DETALLES DEL DISEÑO

PATRONES USADOS

Patrón *MVC*

1. Organización de archivos y paquetes

1.1. Archivos de diseño asociados (Modelio):

1.1.1. proyecto 24-04 Class diagram.png (versión más actualizada).
1.1.2. Class diagram historia 7.png
1.1.3. Class diagram historia 9.png
1.1.4. LoadGame Class diagram.png
1.1.5. Multiplayer Class Diagram.png

1.2. Clases principales asociadas al patrón:

1.2.1. **MatchMaker**: funciona como nexo del modelo, donde se concentra la lógica principal de una partida, incluyendo a los jugadores, tablero, atributos para la elección de partidas contra la IA o en modo multijugador, y una serie de funciones auxiliares (la mayoría *getters*) para intercambiar datos con el resto de clases.

1.2.2. MainController: funciona como conexión entre MatchMaker y la vista, por ello comparte muchos nombres de métodos con MatchMaker, ya que solo son llamadas al mismo, aunque está siendo refactorizado para no tener tal cantidad de funciones repetidas. No contiene casi lógica aunque toma un papel más importante en el patrón *Memento*, ya que incluye un atributo para guardar, pero eso será detallado posteriormente. Implementa a una interfaz *Controller* con un método *playTurnG* que controla la jugada de un turno.

1.2.3. MainView: reúne todas las clases de la vista en un *frame*, como paneles o componentes del juego como *EmptyCircle.* También contiene al Controller y a través de los *ActionListener* comunica al Controller todas las tareas que

tiene que mandar al modelo, así como los resultados que debe devolver.

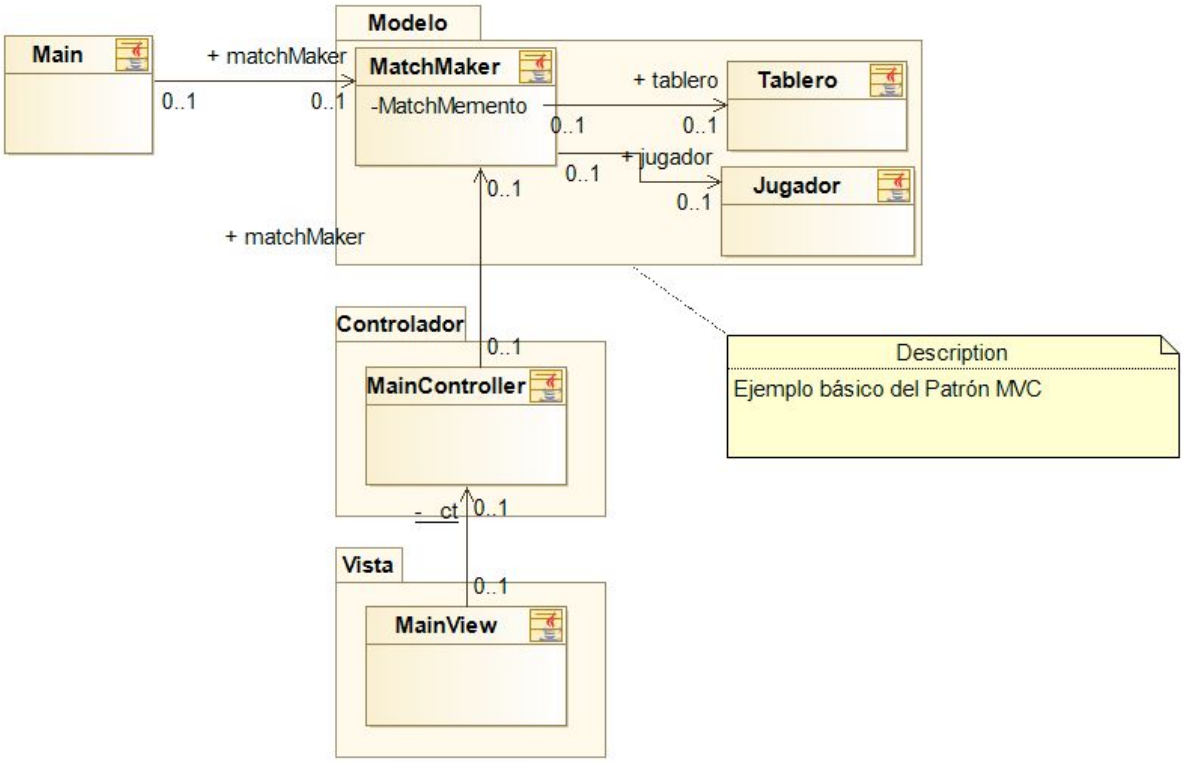
1.2.4. **MultiView:** En el caso del multijugador también hemos intentado usar una especie de patrón MVC en el que el MultiView es la vista.

1.2.5. Server-Player: Harían la función de modelo ya que el servidor posee MatchMaker para poder realizar cada jugada y que el juego se desarrolle de manera correcta informando a las vistas (cada uno de los jugadores) sobre lo que deben mostrar.

1.2.6. MultiController: Como su propio nombre indica es el encargado de controlar las comunicaciones entre modelo y vista ya que es el que posee el socket con el que se comunica el cliente con el servidor.

2. Maquetación del diseño: proyecto 24-04 Class diagram

Este archivo de Modelio es sobre el que mejor se visualiza el patrón MVC. Se observan las relaciones de asociación entre las clases enumeradas anteriormente y las relaciones con las clases adheridas a ellas. Es un esquema completo y general sobre el funcionamiento de todo el proyecto, por eso es más recomendable acceder a los esquemas más concretos para mayor entendimiento del juego. Se explicarán cada una de sus funcionalidades por separado en otro documento. En el caso del multijugador su funcionamiento se ve en MultiplayerClassDiagram.



Patrón *Memento*

1. Organización de archivos y paquetes
   - **1.1. Archivos de diseño asociados (Modelio):**
     - 1.1.1. LoadLastSave Sequence diagram.png
     - 1.1.2. Save Sequence diagram.png
   - 1.2. **Clases principales asociadas al patrón:**
     - 1.2.1. **MenuBar**: desde el menú agregado al JFrame puedes seleccionar alguna de las tres opciones asociada al patrón: Load last save, que cargará el último guardado que se haya hecho durante la ejecución del programa, de modo que si no se ha hecho ningún guardado esta opción no será seleccionable; Save guarda el estado actual de la partida en un *Memento* y se escribe sobre un archivo JSON para poder ser cargado luego; Load from file, abre una ventana para seleccionar un archivo válido JSON y recuperar el estado del juego, independientemente de si haya sido guardado durante la ejecución o en una anterior.
     - **1.2.2. MainController:** tiene dos funciones, guardar el memento creado al guardar una partida y comunicarse con el

MatchMaker, pasándole el memento creado, el archivo seleccionado por el usuario o creando el memento para guardar y meterlo en un archivo.

1.2.3. **MatchMaker:** contiene la clase privada Memento (para no romper los principios de encapsulación), que contiene los atributos necesarios para guardar toda la información de estado, y contiene el método que introduce esos datos en formato JSON a un archivo. Luego se encarga de crear el Memento y llamar a su función para guardar en un archivo; cargar de un archivo haciendo uso de una clase *Parser* para transformar los datos JSON y cargar de Memento guardado en MainController.

2. Maquetación del diseño

2.1. LoadGame Class diagram

Cuando el usuario ordena a través de MenuBar cargar el último guardado se suceden los siguientes pasos:

2.1.1. Se llama al método loadLastSave() de Controller.

2.1.2. El método de Controller llama al método undoToLastSave(mm: MatchMemento) pasándole como argumento el Memento con la información necesaria para la carga.

2.1.3. El método undoToLastSave asignará sus atributos a los atributos del Memento recibido, y luego avisará a todos los observadores a través de reload().

2.2. Save Sequence diagram

Después de elegir un usuario un archivo en el seleccionador de archivos siguen los pasos:

2.2.1. Se llama al método saveToFile(f: File) de Controller pasándole el archivo seleccionado para guardar.

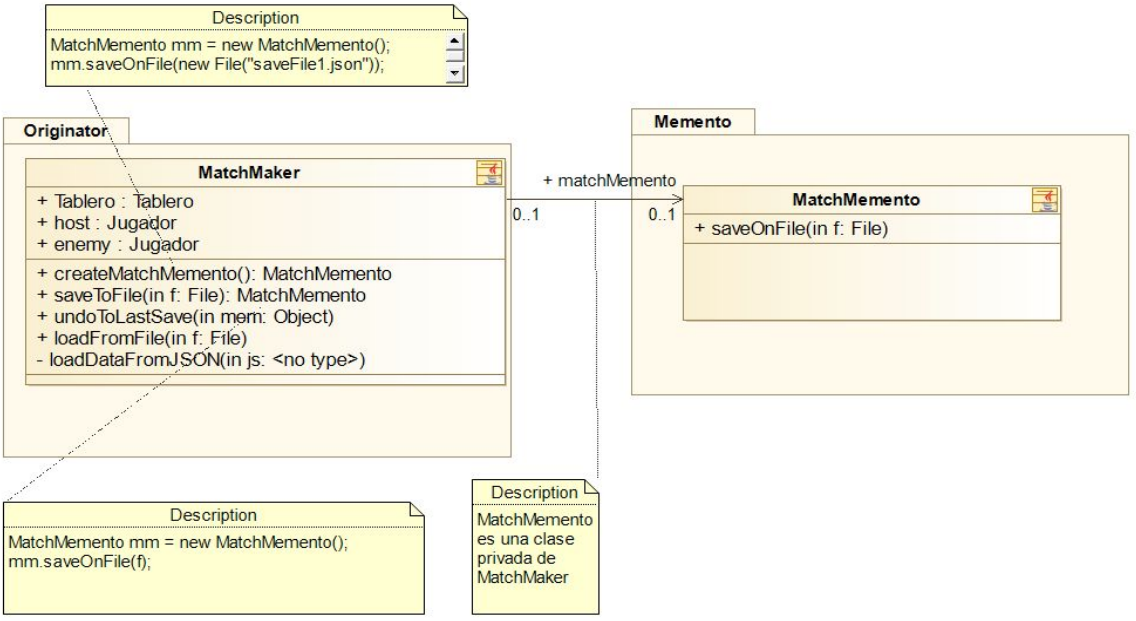
2.2.2. El método de Controller llama al método con el mismo nombre en MatchMaker.

2.2.3. MatchMaker hace uso de su clase privada MatchMemento, creando uno nuevo y llamando a su método saveOnFile(f: File), que ya se encarga de transformar los datos a formato JSON y escribirlos en el archivo indicado. El método de

MatchMaker retorna el Memento generado para que sea guardado por Controller.

2.3. LoadLastSave Sequence diagram

Describe el mismo comportamiento del diagrama de clases para la funcionalidad de cargar del último guardado.



Patrón *Estrategia*

Se usa el patrón Estrategia para el desarrollo de la "inteligencia artificial" y su comunicación con la interfaz.

1. Organización de archivos y paquetes

1.1. Archivos de diseño asociados (Modelio):

1.1.1. Historia 15-IA.png

1.1.2. Historia 16 - Tres niveles de dificultad.png

1.2. Clases principales asociadas al patrón:

1.2.1. **OptionsMenu:** permite seleccionar y deseleccionar la opción de jugar contra la máquina y elegir los niveles.

1.2.2. **MainController:** pasa al modelo el nivel seleccionado por el jugador, y le informa para que habilite el modo IA.

1.2.3. **MatchMaker:** según el nivel elegido, usando el patrón estrategia se transforma al enemigo en uno de los jugadores IA diferentes.

1.2.4. **ComputerInterface:** interfaz de la que heredan las clases de los distintos niveles de dificultad:

1.2.4.1. **WeakAI**

1.2.4.2. **MediumAI**

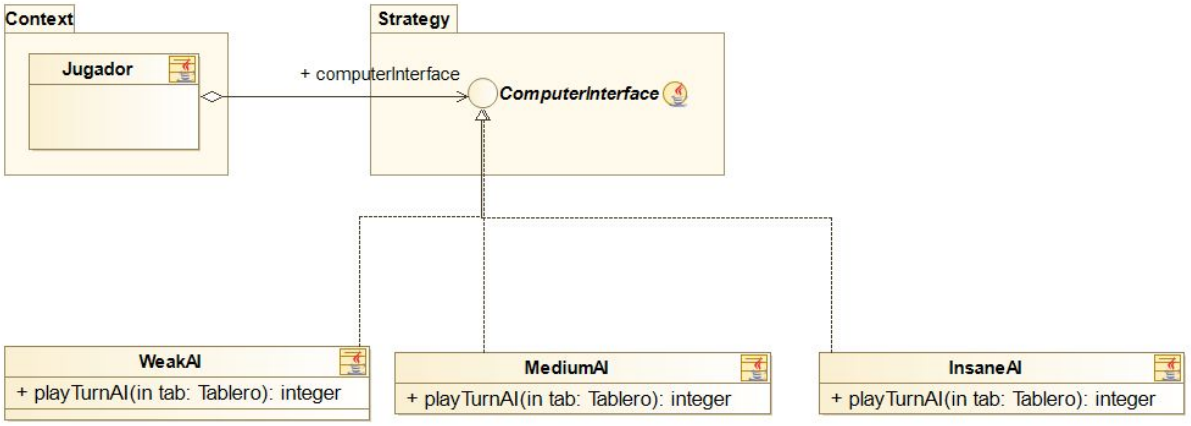
1.2.4.3. **InsaneAI**

1.2.5. **AILevel:** enumerado de niveles.

2. Maquetación del diseño

2.1. Historia 16 - Tres niveles de dificultad

Según el nivel elegido por el usuario, se le pasa al controlador para que notifique al modelo. Haciendo uso del patrón estrategia, el modelo crea uno u otro jugador IA, que implementan la interfaz ComputerInterface y que implementan diferentes algoritmos para jugar los turnos.



Patrón Composición

Se usa este patrón implícitamente en la creación de interfaces debido al uso de Java Swing. Java Swing es una herramienta de interfaz gráfica (GUI) ligera que incluye un amplio conjunto de widgets.

La biblioteca Swing está construida sobre el conjunto de herramientas de widgets abstractos de Java (AWT), un kit de herramientas GUI más antiguo que depende de la plataforma. Puede utilizar los componentes de la GUI de Java como el botón,

el cuadro de texto, etc. de la biblioteca y no tiene que crear los componentes desde cero. Algunos componentes están formados por otros componentes más básicos.

1. Organización de archivos y paquetes
   - **1.1. Archivos de diseño asociados (Modelio):**
   - **1.2. Clases principales asociadas al patrón:**

     Todas las del paquete view.
     - **1.2.1. AIOptionPanel**
     - **1.2.2. BombButton**
     - **1.2.3. CronoPanel**
     - **1.2.4. EmptyCircle**
     - **1.2.5. FilledCircle**
     - **1.2.6. FinalScreen**
     - **1.2.7. InitMenu**
     - **1.2.8. MainView**
     - **1.2.9. MenuBar**
     - **1.2.10. MultiInitMenu**
     - **1.2.11. MultiView**
     - **1.2.12. OptionsMenu**
     - **1.2.13. ScoreBoardTable**
     - **1.2.14. SelectCronoPanel**
     - **1.2.15. TabPanel**
     - **1.2.16. TextPrompt**
     - **1.2.17. WinnerScreen**

Patrón Decorador

1. Organización de archivos y paquetes
   - **1.1. Archivos de diseño asociados (Modelio):**
   - **1.2. Clases principales asociadas al patrón:**
     - **1.2.1. FinalScreen**

       Se le añaden bordes al panel del ScoreBoard

Patrón Fachada

1. Organización de archivos y paquetes

1.1. Archivos de diseño asociados (Modelio):

1.1.1. Fachada Class diagram.png

1.2. Clases principales asociadas al patrón:

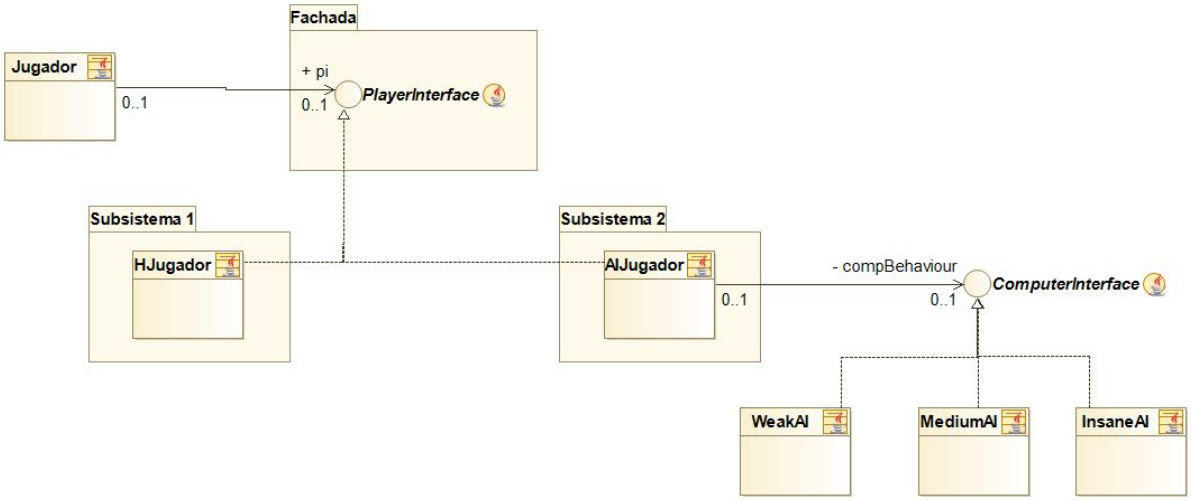
1.2.1. PlayerInterface: funciona como la interfaz que opera los diferentes subsistemas. En este caso, el sistema es la estructura de los jugadores. PlayerInterface es implementada siguiendo la estructura de un patrón estrategia, siendo un atributo privado en la clase Jugador.

1.2.2. HJugador: creado por la clase Jugador aplicando la estructura del patrón estrategia. Es el subsistema 1 del patrón fachada.

1.2.3. AIJugador: creado por la clase Jugador aplicando la estructura del patrón estrategia. Es el subsistema 2 del patrón fachada. Luego esta clase aplica la interfaz ComputerInterface aplicando el patrón estrategia explicado anteriormente.

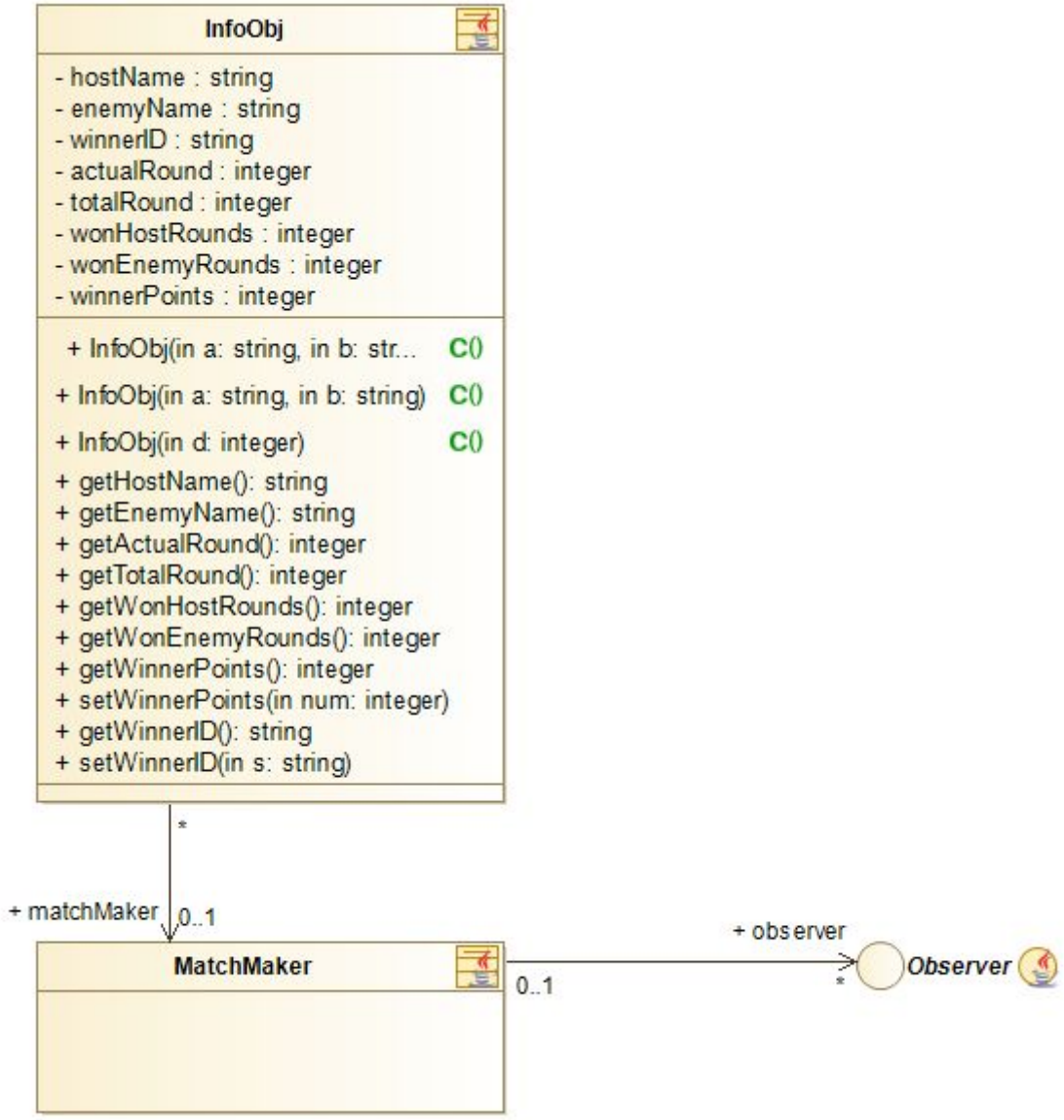
2. Maquetación del diseño: Fachada Class diagram

2.1. Este archivo de Modelio es sobre el que mejor se visualiza el patrón Fachada aplicado a la estructura para crear los diferentes jugadores. Los subsistemas son los diferentes jugadores que se pueden crear en el juego. En nuestro juego solo hay, hasta la fecha, dos tipos de jugadores. Uno es el jugador normal y el otro es la CPU o IA. Dependiendo de la opción de juego que se escoja, el modelo creará un jugador u otro y de esa manera cada jugador realiza los métodos necesarios para realizar su turno.



Patrón DTO

El patrón DTO (Data Transfer Object) fue propuesto por Martin Fowler en su libro de patrones y es bastante simple. Consiste en el uso de un objeto que transporta datos entre procesos para reducir el número de llamadas a métodos.



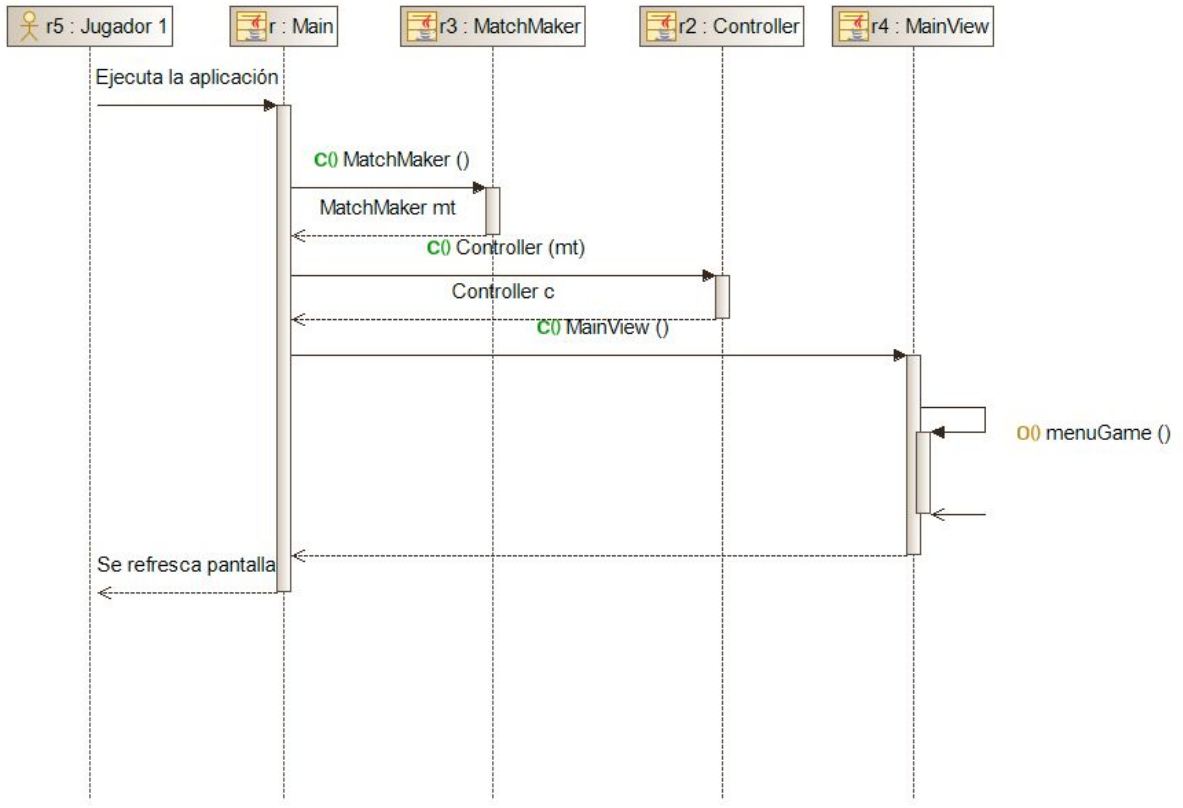
DISEÑOS POR HISTORIAS

Creemos que la mejor forma de plasmar la funcionalidad de las historias es a través de los diagramas de secuencia por ello les hemos dado más importancia. Además consideramos que las historias que no tienen diagrama de clases no lo necesitan ya que la información que proveerían sería nula o muy baja.

HISTORIA 01 - Inicio del Juego

Al ejecutarse el juego y la aplicación se inicia, el main crea dos objetos. El primero es el Matchmaker (MatchMaker mt = new MatchMaker()) y el segundo es el Controller al cual se le pasa el MatchMaker. Finalmente se crea otro hilo con el MainView (interfaz gráfica) y a este se le pasa el Controller. De esta manera utilizamos el patrón MVC (Vista: MainView, Controlador: Controller, Modelo: MatchMaker).

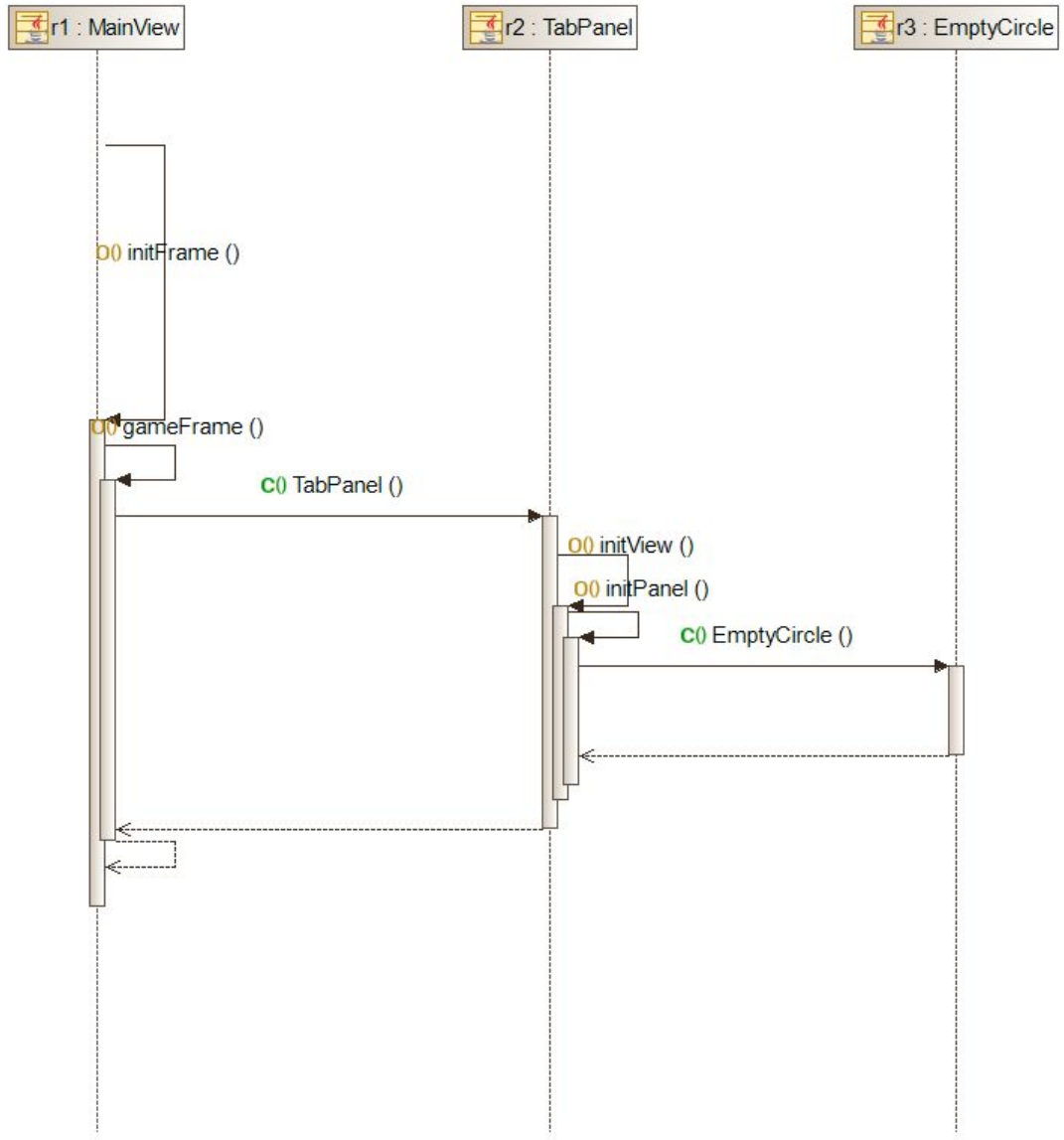
El mainView ejecuta el método menuGame que se encarga de crear cada una de las pantallas del juego y se escoge a la menuPanel como la pantalla a mostrar al principio.



HISTORIA 03 - Tablero de juego (GUI)

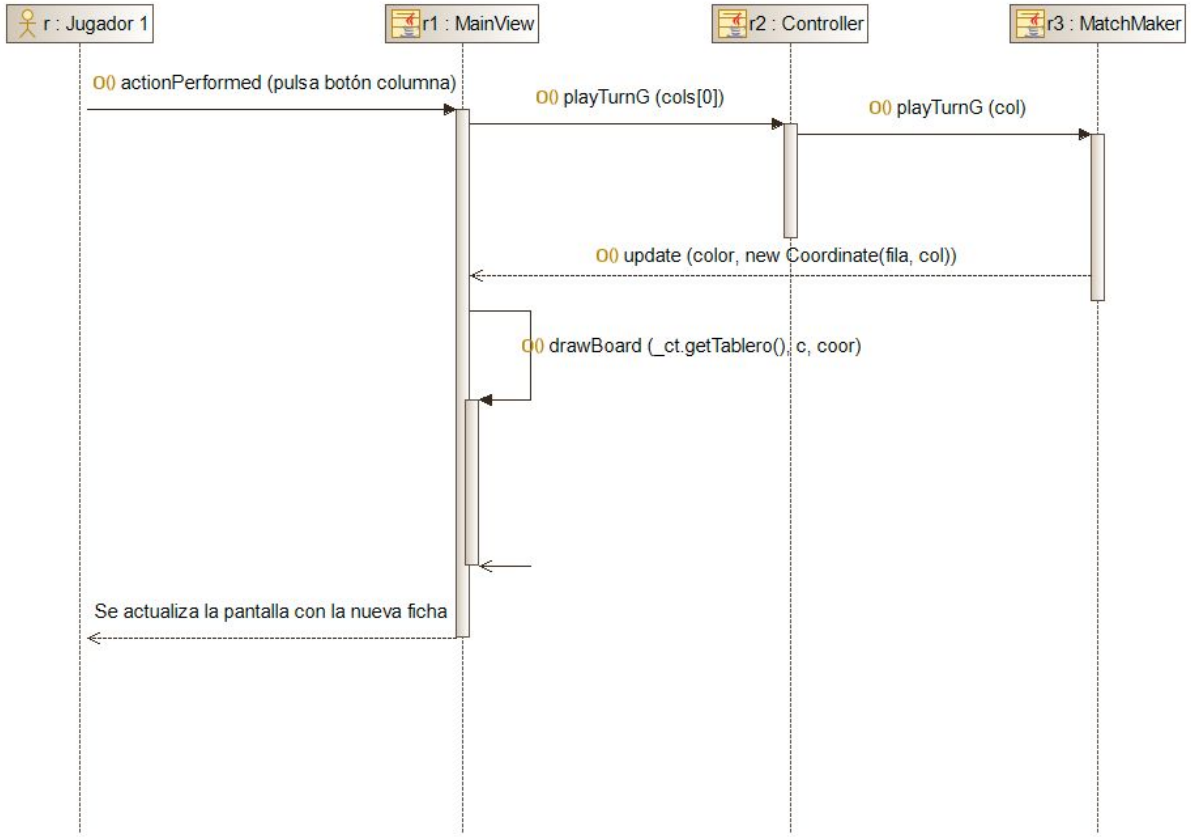
Al inicializarse el MainView, se ejecuta el método InitFrame que llama al método GameFrame del tablero, pasando como parámetro el propio tablero. Estos dos métodos se encargan de inicializar el JFrame del juego y GameFrame crea la vista del tablero.

GameFrame crea un nuevo TabPanel (clase TabPanel) que es el panel donde estará el Tablero. Al crearse esta clase se ejecuta el método InitView que se encarga de inicializar toda la vista del tablero, que ejecuta a su vez el método InitPanel. El método InitPanel crea una matriz de EmptyCircle (clase EmptyCircle) que dibuja todos los círculos en blanco para mostrar el tablero vacío.
Esto al pintarse crea el tablero deseado, formado por círculos vacíos que posteriormente se rellenarán.



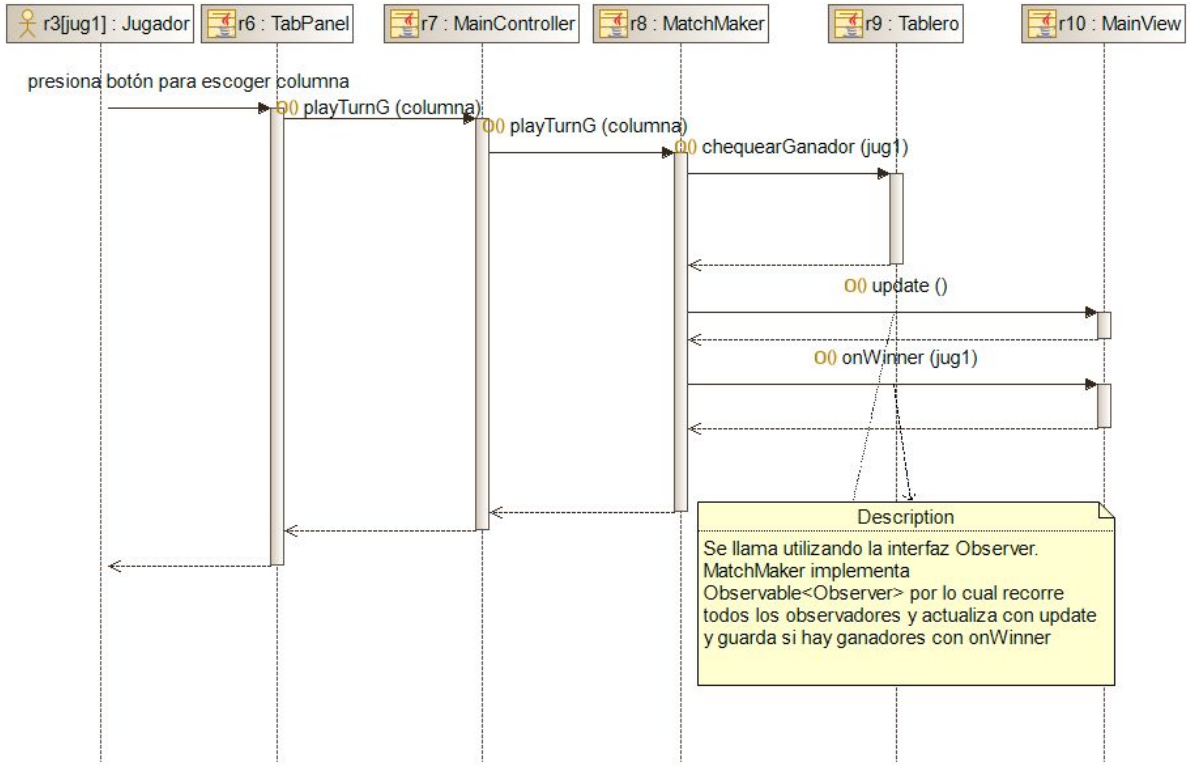
HISTORIA 04 - Introducir ficha

El jugador pulsa un botón de una de las columnas y al ejecutarse el actionPerformed de ese botón se ejecuta el playTurnG del controller y se le pasa por parámetro la columna seleccionada y ese método llama a PlayTurnG del matchmaker pasandole la columna.
Tras eso el matchmaker hace un update a la vista y se ejecuta el método DrawBoard en MainView con el tablero actualizado. De esta forma se inserta una nueva ficha en el tablero



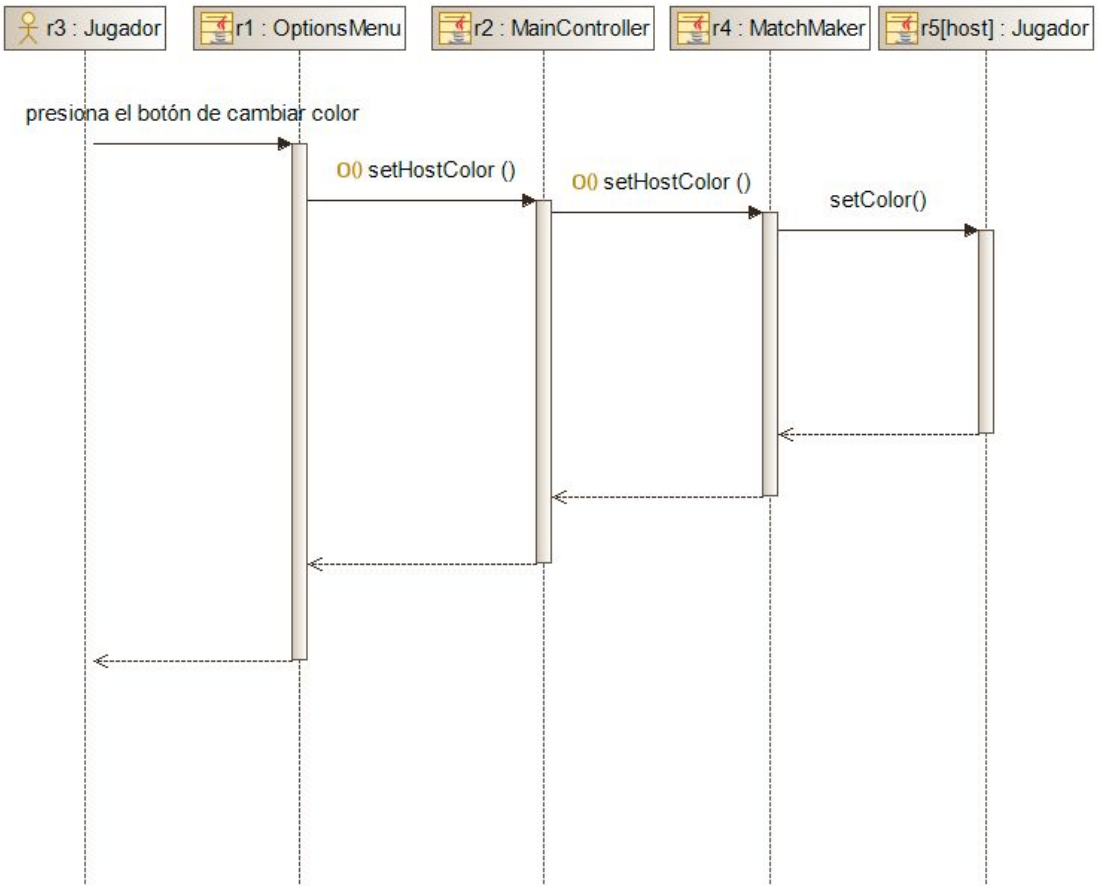
HISTORIA 05 - Finalización de la Partida (Gana un jugador)

En esta historia estamos en el paso donde al jugador jug1 le queda sólo una ficha para ganar. Este escoge la columna que lo hará ganar y TabPanel actualiza que esa columna tiene una ficha nueva y se lanza un evento que llama al ActionPerformance del botón escogido. Este método llama al playTurnG de MainController pasando por los argumentos la columna que se escogió. Este mismo llama al del MatchMaker ya que este es el modelo.
Este método controla las partidas donde chequea si ganó o no el jugador actual. Para chequear llama al Tablero y este devuelve el jugador que ha ganado la partida. Si no ha ganado nadie devuelve null.
Luego haciendo uso de la interfaz observer, todos los observadores reciben las actualizaciones a través del update y si hay un ganador, llama onWinner con el ganador de la partida. De esta manera la vista podrá actualizar la pantalla con la ficha ganadora y con la pantalla de finalización de ronda o partida.



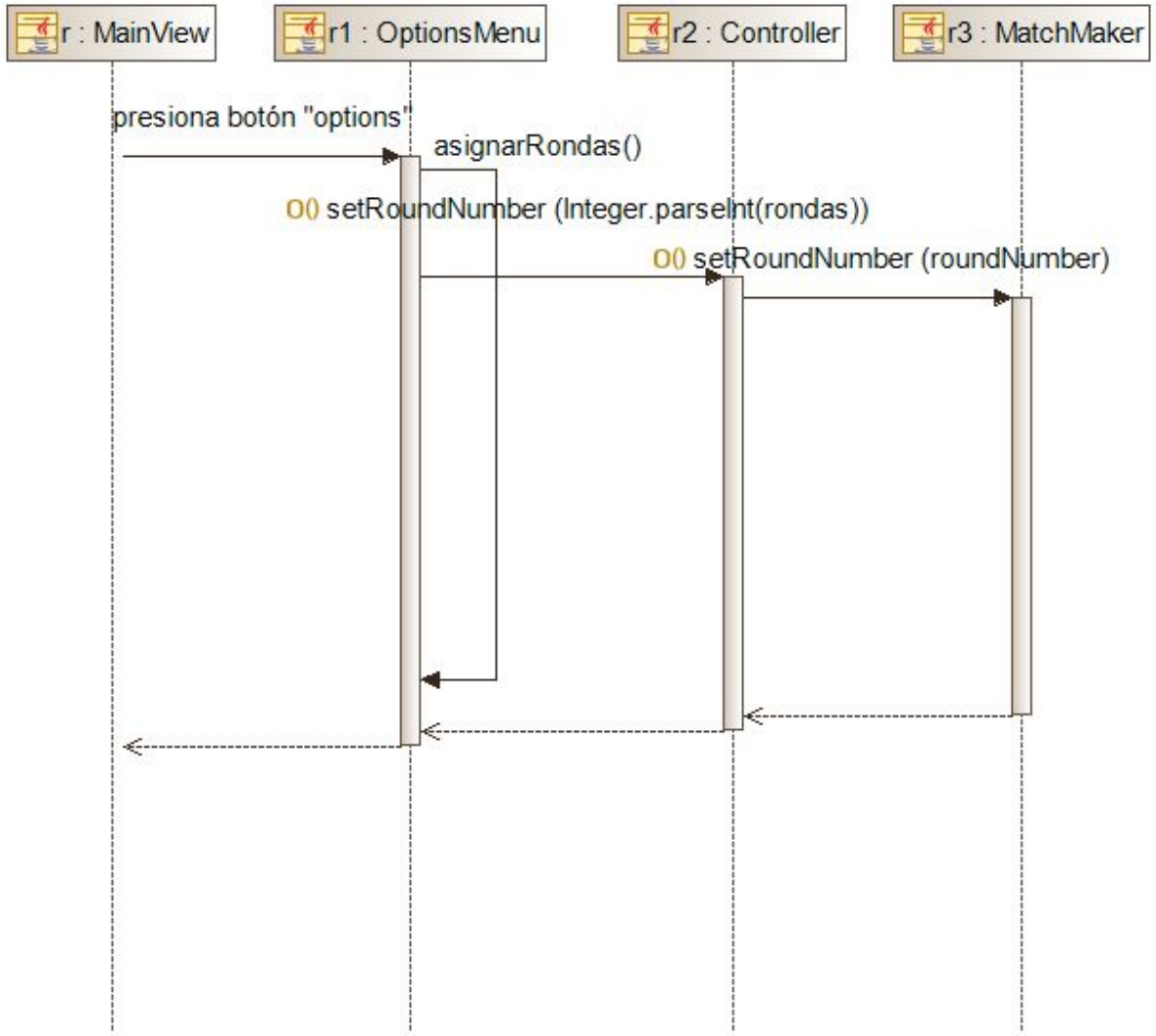
HISTORIA 06 - Color fichas

Cuando el jugador quiere cambiar el color de su ficha escoge el botón options en el MainView y este cambia a la otra pantalla que es la de OptionsMenu. Este posee varios botones entre los cuales está el de cambiar color de las fichas. Al seleccionarlo se abre un JColorChooser y este Color se pasa al MainController y luego al MatchMaker (modelo). Este con el jugador que ha seleccionado cambiar su color, accede a su objeto de jugador en el matchmaker haciendo un set del color nuevo.



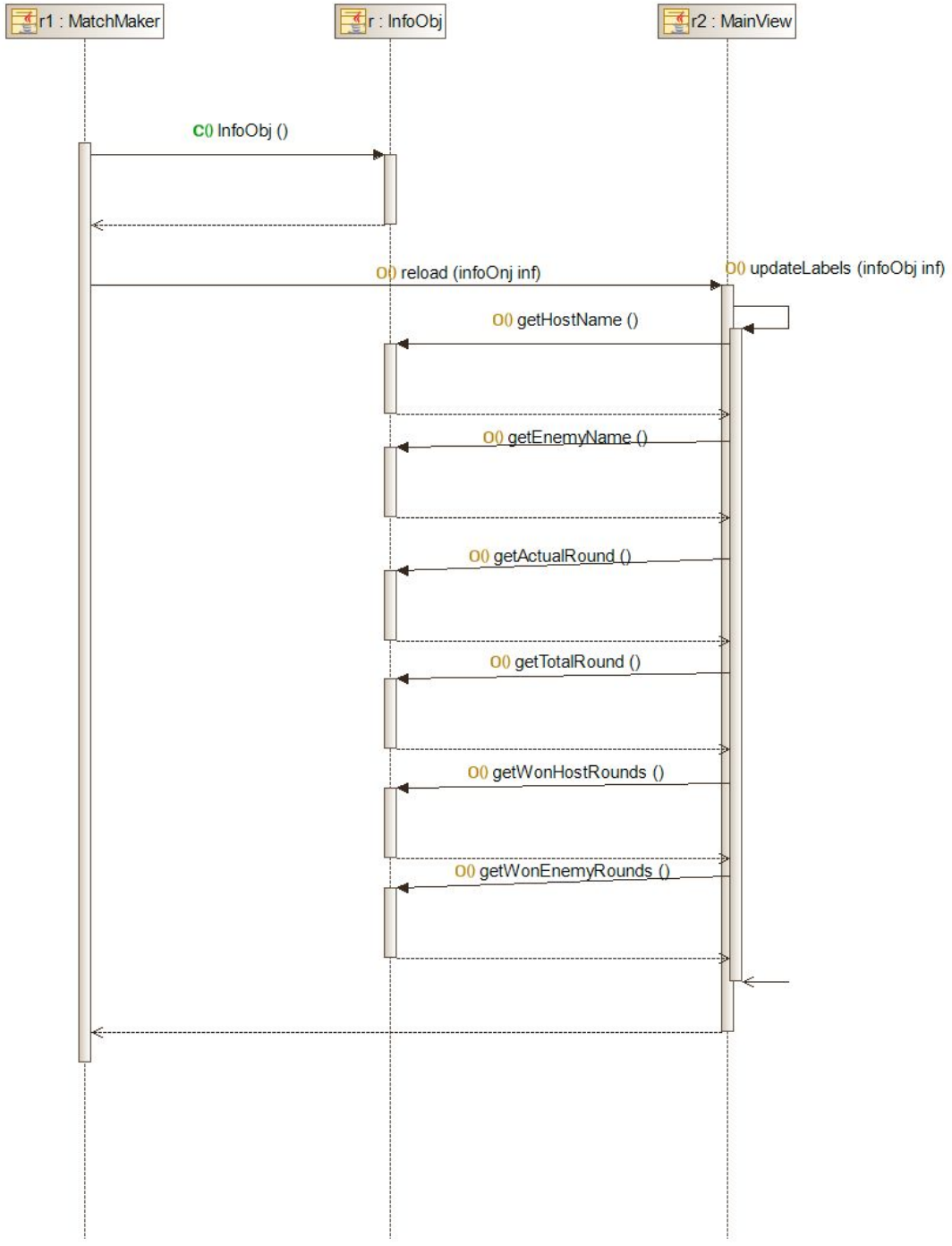
HISTORIA 07 - Número de Rondas

Al pulsar el botón options de la vista(clase Mainview) se accede a la clase OptionsMenu. En ella está la opción de cambiar rondas. Al pulsar ese botón se ejecuta el método AsignarRondas que obtiene el número de rondas que se desean. Este método llama al método setRoundNumber del controlador (clase MainController con interfaz Controller) con las rondas deseadas como parámetro. Finalmente este método llama al método setRoundNumber del modelo (clase MatchMaker) y modifica el número de rondas que el juego va a tener.



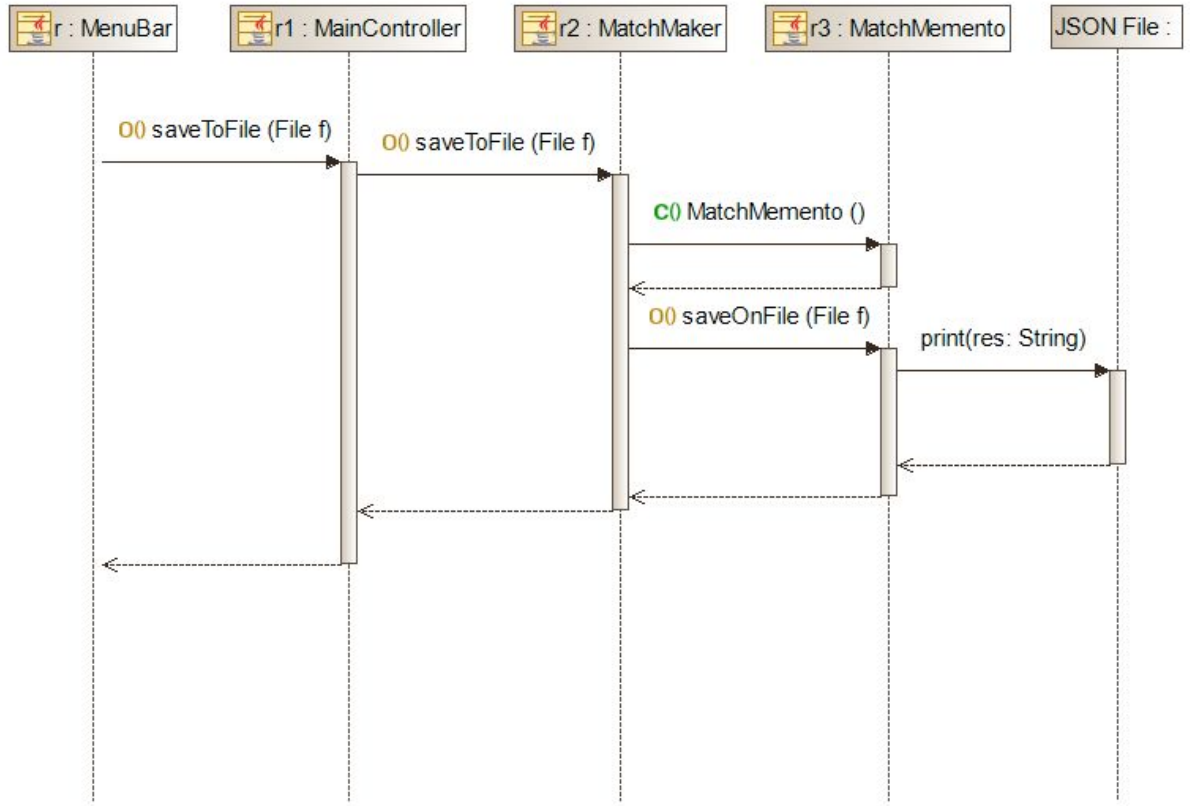
HISTORIA 09 - Marcador

El MatchMaker crea una instancia de InfoObj con los parámetros deseados.
El MainView implementa la interfaz Observer que posee una funcion llamada reload que como parámetro tiene ese InfoObj creado en MatchMaker.
Cada vez que se realizan cambios relacionados con el InfoObj se notifica al observador(MainView) a través de la función reload.
El MainView obtiene los datos necesarios del InfObj a través de gets.



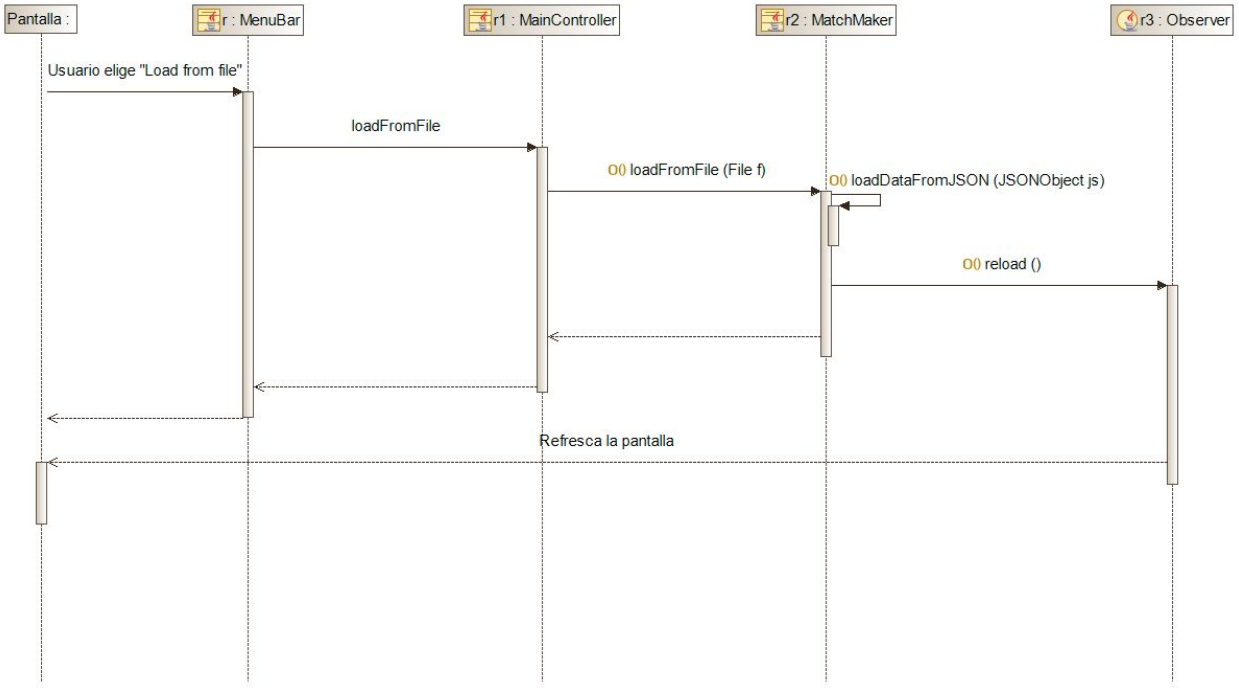
HISTORIA 11 - Guardar Partida

Cuando el jugador quiere guardar una partida, selecciona en el MenuBar “guardar” y este llama al MainController (controlador) para contactar al MatchMaker (modelo) que posee una clase privada llamada MatchMemento (Patrón Memento). MatchMaker llama al saveOnFile del MatchMemento para salvar la ejecución en un archivo JSON.



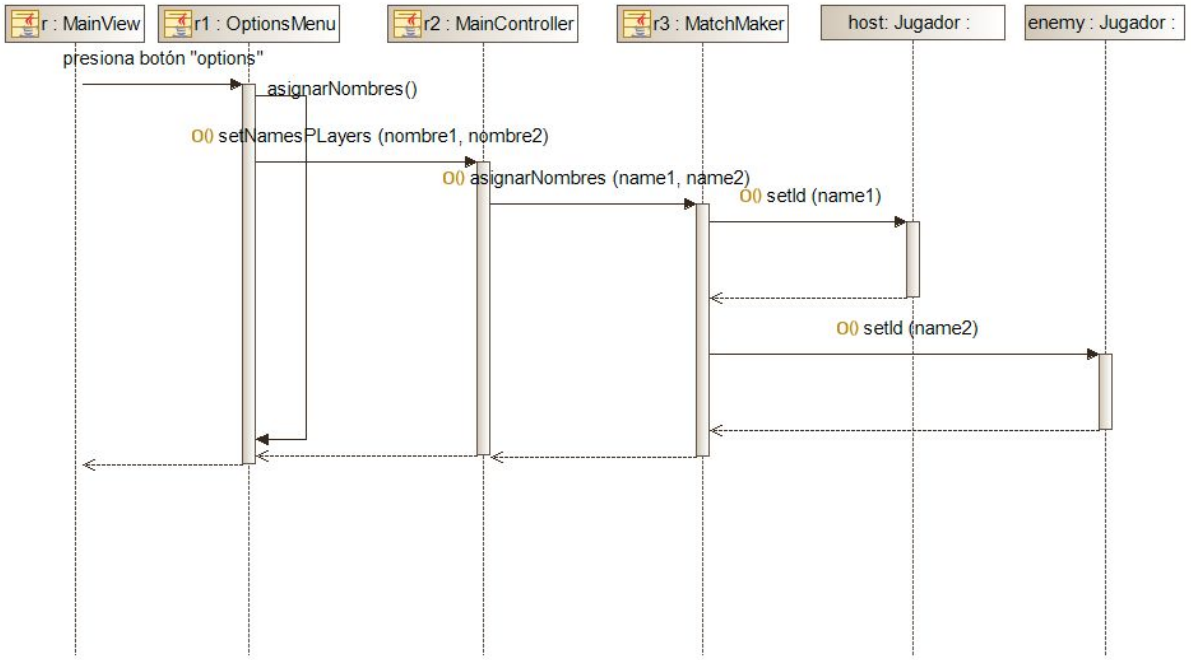
HISTORIA 12 - Cargar Partida

El jugador selecciona en pantalla cargar desde un fichero, se abre un JFileChooser para escoger el File y este se carga al MenuBar y se pasa al MainController. Como este es el controlador, transmite al modelo (MatchMaker) para cargar todos los objetos con loadDataFromJSON y usa el reload para informar a los observadores que se han creado nuevos objetos. Finalmente se refresca la pantalla con la información nueva.



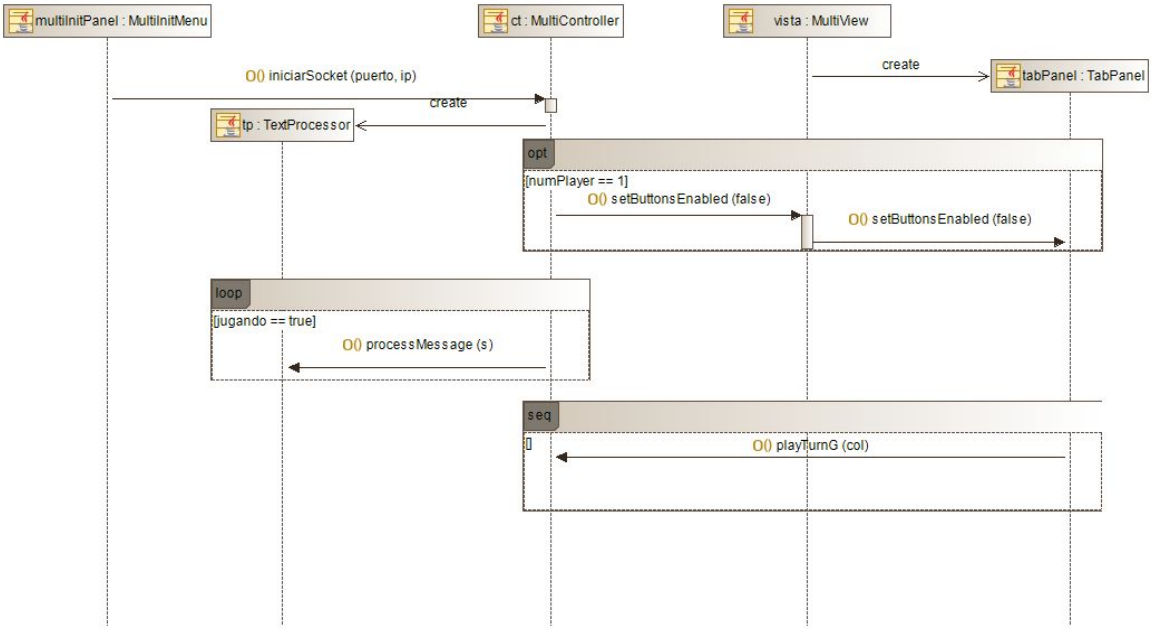
HISTORIA 13 - Cambiar nombre de los jugadores

Cuando los jugadores quieren cambiar sus nombres, en el MainView presionan el botón de options y se cambia a la pantalla de OptionsMenu. En esta hay varios botones, el que escogen es el de “CAMBIAR NOMBRE DE LOS JUGADORES”. Este utiliza el método asignarNombres() de OptionsMenu el cual, haciendo uso de MainController, se le pasa los nombres escritos en los JDialogs al MatchMaker para que este pueda guardar en el modelo los nuevos Ids. El MatchMaker posee dos atributos de jugador (host y enemy) para los dos jugadores del Conecta 4. Se realizan dos setId: host.setId(name1) y enemy.setId(name2) y así cada jugador guarda los nuevos nombres.



HISTORIA 14.1- Cliente

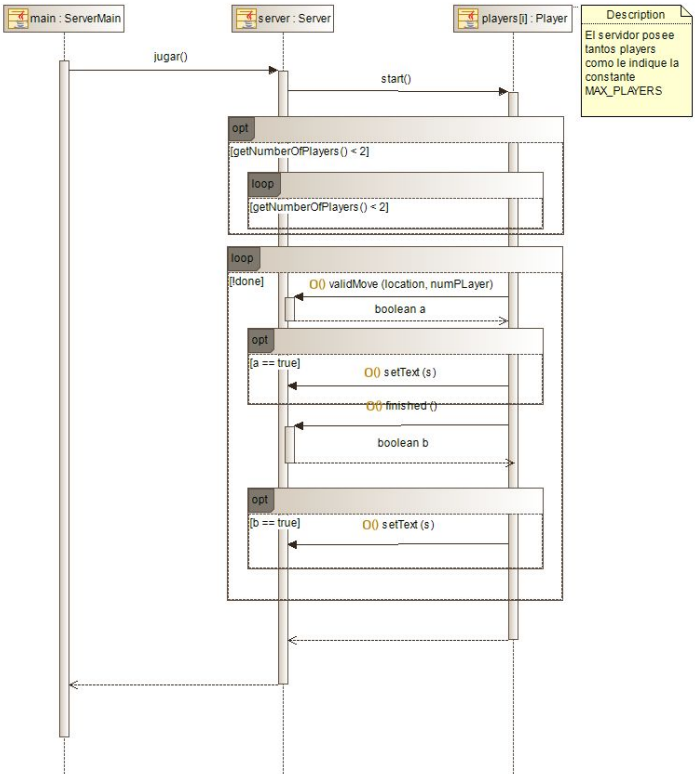
El diagrama representa el funcionamiento de la parte del cliente a la hora de jugar el modo multijugador. Podría dividirse en 2 partes que se comunican (MultiView + TabPanel como vista y MultiController) y una inicial que es el multiInitMenu, esta última simplemente se encarga de crear lo necesario para iniciar la conexión con el servidor (iniciarSocket). Toda conexión con el servidor se realizará a través del controlador por lo que es el encargado de decirle a la vista lo que debe mostrar y a la vez de comunicarle al servidor la acción que la vista quiere realizar (playTurnG).
Puede crear confusión la forma en la que se le informa a la vista de lo que debe hacer por lo que lo explicaré aquí. De ello se encarga el TextProcessor que es una clase interna al controlador que se encarga de procesar los mensajes recibidos del servidor y de realizar las acciones que convengan en cada caso llamando a funciones de MultiView.

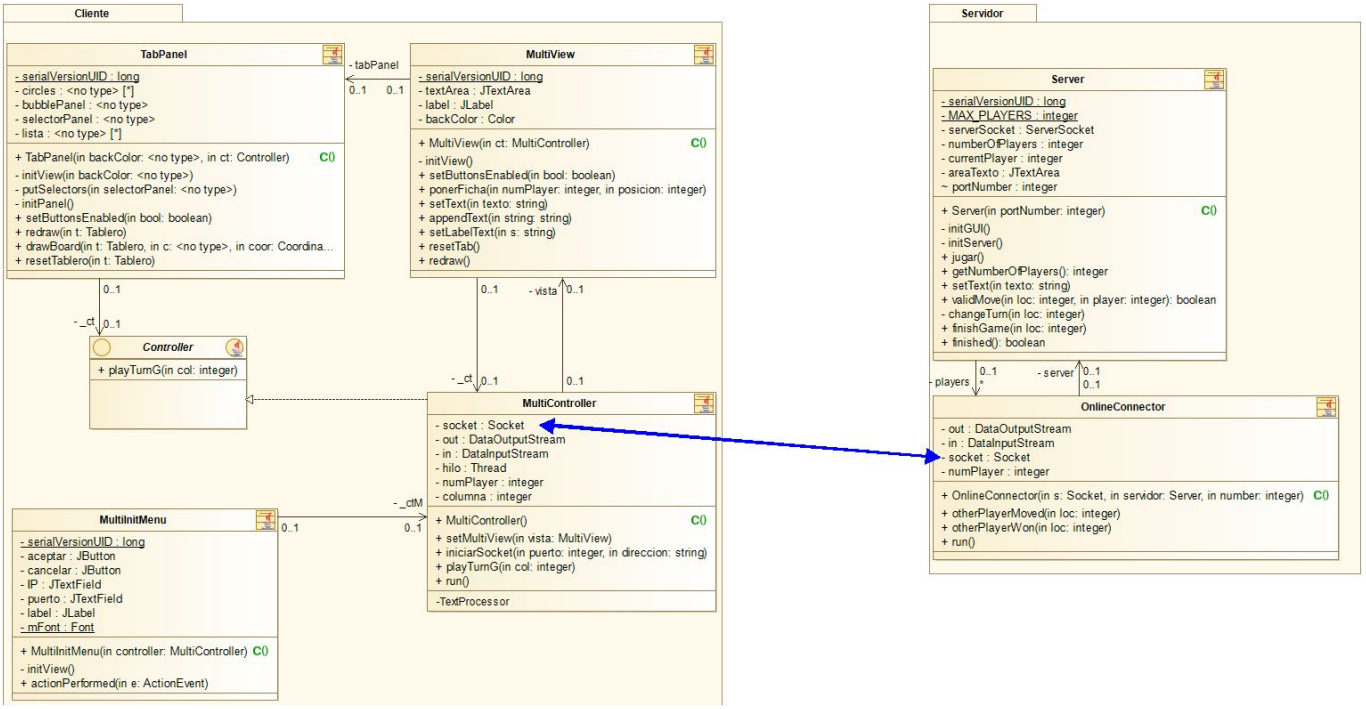


HISTORIA 14.2 - Servidor

El diagrama representa el funcionamiento de la parte del servidor a la hora de jugar el modo multijugador. Podría dividirse en 2 partes que se comunican (Server y Player) y una inicial que es el ServerMain, esta última simplemente se encarga de crear el servidor e iniciar el proceso de juego (jugar). En cuanto al servidor su función principal es aceptar solicitudes de conexión creando nuevos Players y comenzar sus procesos de juego en un nuevo hilo ya que heredan de Thread, sin embargo, también tiene métodos de inserción y consulta de distintos datos sobre el estado de la partida.

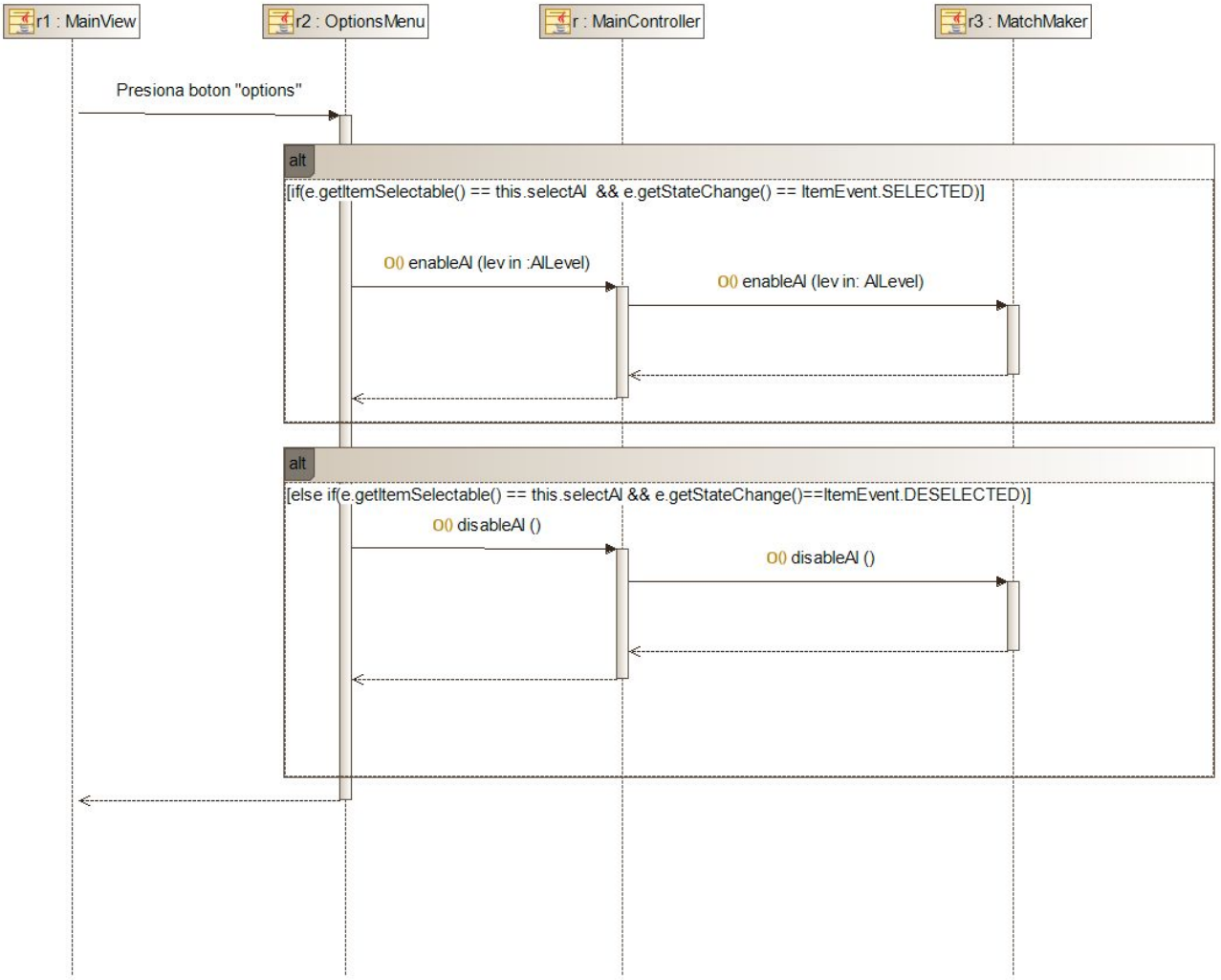
El más importante de ellos es el que sale en el diagrama como validMove ya que es el que inserta datos en el tablero de juego, por ello se realiza en bucle.
Como aclaración extra he de decir que toda la comunicación por red se realiza entre cada uno de los Player y MultiController de las personas que estén conectadas al servidor a través de lo que se conoce como Socket, esta comunicación se realiza únicamente mediante Cadenas de texto y enteros.





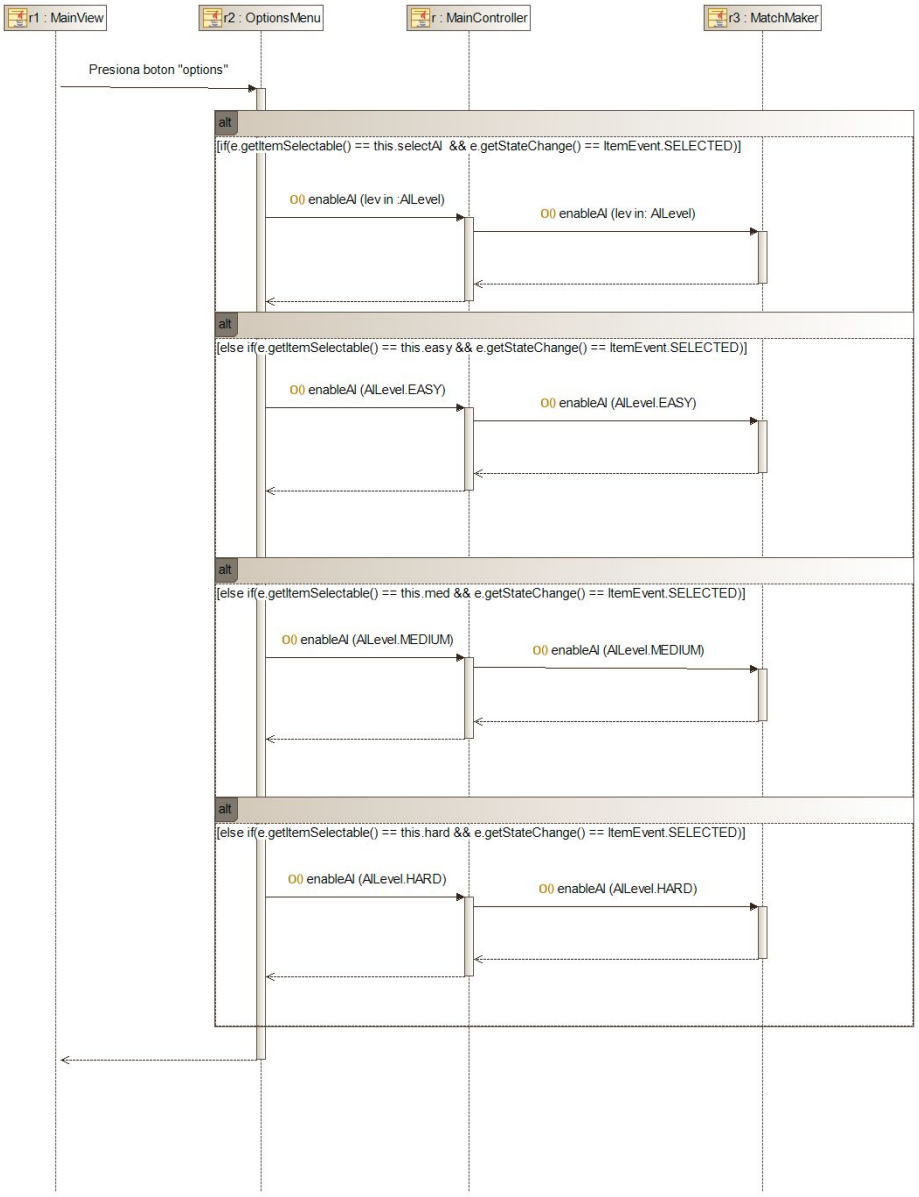
HISTORIA 15-IA

Al pulsar el botón options de la vista(clase Mainview) se accede a la clase optionsMenu. En ella está la opción de IA. Al seleccionar la casilla se activa el modo de jugar contra la IA.
Si la casilla está seleccionada se llama al método enableAI (clase MainController), el parámetro tiene el nivel deseado. Este método llama a su vez al método enableAI de la clase MatchMaker y se le pasa por parámetro el nivel de IA deseado.
Al seleccionar otra vez la casilla se comprueba el estado.
Si está deseleccionada se ejecuta el método disableAI en el MainController, que a su vez llama al método disableIA del Matchmaker que desactiva el modo de jugar contra la IA.



HISTORIA 16- Tres Niveles de Dificultad

Usamos patrón Strategy (Estrategia). Cada nivel de dificultad es una subclase con una funcionalidad distinta.
Al pulsar el botón options de la vista(clase Mainview) se accede a la clase optionsMenu. En ella está la opción de IA. Al pulsar esta casilla se despliegan varias opciones EASY; MEDIUM; HIGH (si no se selecciona ninguna opción se pone EASY por defecto).
En el método itemStateChanged de la propia clase hay una serie de ifs que comprueban una serie de cosas.
Si la casilla se selecciona, según si está seleccionado o no, se activará o no la AI y una serie de cosas visuales para saberlo.
Si se selecciona EASY, MEDIUM o HIGH se activa la AI. La opción se pasa por parámetro a la clase MainController a través del método enableAI. Posteriormente el MainController llama al método enableAI de la clase MatchMaker con el nivel como parámetro.
Según el nivel del parámetro el método playVsAi del matchmaker usará un algoritmo u otro ya que le pasamos la clase del nivel por parámetro con el método usando el algoritmo necesario.

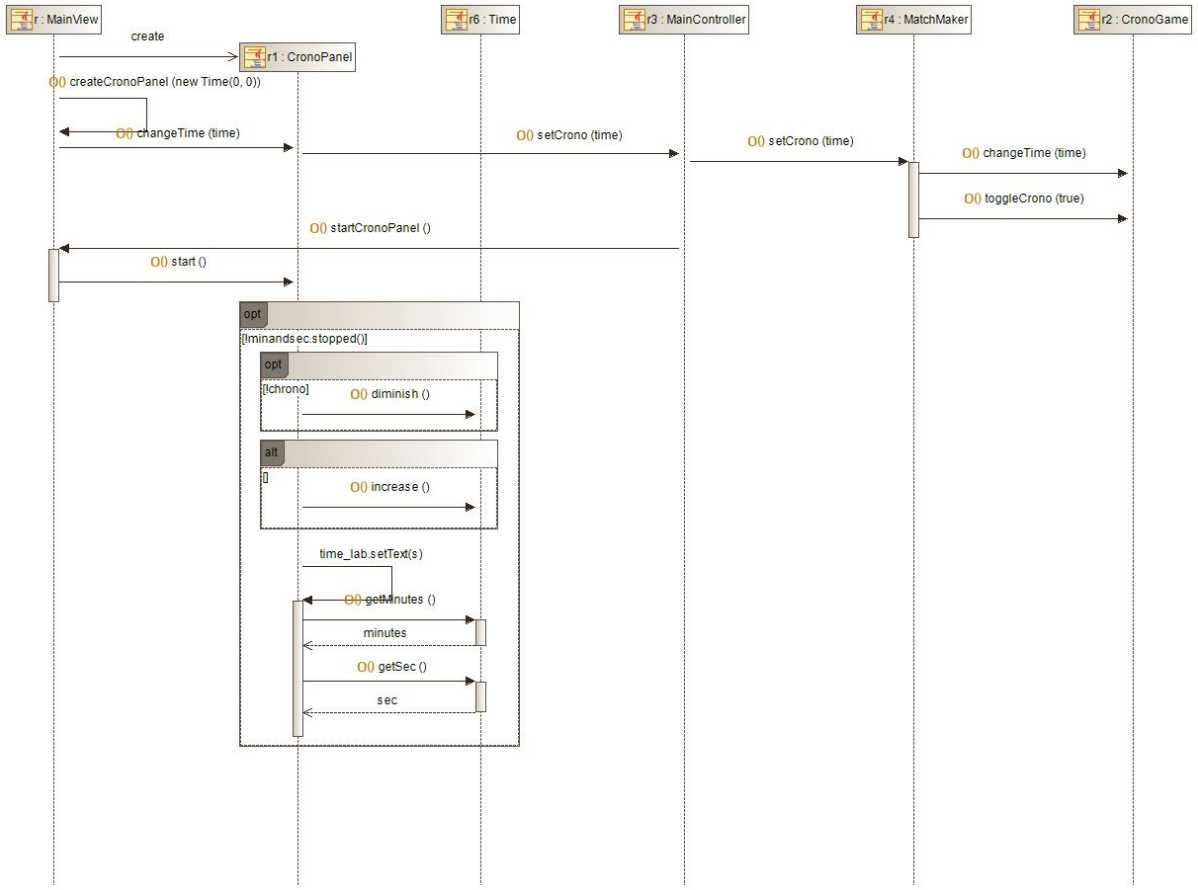


HISTORIA 17 - Lógica del ScoreBoard

Cada vez que el MatchMaker ejecuta playTurnG (método que maneja las partidas y a qué jugador le toca), carga de un archivo todos los ganadores que están guardados en un fichero y los guarda en una lista de Jugadores en el Scoreboard. Cuando un jugador gana, llama a addPlayer para agregarlo a esa lista. Luego crea otro fichero con los nuevos datos. Cuando finalizan las rondas se muestra una tabla que será el ScoreBoard, este recibe la lista de los jugadores y los muestra en pantalla.

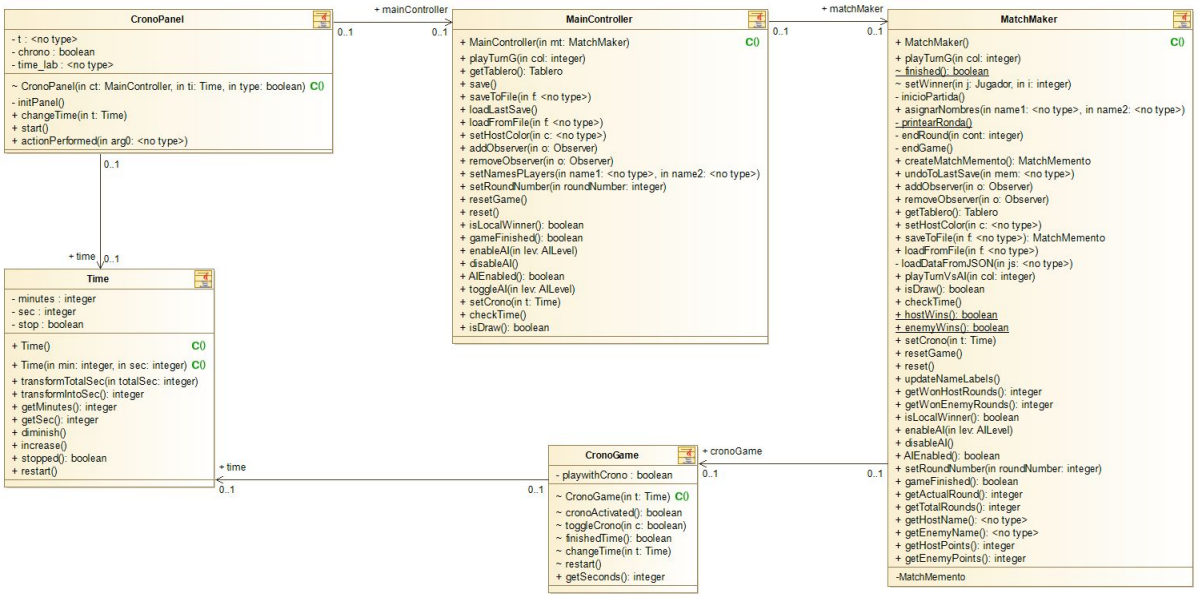
HISTORIA 18 - Cronómetro

Cuando iniciamos una partida que no es del tipo contrarreloj nos aparecerá un cronómetro ascendente que nos indique la cantidad de tiempo que estemos invirtiendo en acabar la partida, ese tiempo será utilizado para la Scoreboard. El proceso es sencillo ya que se utiliza un panel con el cronómetro (Time) en modo ascendente por lo que 'chrono' será true y se irá aumentando el valor del reloj.



HISTORIA 19 - Partida Limitada

El diagrama de esta funcionalidad es idéntico al de la Historia 18 pero empezando desde un tiempo mayor que 0.



CLASES E INTERFACES QUE COMPONEN EL CÓDIGO DE CONNECT4

1. Paquete Launcher

1.1. Main

Clase que posee el punto de entrada de la aplicación. Se encarga de crear el MatchMaker (clase principal del modelo), el MainController (controlador) y el hilo para la vista del juego.

1.2. ServerMain

Clase que inicia la ejecución del servidor creando un objeto de la clase Server con el parámetro 55555, en caso de querer cambiar el puerto de conexión al servidor este es el número que se debe cambiar. Llama a la función jugar() de Server para iniciar su funcionamiento.

2. Modelo

2.1. MatchMaker

Clase que concentra la lógica de la partida y la creación de las clases del modelo al iniciar la ejecución del juego. Crea el tablero, los jugadores iniciales del juego (siendo los dos jugadores normales), el cronómetro, la lista de observadores y el Scoreboard).

Tiene los siguientes métodos:

2.1.1. playTurnG(int col)

Se encarga de la lógica de los turnos y de avisar a los observadores los nuevos cambios. Al iniciar el juego crea un objeto de tipo infoObj y se lo pasa los observadores para la vista.

Luego existe un if para revisar si la partida ha terminado o si el tablero está lleno; si no es así, entra en la lógica para determinar cuál jugador va a realizar su turno. Se realiza con un entero. Si el entero es par, es el turno del jugador “host”. Este realiza su turno llamando a playTurn a través del PlayerInterface. Coloca la ficha en el tablero y devuelve la fila donde fue colocado. Al saber la columna y la fila, es más fácil indicarle a la vista dónde tiene que pintar una nueva ficha insertada, junto al color escogido (o predefinido) como color de esa ficha.

Después, chequea en el tablero si existe algún jugador que haya hecho alguna línea de 4 filas. Si es así, guarda en el objeto Jugador win este valor.

A continuación, actualiza a la vista de los cambios en el modelo con los observadores pasándole información con infoObj y la coordenada de la ficha insertada.

Si fuese el turno del jugador “enemy” el procedimiento es parecido más se agregaron algunas líneas de código para que este enemy pueda ser un jugador automático. Los cambios consisten en que se le avisa a los observadores y la vista que es el turno del jugador automático (si es el caso) y que se debe esperar a que este juegue.

Finalmente, playTurnG revisa si existe algún ganador en este turno. Si es así, guarda al jugador con el método “setWinner()” y actualiza todos los atributos necesarios para iniciar una nueva partida y le concede al ganador los puntos por la partida ganada. Igualmente en este caso, se vuelve a llamar a la vista para informarle que hay un ganador y que debe mostrar en pantalla el mensaje adecuado junto con la información enviada a través de infoObj. Luego se le avisa al Scoreboard que debe guardar todos los datos de los ganadores que hay en el Scoreboard junto con el nuevo ganador.

En el caso que el tablero se haya llenado, se actualizan los atributos necesarios y se aumenta en el contador que decide a qué jugador le toca el siguiente turno “nextTurn”.

Al final del método se vuelven a cargar los datos de todos los ganadores hasta la fecha y se crea una lista de jugadores que se le envía a la vista del Scoreboard para enseñar los ganadores.

Hay que añadir que en el caso de que esté activado el jugador automático y sea el turno del mismo; antes de terminar el método, playTurn se llama así mismo para que ejecute el siguiente turno que es el del CPU.

2.2. **Ficha**

Clase que representa la ficha que se pone en el tablero, sus atributos son el color y las coordenadas en el tablero.

2.3. **Tablero**

Clase que representa el tablero de juego mediante una matriz de fichas, sus funciones son colocar fichas, comprobar que se ha acabado la partida, que el tablero esté lleno o que una columna esté llena.

2.4. **Jugador**

Clase que representa a los jugadores de la partida, contiene el número de puntos que lleva el jugador, el color de sus fichas, etc.

2.5. **CronoGame**

Clase que alberga la lógica necesaria para administrar el funcionamiento de los cronómetros.

2.6. **ScoreBoard**

Es una clase que tiene una lista de jugadores y usa el patrón memento, su función principal es cargar y guardar los datos de los jugadores una vez finalizada la partida, para poder crear un un marcador con esos datos (los datos se guardan en un fichero json).

Tiene una lista de jugadores para guardar los ganadores de las partidas y un mapa de strings (que sería el Id del jugador) que guarda el jugador. Esto sirve para que se actualicen los jugadores dependiendo de su ID y no se repitan en el Scoreboard el mismo Jugador. Igualmente la lista de jugadores está ordenada usando la clase SortedArrayList y DescendentOrder.

2.7. **AI**

2.7.1. ComputerInterface

Esta interfaz se encarga de simular el comportamiento que debe tener cada nivel de la IA que lo implemente. Para ello incluye un método para poder jugar el turno; un método para acceder a la columna escogida por el comportamiento y un getter del nombre que se le asigna a la máquina.

2.7.2. WeakAI

Implementa el juego de turno escogiendo una columna libre aleatoria. Le asigna un nombre y guarda en una variable la última columna escogida.

2.7.3. MediumAI

Mezcla la estrategia Weak y la Insane para escoger una columna libre. Usa un seleccionador Random para elegir la estrategia aleatoria o la *minimax*.

2.7.4. InsaneAI

Implementa el juego de turno escogiendo una columna libre usando el algoritmo minimax con una profundidad en el árbol limitada, y se usa poda alpha-beta para mejorar la eficiencia. Le asigna un nombre y guarda en una variable la última columna escogida.

2.8. InfoObj

Es una clase dedicada a pasar información entre el Matchmaker y sus observadores, de esta manera quitamos funcionalidad innecesaria al Matchmaker y al controlador y pasamos toda la información necesaria en una sola clase.

2.9. PlayerInterface

Interfaz que contiene los métodos para Jugador, estos son jugar turno, modificar datos del jugador, elegir comportamiento…

2.10. HJugador

Es una clase que implementa la interfaz PlayerInterface y sobrescribe la función de jugar turno, modificar el color modificar el id y sus correspondientes getters.

2.11. AIJugador

Es una clase que implementa la interfaz PlayerInterface y sobrescribe la funcion de jugar turno, getId, y modicar el comportamiento de la IA.

3. Paquete controller

3.1. Controller

Interfaz que contiene el método playTurnG, esta interfaz es implementada tanto por MainController como por MultiController. Su principal función es asegurarse de que todo controlador que se cree en el programa sea válido para su uso.

3.2. **MainController**

Controlador utilizado para la ejecución del juego local. Se encarga de comunicar el Mainview con el MatchMaker. Su función es mandar las órdenes del MainView al MatchMaker.

3.3. **MultiController**

Controlador usado para la correcta ejecución del multijugador, se encarga de comunicar la vista del multijugador con el modelo contenido en el servidor y viceversa a través de la comunicación que crea (socket).

Cabe recalcar que contiene una clase interna encargada de procesar la información recibida por el socket para traducirla a acciones que deben ser comunicadas a la vista.

4. View

4.1. **MainView**

Es la vista principal del juego, extiende a JFrame e implementa Observer (para obtener los datos). Contiene todos los paneles del juego y según la información obtenida por las funciones de observer, muestra unos paneles u otros y actualiza la información.

4.2. **InitMenu**

Es una clase que hereda a JPanel e implementa ActionListener.

Representa el menú inicial del juego en el que muestra las opciones de “jugar”, “jugar en multijugador”, “opciones” y “salir del juego”.

4.3. **TabPanel**

Clase derivada de una refactorización que hereda de JPanel, fue creada para evitar repetir código muy parecido tanto en la vista del

multijugador como en la normal y dejar estas clases mucho más limpias. Contiene los elementos que componen el tablero de juego que vemos, es decir, los botones y el tablero con las fichas que se hayan colocado. Algo a destacar es que funciona independientemente de la clase de juego al que estemos jugando gracias al atributo Controller que tiene, esto le permite llamar a playTurnG tanto en una modalidad como en otra al pulsar un botón.

4.4. **OptionsMenu**

Clase derivada de una refactorización que hereda de JPanel e implementa la interfaz ActionListener (para los botones). Su función es realizar los cambios deseados en el juego, tienes la opciones de cambiar los nombres, el número de rondas y los colores de las fichas.

4.5. **FilledCircle**

Se encarga de crear y dibujar los círculos del tablero sobre los que se ha insertado alguna ficha, el color del que será dicho círculo le llega como parámetro en el constructor.

4.6. **EmptyCircle**

Se encarga de crear y dibujar los círculos del tablero sobre los que no se ha insertado ninguna ficha.

4.7. **BombButton**

Se encarga de crear y dibujar los botones del tablero con la imagen de la bomba. Poseen un efecto visual a la hora de pasar el ratón por encima que permite que una persona que nunca ha jugado deduzca de forma sencilla que es sobre estos botones sobre los que tiene que pulsar para colocar las fichas en las distintas columnas.

4.8. **WinnerScreen**

Clase derivada de una refactorización que hereda de JPanel e implementa la interfaz ActionListener (para los botones).

Su función es mostrar una pantalla con el ganador y sus puntos entre ronda y ronda.

4.9. **FinalScreen**

Clase derivada de una refactorización que hereda de JPanel e implementa la interfaces ActionListener y Observer (para obtener la información del marcador). Su función es mostrar (cuando acaba) el marcador de la partida actual y de las partidas anteriores.

4.10. **CronoPanel**

Parte de la vista de juego que muestra el cronómetro, ya sea ascendente o descendente.

4.11. **SelectCronoPanel**

Hereda de JPanel, implementa la sección de la vista de opciones en la que se selecciona el modo de juego de la cuenta atrás y el tiempo que dura.

4.12. **MenuBar**

Clase que hereda de JMenuBar e implementa ActionListener, ofrece un menú con las opciones de guardar/cargar partida y de terminar la partida.

4.13. **MultiInitMenu**

Vista que precede a meterse en una partida multijugador en red, su función es obtener los datos necesarios para iniciar la conexión con el servidor (puerto y dirección IP). Tiene 2 botones, uno de cancelar que nos devolverá al menú principal y otro de aceptar, al pulsar este se recopilan los datos recibidos del humano y se intenta iniciar la conexión con ellos, en caso de error al iniciarla se informará de ello en un JLabel para revisar los datos introducidos, en caso de éxito se pasará a la MultiView.

4.14. **MultiView**

Clase principal de la vista multijugador, contiene un objeto TabPanel sobre el que se jugará, un área de texto en la que se escribirá información sobre el estado de la partida, un JLabel sobre el que se

pondrá el número de jugador que es dentro de la partida y por último el controlador con el que se comunica para que este le envíe información al servidor.

4.15. **ScoreBoardTable**

Es una clase que extiende a AbstractTableModel e implementa la interfaz Observer (para obtener la información de la tabla).

Su función es crear una tabla con la información (id, puntos, tiempo) de los jugadores que han jugado las partidas anteriores.

4.16. **TextPrompt**

Esta clase es prestada por Rob Camick de: https://tips4java.wordpress.com/2009/11/29/text-prompt/

5. Paquete Observer

5.1. **Observer**

Una clase puede implementar la interfaz Observer cuando quiere estar informada de los cambios en los objetos observables.

5.2. **Observable**

Esta clase representa un objeto observable, o "datos" en el paradigma de vista de modelo. Se puede subclasificar para representar un objeto que la aplicación quiere haber observado.

Un objeto observable puede tener uno o más observadores. Un observador puede ser cualquier objeto que implemente la interfaz Observer. Después de que una instancia observable cambia, una aplicación que llama al método notifyObservers del Observable hace que todos sus observadores sean notificados del cambio mediante una llamada a su método de actualización.

6. Paquete Multiplayer

6.1. **Server**

Clase principal del lado del servidor, extiende JFrame ya que posee una GUI (muy básica) en la que se muestra información sobre lo

que ocurre en el servidor, el puerto en el que se está ejecutando y la partida que se esté llevando a cabo. Se encarga de abrir el ServerSocket pero hasta que no se llama a jugar() no acepta conexiones. El método jugar() consiste en aceptar conexiones de red hasta que se cierre el programa, dichas conexiones se acumulan en forma de objetos de la clase OnlineConnector en un array de tamaño MAX_PLAYERS. Esta clase también contiene el tablero sobre el que se está jugando la partida online y que es reiniciado cada vez que se acaba una partida y además métodos públicos que utilizarán los OnlineConnector para consultar el estado del tablero e informar de acciones que se deben realizar sobre el mismo y la partida.

6.2. **OnlineConnector**

Hereda de Thread ya que cada objeto de esta clase será el encargado de administrar la conexión con los jugadores de forma independiente a los demás para que no se cuelgue el programa. Su función es enviar y recibir datos por el socket hasta que el servidor le diga que hay un ganador y acabe su ejecución.

7. Paquete Utils

Conjunto de clases con funcionalidades variadas y reducidas que permiten simplificar el resto de clases del programa, podrían ser reutilizadas en otros proyectos.

- 7.1. **AllLevel**
- 7.2. **Coordinate**
- 7.3. **Parser**
- 7.4. **DescendentOrder**
- 7.5. **SortedArrayList**
- 7.6. **Time**

8. Paquete testing (JUNIT)

Tests que prueban las distintas clases del programa.

- 8.1. **JugadorTest**

8.2. **FichaTest**
8.3. **InfoObjTest**
8.4. **TableroTest**
8.5. **MatchMakerTest**
8.6. **MultiplayerTest**