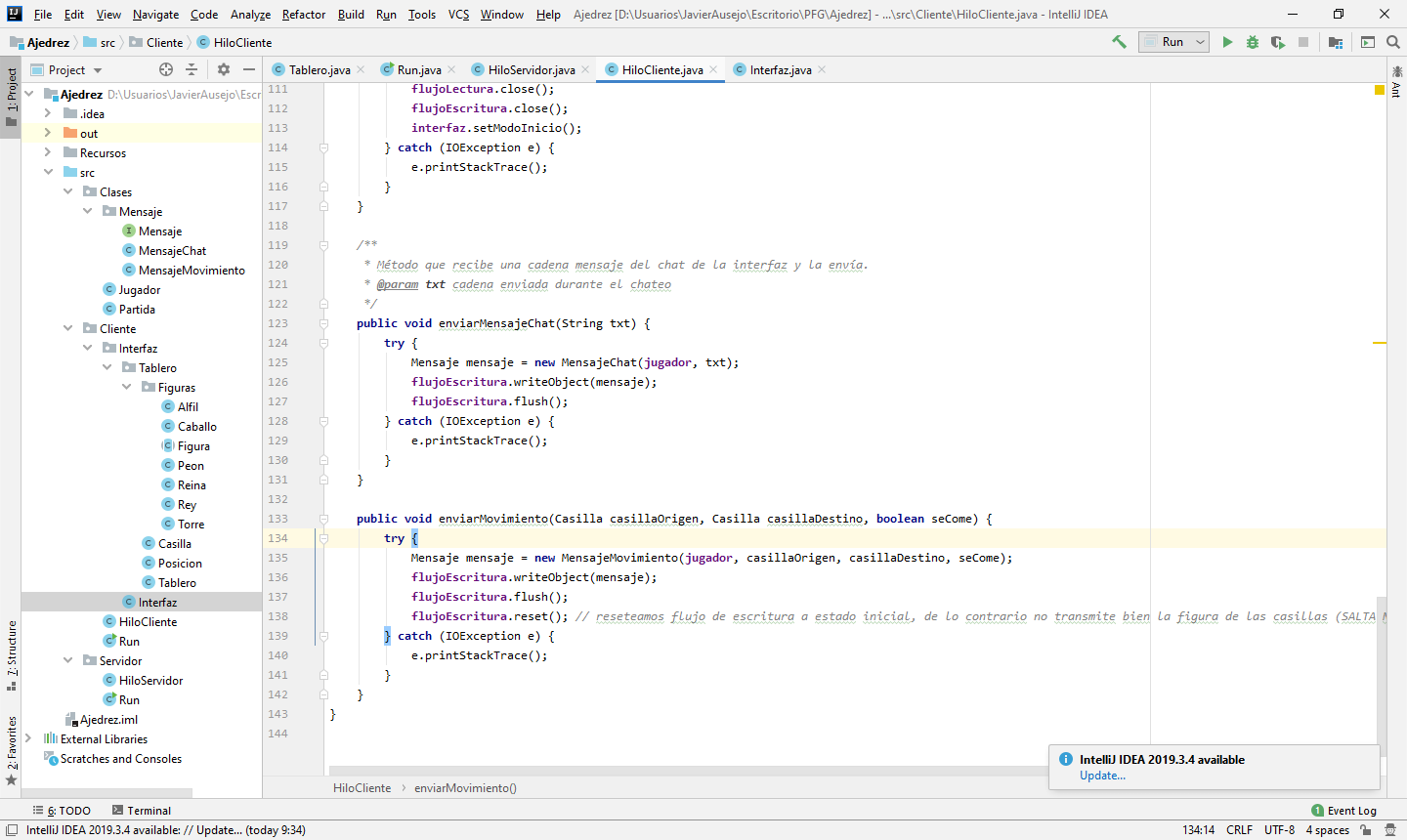


En este punto del documento, una vez hemos simulado el arranque del servidor y su gestión de las conexiones entrantes hasta el punto de emparejarlos, debemos proporcionar a cada cliente las herramientas necesarias para poder generar instancias de *Mensaje.java* y enviarlas al servidor. A continuación, **nos centramos en el servicio chat de la aplicación**.

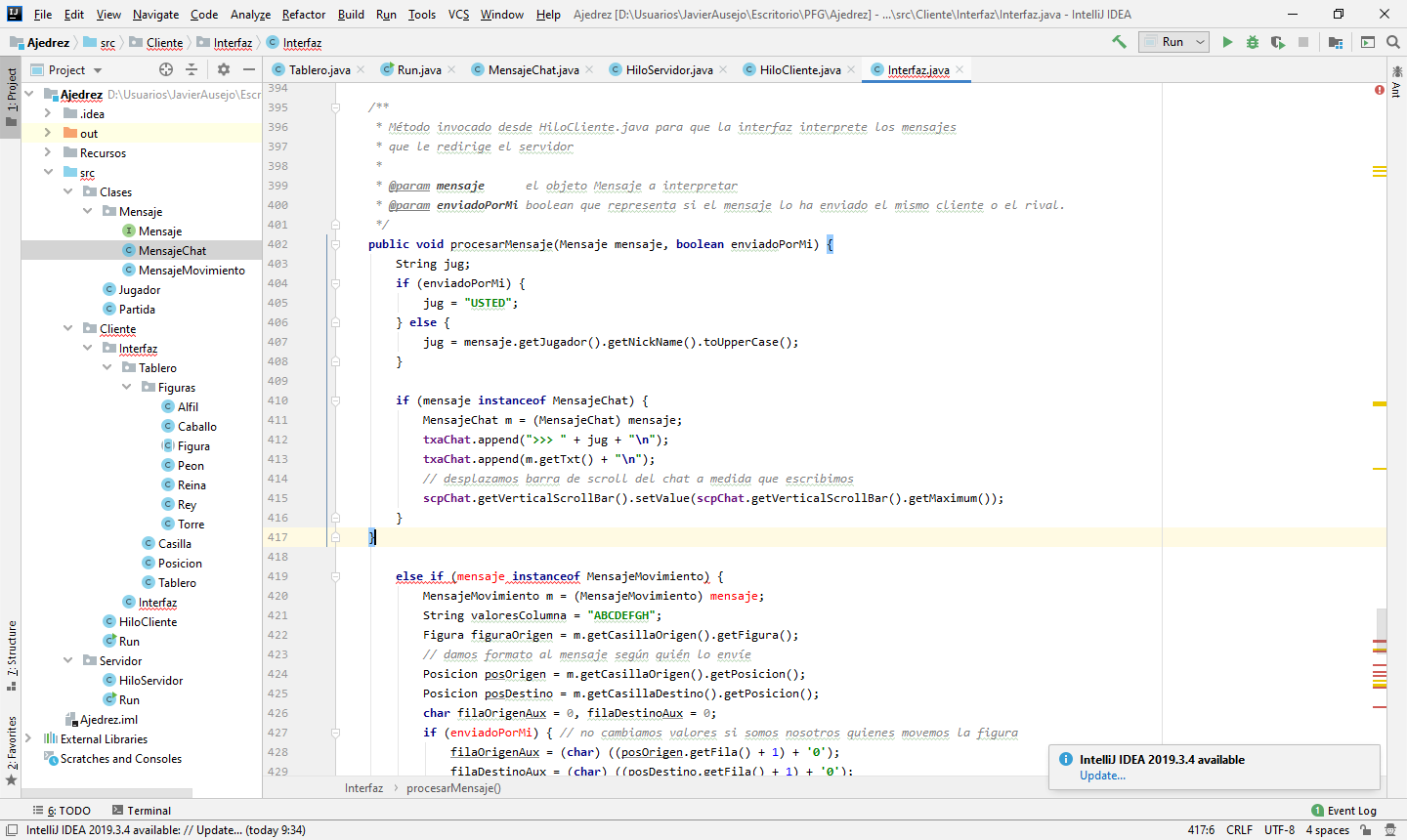
Si recordamos la GUI, nos acordaremos de que hay un *JTextField* dedicado a que los jugadores escriban los mensajes que quieren enviar por chat a sus rivales. Es por ello que, cada vez que pulsemos la tecla *Enter* sobre este *JTextField* invocaremos al método ***enviarMensajeChat()*** que hemos definido en la clase *HiloCliente.java*  para enviar instancias de *MensajeChat.java*.

Como podemos intuir de la siguiente imagen, **añadimos a *Mensaje.java* el método abstracto** ***getJugador()***.



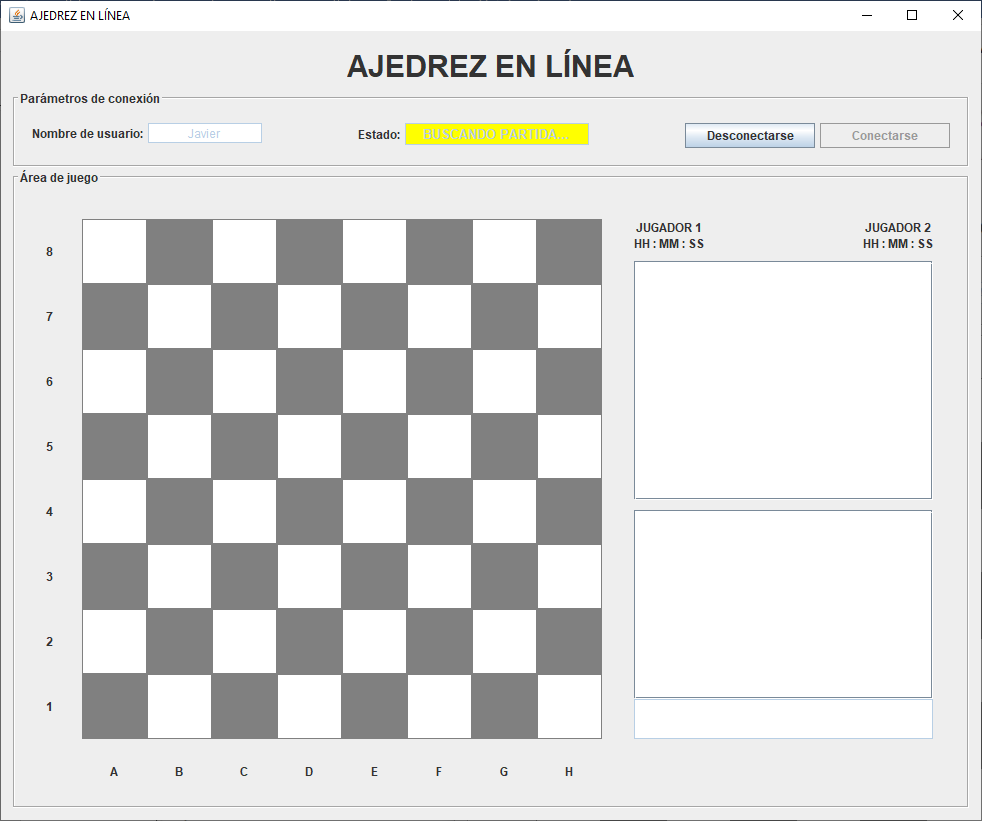


Una vez explicado el flujo de mensajes entre los jugadores a través del servidor, es momento de explicarle a la interfaz cómo interpretar los *MensajeChat.java*.



Para acabar con este sprint, observamos el aspecto final que debe tomar nuestra interfaz durante las diferentes fases del emparejamiento, así como una vez cargadas las figuras en el tablero y habiendo establecido la comunicación vía chat entre los diferentes jugadores:

\*VISTA JUGADOR ESPERANDO RIVAL:



\*VISTA JUGADOR LOCAL EMPAREJADO:



\*VISTA JUGADOR VISITANTE EMPAREJADO:



# AMPLIACIÓN Y POSIBLES MEJORAS.

# CONCLUSIÓN.

# BIBLIOGRAFÍA.