

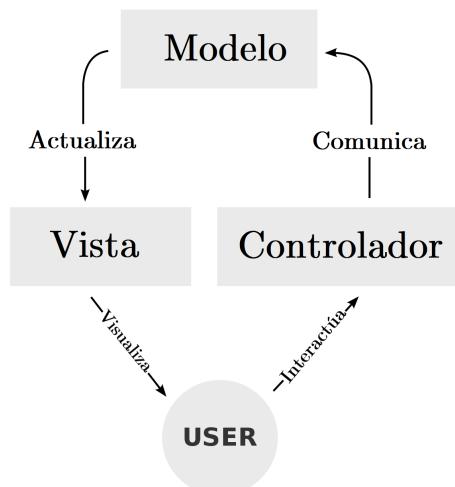
DISEÑO

Grupo PMC

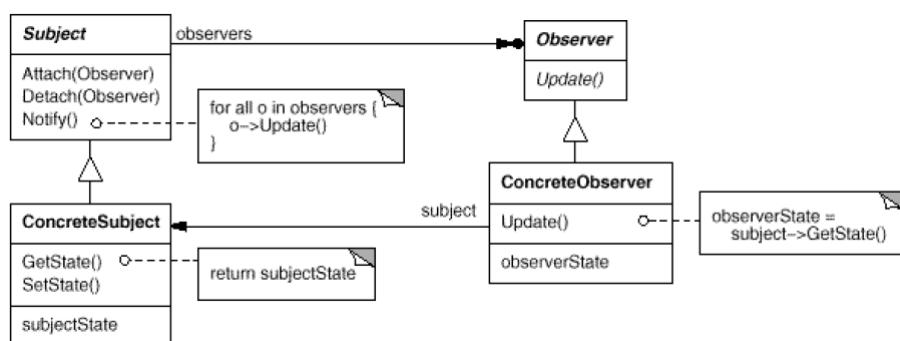
22 de mayo de 2022

1. Arquitectura

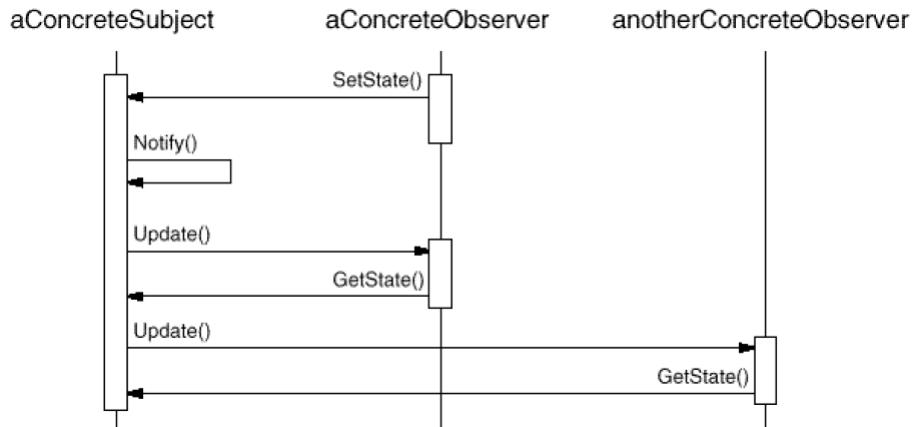
El patrón de diseño principal en el proyecto es el **Modelo-Vista-Controlador**; ideal para un juego de las características de Rolit.



El funcionamiento básico, que pormenorizaremos a lo largo del documento, es el siguiente: las vistas recogen las peticiones del usuario (por ejemplo, poner un cubo), el cual procede a notificar al controlador, quien a su vez notifica al modelo para que ejecute las correspondientes acciones. El modelo a su vez, una vez finaliza la ejecución de esas peticiones, actualiza la vista por medio del **patrón observador**.

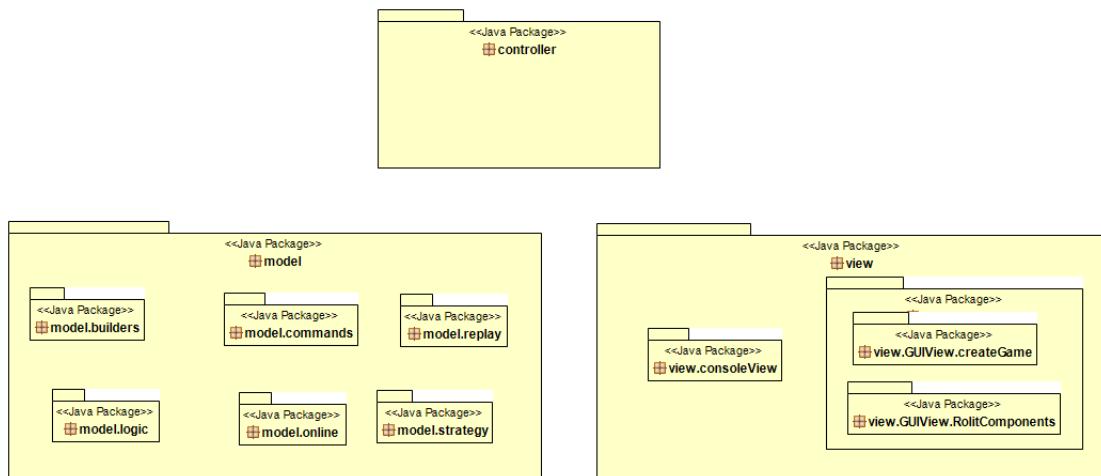


Diseño diagrama de clases UML



Diseño diagrama de secuencia UML

Para implementar el patrón MVC, hemos organizado las clases del proyecto en tres paquetes principales: modelo, vista y controlador. En cada uno de los paquetes se han incluido subpaquetes correspondientes a distintas funcionalidades.



Diseño diagrama de secuencia UML

A continuación se desarrollará el contenido de cada parte del proyecto, representado mediante paquetes.

Modelo-Vista-Controlador

Se ha dividido el código en 3 paquetes con el fin de seguir el patrón modelo-vista-controlador (MVC). De esa forma, separamos de forma encapsulada el modelo (encargado de la lógica del juego), el controlador (intermediario entre la vista y el modelo) y la vista (la parte de la aplicación que interactúa con el usuario).

Haga click [aquí](#) para ver la distribución de estos paquetes.

Con respecto al modelo, se usa el paquete model; con respecto al controlador, el modo consola y el controlador del cliente (online) usa el paquete Controller y el

modo GUI aprovecha de la estructura controlador-vista proporcionada por Swing en los actionPerformed; por último, tenemos la vista en el paquete view, que a su vez se divide en subpaquetes para ofrecer distintas vistas según si se juega en consola o en GUI.

De esta forma se implementa el patrón MVC de una forma eficiente, organizada y respetando la encapsulación en todo momento.

Modelo

El modelo contiene la lógica y los datos del juego; es el núcleo de Rolit.

-  **model**
-  **model.builders**
-  **model.commands**
-  **model.logic**
-  **model.online**
-  **model.replay**
-  **model.strategy**

Listado de subpaquetes del modelo

- El paquete **model.builders** paquete agrupa las clases cuya responsabilidad es saber construir un objeto del tipo Game a través de un conglomerado de información.
- El paquete **model.commands** recoge los distintos comandos específicos por medio de los cuales el usuario puede interactuar con el modelo, según los requerimientos del **patrón Command**.
- El paquete **model.logic** agrupa la lógica principal encargada del funcionamiento de los elementos que componen el juego.
- El paquete **model.online** recoge los aspectos del juego en red relacionados con el modelo.
- El paquete **model.replay** contiene a las principales clases involucradas en el funcionamiento de las repeticiones de partidas.
- En el paquete **model.strategy** se recogen las distintas estrategias y clases auxiliares para la implementación de los usuarios IA, según los requerimientos del **patrón Strategy**.

La visión completa del modelo es la siguiente:



Modelo completo

Vista

La vista se encarga de elaborar adecuadamente cómo se visualiza el juego. En nuestro juego existen dos vistas: la de consola y la de GUI.

Los paquetes que hemos empleado para ello son los siguientes:

```
>  view
>  view.consoleView
>  view.GUIView
>  view.GUIView.createGame
>  view.GUIView.RolitComponents
```

Listado de subpaquetes de la vista

- En el paquete **view.consoleView** se incluyen las clases relacionadas con la vista en el modo consola.
- El paquete **view.GUIView** contiene las clases relacionadas con la vista en el modo GUI.
- El paquete **view.GUIView.createGame**, subpaquete de **view.GUIView**, contiene las clases relacionadas con los diálogos de creación de partidas.
- El paquete **view.GUIView.RolitComponents** está formado por un conjunto de clase que extienden a los que Java proporciona por defecto. Estos componentes propios aportan uniformidad visual al juego, evita la repetición de código para configurar los componentes y facilita la adaptabilidad a futuros cambios.

La visión completa de la vista es la siguiente:



Vista completa

Controlador

El controlador supone el intermediario entre la vista y el modelo, notificando a ambos si es preciso que se actualicen en base a las interacciones realizadas para el juego o desde el juego.

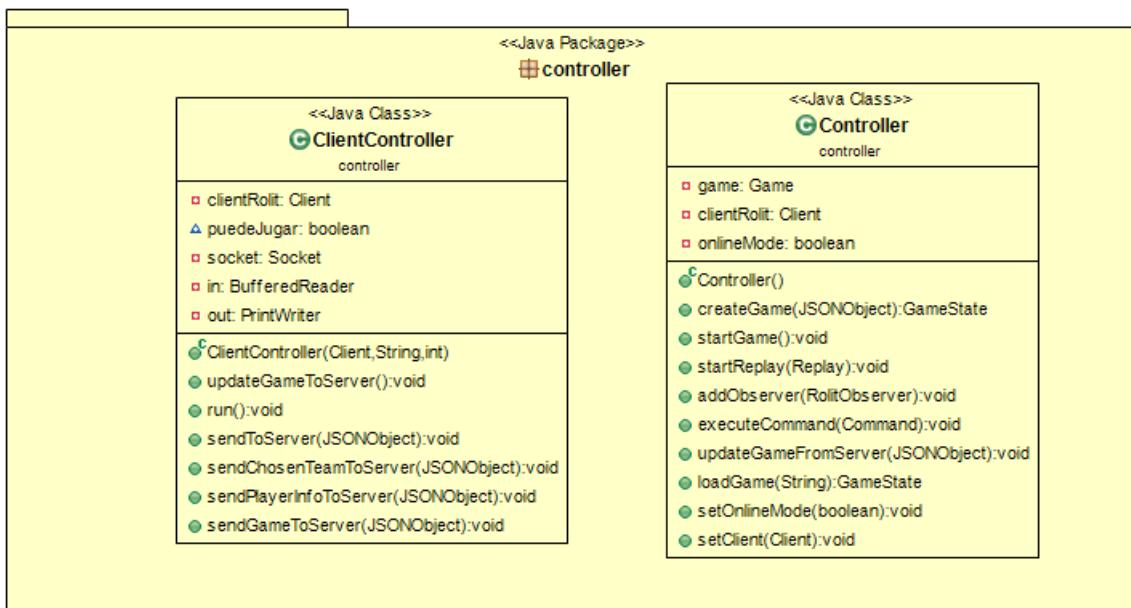


Paquete controller

En el paquete controller se pueden distinguir dos tipos de controladores:

- **Controller**: clase encargada de controlar la lógica del juego por medio de la creación y el arranque del mismo y de crear los comandos. Es el nexo entre el modelo y la vista GUI en pro del modelo-vista-controlador.
- **ClientController**: clase desde la perspectiva del cliente, única por cliente. Es el controlador envía y recoge información de y al servidor.

La visión completa del controlador es la siguiente:

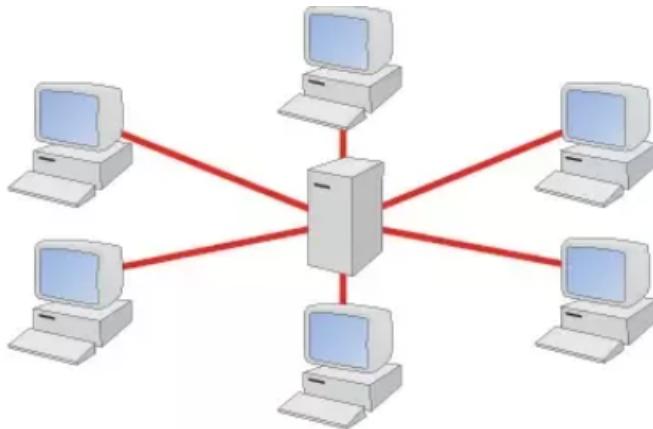


Controlador completo

Cliente-Servidor

Para implementar la funcionalidad de red en Rolit, hemos empleado el patrón cliente-servidor, adaptándolo a nuestras necesidades.

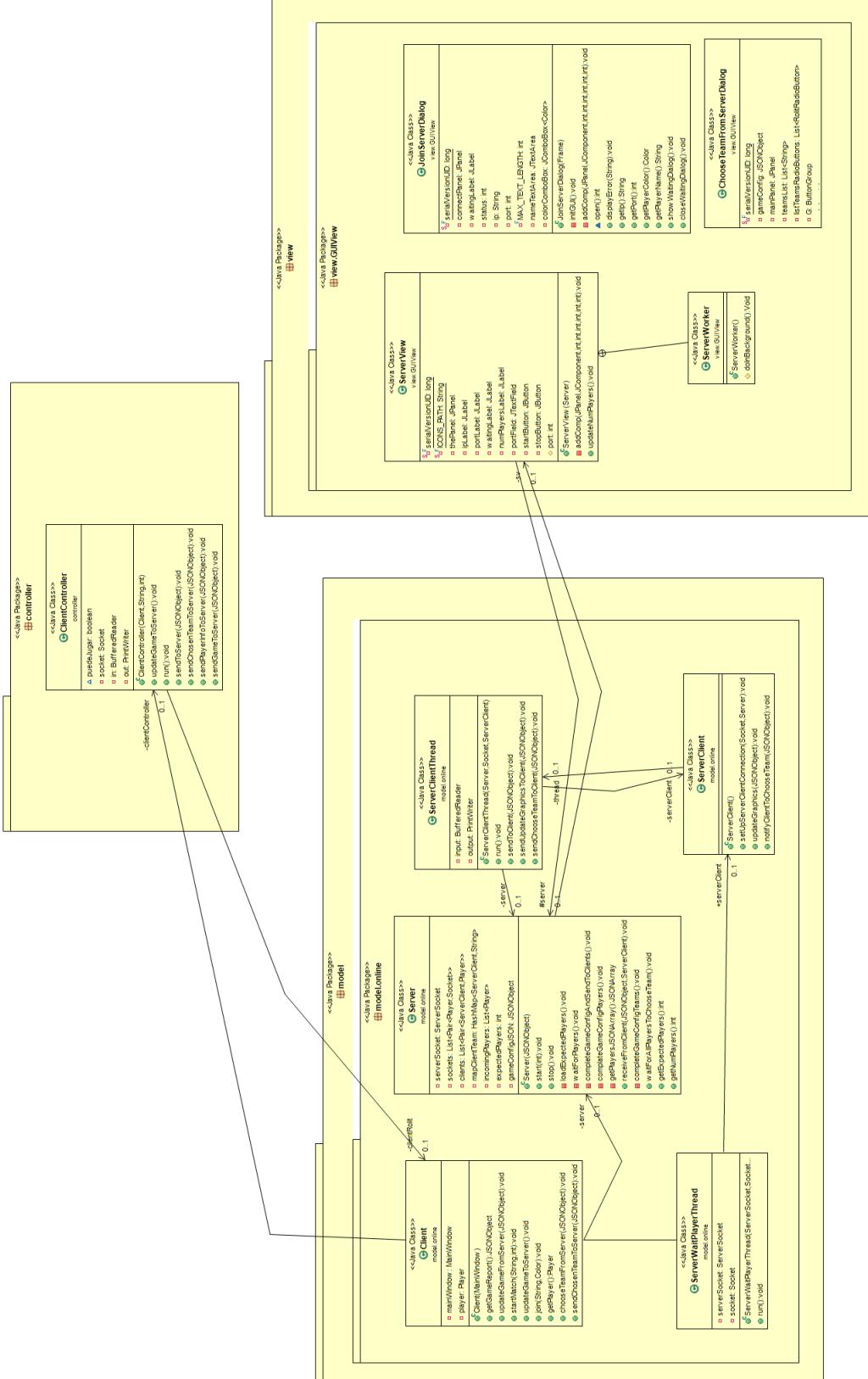
En particular, nuestra conexión en red se basa en una estructura en el que **los clientes poseen el modelo**, realizan cambios en el mismo y lo notifican al servidor. El servidor procede a enviar al resto de clientes la información nueva según la cual deben actualizar sus modelos. Nótese que el cliente sólo tiene contacto con el servidor, y el servidor tiene contacto con todos los clientes. Es, por tanto, un **modelo de red centralizado**, con nodo central el servidor.



Para poder implementar dicho patrón, tuvimos que completar el modelo, vista y controlador, introduciendo las siguientes clases:

- **Modelo:** paquete model.online. En particular, Client, Server, ServerClient, ServerClientThread, ServerWaitPlayerThread.
- **Vista:** ChooseTeamFromServerDialog, JoinServerDialog, ServerView
- **Controller:** ClientController.

Por tanto, la visión completa del juego online es la siguiente:

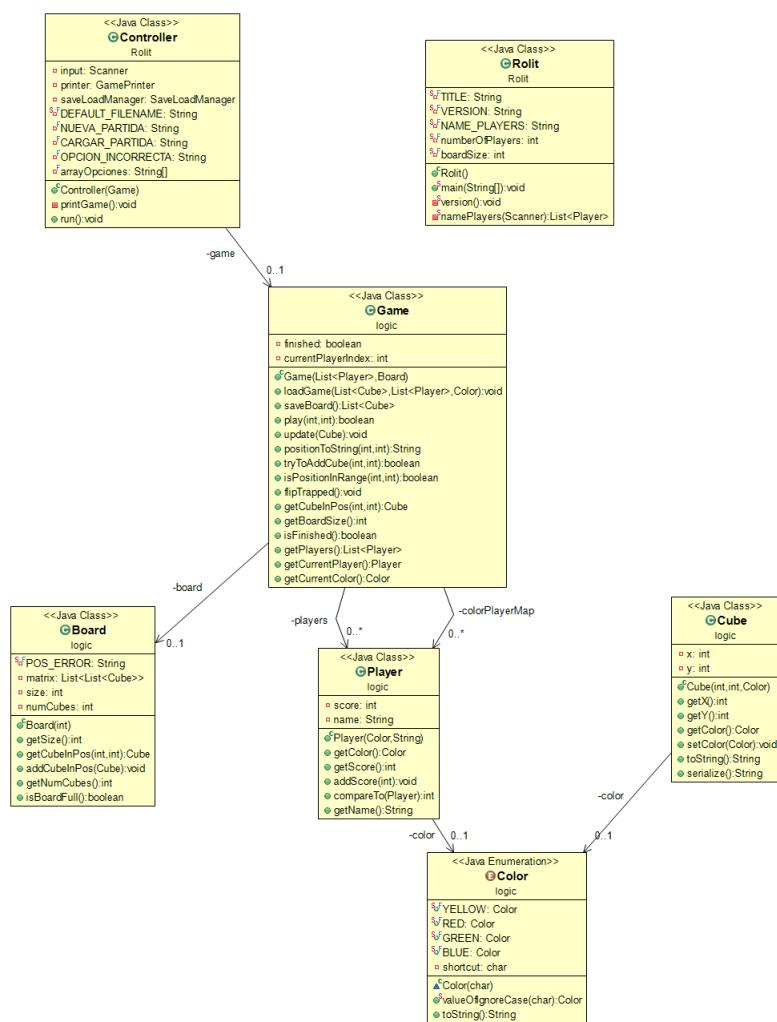


2. Diseño y evolución de las Historias de Usuario

Como usuario quiero poder jugar a Rolit siguiendo un conjunto mínimo de normas

Sprint (1)

Diagrama de clases:

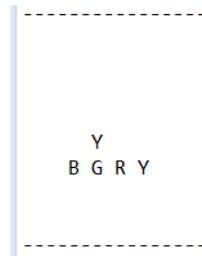


Esta es la versión más básica de Rolit.

El tablero es un cuadrado 8x8. Esto se hace por medio de una constante llamada `boardSize`, dentro de la clase `Rolit` y cuyo valor era trivialmente 8.

Hay una cantidad invariable de 4 jugadores cuyos colores son: amarillo, rojo, verde y azul. El enumerado `Color` facilita la gestión de dicha tarea, ya que cada cubo y cada jugador tienen asignado un `Color`, como bien muestra el diagrama de clases.

Aunque la intención final es que los cubos sean redondos, en esta versión de Rolit decidimos no implementarlo aún. Por ello el tablero se muestra gracias a `GamePrinter` como una matriz 8x8 en la que los cubos están representados por la letra inicial del color, tal y como se muestra en la imagen inferior.



Los jugadores están almacenados en una lista de `Player` y en un mapa de `Color`, `Player` en la clase `Game`. Esto es necesario para el funcionamiento del método `update(Cube)`, que es el encargado de llevar a cabo las siguientes tareas:

- Solo se puede colocar un cubo en una casilla vacía adyacente a una ya ocupada (ver imagen inferior), excepto el primer cubo que se pone donde se deseé.

```
-----  
-----  
R  
Y  
B G R Y  
-----  
  
Turno de dani (G)  
Introduce un comando:  
c : Poner un cubo  
s : Guardar partida  
c  
Introduce la posicion x:  
1  
Introduce la posicion y:  
1  
La posicion no es valida  
Turno de dani (G)  
Introduce un comando:
```

- Cuando se coloca un cubo, se cambian de color todos los que quedan atrapados en las direcciones válidas. Un cubo se dice atrapado cuando se encuentra en la línea que une un cubo recién puesto con otro del mismo color y las direcciones válidas son las líneas rectas dadas por alguna de las casillas adyacentes.

```

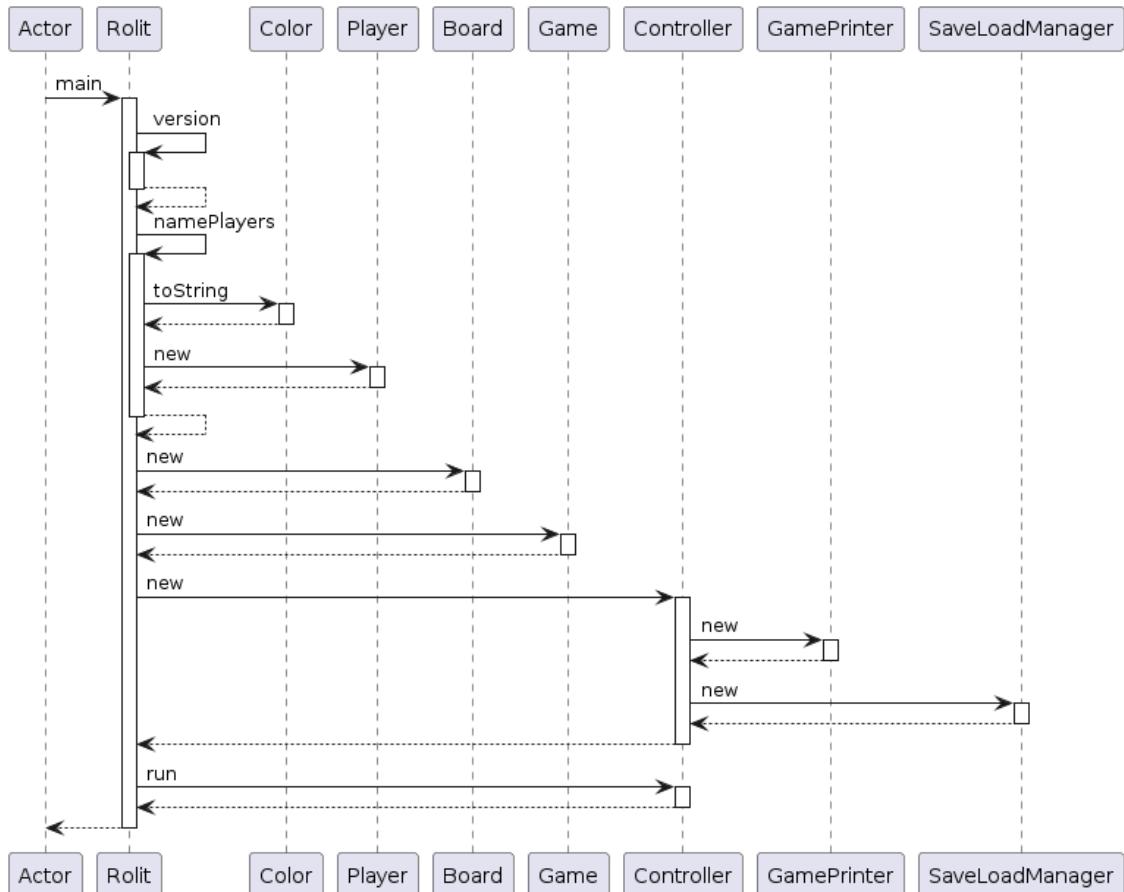
Turno de dani (G)
Introduce un comando:
c : Poner un cubo
s : Guardar partida
c
Introduce la posicion x:
2
Introduce la posicion y:
3
-----
```

```

G
G
G
B G R Y
-----
```

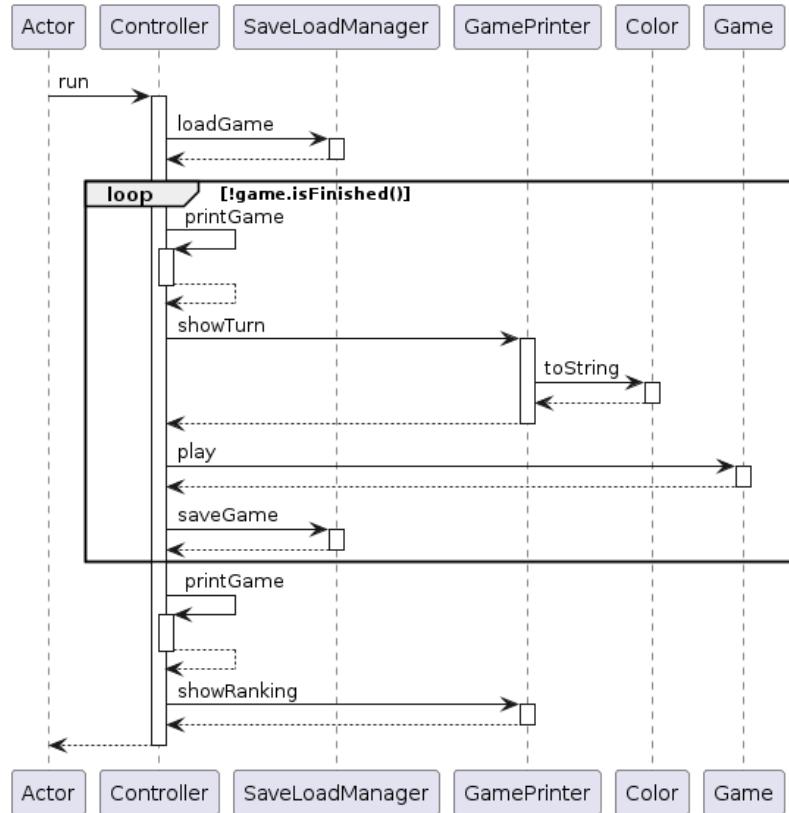
- En cada turno un jugador pone un único cubo y se pasa de turno al siguiente jugador después de haber cambiado de color a los posibles cubos atrapados.
- El juego acaba cuando el tablero se llena completamente.

Para poder entender el método `update(Cube)` de una forma más simple hay que contextualizarlo primero. El `main()` está dentro de la clase `Rolit` y es el encargado de instanciar `Controller`, `Game` y `Board`, así como de crear la lista de `Player`.

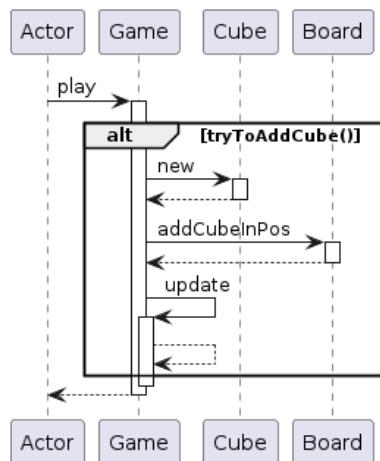


Por otro lado el método `run()` se encarga de gestionar la ejecución global del juego, el cual será ejecutado hasta que haya terminado, es decir, cuando el tablero se

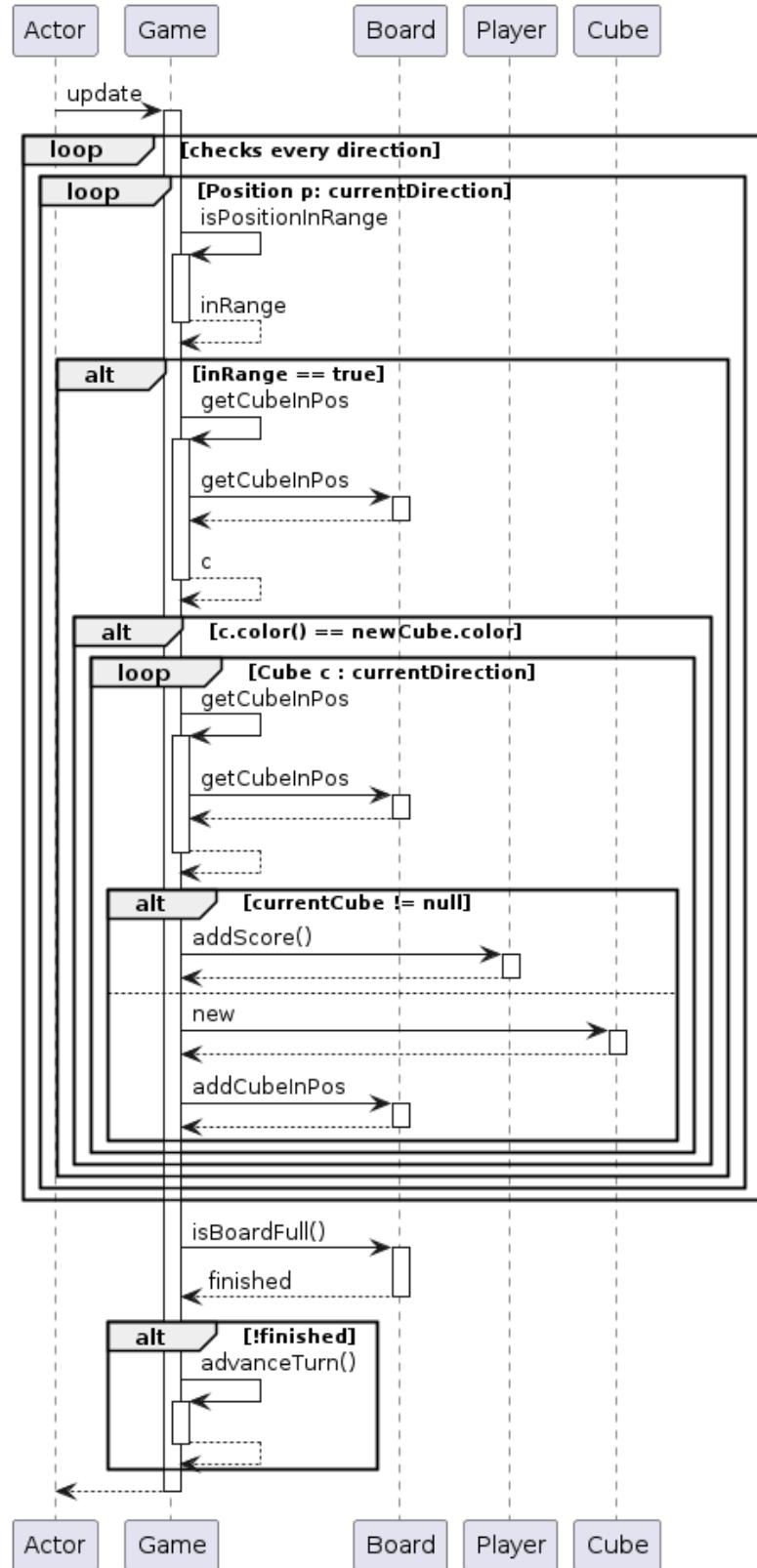
esté completamente lleno. Esto se gestiona por medio de un bucle cuya condición es `!game.isFinished()`, esta función actúa como un getter del atributo privado `finished` de la clase `Game`, como bien se puede apreciar en el diagrama de clases. Por último, se mostrará un ranking, por medio de `GamePrinter` a través del método `showRanking()`, que muestra una lista de los jugadores ordenados según su atributo `score`, es decir, ganará el jugador con más puntos.



El método `play()` comprueba en primer lugar que la posición en la que se quiere colocar el cubo es válida. En caso afirmativo se crea un `Cube` y se llama al método `update()`, al que queríamos llegar desde un comienzo y podríamos considerar núcleo de la lógica de Rolit.

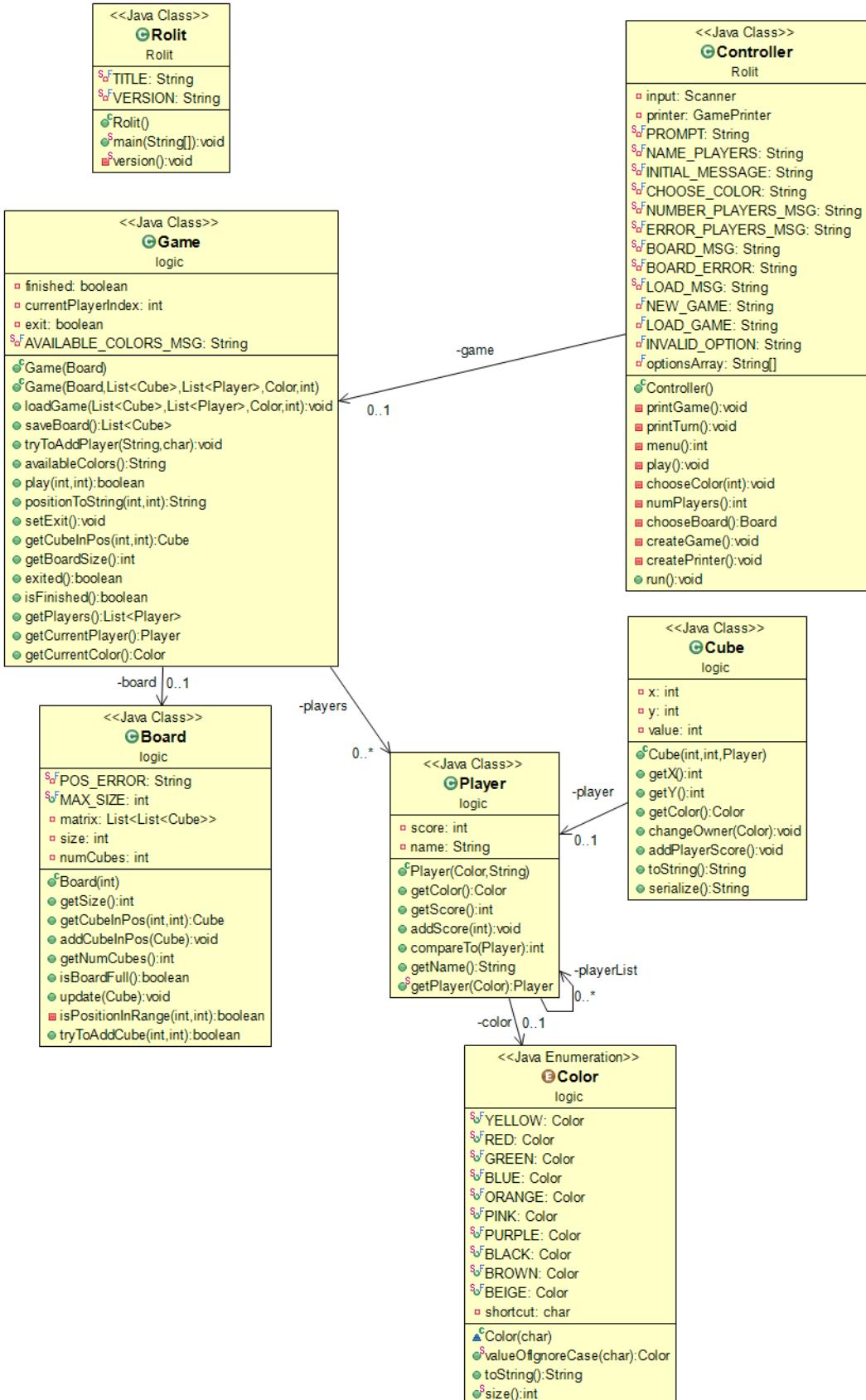


Como bien se puede observar en el diagrama de secuencias inferior, el método `update()` de Game es el encargado de cambiar de color todos los cubos que quedan atrapados en las direcciones válidas, así como de actualizar el atributo `finished` si la partida ha acabado, en otro caso avanza el turno para que cada jugador solo colocar un Cube en cada turno.



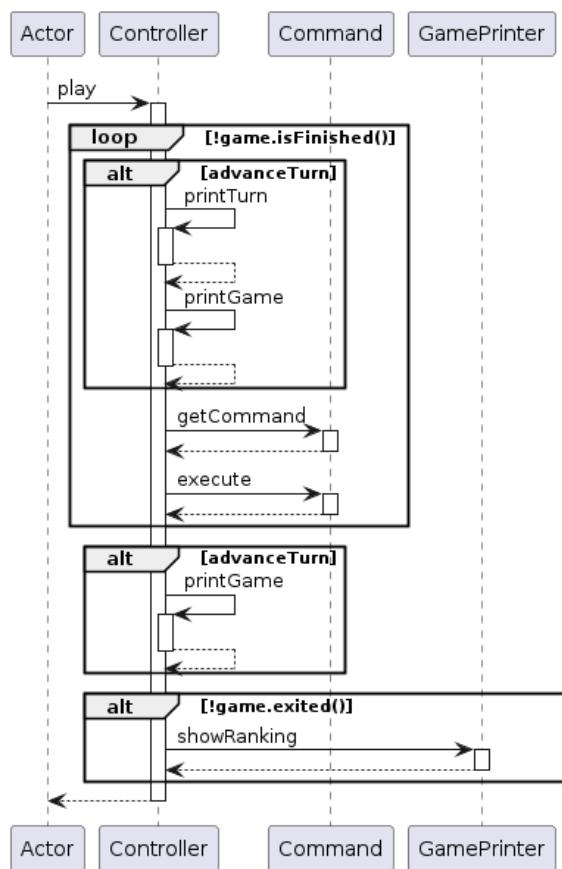
Sprint (2)

Diagrama de clases:



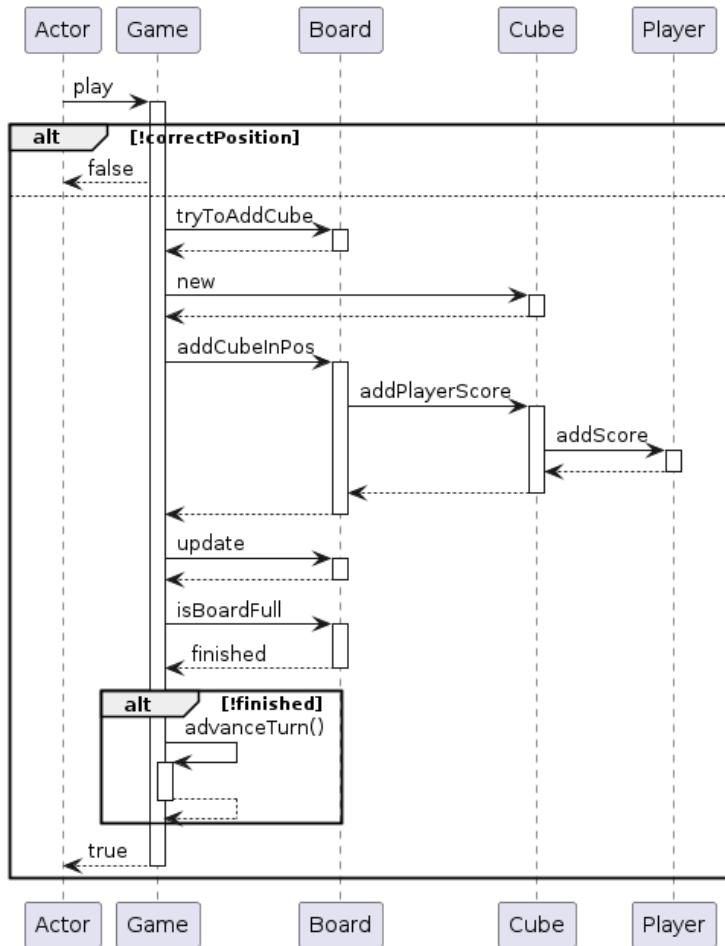
En esta segunda versión de Rolit esta historia de usuario a penas sufre modificaciones, no obstante, la forma de implementarlas es algo distinta. Para poder apreciarlo mejor conviene comparar el diagrama de clases de este Sprint con el del Sprint anterior. Entre los cambios podemos destacar:

- Game ya no posee el mapa Color, Player.
- Player tiene una lista de los Player.
- Cube ya no tiene un atributo Color, ahora tiene un atributo Player
- El tablero es un cuadrado de tamaño variable (entre 8 y 15), así como el número de jugadores también lo es (entre 2 y 10), por ello habrá 10 Color posibles. Para poder hacer un juego a gusto del cliente el Controller se encarga de comunicarse con él por medio de funciones como ChooseColor() o ChooseBoard().
- Los cubos siguen siendo representados por letras significativas para cada Color.
- el main() de Rolit tiene como única funcionalidad la creación del Controller y una llamada a su método run().
- Ahora el método run() es el encargado de crear el juego según desee el usuario y el bucle principal del juego estará en el método play(), un método nuevo que sería un intermediario entre run() de Controller y play() de Game, al que accedemos desde el execute() del comando PlaceCube.

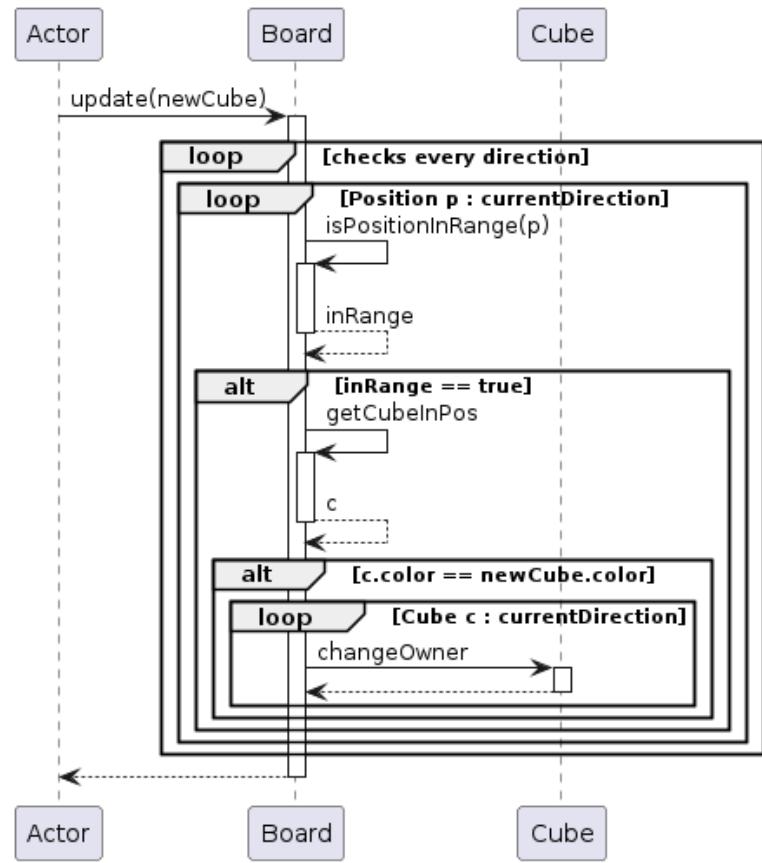


Como podemos apreciar en la imagen superior, ahora es `play()` el encargado de gestionar cuando termina el juego, así como de mostrar el ranking final del juego, que al igual que el anterior Sprint vendrá determinado por el `score` de los `Player`. Al añadir el comando `Exit` el bucle de `play()` de `Controller` podría salir del bucle y que el juego no haya terminado, por eso es necesario hacer la comprobación al saltar de turno.

- la función `play()` de `Game` es muy parecida a la del Sprint 1, no obstante, al pasar el método `update()` de la clase `Game` a la clase `Board` debemos actualizar el atributo `finished` y pasar de turno en el método `play()` de `Game`.

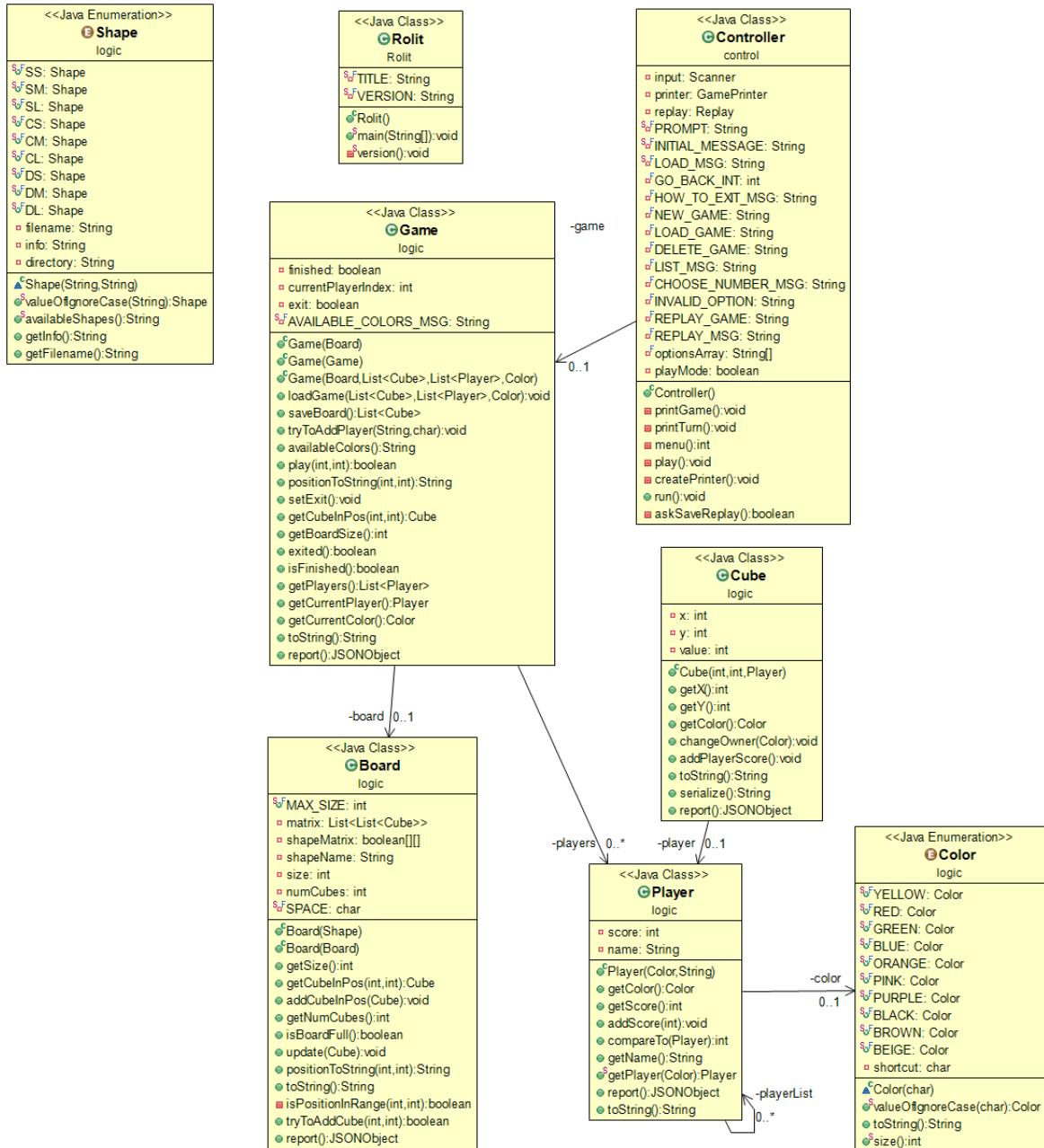


- Tal y como se mencionó anteriormente el método `update()` pasó a estar en `Board` y en este caso su única funcionalidad es cambiar de color todos los cubos que quedan atrapados en las direcciones válidas. Esta será la versión final del método para todos los Sprints siguientes.



Sprint (3)

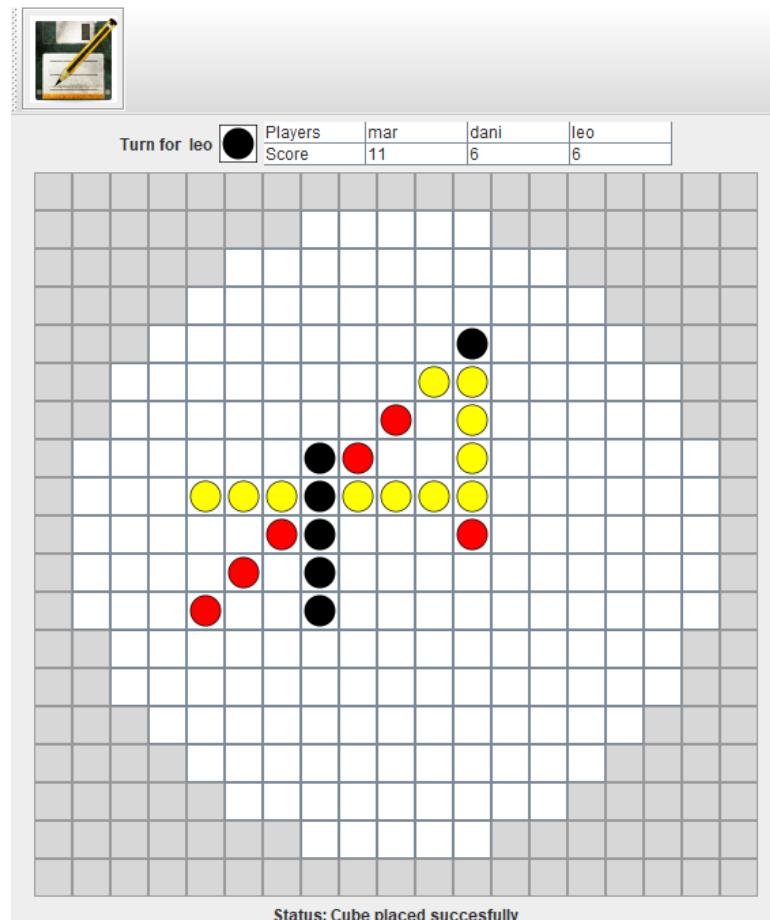
Diagrama de clases:



En esta tercera versión no hay apenas cambios, todo se gestiona de la misma manera salvo el Board. Ahora existen 9 tipos de tablero, hay tres formas y tres tamaños (grande, mediano, pequeño) para cada forma (rombo, círculo, cuadrado). Esto es posible gracias a métodos como `getShapeSize()` o `loadShape()` de la clase `SaveLoadManager`.

Sprint (4)

A partir de este Sprint los cubos pasan a ser redondos y del Color que representan y se van introduciendo en las casillas del tablero.

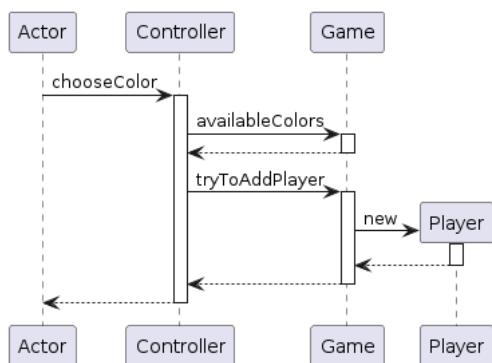


Como usuario quiero poder jugar a Rolit con una interfaz agradable

Como usuario, me gustaría que se pudiesen personalizar los colores con los que jugamos cada jugador porque hace más visual el juego.

Sprint (2)

En este momento, la creación del juego se realizaba a través del controlador y éste poseía los métodos necesarios para preguntar por las características concretas que se querían del mismo. De este modo, los métodos para elegir color se encontraban en la clase `Controller`, concretamente el responsable de escoger el nombre y el color de los jugadores era `void chooseColor(int numPlayers)`.



El funcionamiento era iterativo para cada jugador:

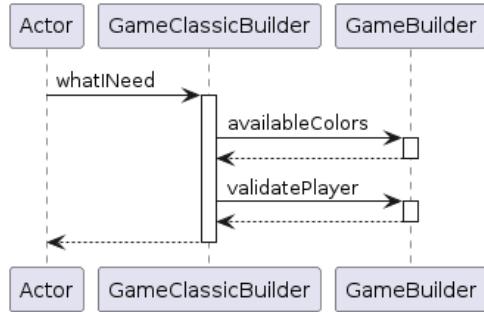
- Se consultaba al `Game` que colores seguían disponibles (puesto que él va sabiendo qué jugadores tiene añadidos)
- Se escoge el color y se intenta añadir el jugador con dicho color al juego, con resultado positivo o con una excepción de error.

Sprint (3)

Con la creación de la nueva clase `GameGenerator` que constituye una primera aproximación a las factorías, se han movido los métodos de los que hablábamos en el sprint anterior a esta nueva clase.

Sprint (4)

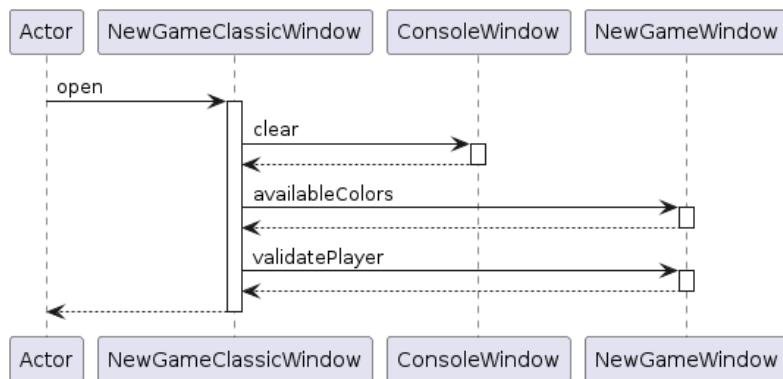
Este caso es más complejo que los anteriores. En este sprint nos encontramos en un punto intermedio del proyecto en el que se han implementado los Builder y las factorías, pero no se ha llegado a una interfaz gráfica de consola adecuada siguiendo el *Modelo-Vista-Controlador*.



Como todavía no se tiene una vista de consola adecuada, los Builder en este caso tienen una función llamada `void WhatINeed(JSONObject o)` que se encarga de pre-guntar al usuario lo necesario para terminar de construir el Game. El método que permite conocer los colores disponibles se sigue llamando `String availableColor()`, solo que ahora está situado en la clase `GameBuilder` y el método que me permitía intentar añadir un jugador al juego una vez elegido el color se llama ahora `bool validatePlayer()` y de nuevo se encuentra en `GameBuilder`. La diferencia fundamental es que en este caso no se añade el jugador al juego, sino que se añade al `JSONObject` del Game para que luego el Builder concreto lo construya adecuadamente.

Sprint (5)

En este sprint el funcionamiento es básicamente el mismo que en el anterior, sin embargo, cambia la ubicación de los métodos pues se introdujo la interfaz de consola. En este caso, las funcionalidades de `WhatINeed()`, `availableColors()` y `validatePlayer()` han pasado a la clase que representa la ventana de creación de juego de la consola `NewGameWindow` dejando el funcionamiento general como:



Como usuario, me gustaría que tenga una interfaz gráfica amable porque hace más fácil jugar.

En este apartado, los primeros sprint realmente no son relevantes para comprender cuál ha sido la evolución del proyecto con respecto a esta historia de usuario, pues el desconocimiento de técnicas como el patrón del *Modelo-Vista-Controlador* y la poca

experiencia con proyectos grandes hace que el soporte gráfico y la encapsulación de la entrada salida en los primeros sprint sea realmente pobre una vez visto en perspectiva con el resultado final.

Sin embargo, poseen gran importancia los sprint 4 y 5, pues son en los que se incluye por primera vez una vista gráfica que no sea la de consola y porque por primera vez se formaliza una entrada salida controlada y a través de clases especializadas que son usadas a modo de componentes gráficos, pero para la consola.

Sprint (1)

Al comienzo del proyecto, el juego comenzaba (tras la generación de los objetos correspondientes) con la llamada al método `run()` del `Controller`. Este método contenía en su interior el propio menú en forma de sentencias de código:

```
1 public void run() {  
2  
3     System.out.println();  
4     System.out.println("Que desea?");  
5     System.out.println();  
6     for (int i = 0; i < arrayOpciones.length; ++i)  
7         System.out.println((i+1) + ". " + arrayOpciones[i]);  
8  
9     int respuesta = 1;  
10    boolean repeat = true;  
11    ...  
12}
```

y también la parte de impresión del tablero y de las jugadas conforme avanzaba el juego:

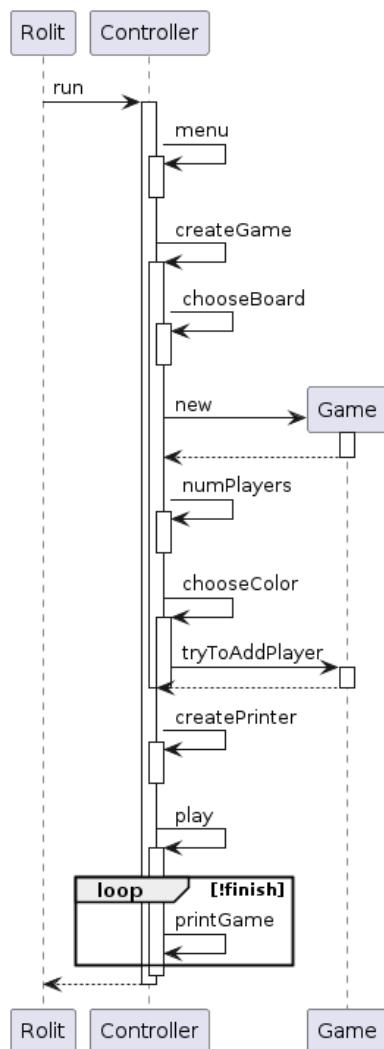
```
1 while(!game.isFinished()) {  
2     String command;  
3     int posx, posy;  
4     boolean valido = false;  
5  
6     printGame();  
7  
8     while (!valido) {  
9         System.out.println(printer.showTurn());  
10        System.out.println("Introduce un comando:");  
11        System.out.println("c : Poner un cubo");  
12        System.out.println("s : Guardar partida");  
13        command = input.next();  
14        if ("c".equals(command)) {  
15            System.out.println("Introduce la posicion x: ");  
16            posx = input.nextInt();  
17            System.out.println("Introduce la posicion y: ");  
18            posy = input.nextInt();  
19            valido = game.play(posx, posy);  
20        }  
21        else if("s".equals(command))  
22            saveLoadManager.saveGame();  
23        else  
24            System.out.println("Invalid Command");  
25    }  
26}  
27}
```

Tal y como se ve, principalmente la función más importante de impresión es la función `void printGame()` que se encargaba simplemente de hacer una llamada al método `String toString()` de la clase `GamePrinter` que era la experta en representación por pantalla de todo lo relativo al juego. Fundamentalmente, el desconocimiento del patrón Modelo-Vista-Controlador hizo que las instrucciones de uso del jugador o los diferentes menús de carga y guardado se hicieran en el propio método

que se encargaba de hacer cada cosa, dificultando el poder hacer un diagrama de secuencia de cómo se imprime el menú o cómo se piden acciones.

Sprint (2)

En base a la experiencia en proyectos anteriores, durante este sprint se consolidó la clase **Controller** como la encargada de todos los procesos externos a los bucles del propio juego, es decir, se encargaba determinar el número de jugadores, el tipo de tablero, creación de los elementos del **Game**... Por tanto, el menú principal estaba en la clase **Controller**, concretamente la funcionalidad la encapsulaba la función `int menu()` que interactuaba con otros métodos privados de la clase para que el usuario pudiese decidir el resto de cosas y la funcionalidad de mostrar el juego seguía estando representada en la clase **GamePrinter** con el método `toString()`.



Para formalizar esta primera aproximación a una encapsulación formal de cada una de las pantallas a métodos o clases concretas y especializadas, se generan las siguientes constantes en la clase **Controller** para tener unificados los mensajes que se envían:

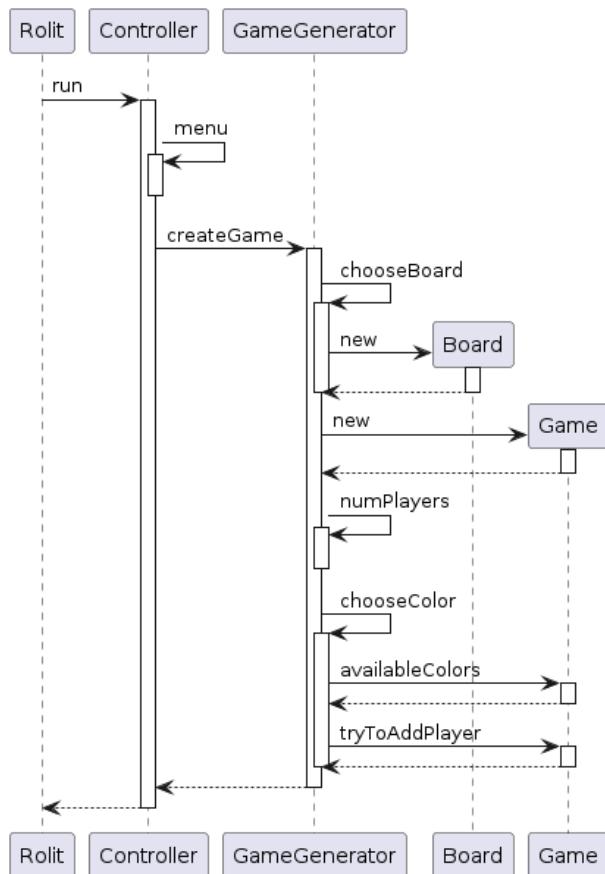
```

1 private static final String PROMPT = "Command > ";
2     private static final String NAME_PLAYERS = "Name the players: ";
3     private static final String INITIAL_MESSAGE = "Choose an option: ";
4     private static final String CHOOSE_COLOR = "Choose a color shortcut: ";
5     private static final String NUMBER_PLAYERS_MSG = "Choose the number of players... ";
6     private static final String ERROR_PLAYERS_MSG = "Number of players must be... ";
7     private static final String BOARD_MSG = "Choose your board size [8 - ...";
8     private static final String BOARD_ERROR = "Board size must be a number between... ";
9     private static final String LOAD_MSG = "Type the name of the file (. to load... ");

```

Sprint (3)

El equipo de desarrollo observó que la clase **Controller** estaba perdiendo cohesión al asumir muchas responsabilidades que ciertamente no eran de un controlador. Para ello, se creó la clase **GameGenerator** como una antesala de lo que sería el patrón factoría. De este modo, el menú principal queda encapsulado en la función `int menu()` de la clase **Controller**, pero las pantallas donde se deciden el número de jugadores, el tipo de tablero, ... quedan como métodos privados de la nueva clase **GameGenerator** que es la encargada de saber crear el juego.

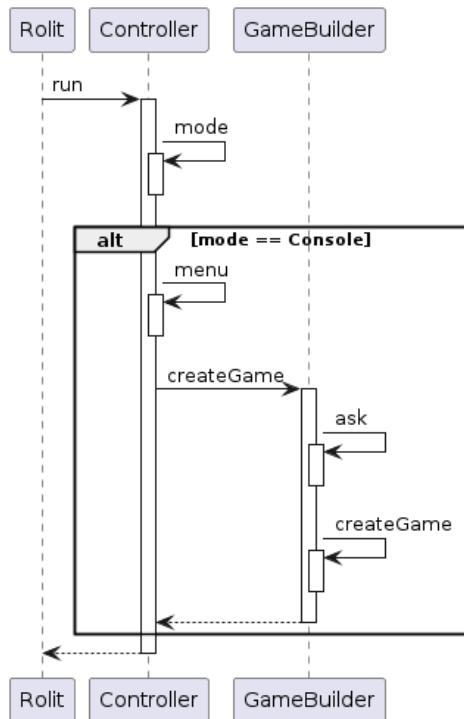


En este caso, la función que imprime cada iteración del juego sigue siendo `printGame()` en **Controller** y aquellas que se encargan imprimir los mensajes de guardado y carga, así como de errores, siguen diseminadas por las respectivas clases que utilizan dichos métodos como sentencias `System.out.println("...")`, pues aún no se ha implementado el Modelo-Vista-Controllador.

Sprint (4)

Console

En este sprint, fundamentalmente se aplica el patrón Builder y Factorías a la situación del sprint anterior. En este caso, el menú principal sigue en el método `int menu()` de la clase `Controller` y los métodos que antes estaban en `GameGenerator` y se encargaban de pedir el número de jugadores, forma del tablero, ... por consola ahora forman parte del método `ask()` de las factorías (que por dentro saben que preguntar en cada caso).

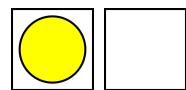


A pesar de que se ha incluido el *Modelo-Vista-Controlador* como patrón de arquitectura, la diferencia fundamental entre tener un nuevo hilo en *Swing* y seguir en el mismo hilo que *Main* ha hecho que no se haya formalizado una vista al uso para la parte de consola y que el código para ciertas excepciones y mensajes siga perdido por los lugares donde se producen dichas incidencias.

GUI

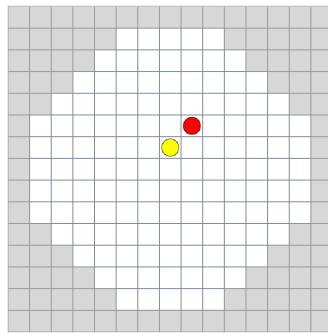
En este sprint se desarrolló una primera versión de la interfaz gráfica, lo cual involucra a un gran número de clases y métodos. Iniciemos un recorrido por cada uno de los componentes visuales explicando su finalidad

CeldaGUI



Esta clase representa cada una de las posiciones del tablero y puede ser vacía o tener un cubo. La mayor parte de funciones de esta clase tienen como objetivo cambiar el color del cubo que representa cada celda.

BoardGUI

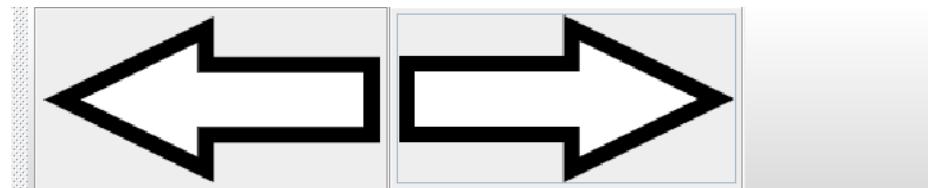


La clase *BoardGUI* extiende de *JPanel* y es la encargada de visualizar el tablero de la partida. Se puede apreciar que el tablero está formado por un conjunto de *CeldaGUI*.

ControlPanel

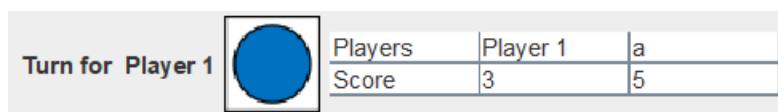


El *ControlPanel* es una *JToolBar* que cuenta con un botón para guardar partidas, que puede ser pulsado en cualquier momento durante la ejecución del juego.



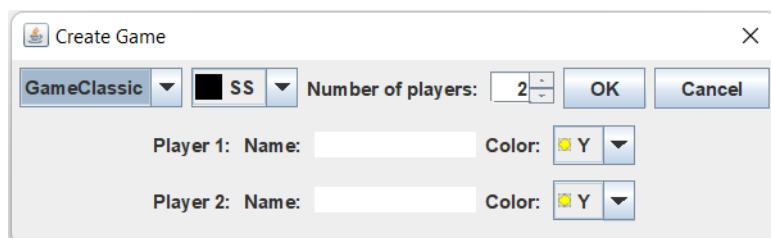
En el caso de las *replays*, en el *ControlPanel* aparecen dos flechas para poder recorrer los estados.

TurnAndRankingBar



Esta clase es un *JPanel* se encarga de mostrar a los usuarios del turno del jugador actual, las puntuaciones de cada uno de los participantes y la modalidad de juego.

CreateGameDialog



Como su propio nombre indica, esta ventana extiende de *JDialog* y tiene como objetivo poder configurar una partida desde cero, combinando todas las características posibles para crear un juego a gusto de los usuarios.

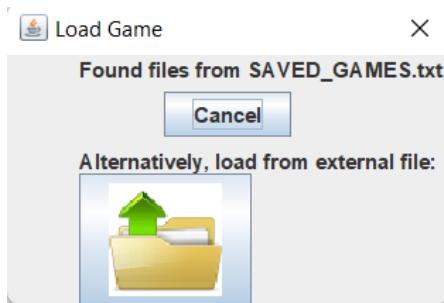
En esta pantalla el usuario elige el modo de juego para la partida (*GameClassic* o *GameTeams*), la forma y tamaño del tablero (cuadrado, círculo o rombo, pequeño, mediano o grande), el número de jugadores (entre 2 y 10, ambos inclusive) y el nombre y color de cada jugador.

En caso de seleccionarse el modo por equipos, el usuario introducirá el nombre de ambos equipos y el equipo al que pertenece cada jugador.

Una vez el usuario presiona el botón *OK* se le lleva a la pantalla de juego.

Si el usuario presiona *Cancel* se le lleva de vuelta a la pantalla principal.

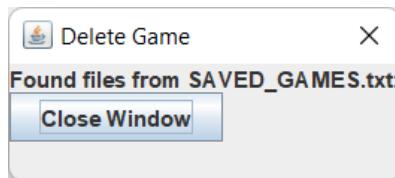
LoadGameDialog



En esta pantalla se muestra una lista con las partidas guardadas, de forma que si se elige una de estas partidas se cargará inmediatamente.

Aparece también debajo un botón de carga de ficheros que abre un *JFileChooser* por si se quiere cargar un juego que no esté incluido en la lista de partidas guardadas.

DeleteGameDialog



En esta pantalla se muestra la lista de partidas guardadas y un botón para confirmar el borrado. Con esto, si se selecciona una de las partidas guardadas y se presiona el botón inferior, la lista se elimina de la lista de partidas guardadas.

Status Bar



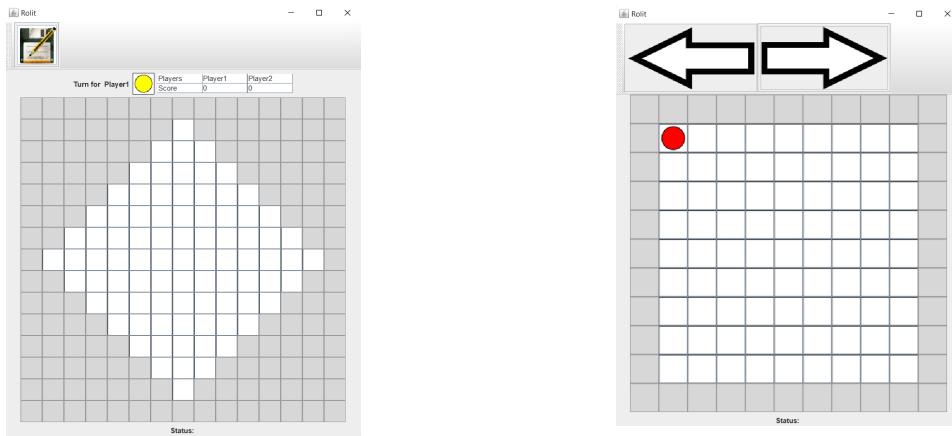
La *Status Bar* es un *JPanel* que se encarga de mostrar información relevante durante la partida, como mensajes de error o de confirmación, otorgándole al usuario un feedback adecuado.

MainWindow



Inicialmente la ventana principal comienza con una pantalla en la que se muestran cuatro opciones a elegir por el usuario: *Create new game*, *Load game*, *Delete game*, *Load replay*.

Si se ha decidido jugar a una partida o cargar una replay, el panel principal de la *MainWindow* será reemplazado por un panel que contiene, de arriba a abajo, un ControlPanel, una TurnAndRankingBar, un BoardGUI y una StatusBar.

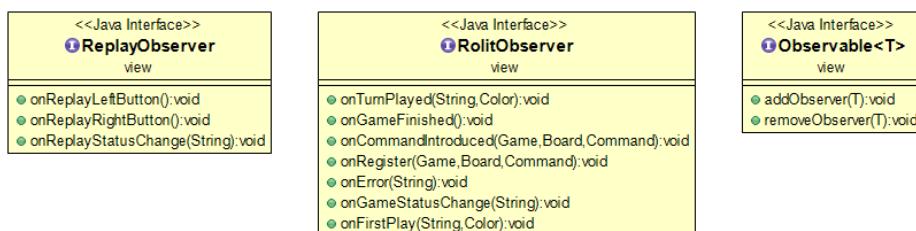


Funcionamiento interno

Una vez visualizadas todas las pantallas es el momento de hablar su funcionamiento interno. Al introducir la GUI decidimos aplicar el patrón MVC, de manera que los futuros cambios en el modelo produzcan modificaciones mínimas en la vista y controlador y viceversa. Para más información sobre el Modelo-Vista-Controlador puede hacer click [aquí](#).

Para la comunicación de las vistas con los modelos se decidió utilizar el patrón observador, así son los propios modelos los que comunican a las vistas cuando deben actualizarse.

La implementación de este patrón se llevó a cabo mediante tres interfaces: Observable, diseñada para los modelos; RolitObserver, pensada para los observadores de la clase *Game*; y ReplayObserver, que utilizan los observadores de la clase *Replay*.

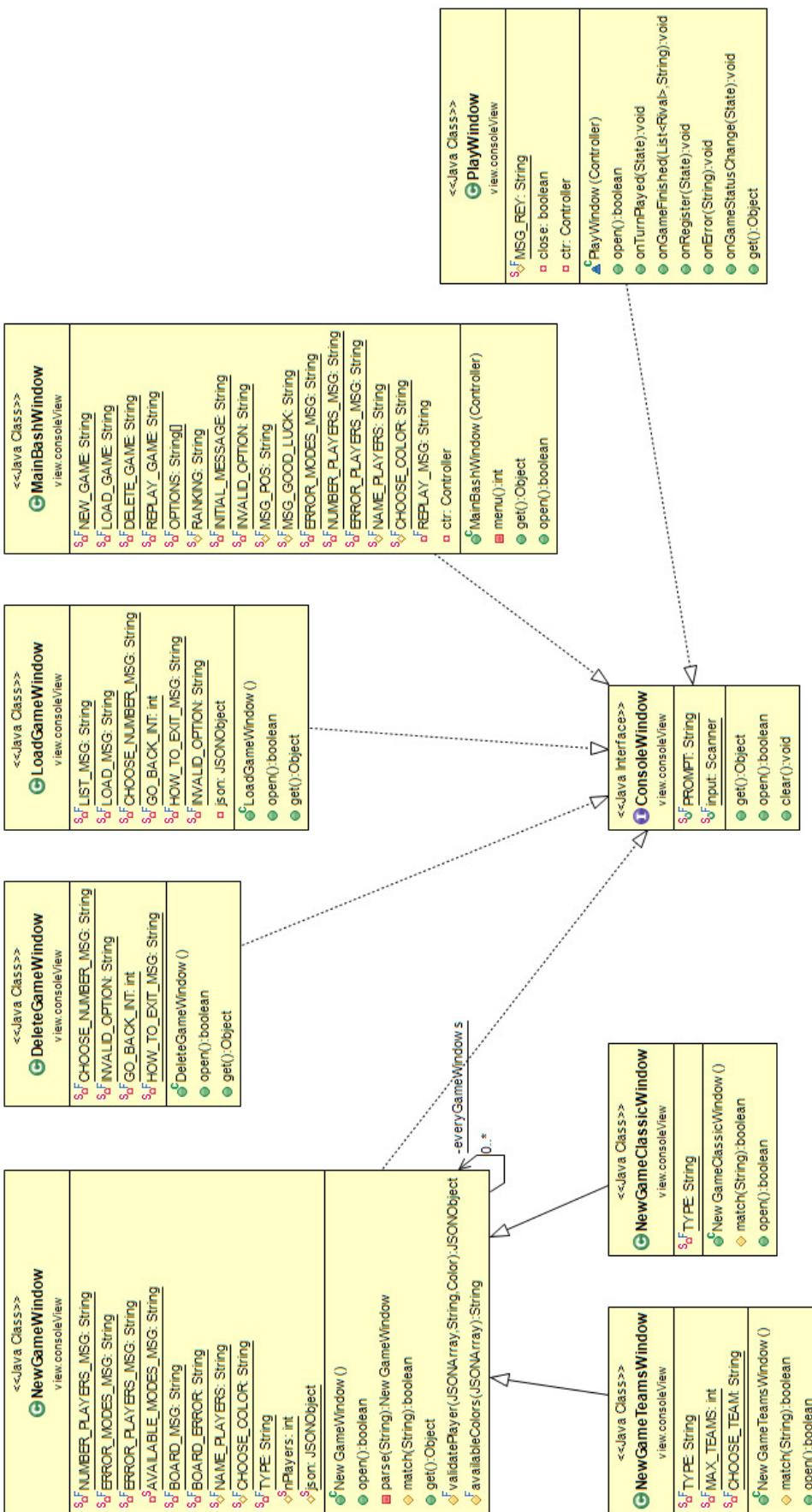


En el caso de la GUI, la comunicación con los modelos se lleva a cabo a través de los *ActionListeners* de los botones, que ejecutan el comando que corresponda.

Sprint (5)

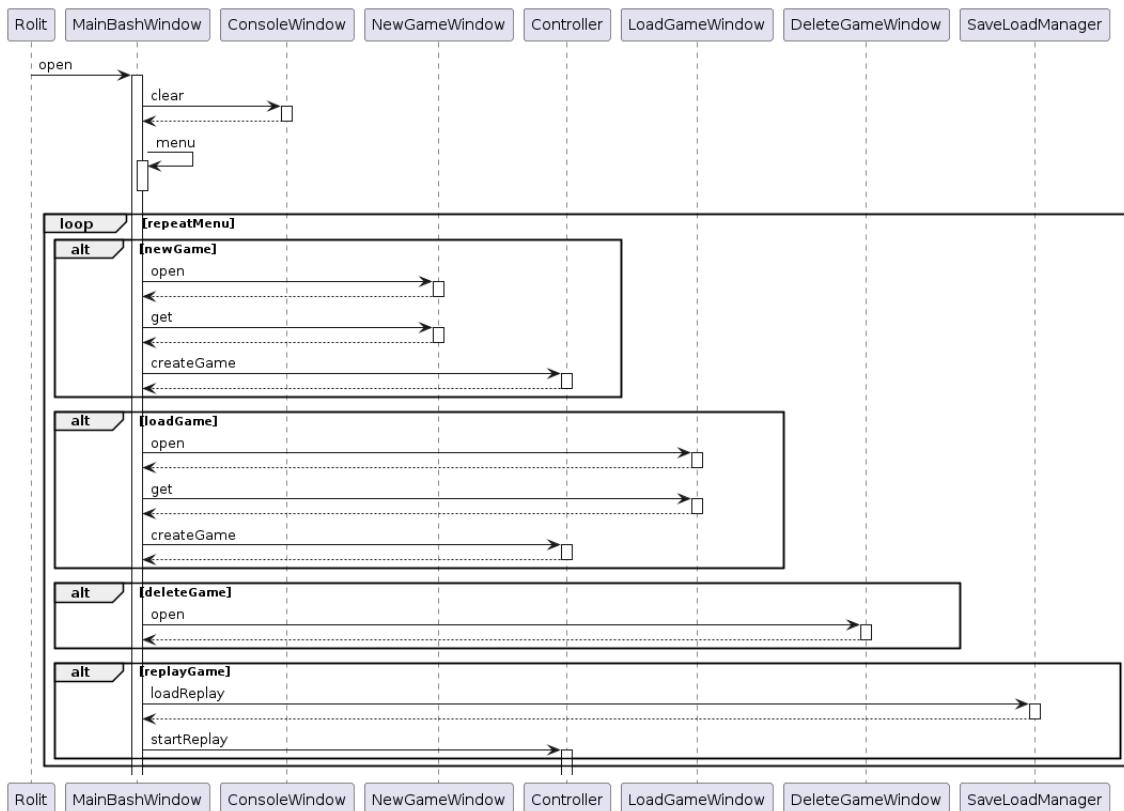
Console

En el sprint 5, la evidencia de las limitaciones que tiene tener la consola tan poco definida en comparación con la programación de componentes de *Swing* y la implementación de un hilo a parte para el modelo hacen posible que se cree una serie de clases a modo de “componentes” para formalizar de una vez la vista del modo consola:



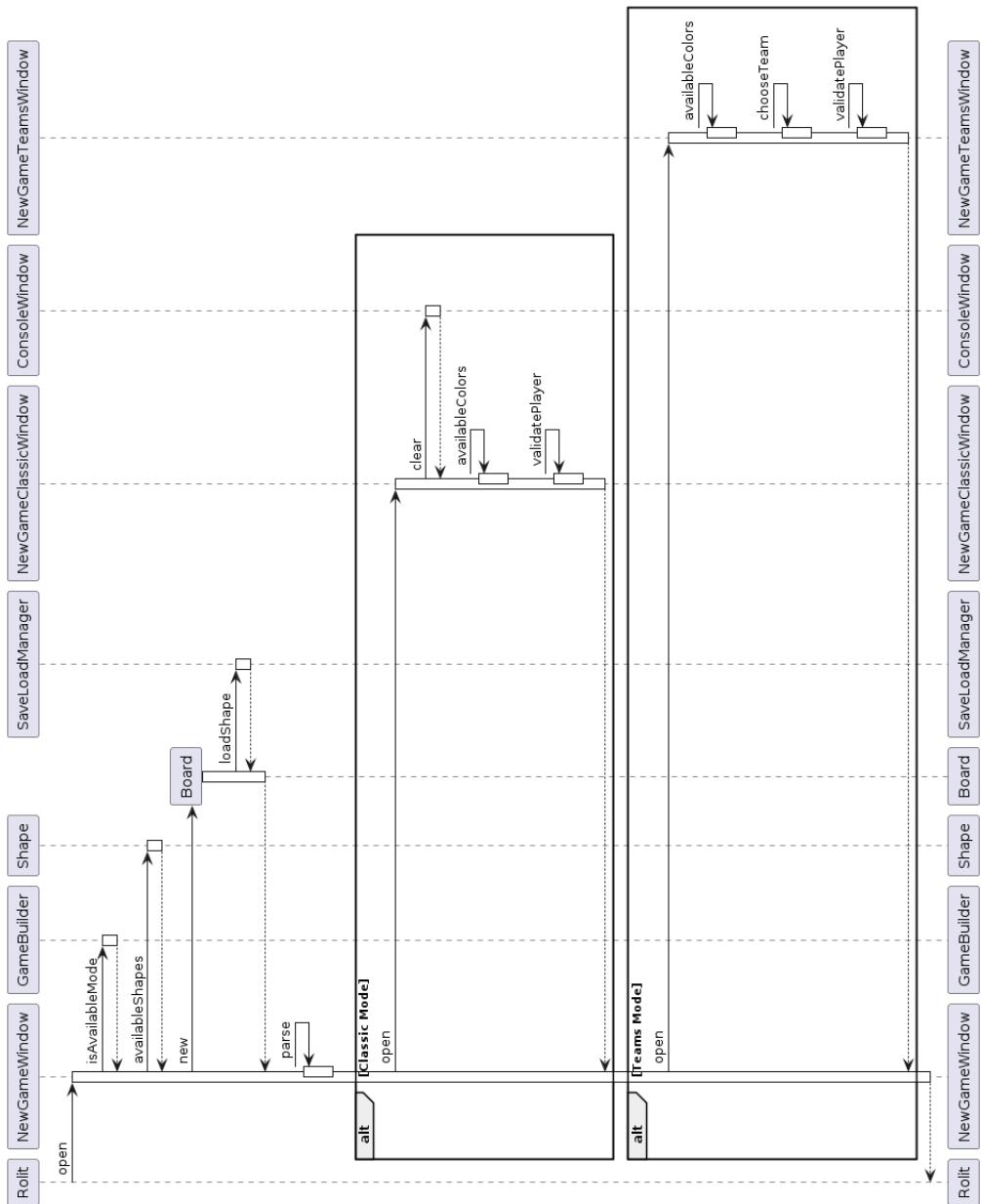
Tal y como se ve en el diagrama de clases, hemos creado una interfaz `ConsoleWindow` que será la abstracción que represente un “componente de la consola”. Todas las posibles ventanas del juego extenderán de dicha interfaz y el método `bool open()` será el que indicará que el componente se muestra y empieza a funcionar. Además, cabe destacar que hacerlo de esta manera permite que las clases que representan a las ventanas en la consola ahora puedan ser observadores, por lo que cubrimos mucho mejor el *MVC* y además nos permite simplificar el comportamiento del modelo al ser unificado para cualquier vista la forma de notificar los cambios.

Con todo esto conseguimos que al iniciar el juego en la clase `Rolit` y seleccionar el modo de vista deseado, se escoja qué componente (la `MainWindow` de *Swing* o la `MainBashWindow` de la consola) se abrirá para empezar la vista. Esto permite que a nivel de vistas solo trabajemos con abstracciones de las clases que conforman cada ventana, por tanto, agregar nuevas ventanas en la consola se reduce a crear una nueva clase que encapsule lo que se espera de la nueva ventana. De este modo, el menú principal de elección sobre un nuevo juego, cargar uno antiguo, etc. queda como:

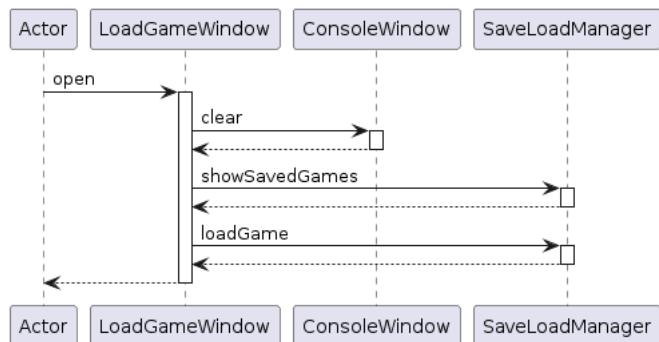


Y el funcionamiento interno de cada una de las ventanas puede verse de la siguiente manera:

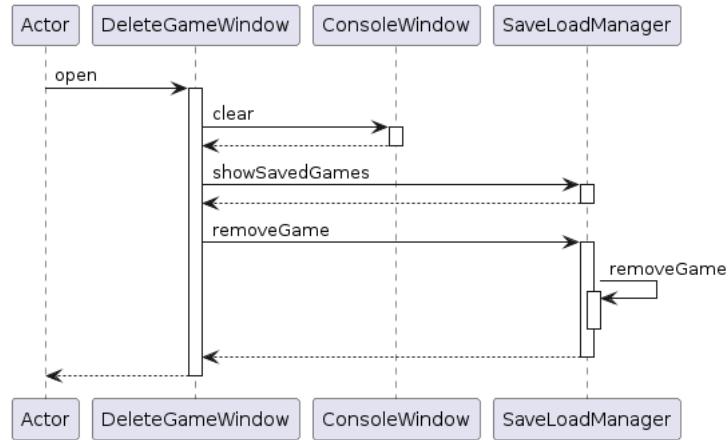
- NewGameWindow: está implementada utilizando el patrón factoría (puesto que son clases íntimamente ligadas a los `Builder` ya que son las ventanas donde se crea el juego) y heredan la funcionalidad que tenían los métodos `void ask()` de los `Builder`.



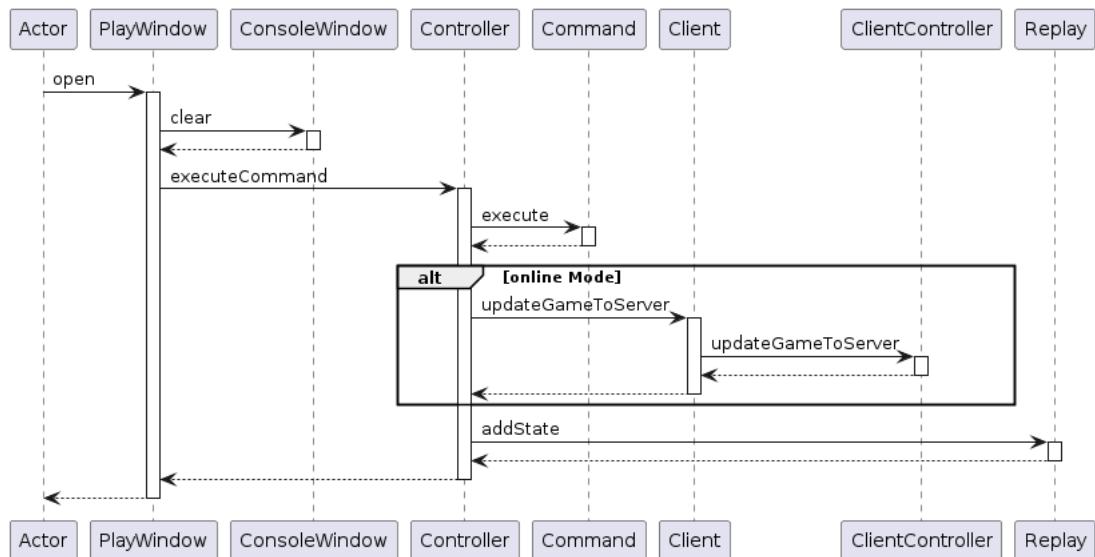
- LoadGameWindow: es una clase de ventana que principalmente hace las llamadas adecuadas a SaveLoadManager que el la clase *Experta* en el manejo de flujo de ficheros de carga y guardado.



- DeleteGameWindow: es una clase de ventana que de nuevo se encarga de gestionar las llamadas correspondientes a SaveLoadManager para borrar uno de los juegos cargados.



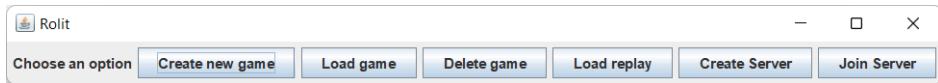
La clase que principalmente soporta la ventana de consola una vez se va a comenzar a jugar es la clase PlayWindow que se mantiene abierta hasta que el juego notifique que ha terminado. En dicha ventana se generan a través de la entrada por teclado, los comandos que posteriormente cambiarán el estado del modelo y está implementada de forma que sea un observador, tanto para percibir los cambios que ocurren este y mostrarlos adecuadamente como para reaccionar en consecuencia mostrando cosas según el tipo de juego que tengamos (aunque ella solo maneja la abstracción de un Game cualquiera):



GUI

MainWindow

La incorporación de la funcionalidad para jugar en red trajo consigo la necesidad de añadir nuevos botones al menú principal para poder acceder a ella.



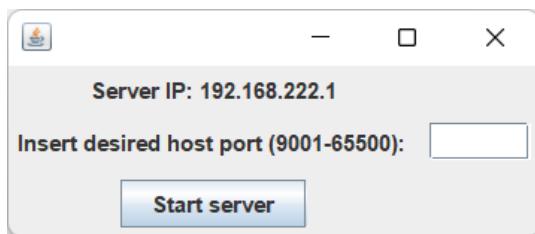
CreateGameDialog



Al configurar la partida de un servidor no es posible introducir la información de los jugadores, pues es cada uno individualmente quien decide su nombre y color desde su ordenador.

Por este motivo, se añadió un atributo al constructor de *CreateGameDialog* que permite ocultar el panel que contiene los componentes para introducir los datos de los jugadores.

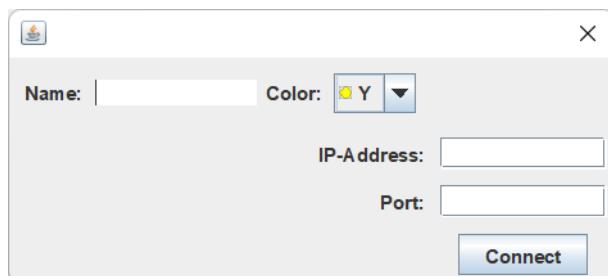
CreateServerDialog



Para poder crear el servidor el usuario debe elegir en qué puerto hostearlo, para ello se creó este *JDialog*, que abre el servidor una vez se pulsa el botón “Start Server”.

En esta versión del juego, la poca experiencia con el manejo de hilos del equipo de desarrollo provocó que esta ventana se quedase pillada hasta que todos los participantes se unieran al servidor.

JoinServerDialog



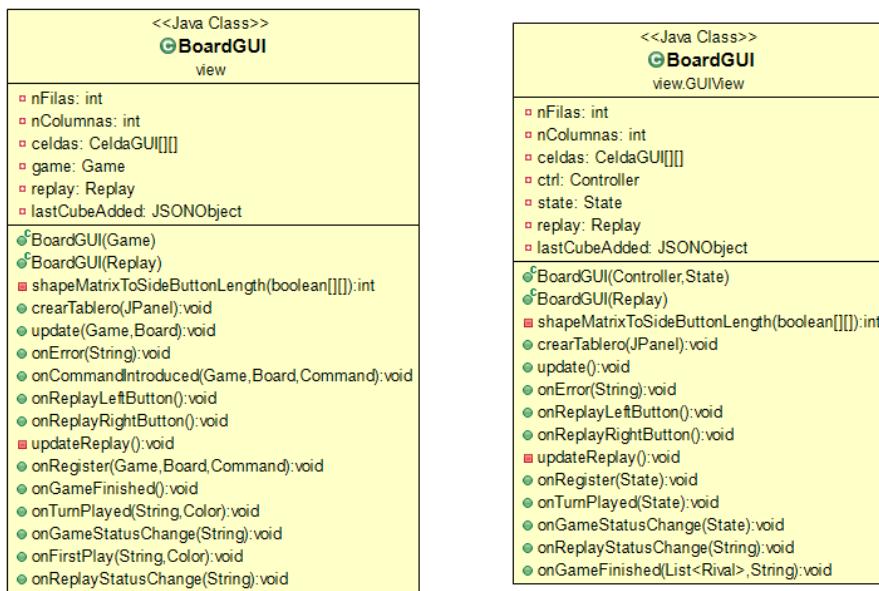
Como ya se ha mencionado anteriormente, es cada usuario al unirse al servidor quien decide su nombre y su color. La clase *JoinServerDialog* es la encargada de ello, además de recoger la IP y el puerto del servidor al que se quiere acceder.

Fucionamiento interno

En el sprint anterior nuestra única intención era hacer una interfaz gráfica funcional, y ese objetivo fue alcanzado exitosamente. Sin embargo, la GUI accedía directamente a todos los métodos que necesitase de la clase *Game*, algo que no es correcto desde el punto de vista de la programación orientada objetos.

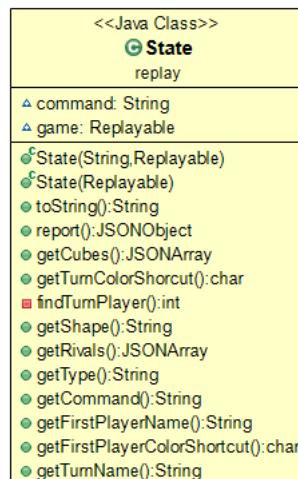
Para solventar este problema planteamos inicialmente el uso de objetos transferencia que restringiesen los métodos de las clases que eran necesarias para visualizar el juego.

Finalmente, esta idea fue rechazada en favor de reutilizar el código ya existente. Así, se decidió que la información se transmitiese mediante la clase *State* creada originalmente para reproducir las *replays*, pues al fin y al cabo, un estado representa una serialización del juego y contiene toda la información necesaria.



Evolución de la clase *BoardGUI*. Sprint 4 (izq) y Sprint 5 (der)

Por consiguiente, todas las instancias de clases relacionadas con el modelo de *Game* fueron eliminadas y reemplazadas por estados y JSONObjects. Para ello fue necesario añadir nuevos métodos a la clase *State*.



Sprint (6)

Console

Los componentes a modo de ventanas de la consola no sufrieron más cambios más allá del sprint 5, únicamente se corrigieron errores en los `input.nextLine()` de la lectura puesto que fallaba al no consumir saltos de línea abandonados en el buffer.

GUI

En este sprint la GUI fue refactorizada por completo, cambiando tanto visualmente, como a nivel de funcionamiento interno en algunas clases para hacer el código más manejable.

Creación de RolitComponents

Hasta este momento el proyecto utilizada los componentes visuales de Java por defecto, dando lugar a una interfaz gráfica funcional, pero con carencias visuales.

Para el estilo de los componentes, se optó por una interfaz minimalista basada en el color blanco y en el azul que aparece en el logo del juego Rolit, al que se denominó *BLUE_ROLIT*.

Los *RolitComponents*, como la clase *RolitButton* o *RolitTextArea*, además de homogeneizar el entorno visual facilitan los cambios de estilo en un futuro.

Veamos el resultado gráfico tras la aplicación de estos componentes, sumado a reestructuraciones en los Layouts y la inclusión de nuevas imágenes e iconos

MainWindow



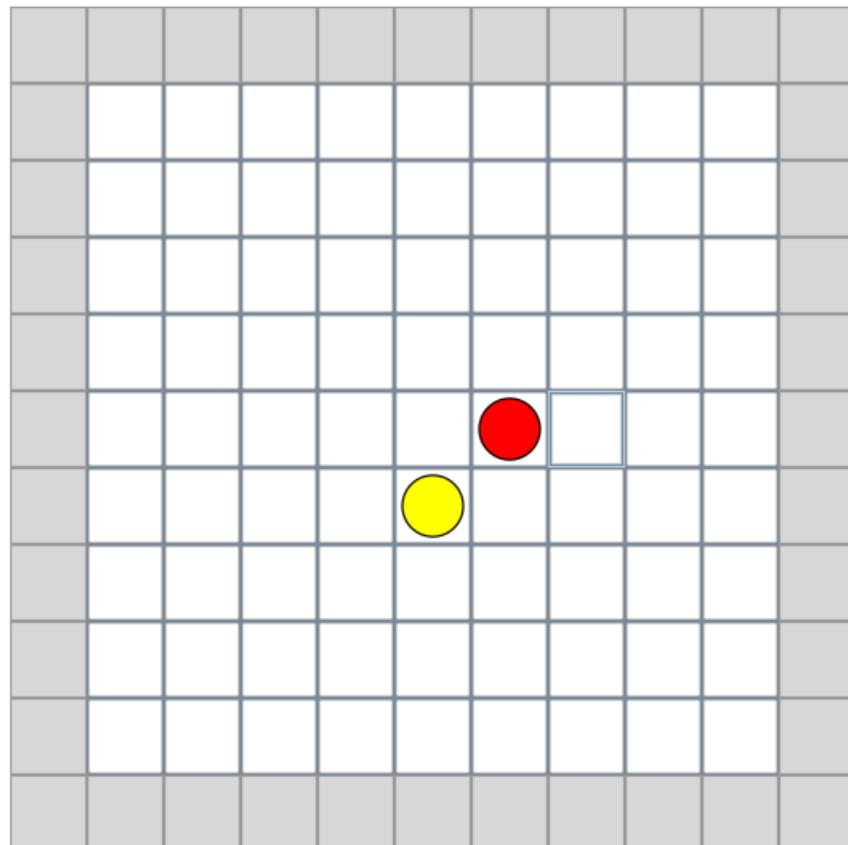
Menú Inicial

 Rolit

— □ ×

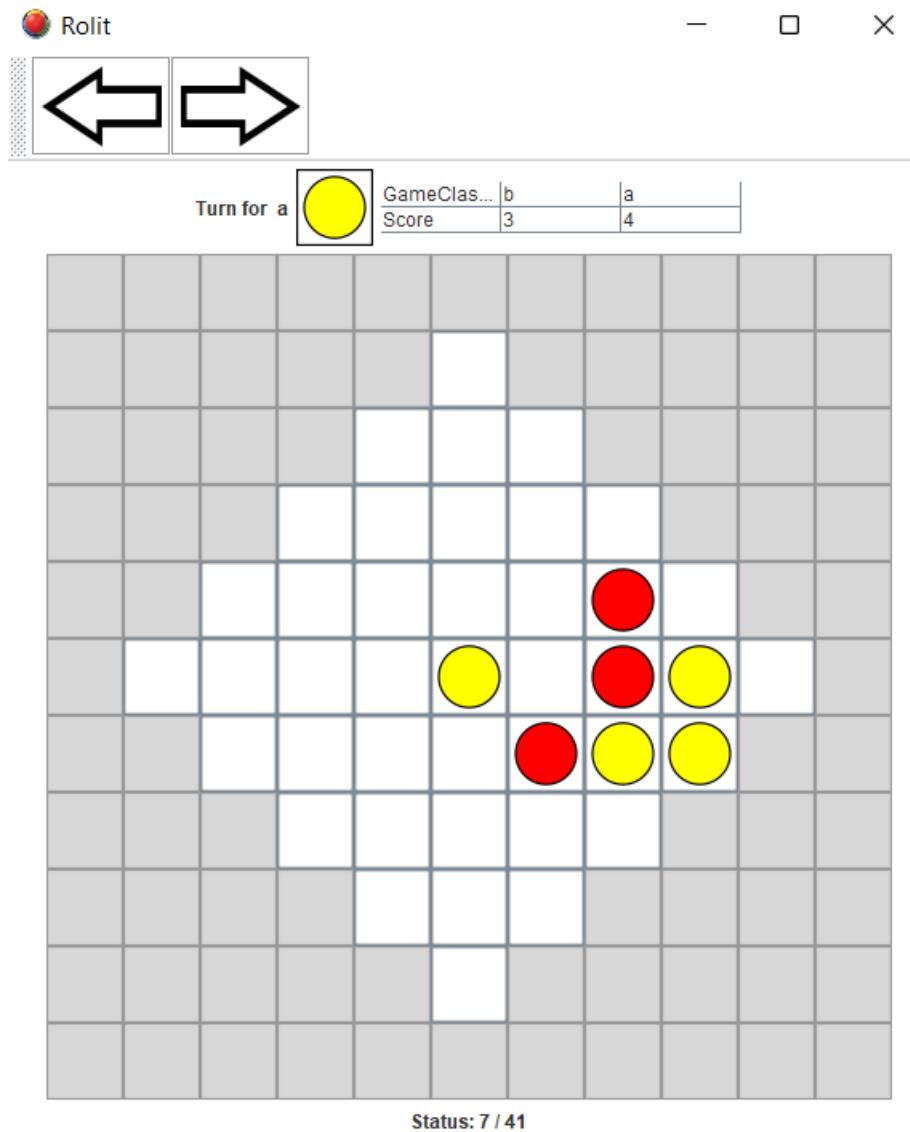


Turn for b	GameClass	a	b
Score	1	1	



Status:

Jugar una partida



Reproducir una *replay*

CreateGame package

En sprints anteriores había una clase encargada de crear y gestionar todos los componentes visuales necesarios para crear un juego nuevo, resultando en una clase demasiado compleja.

Por ello, la antigua clase *CreateGameDialog*, fue dividida en otras más pequeñas. Dando lugar a:

- *PlayerDataPanel*: panel que contiene los componentes necesarios para obtener la información de un jugador.

Player 1: Name: Color: ▾ Al:

- *TeamDataPanel*: panel que contiene los componentes necesarios para obtener la información de un equipo.

Team 1:

- *CreatePlayersPanel*: conjunto de *PlayerDataPanel*.
- *CreateTeamsPanel*: conjunto de *TeamDataPanel*.
- *GameConfigurationPanel*: panel encargado de obtener la configuración básica del juego: modo de juego, forma, numero de jugadores...



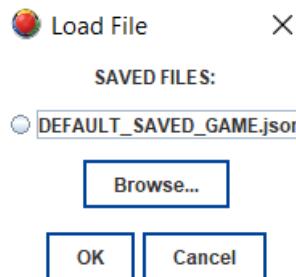
- *CreateGameDialog*: ventana de diálogo que contiene un *GameConfigurationPanel* y un *CreateTeamsPanel*.



- *CreateGameWithPlayersDialog*: extiende a *CreateGameDialog*, añadiendo un *CreatePlayersPanel*.

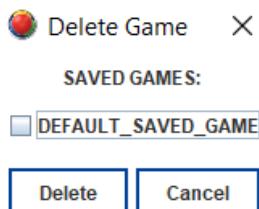


LoadFileDialog

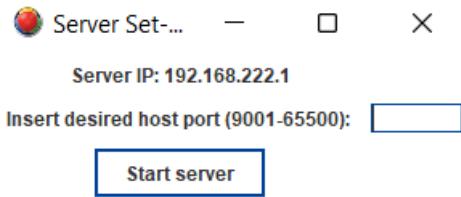


Las clases *LoadGameDialog* y *LoadReplayDialog* fueron abstraídas en *LoadFileDialog*, que permite cargar tanto partidas como *replays*, dependiendo del botón que se pulse.

DeleteGameDialog



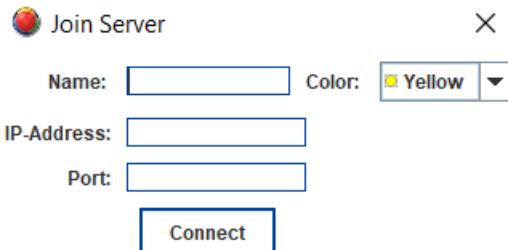
CreateServerDialog



Además, una vez creado el servidor, el usuario que está hosteandolo recibe feedback de cuanta gente se ha unido.



JoinServerDialog



Los usuarios también reciben feedback del servidor, mostrándose una venta que les invita a esperar a los demás.



Como usuario quiero que Rolit introduzca características innovadoras pensando en las posibilidades que brinda el multijugador

Como usuario, me gustaría que se pudiera jugar contra una inteligencia artificial, así como que ellas jugaran solas

Sprint (5)

Este fue el Sprint en el que se empezaron a desarrollar las distintas estrategias de las inteligencias artificiales. Se planearon tres, recogidas en las siguientes clases, todas herederas de la clase abstracta Strategy: RandomStrategy, GreedyStrategy y MinimaxStrategy.

La idea de la estrategia es que, cuando le toque jugar a una inteligencia artificial, la estrategia se encargue de calcular su siguiente movimiento y este se ejecutase inmediatamente después de su cálculo.

Para hacer esto, hemos hecho uso del **patrón estrategia**. Este patrón permite mantener un conjunto de algoritmos para la resolución de una tarea de distintas formas, de forma que se pueda dinámicamente elegir un algoritmo u otro en tiempo de ejecución. La explicación de como se ha llevado a cabo el patrón se explica a continuación.

Para encapsular esta lógica se creó la clase abstracta Strategy, para que cada estrategia en particular fuera una clase heredera de esta.

Se han desarrollado tres estrategias, que suponen tres niveles de dificultad distintos, y la lógica de estas está recogida en las siguientes clases: RandomStrategy, GreedyStrategy y MinimaxStrategy.

RandomStrategy: La idea es que se genere una posición cualquiera en el tablero, siempre y cuando esta sea válida. Esta es la posición que la inteligencia artificial jugará. Lógicamente, la tendencia general de las inteligencias artificiales que aplican esta estrategia es no obtener una gran cantidad de puntos, por lo que esta estrategia es la de nivel fácil.

GreedyStrategy: Esta estrategia tiene por intención analizar el tablero en busca de la posición que le garantiza al jugador el máximo número de puntos en este mismo turno. Esta estrategia lleva a jugadas mucho mejores y elaboradas, pero sigue sin ser la mejor, así que representa el nivel de dificultad medio.

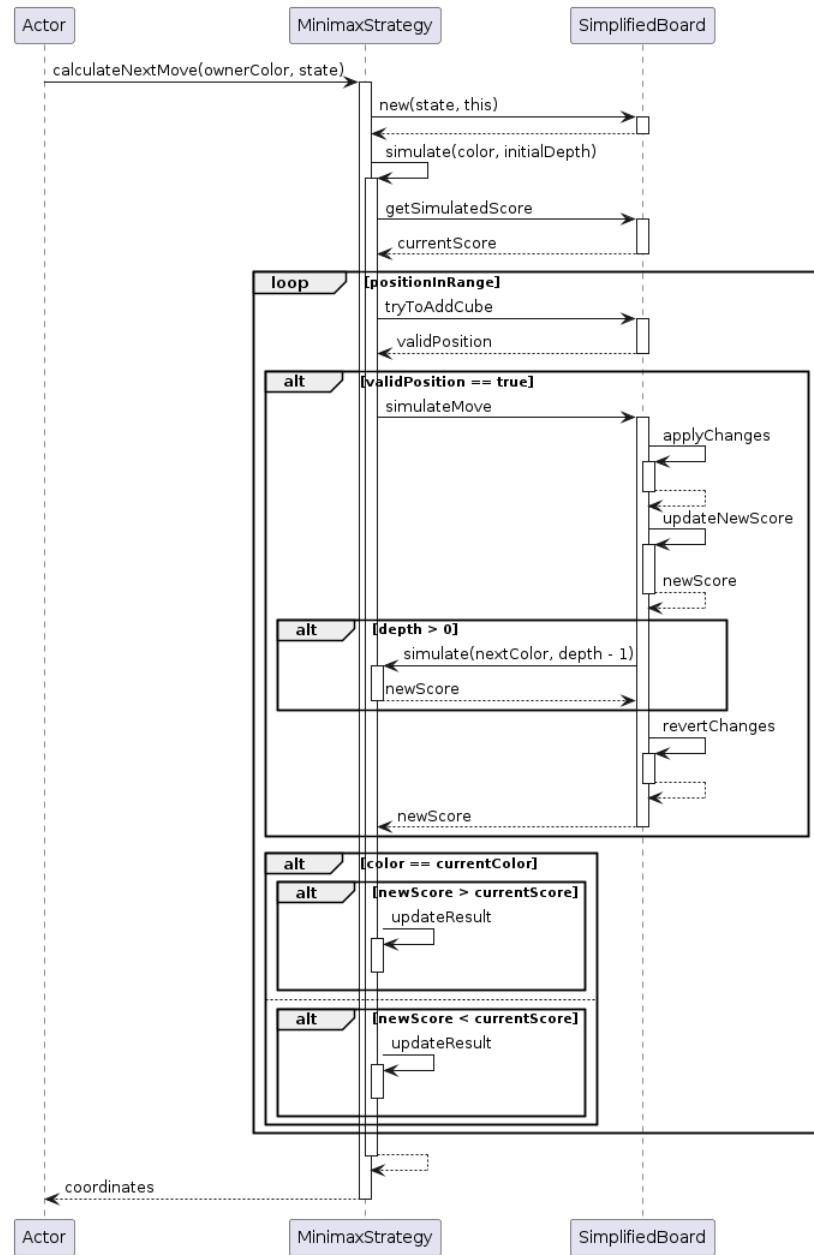
MinimaxStrategy: Esta estrategia realiza el cálculo de la posición en la que jugar el siguiente turno a través de una adaptación a Rolit del algoritmo Minimax. Este algoritmo se explica en el [Anexo III](#). Este es un algoritmo con un nivel de sofisticación considerablemente mayor que los de las anteriores estrategias y por ello (y tras comprobación en base a la simulación de numerosas partidas) este algoritmo representa el nivel de dificultad alto. Se dan más detalles de la implementación en la subsección [MinimaxStrategy](#).

Para hacer el cómputo de todas las simulaciones, usar como representación la que nos proporciona la clase Board resulta inviable, por lo que se ha creado una nueva

clase a modo de representación puramente funcional del tablero orientada a las simulaciones, llamada **SimplifiedBoard**.

Se pueden ver detalles de la implementación de estas clases en la subsección [Strategy](#).

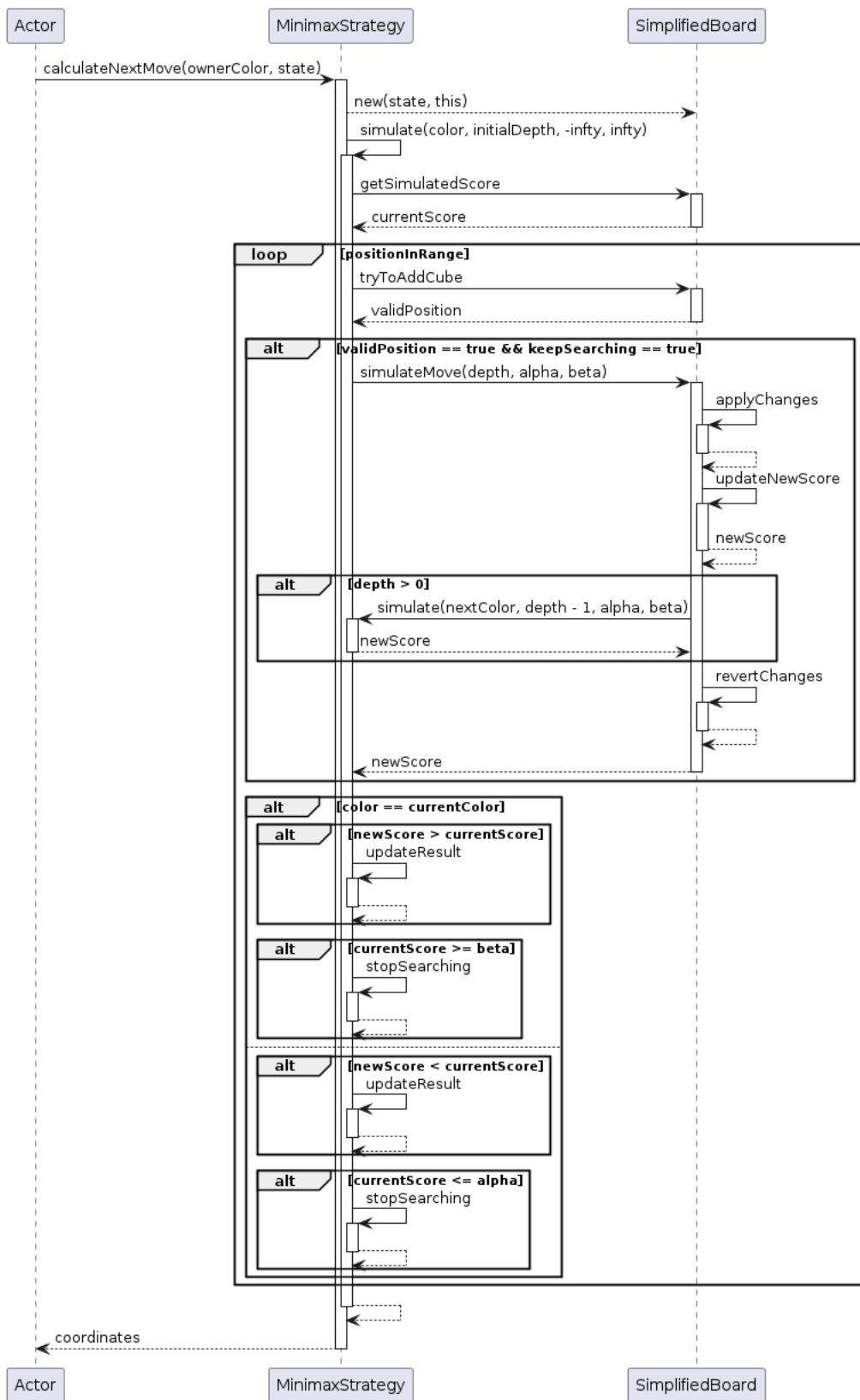
El cómputo del movimiento a jugar a través de la estrategia Minimax, llevado a cabo en el método `calculateNextMove()`, se ilustra en el siguiente diagrama:



Sprint (6)

Por motivos de eficiencia de los cálculos de **MinimaxStrategy**, se ha implementado de forma complementaria la poda Alfa-Beta. Los detalles acerca de este concepto se explican en el [Anexo IV](#).

Esta poda ha sido muy útil para reducir costes de cálculo, y el nuevo algoritmo mejorado queda reflejado en el siguiente diagrama:



Como usuario, me gustaría que se pudiera jugar en red.

Sprint (5)

En cuanto a la tarea de añadir un modo de juego en red, se concibe, planea e implementa la funcionalidad de red casi por completo, consigue llegar a una versión funcional de juego en red en **GameClassic**.

En cuanto al servidor, se define y se mantiene en todos los sprints la característica de que el servidor no posee el modelo, si no que es un intermediario entre clientes (pasando información procedente de un cliente al resto de clientes, para que estos últimos actualicen sus modelos). El diálogo **ServerView** interactúa con el servidor a la hora de dejar al usuario especificar los detalles sobre los que el servidor operará (puerto).

En cuanto al cliente, se define y se mantiene en todos los sprints la característica de que el cliente es el que posee el modelo, Cada vez que se hace una modificación en el modelo del cliente, se envía al servidor la información del juego; el servidor procede a actualizar la información del modelo del resto de clientes conectados.

Al final del Sprint, los objetivos alcanzados son los siguientes:

- Determinar si el servidor, o por el contrario el cliente, debería poseer el modelo. Optamos por la segunda opción.
- Definir toda la estructura en cuanto a relaciones jerárquicas de clases y dependencias entre las mismas.
- Crear un número suficiente de diálogos en GUI que permitan la conexión en red. Entre ellos, se encuentran:
 - **ServerView**
 - **JoinServerDialog**

El diseño y evolución pormenorizados de estos diálogos se encuentran en el [apartado de la historia de usuario dedicada a la interfaz](#).

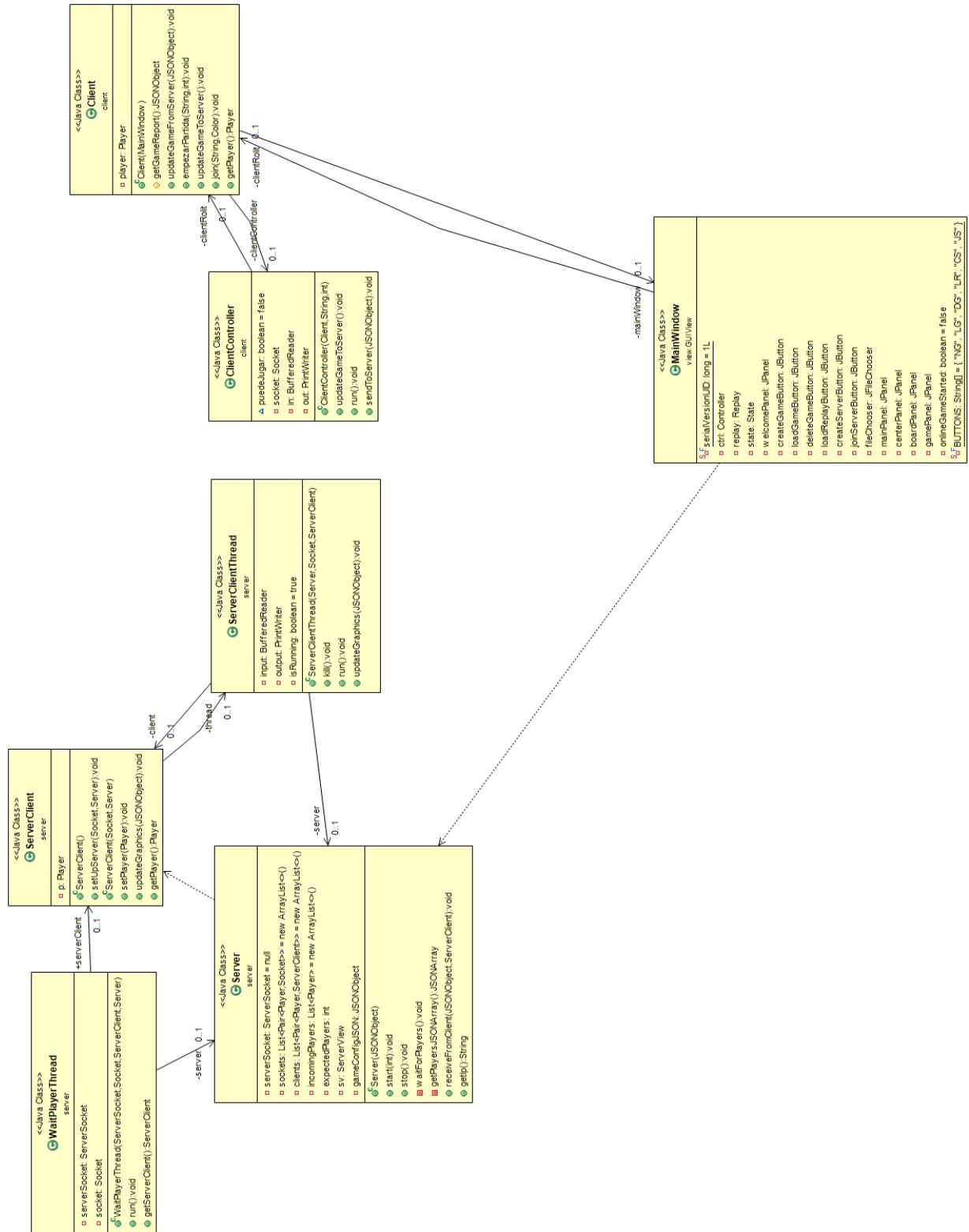
- Reutilización del código del diálogo de crear partida, adaptado a las circunstancias de red.
- Poder jugar a una partida **GameClassic** en red de forma satisfactoria.

Aun así, otros de los objetivos propuestos no son implementados por falta de tiempo y de dependencia con otras de las partes del desarrollo no concluidos. Estos objetivos son:

- Crear más diálogos que aporten *feedback* para la conexión, tanto de la perspectiva del cliente como del servidor.
- Llegar a una versión completamente refactorizada y con métodos simplificados.
- Implementar el juego en red en **GameTeams**.

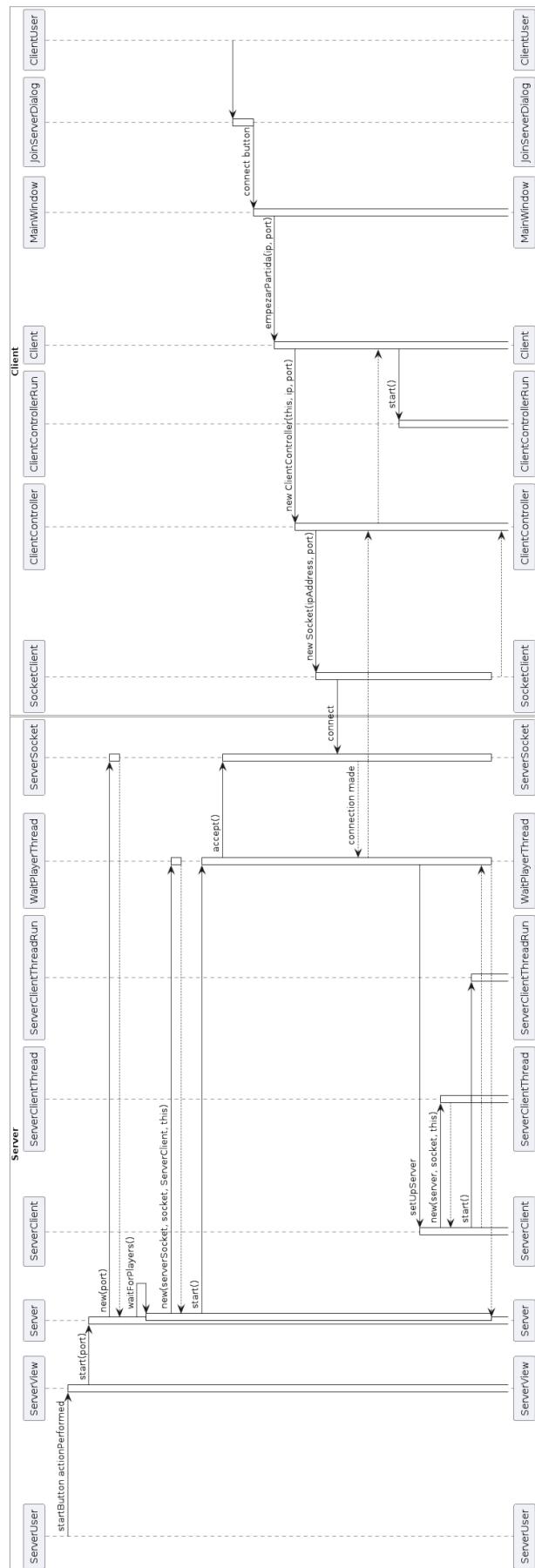
Presentamos los diagramas UML de este Sprint.

El UML de clases es el siguiente:

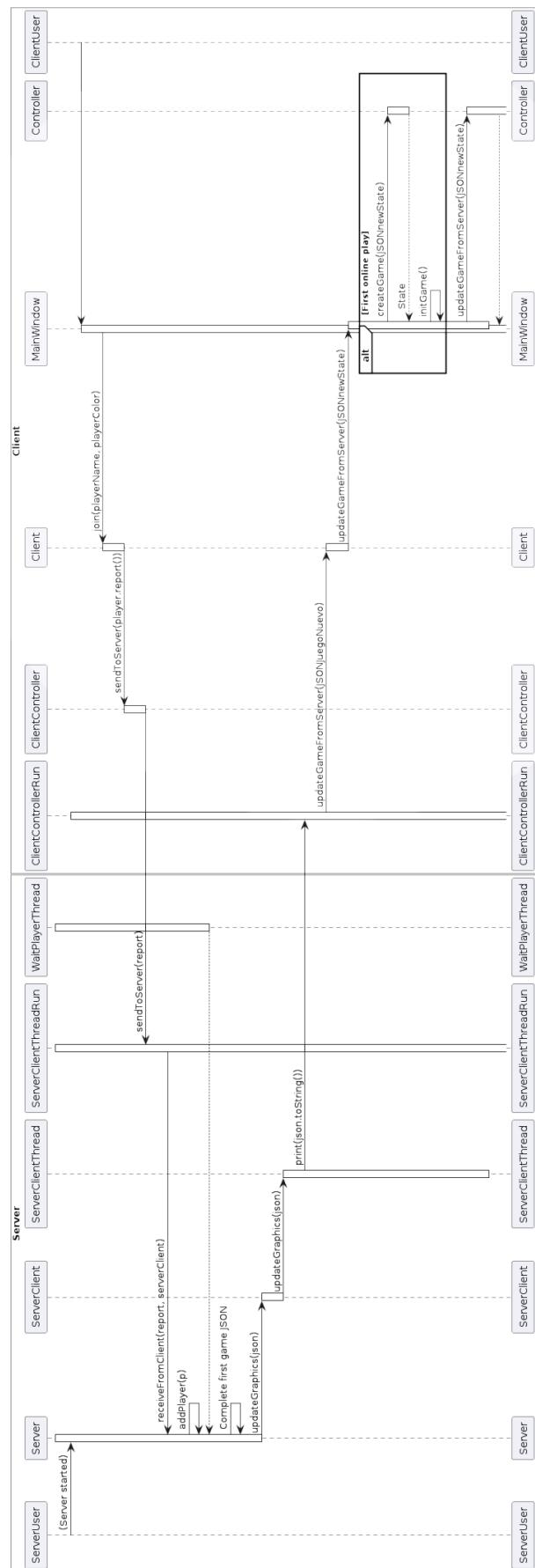


Presentamos los diagramas de secuencia de este Sprint.

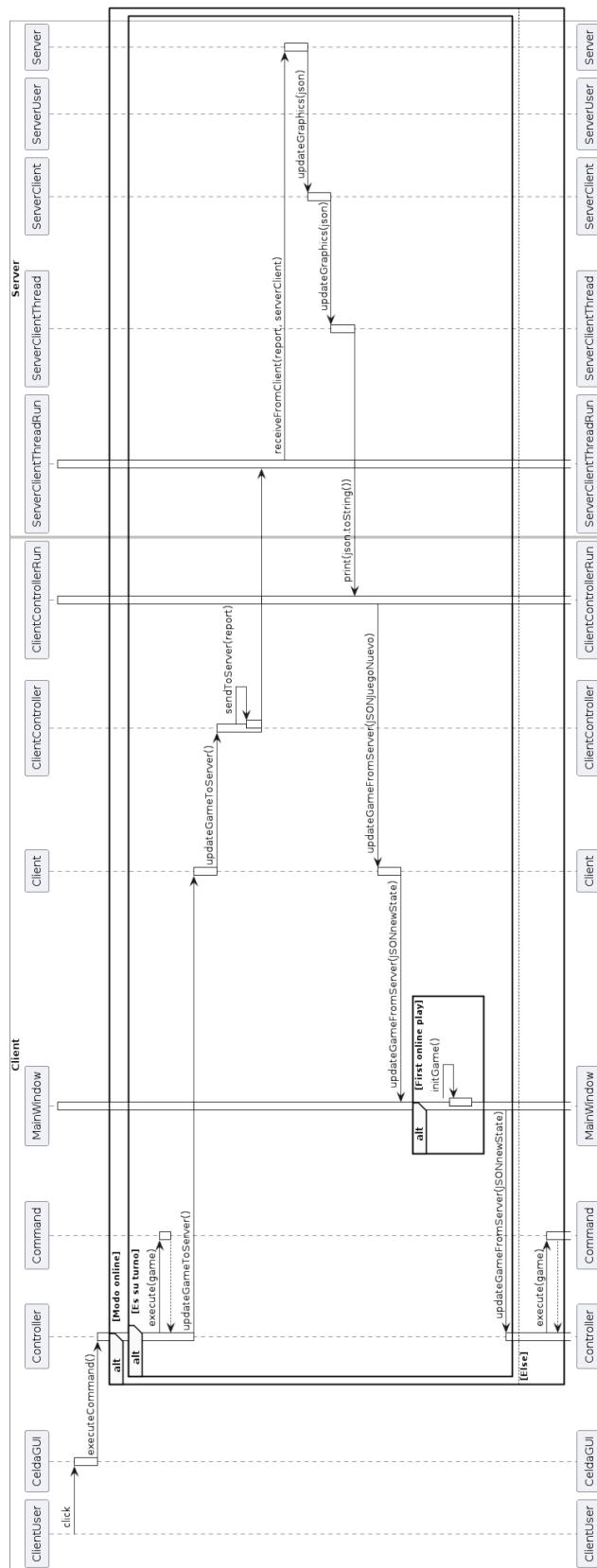
- Cómo se realizan las conexiones



■ Cómo se inicia una partida



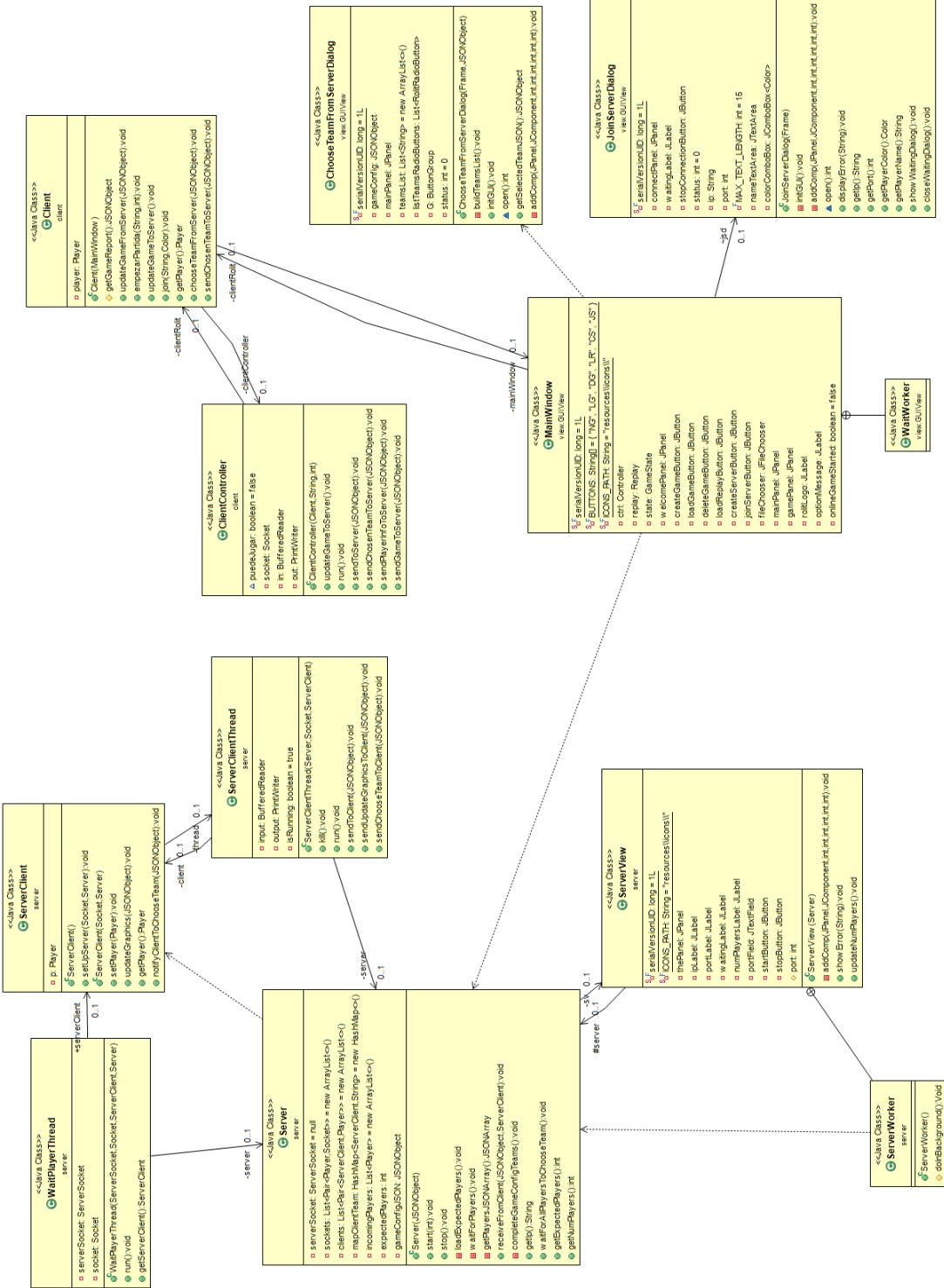
- Cómo se envía y recibe una jugada



Sprint (6)

En cuanto a la red, se alcanzan los objetivos propuestos que quedaron pendientes en el Sprint 5. Las características implementadas son:

- Implementación del modo red para `GameTeams`.
- Creación de diálogos y de código auxiliar para soportar `GameTeams`.
 - Una clase creada para tal efecto es `ChooseTeamFromServerDialog`.
- Perfeccionamiento de la estructura y relaciones de las comunicaciones cliente-servidor y viceversa, incluyendo mecanismos como un campo en el JSON mensaje que especifica qué tipo de notificación constituye el mensaje enviado.
- Adaptación de los diálogos GUI de red al nuevo diseño de la interfaz gráfica introducido en este Sprint.
- Se introduce un `ServerWorker` que soluciona un *bug* complejo que afectaba a los *threads* de Red y Swing.
- Simplificación y refactorización del código de Red.



Por último las sucesivas refactorizaciones pugnadas desde otras partes del código (para, por ejemplo soportar la IA) precisan de un debug extensivo en el modo Red. Este debug es satisfactorio.

Los nuevos diagramas de secuencia, así como los comentarios acerca de las implementaciones de las nuevas funcionalidades se describen en el séptimo Sprint, pues en aquél se refinan ligeramente estas.

Las clases cambiadas, desde la perspectiva del servidor, son las siguientes:

- **ServerView**

En vez de llamar a `server.start()`, se invoca un `SwingWorker`, llamado `ServerWorker`, que se encarga de llamar a `server.start()`; esto arregla un bug de solapamiento del *thread* del servidor y el de Swing.

- **Server**

Se modifica el método `receiveFromClient()` para que se pueda examinar el tipo de notificación (elegir equipo, actualizar gráficos, recibir información de jugador) y hacer así procesamientos distintos de la información recibida en cada caso.

En el método `waitForPlayers()` se distingue si se juega por equipos o no. Si se juega por equipos, una vez se realicen todas las conexiones de todos los usuarios esperados, se notifica a los mismos que deben coger un equipo, junto con la lista de equipos de los que pueden elegir.

Cuando un jugador envía su equipo, se añade una entrada en el mapa `mapClientTeam`, nuevo atributo el cual adjudica a cada cliente, el equipo que ha escogido.

Se espera hasta que todos los jugadores envíen la elección de equipo (de ahí la introducción del nuevo método `waitForAllPlayersToChooseTeam()`); una vez conseguido, se llama al nuevo método `completeGameConfigTeams()` para llenar el JSON del `gameConfig`.

- **ServerClient**

Añadido `notifyClientToChooseTeam()`; indica al `ServerClientThread` que debe informar al cliente que debe escoger equipo.

- **ServerClientThread**

Añadido `sendUpdateGraphicsToClient()` y `sendChooseTeamToClient()`, que se encargan de pasar el mensaje report del parámetro más un nuevo campo en el JSON, la notificación (`updateGraphics`, `chooseTeam` respectivamente).

Las clases cambiadas, desde la perspectiva del cliente, son las siguientes:

- **ClientController**

Distingue, en su método `run()`, de las notificaciones `updateGraphics` o `chooseTeam`; en función de eso, llama a `updateGameFromServer` o `chooseTeamFromServer` (de `Client`) respectivamente.

Además, incorpora los métodos `sendChosenTeamToServer()`, `sendPlayerInfoToServer()`, `sendGameToServer()`, que se encargan de pasar el mensaje report del parámetro más un nuevo campo en el JSON, la notificación (`chooseTeam`, `playerInfo`, `updateGraphics` respectivamente).

- Client

Client incorpora los métodos `chooseTeamFromServer()` y `sendChosenTeamToServer()`, ambos sirven de pasar el mensaje del parámetro de `ClientController` a `MainWindow` y viceversa respectivamente.

Sprint (7)

En cuanto a la red, si bien está conclusa llegados a este Sprint, el debug realizado en otras zonas del código influyen directamente en la funcionalidad de Red. Se lleva a cabo un debug rápido pero extensivo para verificar que el modo de juego en red no ha sido afectado, llevándose a cabo con éxito.

Por último, se procede a crear y solventar algunas *issues* relativas a la experiencia de usuario jugando en red, conllevando una serie de modificaciones que hacen más intuitiva las instrucciones para realizar la conexión.

En cuanto a la perspectiva del servidor, se modifican las siguientes clases:

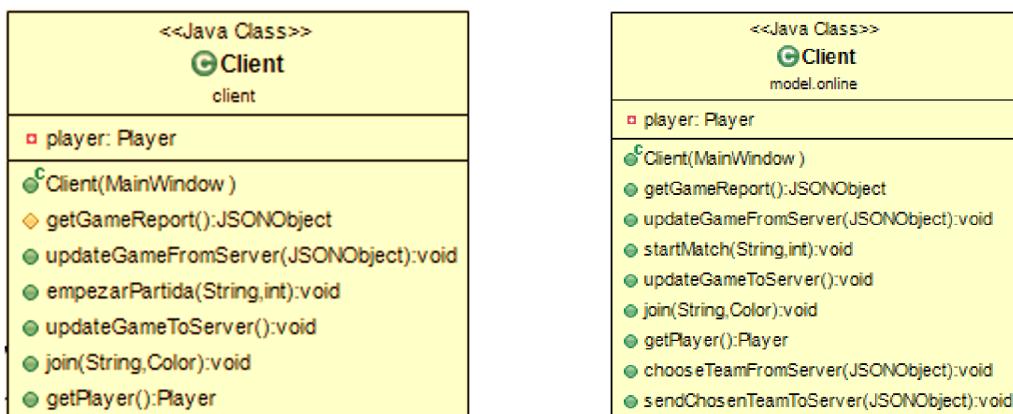
- WaitPlayerThread

Pasa a llamarse, con fines aclaratorios, `ServerWaitPlayerThread`.

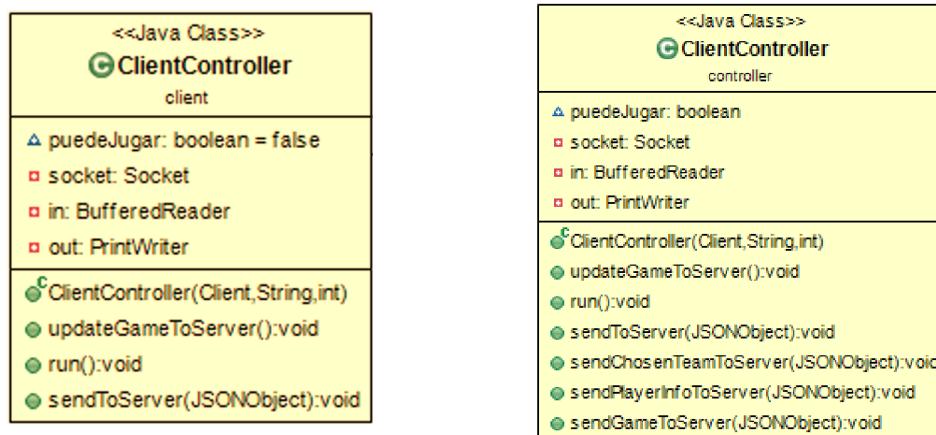
- Server

Refactorización del método `waitForPlayers()`; se crea el método `completeGameConfigSendTo` para este efecto.

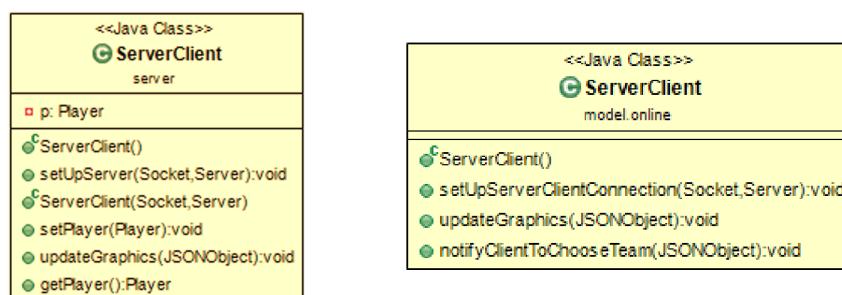
Cambios de importancia para comparar los UML del Sprint 5 con los del Sprint 7 (final) son los siguientes:



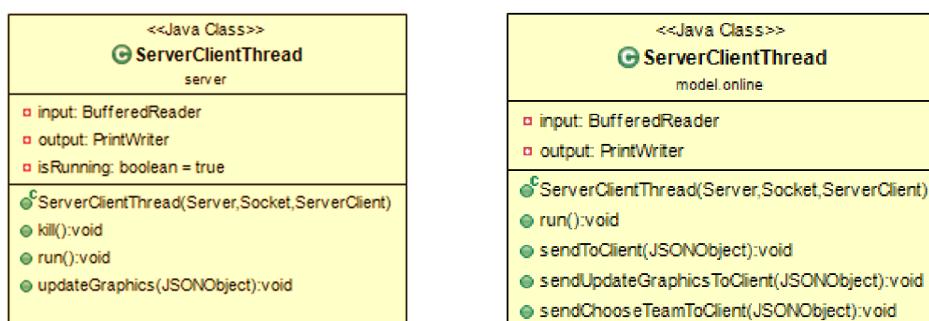
Sprint 5 (izquierda) vs Sprint 7 (derecha)



Sprint 5 (izquierda) vs Sprint 7 (derecha)



Sprint 5 (izquierda) vs Sprint 7 (derecha)



Sprint 5 (izquierda) vs Sprint 7 (derecha)

<pre><<Java Class>> Server server</pre>	<pre>serverSocket: ServerSocket = null sockets: List<Pair<Player,Socket>> = new ArrayList<>() clients: List<Pair<Player,ServerClient>> = new ArrayList<>() incomingPlayers: List<Player> = new ArrayList<>() expectedPlayers: int sv: ServerView gameConfigJSON: JSONObject</pre>
<pre>Server(JSONObject) start(int):void stop():void waitForPlayers():void getPlayersJSONArray():JSONArray receiveFromClient(JSONObject,ServerClient):void getIP():String</pre>	<pre>Server(JSONObject) start(int):void stop():void loadExpectedPlayers():void waitForPlayers():void completeGameConfigAndSendToClients():void completeGameConfigPlayers():void getPlayersJSONArray():JSONArray receiveFromClient(JSONObject,ServerClient):void completeGameConfigTeams():void waitForAllPlayersToChooseTeam():void getExpectedPlayers():int getNumPlayers():int</pre>

Sprint 5 (izquierda) vs Sprint 7 (derecha)

<pre><<Java Class>> WaitPlayerThread server</pre>	<pre>serverSocket: ServerSocket socket: Socket</pre>
<pre>WaitPlayerThread(ServerSocket,Socket,ServerClient,Server) run():void getServerClient():ServerClient</pre>	<pre>serverSocket: ServerSocket socket: Socket</pre>

Sprint 5 (izquierda) vs Sprint 7 (derecha)

El estado final del juego, especificando el UML y el diseño últimos, se encuentra en el apartado de [Diseño final](#).

Como usuario quiero que Rolit introduzca características innovadoras siendo intuitivo y cómodo de jugar

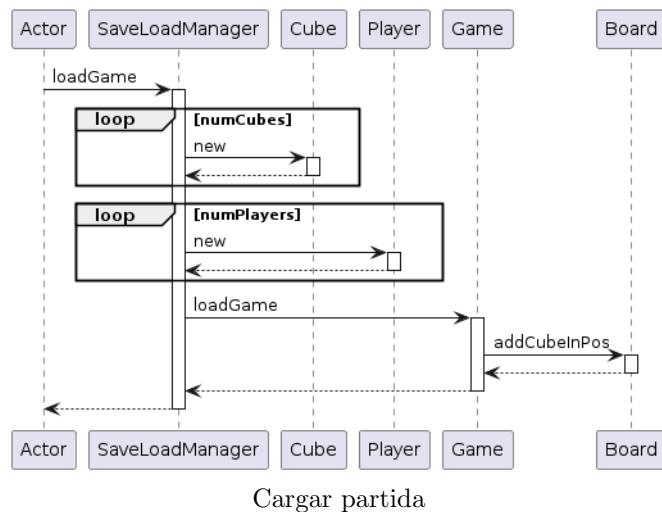
Como usuario, me gustaría que se pudiese guardar y cargar partida para continuar más tarde porque permite poner en pausa el juego

Sprint (1)

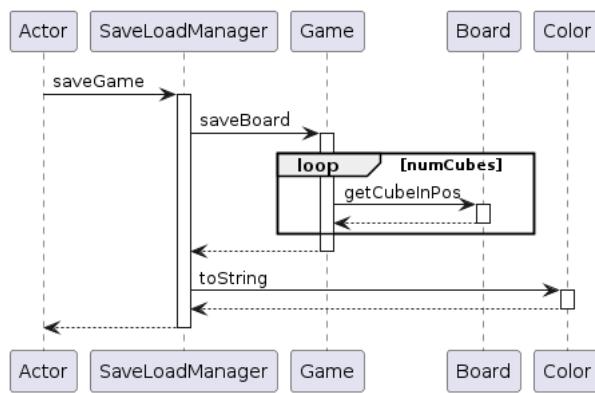
Desde el primer sprint tuvimos clara la necesidad de tener una clase que se encargase de gestionar la comunicación del programa con el exterior para cargar y guardar partidas, el SaveLoadManager.



Como se observa, inicialmente era una clase muy sencilla, pues solo tenía funciones para cargar y guardar un Game.



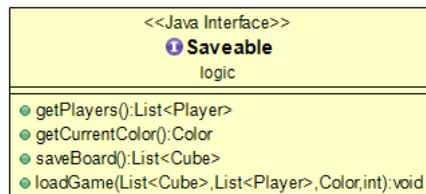
Cargar partida



Guardar partida

Sprint (2)

En la versión anterior del `SaveLoadManager`, la clase trabajaba a nivel de `Game`, lo cual no tenía mucho sentido desde el punto de vista de la programación orientada objetos, pues no es necesario acceder a todos sus métodos. Por esta razón, se creó una interfaz `Saveable` que debía implementar `Game` para poder ser guardada y cargada de en un fichero.



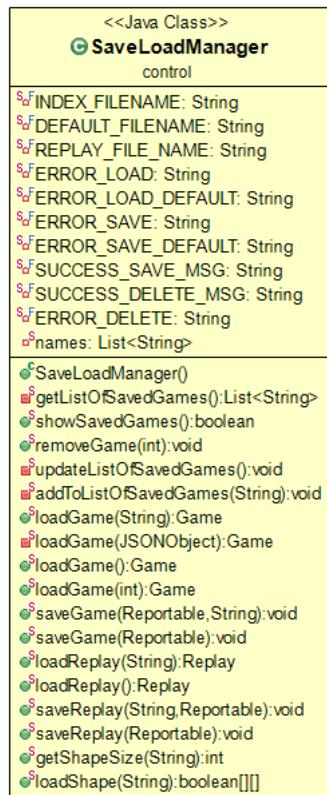
Sprint (3)

Aunque el sprint 2 supuso un cambio positivo, seguía sin ser lo que buscábamos para el `SaveLoadManager` en nuestro proyecto. Si bien es cierto que la interfaz `Saveable` restringía los métodos de `Game`, era una interfaz muy concreta y que solo se podía implementar en dicha clase.

La adquisición de nuevos conocimientos nos permitió tomar una decisión de diseño que marcaría el desarrollo de la aplicación de aquí en adelante. Desde este sprint toda la comunicación con el exterior se realizaría mediante `JSONObject`s.



De esta forma, la interfaz `Saveable` fue reemplazada por `Reportable`, que se podía implementar en cualquier clase del proyecto y que contaba con un único método `report()`, que devuelve una serialización del objeto en formato `JSON`. Además, se creó un documento en el que se especificaban todos los reports de cada clase. Puede accederse al documento actual de reports haciendo click aquí.



Esto supuso una refactorización completa del `SaveLoadManager` y, aunque se mantuvo la esencia de los métodos que contenía fueron, todos fueron reimplementados completamente.

Sprint (4)

Las introducción de formas para los tableros obligó a cambiar los reports y a implementar algunos métodos extras en el `SaveLoadManager` para poder mantener esta funcionalidad.

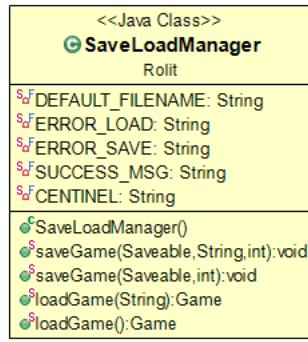
Sprint (6)

La similaridad entre los métodos para guardar partidas y *replays* en el `SaveLoadManager` dio lugar a una pequeña generalización de los métodos privados que permitiese reutilizarlos para guardar cualquier tipo de archivo.

Como usuario, me gustaría poder guardar y cargar distintas partidas, eligiendo el nombre del fichero donde se cargan/guardan

Sprint (2)

La refactorización llevada a cabo en el `SaveLoadManager` incluyó la libre elección del nombre del fichero que contiene la partida guardada, así como su ruta.



Como usuario, me gustaría que se pudiera guardar repeticiones de partida para poder revisarlas más tarde

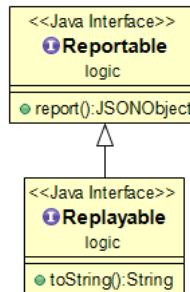
Sprint (3)

Durante el sprint 3 comenzó el desarrollo de esta funcionalidad, que de ahora en adelante denominaremos como *replays*. Para su implementación, se planteó abstraer la clase Game mediante estados que representen cada uno de los momentos que atraviesa el juego a lo largo de una partida. Así, mediante un conjunto de estados es posible replicar una partida al completo.

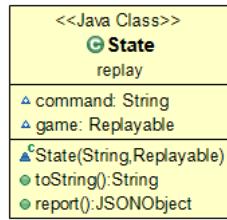
Inicialmente se planteó que los estados almacenaran únicamente el cubo que se añade en su turno, pero esta idea fue descartada debido a que la colocación de un cubo puede llegar a afectar a cualquier parte del tablero, teniendo que incluir la lógica correspondiente para calcular los cambios. Por ello, se decidió que cada estado guardase una copia de Game en el momento deseado.

Sin embargo, durante una *replay* no se debería de poder alterar el juego, haciendo que carezca de sentido que los estados tengan acceso a los métodos de la clase Game, pues el único objetivo es mostrar una información inmodificable al usuario.

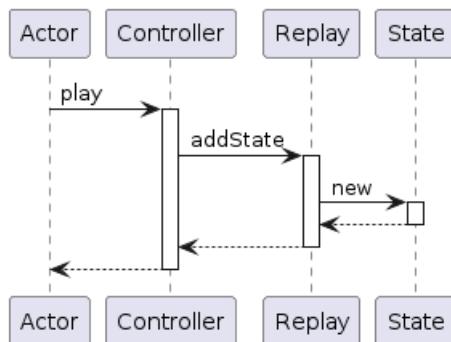
De esta manera surge la interfaz **Replayable**, que extiende de la interfaz **Reportable** y que contiene dos métodos: `toString()`, para facilitar la visualización en la vista de consola, y `report()`, para poder almacenar los estados en formato JSON.



Una vez ideada esta interfaz, ya es posible definir la clase **State**, que representa los estados que hemos estandarizado hasta ahora y que, además, cuenta con el comando que la generó.

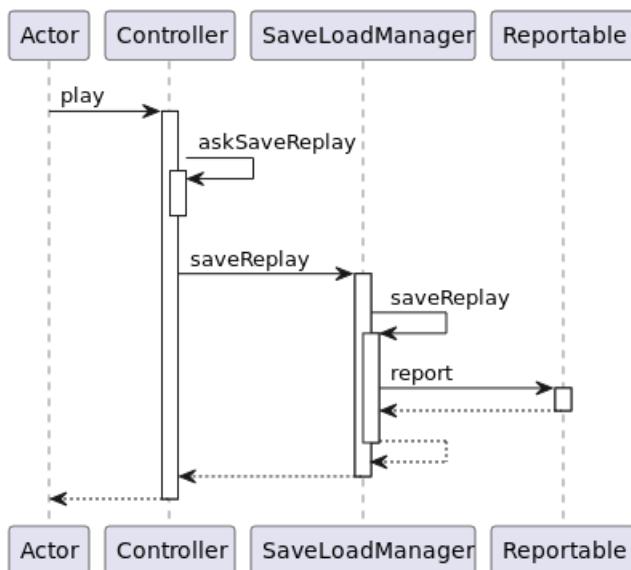


El siguiente paso fue la creación de una clase contenedora de los estados, con la responsabilidad añadida de saber gestionarlos, la clase `Replay`. Esta clase contiene una lista de `State` que se completa durante una partida.



Generación de lista de estados

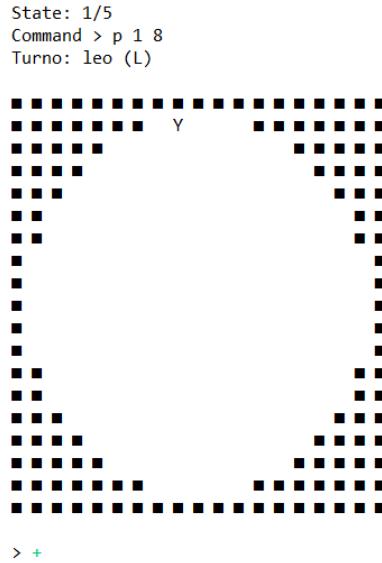
Cuando la partida acaba o el usuario hace que acabe (mediante el uso del comando `exit`), se pregunta si desea que se guarde la repetición y se procede según su respuesta.



Guardado de Replay

Finalizada la generación de las *replays* era el momento de hacer que pudieran cargarse y ser interpretadas para su correcta visualización. Como ya mencionamos anteriormente, la clase `Replay` es la encargada de gestionar los estados y, por ello, la que contendrá esta lógica.

La función para cargar una *replay* es análoga a la utilizada para guardarla y es gestionada por el **SaveLoadManager**. La ejecución de una repetición comienza llamando al método **startReplay()**, la clase comienza a visualizar el primer estado de su lista y entra en un bucle que permite recorrerla mediante el uso de los símbolos “+” “-”.



Con estas repeticiones de partidas funcionales dimos por completada esta tarea en el Sprint 3.

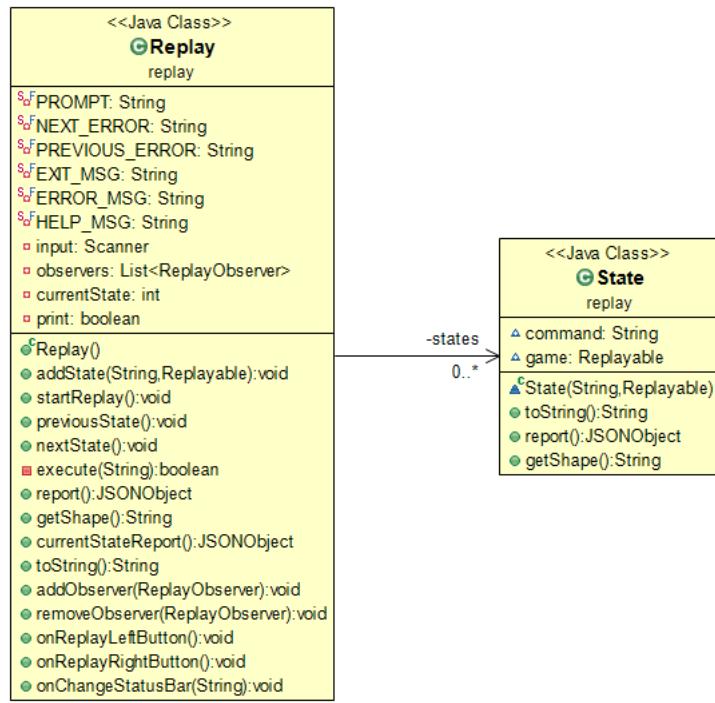
Sprint (4)

Este sprint se caracterizó por la implementación de una GUI funcional, lo que conllevo la adaptación de las *replays* a este nuevo formato. Decidimos que la comunicación entre modelo y vista se lleva a cabo mediante el uso del patrón observador, lo cual supuso un reto inicial para las repeticiones de partidas.

Basta detenerse a pensar unos minutos para darse cuenta que, cuando cargamos, procesamos y visualizamos una *replay* desde un fichero, la clase **Game** no interviene en el proceso, pues es la clase **Replay** la encargada de gestionar toda la lógica de las repeticiones. Por tanto, la clase **Replay** es también un modelo que debe ser observado.



Para implementar los observadores se creó la interfaz **ReplayObserver** que tiene métodos para notificar cuando se avanza a la izquierda, a la derecha y para mostrar mensajes en la barra de estado. Además, se añadieron algunos getters necesarios a las clases **Replay** y **State**.



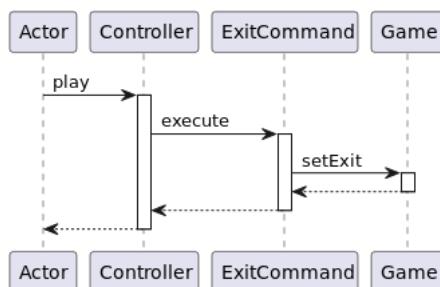
Sprint (5)

La GUI sufrió una refactorización, siendo necesario para ello la creación nuevos getters. Además, la refactorización de la clase **Controller** trajo consigo problemas con las *replays* que no serían solucionados en este sprint por falta de tiempo.

Sprint (6)

El cambio de paradigma en el funcionamiento mediante la introducción de su propio hilo y el nuevo controlador del sprint anterior hicieron que fuese necesario modificar cómo se guardan los estados en la clase **Replay**.

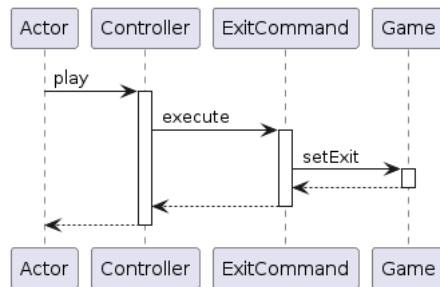
Además, la necesidad de notificar a la vista con el **Replay** para que sea guardada en caso de que así lo desee el usuario, tiene como consecuencia que la clase **Game** sea la encargada de generar sus propias repeticiones. Este proceso puede verse reflejado en el siguiente diagrama de secuencias.



Como usuario, me gustaría poder salir del juego en cualquier momento

Sprint (2)

Con la introducción de los comandos vino acompañado el comando exit, que puede ser ejecutado en cualquier momento durante la partida para detener la ejecución del juego.

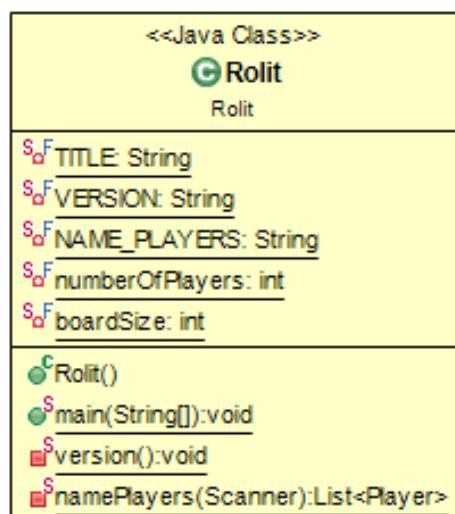


En sprints posteriores este diagrama dejará de ser válido debido a refactorizaciones en el controlador, aunque la clase ExitCommand ha permanecido invariante. Los cambios sufridos para este comando son análogos a los del resto, que pueden ser consultados en [Commands](#).

Como usuario, me gustaría que el número de jugadores fuese variable porque así se adapta a diferentes grupos de personas

Sprint (1)

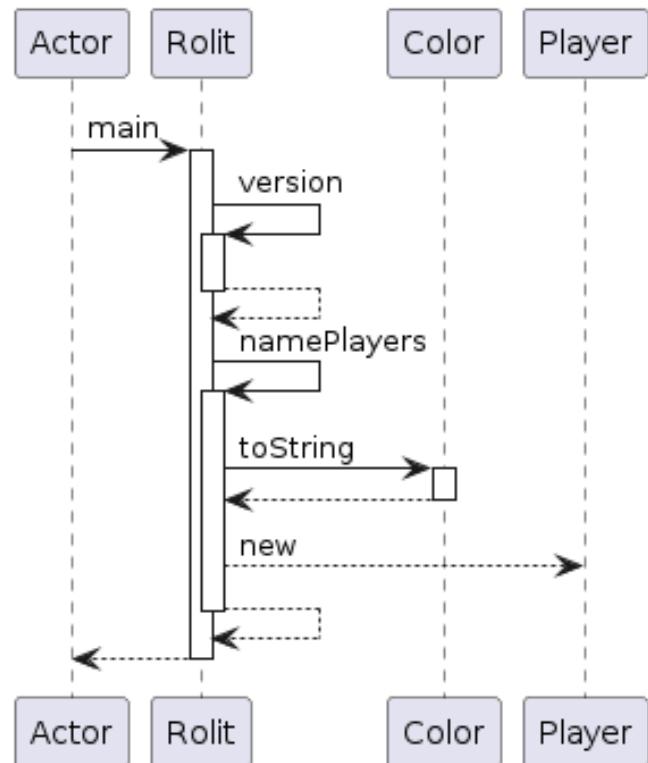
En este primer Sprint el número de jugadores es fijo. Este número se guarda como atributo de la clase Rolit, cuyo estado en este momento es el siguiente:



Para la creación de estos jugadores, los colores también están fijados, siendo estos amarillo, rojo, verde y azul, asignados a los jugadores en ese orden. Por tanto, en

este momento, de cara a la creación de los jugadores (y de la partida en general) la entrada se limita a leer el nombre de los jugadores, que se hace directamente en el método main de Rolit.

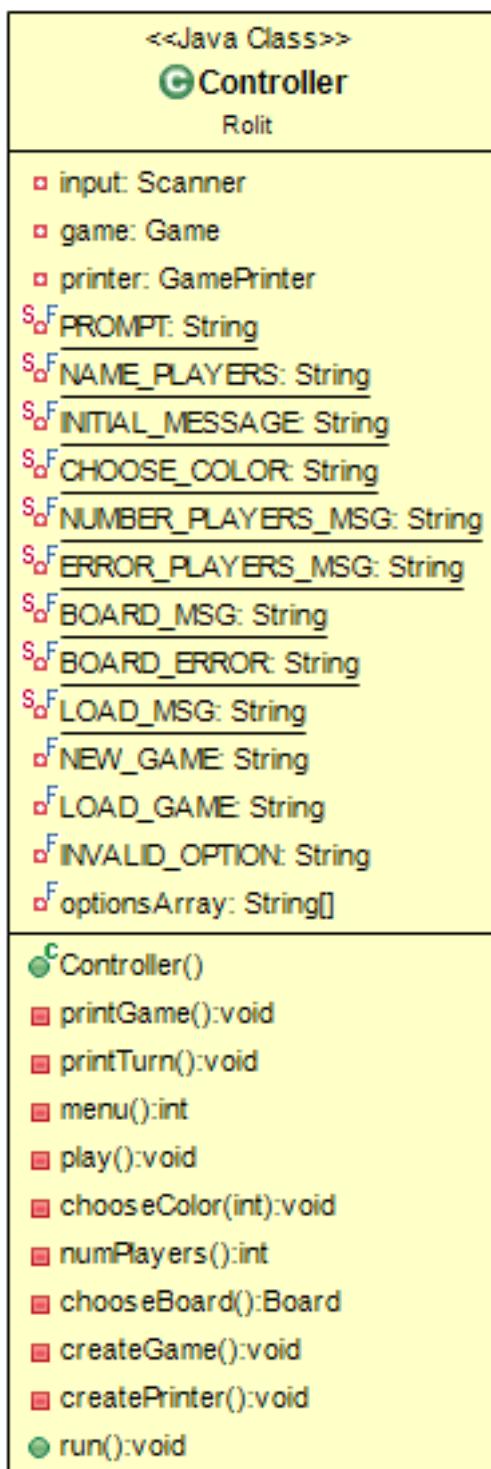
El flujo de creación de los jugadores actualmente se refleja en el siguiente diagrama:



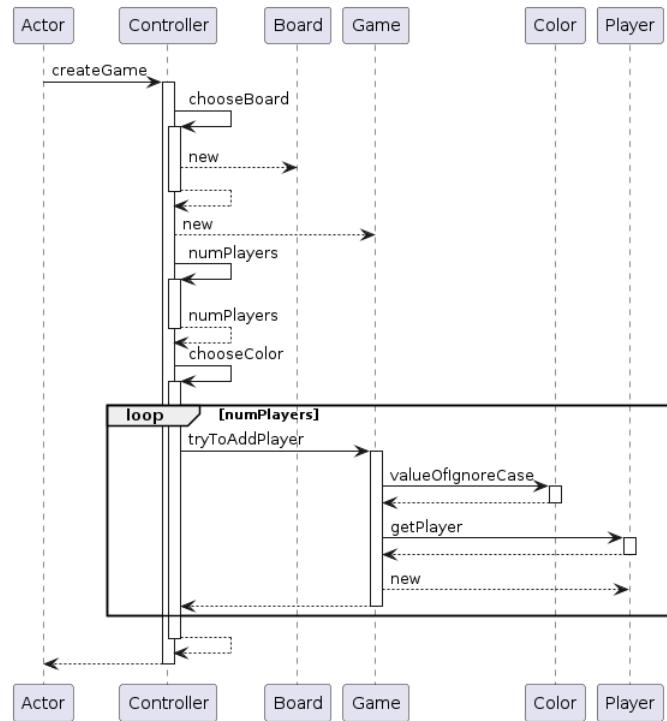
Sprint (2)

En este Sprint ya se da la opción al usuario de elegir el número de jugadores que van a participar en la partida. Aparte, ahora se delega la tarea de lectura de la entrada para la creación de la partida a la clase `Controller` en su método `createGame()`.

En este momento, la clase `Controller` se haya en el siguiente estado:



El flujo de dicha creación es la siguiente:



A parte, dentro del bucle con `tryToAddPlayer` vemos que se da la opción al jugador de elegir color. Los colores disponibles son: Amarillo, rojo, verde, azul, naranja, rosa, morado, negro, marrón y beige.

Para conseguir esto, la representación interna de los colores se hace a través de la clase enumerada `Color`, cuyo estado en este momento es el siguiente:



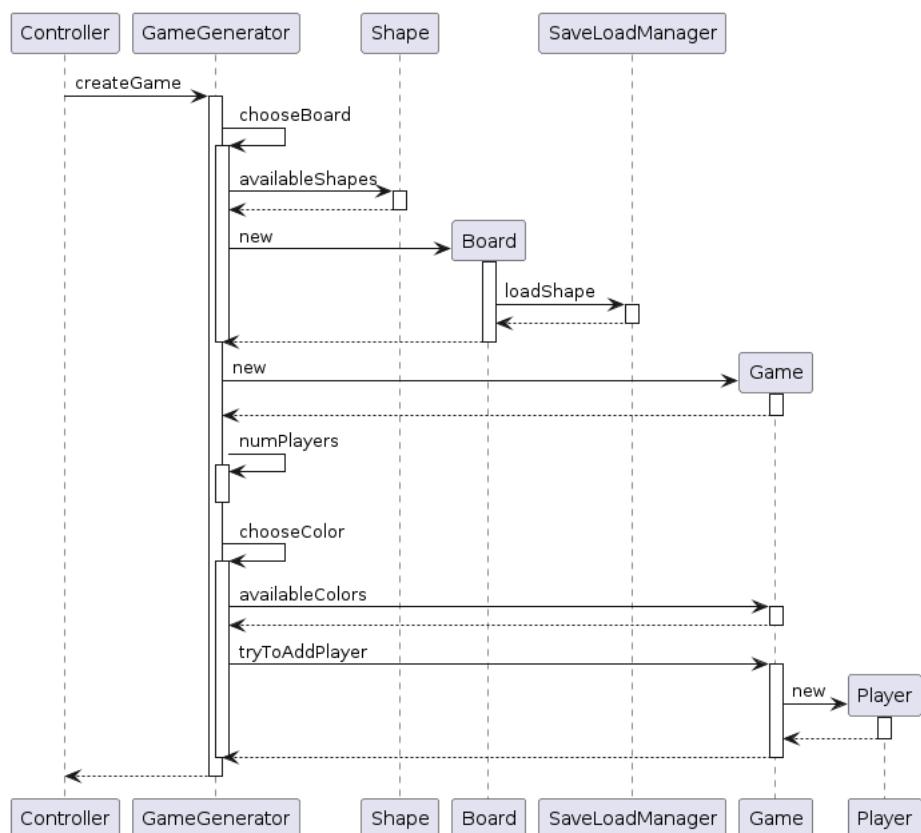
3. Diseño y evolución de otros aspectos del proyecto

Builders

Hasta que el proyecto no aumentó lo suficiente de tamaño y hasta que no se estudiaron los patrones adecuados, la necesidad de tener unas clases específicas especializadas en la creación de ciertos objetos no tuvo relevancia en el proyecto (pues no surgió la necesidad o no teníamos los conocimientos para cubrirla).

Sprint (3)

El flujo de creación del juego quedaría como:



El aumento de tamaño del `Game` y, con mucha más relevancia, la complejidad de la creación del mismo que estaba adquiriendo dicha clase hizo que fuese necesario

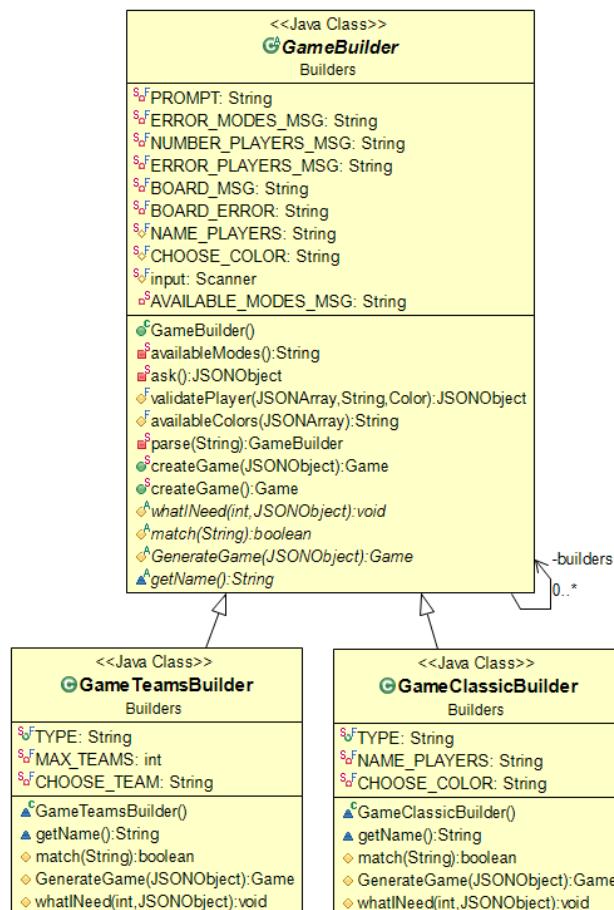
encapsular la lógica de creación de este objeto más allá de un simple constructor, pues no era responsabilidad del controlador o las clases que iniciaran el juego de traducir los datos que el usuario quería del mismo a una instancia concreta de `Game`, pero tampoco podíamos hacer una cantidad ingente de constructores para cada posible situación de juego.

En base a esta necesidad, y con vistas a que en el futuro iba a haber más tipos de `Game` a los que jugar, se crea por primera vez la clase `GameGenerator`. Esta clase, debido a la falta de una implementación del *Modelo-Vista-Controlador* útil y funcional, se encarga no sólo de traducir los datos que el usuario introduce por pantalla para generar el juego sino de preguntar directamente por consola cuáles son las características del juego que se va generar.

Es decir, esta clase constituye una aproximación al patrón de Factorías, pues ahora `Controller` solo debe encargarse de pedirle a la factoría que genere el juego y después hacer su verdadera función: controlar los eventos del juego.

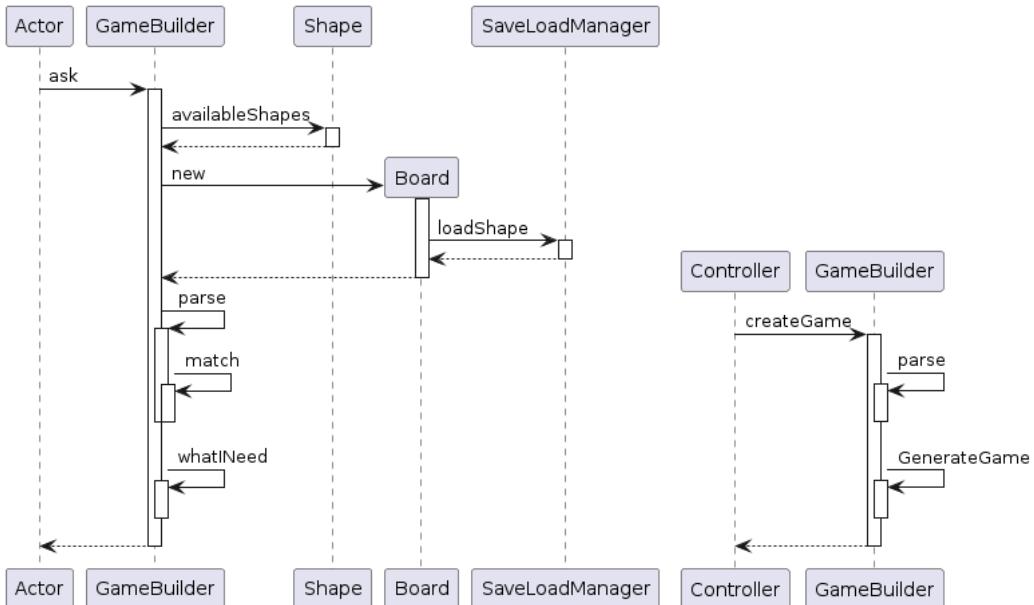
Sprint (4)

La aparición de nuevos tipos de `Game` complica, más si cabe, la creación de los mismos para comenzar la partida. Ya no sólo el usuario debe introducir características que desea del juego sino que tiene que introducir cosas distintas en función del tipo de juego que desee jugar. Esta nueva funcionalidad complica mucho las cosas porque, tal y como estaba implementado en el sprint anterior, tendríamos que saber generar una factoría distinta para cada tipo de juego y que dicha factoría generase el juego.



Es en este punto donde implementamos los patrones de *Abstract Factory* y *Builder*. Generamos un nuevo paquete y unas nuevas clases que van a sustituir a la clase **GameGenerator** del sprint anterior y que principalmente van a tener dos familias de métodos: un grupo de métodos para **preguntar** y recopilar la información que el usuario teclee sobre el juego y otra familia para **construir** el juego una vez se tiene la información.

La clase principal y la cual es la encargada de la generación de los **Game** como tipo abstracto es la clase **GameBuilder**. Dicha clase pregunta **los elementos globales a cualquier juego** por pantalla a través del método **ask()** y después indica al constructor concreto del juego concreto que se quiere jugar que pregunte lo que considere necesario sobre su juego a través de su método **whatIneed()**. Todo este proceso genera un conglomerado de datos **en forma de JSONObject** que se utiliza como receta para el método de construcción del juego. Dicho método funciona exactamente igual a los procesos anteriores: la factoría general **GameBuilder** construye a partir del **JSONObject** todo lo que un juego genérico tendría y después indica al Builder concreto que termine de construir el juego con el mismo **JSONObject**.



La parte de preguntar podría omitirse y tener un método general de crear el juego recibiendo un **JSONObject** y otro sin **JSONObject**, sin embargo el cuello de botella de sólo poder llamar al método **Game createGame()** de **GameBuilder** es una característica fundamental del diseño que se va a tomar posteriormente. En este caso, los Builder preguntaban porque aún no había una interfaz gráfica de consola en condiciones que pudiese encargarse de un manejo adecuado de entrada salida, pero con la decisión anterior el mensaje es tajante: todo aquel que desee generar un juego debe pasarle la información bien estructurada a las factorías. Y este detalle es fundamental, pues permite que la forma de obtener la información no sea relevante, es decir, la información del juego puede ser recopilada desde un fichero de carga, por consola, por GUI o por red y que el método para crear el juego siempre sea el mismo, pues sólo se necesita saber la información en formato **JSONObject**.

Sprint (5)

La aparición de una interfaz de consola adecuada posibilitó que los métodos `JSONObject ask()` y `void WhatIneed(JSONObject o)` salieran de las factorías y se incorporaran a la vista de consola correspondiente. De este modo, ahora simplemente los Builder mantienen su funcionalidad de contener la lógica de creación del juego a partir del saco de datos que suponen los `JSONObject` de cada juego.

Sprint (6)

Para este sprint los Builder simplemente cambian la forma instanciar el tipo `Player` que se añade al `Game` porque cambia su estructura interna (ahora tenemos estrategias para jugadores automáticos y forman parte de la estructura interna del `Player`).

Commands

La necesidad de tener un paquete de comandos con clases específicas especializadas en convertir los comandos en objetos surgió en el Sprint 2 y se mantuvo hasta el final del proyecto.

Sprint (2)

Del aumento del tamaño de la clase `Game` surgió la necesidad de crear un paquete con los comandos necesarios para mantener la encapsulación y conseguir que los comandos funcionasen como objetos independientes, todos ellos extendiendo a la clase abstracta `Command`. Se utiliza así el patrón de diseño `Command`.

La clase abstracta `Command` tiene el constructor común a todas sus subclases (así como otros métodos), una lista con los comandos disponibles y un método `execute()` en el cual cada subclase implementará la ejecución del comando. En este Sprint se crean las subclases:

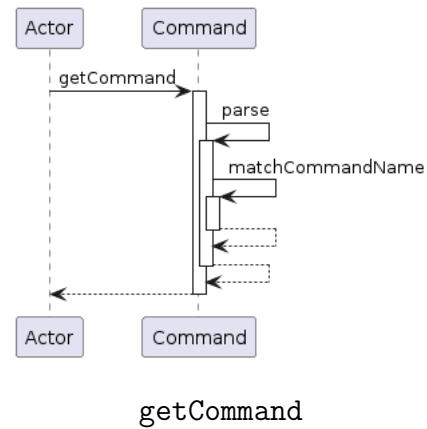
`ExitCommand`: se utiliza para salir del juego

`HelpCommand`: se utiliza para mostrar una lista de los comandos disponibles para el jugador

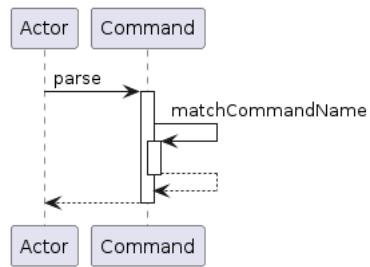
`PlaceCubeCommand`: se utiliza para poner un cubo en una posición del tablero (teniendo en cuenta las posiciones disponibles)

`SaveCommand`: se utiliza para guardar una partida.

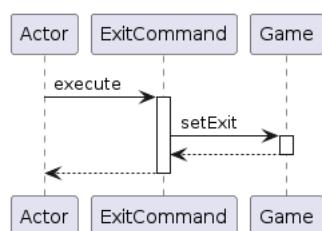
Los diagramas de secuencia UML que muestran procesos clave son los siguientes:



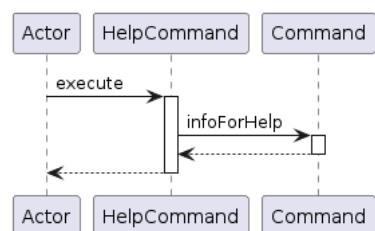
`getCommand`



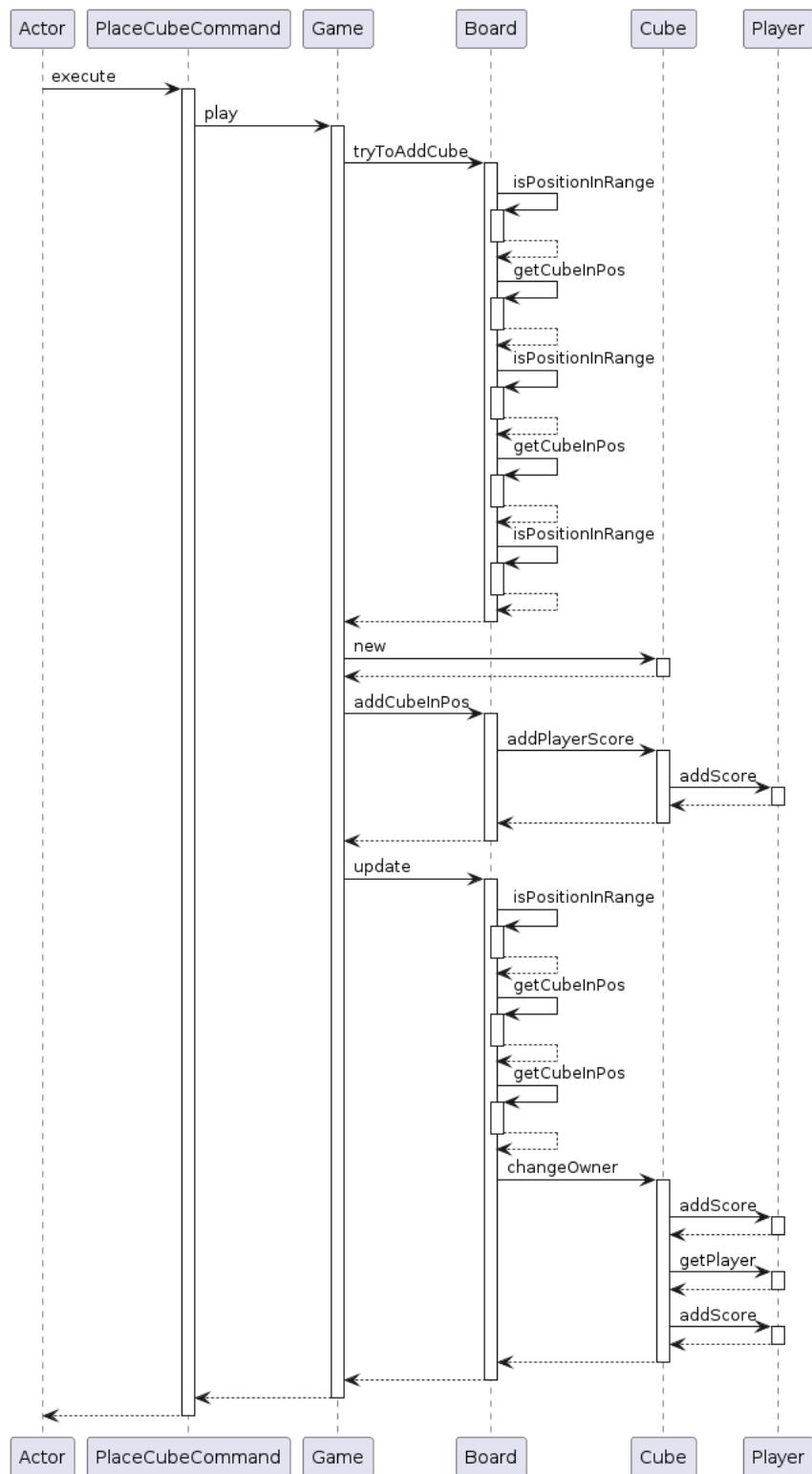
`parse`



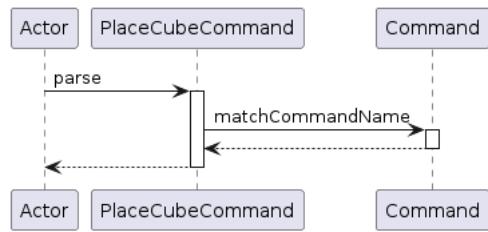
`ExitCommand.execute`



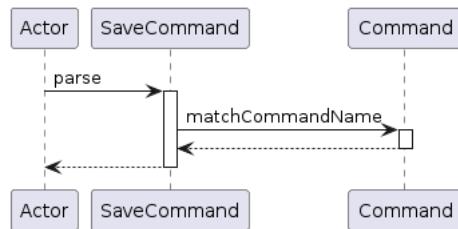
`HelpCommand.execute`



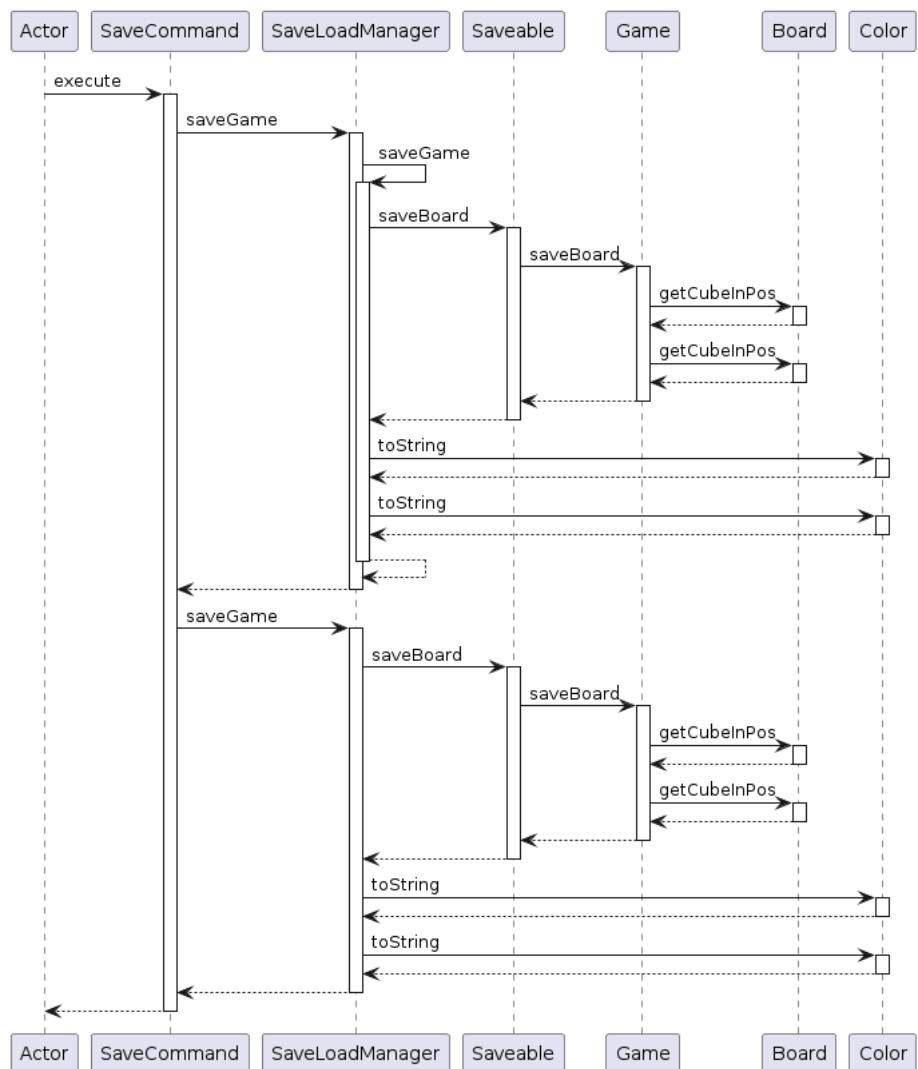
PlaceCubeCommand.execute



`PlaceCubeCommand.parse`



`SaveCommand.parse`



`SaveCommand.execute`

Sprint (3)

En este Sprint los comandos específicos no cambian nada de su implementación ya que los cambios hechos en el proyecto no afectan a este paquete. La clase abstracta `Command` comienza a implementar la interfaz `Replayable` y añade un método `toString()` y un método `report()` como resultado de la adición al proyecto de `JSONObjects`.

Sprint (4)

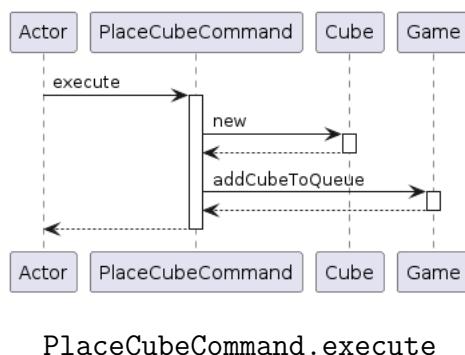
En este Sprint los comandos no cambian nada de su implementación ya que los cambios hechos en el proyecto no afectan a este paquete.

Sprint (5)

En este Sprint los comandos no cambian nada de su implementación ya que los cambios hechos en el proyecto no afectan a este paquete. El único mínimo cambio es la eliminación del mensaje de despedida por consola en el comando `ExitCommand` como resultado de la incorporación de la GUI.

Sprint (6)

En este Sprint cambia la implementación del método `execute()` del comando `PlaceCubeCommand` y pasa a delegar la responsabilidad de poner un cubo en el tablero a `Game`, llamando a un método de esta clase, que se encargará de crear un nuevo cubo y asociar este cubo con el jugador correspondiente.



4. Tests de JUnits

4.1 Paquete Builders

4.1.1 GameClassicBuilderTest

test_1()

Se crea un `GameClassic` por medio de un `inputJson` y se comprueba que el report de dicho game es igual al JSON inicial.

test_2()

Probamos a meter una `Shape` que está mal y comprobamos que se lanza adecuadamente la excepción.

test_3()

Probamos a meter un tipo que no es `GameClassic` y comprobamos que se lanza adecuadamente la excepción.

test_4()

Probamos a meter un `Color` que está mal y comprobamos que se lanza adecuadamente la excepción.

test_5()

Probamos a meter una pos `x` que está mal y comprobamos que se lanza adecuadamente la excepción.

test_6()

Probamos a meter una clave que está mal y comprobamos que se lanza adecuadamente la excepción.

4.1.2 GameTeamsBuilderTest

test_1()

Se crea un `GameTeams` por medio de un `inputJson` y se comprueba que el report de dicho game es igual al JSON inicial.

test_2()

Probamos a meter una coordenada de un `Cube` que está mal y comprobamos que se lanza adecuadamente la excepción.

test_3()

Probamos a meter un tipo que no es `GameTeams` y comprobamos que se lanza adecuadamente la excepción.

test_4()

Probamos a meter un `Color` que está mal y comprobamos que se lanza adecuadamente la excepción.

test_5()

Probamos a meter una clave que está mal y comprobamos que se lanza adecuadamente la excepción.

test_6()

Probamos a meter una `Shape` que está mal y comprobamos que se lanza adecuadamente la excepción.

4.2 Paquete Commands

4.2.1 ExitCommandTest

executeTest() y executeTest2()

Creamos un `Board`, dos `Player` y un `Cube` para cada `Player`. Creamos una lista con los `cubos` y otra con los `Player`. Creamos un `GameClassic` con la información necesaria y creamos un `ExitCommand` para después llamar a su método `execute()` y comprobar que el método `exited()` del `Game` devuelve `true`.

4.2.2 PlaceCubeCommandTest

parseTest()

Creamos un `PlaceCubeCommand` y comprobamos que funciona correctamente su método `parse()` viendo que lanza una excepción cuando los argumentos que se le pasan no son correctos.

4.2.3 SaveCommandTest

parseTest()

Creamos un `SaveCommand` y comprobamos que funciona correctamente su método `parse()` viendo que lanza una excepción cuando los argumentos que se le pasan no son correctos.

exceptionTest()

Comprobamos que se lanza una excepción cuando se intenta llamar al método `getCommand()` con argumentos incorrectos.

4.3 Paquete IA

4.3.1 IATest

IA_test1()

Se crea un juego con un `Player` que es una persona y un `Player` que es una inteligencia artificial, justo le toca poner el último `Cube` al `Player`. Comprobamos que no salta excepción, es decir, que la IA no intenta colocar un `Cube` cuando el `Board` ya está lleno.

IA_test2()

Se crea un juego con un `Player` que es una persona y dos `Player` que son inteligencias artificiales, ambas `Minimax`. Quedan dos huecos en el `Board`, el próximo turno es de la persona y el último de una de las IAs. Comprobamos que la última IA no intenta poner un `Cube` cuando el `Board` ya está lleno, ya que si esto sucediese saltaría una excepción.

IA_test3()

Se crea un juego con tres `Player` que son inteligencias artificiales, una `Minimax`, otra `Random` y la última `Greedy`. Las ponemos a jugar solas y comprobamos que no se produce ningún error, es decir, no se lanza ninguna excepción.

4.4 Paquete Logic

4.4.1 BoardTest

addCubeInPos()

Creamos un `Board`, un `Player` y cuatro `Cube` del `Player` creado. Se añaden dos de ellos en posiciones correctas y se comprueba que se lanzan excepciones al intentar colocarlos en posiciones inválidas del tablero, ya sea porque ya están ocupadas con otro `Cube` o porque las coordenadas x e y no son correctas. Finalmente, se comprueba si el `Cube` que se ha añadido en la posición correcta es el adecuado.

Board_full_numCubes()

Creamos un `Board`, un `Player` y un `Cube`, añadimos el `Cube` al `Board` y comprobamos que el método que comprueba si el `Board` está lleno devuelve `false`.

Update_test()

Se crea un `Board` y dos `Player`, se van creando y poniendo en el `Board` objetos `Cube` y llamando al método `update()` de `Board` y finalmente se comprueba si el `Color`

del **Cube** al final del test es el que debería ser por las posiciones en las que se han ido colocando los mismos.

Update_test2()

Se crea un **Board** y dos **Player**, y se van creando y colocando en el tablero objetos **Cube** y llamando al método **update()** de **Board**. Finalmente, se comprueba si se lanza una excepción al intentar añadir un **Cube** en una posición no válida, así como si el **Color** del **Cube** al final del test es el que debería.

Test_report() y test_report2()

Creamos un **Board**, dos **Player** y varios **Cube**, los cuales se van colocando en el tablero y llamando al método **update()** de **Board**. Finalmente, se comprueba si el método **report()** de la clase **Board** devuelve el **JSONObject** esperado.

4.4.2 CubeTest

Change_owner()

Creamos un **Player** y un **Cube** y tratamos de cambiar el "propietario" (realmente es el color) del **Cube**. Finalmente se comprueba si se ha cambiado correctamente.

Test_report()

Creamos un **Player** y un **Cube**. Finalmente, se comprueba si el método **report()** e la clase **Cube** devuelve el **JSONObject** esperado.

4.4.3 GameClassicTest

play_test()

Creamos todos los objetos necesarios para poder instanciar un **GameClassic**. Tratamos de poner algunos **Cube** en posiciones incorrectas y comprobamos que salte una excepción. El primer **Cube** tratará de ocupar una posición que no está libre y el segundo una posición que no tiene ninguna adyacente ocupada. Por último trata de posicionar un **Cube** correctamente, comprobamos que el **Color** esperado y el del **Cube** de dicha posición coinciden.

test_report()

Creamos todos los objetos necesarios para poder instanciar un **GameClassic**. Usamos el método **play()** para dar un poco de actividad y finalmente comprobamos que el método **report()** nos proporciona el **JSONObject** esperado.

play_test2()

Creamos todos los objetos necesarios para poder instanciar un **GameClassic**. Tratamos de poner algunos **Cube** en posiciones incorrectas y comprobamos que salte una excepción. El primer **Cube** tratará de ocupar una posición que no tiene ninguna adyacente ocupada y el segundo una que no pertenece al **Board**. Por último trata de posicionar un **Cube** correctamente, comprobamos que el **Color** esperado y el del **Cube** de dicha posición coinciden.

test_report2()

Creamos todos los objetos necesarios para poder instanciar un `GameClassic`. Usamos el método `play()` para dar un poco de actividad y finalmente comprobamos que el método `report()` nos proporciona el `JSONObject` esperado.

play_test3()

Creamos todos los objetos necesarios para poder instanciar un `GameClassic`. Tratamos de poner algunos `Cube` en posiciones incorrectas y comprobamos que salte una excepción. El primer `Cube` tratará de ocupar una posición que no está libre y el segundo una que no pertenece al `Board`. Por último trata de posicionar un `Cube` correctamente, comprobamos que el `Color` esperado y el del `Cube` de dicha posición coinciden.

test_report3()

Creamos todos los objetos necesarios para poder instanciar un `GameClassic`. Usamos el método `play()` para dar un poco de actividad y finalmente comprobamos que el método `report()` nos proporciona el `JSONObject` esperado.

4.4.4 GameTeamsTest

play_test()

Creamos todos los objetos necesarios para poder instanciar un `GameTeamsTest`. Tratamos de poner algunos `Cube` en posiciones incorrectas y comprobamos que salte una excepción. El primer `Cube` tratará de ocupar una posición que no está libre y el segundo una posición que no existe. Por último trata de posicionar un `Cube` correctamente, comprobamos que el `Color` esperado y el del `Cube` de dicha posición coinciden.

test_report()

Creamos todos los objetos necesarios para poder instanciar un `GameTeamsTest`. Usamos el método `play()` para dar un poco de actividad y finalmente comprobamos que el método `report()` nos proporciona el `JSONObject` esperado.

play_test2()

Creamos todos los objetos necesarios para poder instanciar un `GameTeamsTest`. Tratamos de poner algunos `Cube` en posiciones incorrectas y comprobamos que salte una excepción. El primer `Cube` tratará de ocupar una posición que no está libre y el segundo una posición que no tiene ninguna posición adyacente ocupada. Por último trata de posicionar un `Cube` correctamente, comprobamos que el `Color` esperado y el del `Cube` de dicha posición coinciden.

test_report2()

Creamos todos los objetos necesarios para poder instanciar un `GameTeamsTest`. Usamos el método `play()` para dar un poco de actividad y finalmente comprobamos que el método `report()` nos proporciona el `JSONObject` esperado.

play_test3()

Creamos todos los objetos necesarios para poder instanciar un `GameTeamsTest`. Tratamos de poner algunos `Cube` en posiciones incorrectas y comprobamos que salte una excepción. El primer `Cube` tratará de ocupar una posición que no está libre y el segundo una posición que no existe en el `Board`. Por último trata de posicionar un `Cube` correctamente, comprobamos que el `Color` esperado y el del `Cube` de dicha posición coinciden.

test_report3()

Creamos todos los objetos necesarios para poder instanciar un `GameTeamsTest`. Usamos el método `play()` para dar un poco de actividad y finalmente comprobamos que el método `report()` nos proporciona el `JSONObject` esperado.

4.4.5 PlayerTest

Add_score_test()

Creamos dos `Player` y se llama al método `addScore()` de uno de ellos para sumarle 1 a su puntuación. Se comprueba si se ha realizado correctamente. Después, se crea un `Cube` del primer `Player` y se le cambia el "propietario" al segundo. Finalmente se comprueba que las puntuaciones de ambos `Player` son las correctas.

test_report() y test_report2()

Creamos dos `Player` y se le añade determinada puntuación al primero. Se comprueba que es correcta, se crea un `Cube` al primer `Player` y se le cambia el "propietario" al segundo. Finalmente se comprueba que los métodos `report()` de ambos `Player` devuelven los `JSONObject` esperados.

4.4.6 TeamTest

Update_test()

Creamos dos `Player` y los añadimos a una lista de `Player`. Creamos un `Team` con esa lista y añadimos puntuación a ambos `Player` por separado. Finalmente, se llama al método `update()` del `Team` creado y se comprueba que la puntuación del `Team` es igual a la suma de las puntuaciones individuales de los `Player`.

test_report() y test_report2()

Creamos dos `Player` y los añadimos a una lista de `Player`. Creamos un `Team` con esa lista y finalmente comprobamos que el método `report()` de `Team` devuelve el `JSONObject` esperado en cada caso.

4.5 Paquete Replay

4.5.1 ReplayTest

Test_1() y Test_2()

Creamos un **Board** y tres **Player** que se añaden a una lista de **Player**. Se crean dos **Cube** y se añaden a una lista de **Cube**. Se crea un **GameClassic** con la información necesaria y se añade un nuevo **Cube** a la cola de **Cube** pendientes de añadir al **Game**. Se llama al método **play()** de **Game** y se crea un **GameState**. Después, se crea un objeto **Replay** y se le añade el estado recién creado. Finalmente, se comprueba que el método **report()** de **Replay** devuelve el **JSONObject** esperado con la información correcta.

4.5.2 StateTest

Report_test() y report_test2()

Creamos un **Board** y tres **Player** que se añaden a una lista de **Player**. Se crean dos **Cube** y se añaden a una lista de **Cube**. Se crea un **GameClassic** con la información necesaria y se añade un nuevo **Cube** a la cola de **Cube** pendientes de añadir al **Game**. Se llama al método **play()** de **Game** y se crea un **GameState** con el **Game** creado. Finalmente, se comprueba que el método **report()** de **GameState** devuelve el **JSONObject** esperado con la información correcta.

5. Diseño final

5.1 Paquete model

* **SaveLoadManager** *

Esta clase incluye un amplio abanico de métodos cuya finalidad es la carga, guardado y borrado de información desde archivos externos al mismo. En particular, implementa métodos que posibilitan:

- Obtener la lista de partidas y replays guardados
- Cargar partidas y replays guardados
- Guardar partidas y replays
- Borrar partidas y replays guardados
- Cargar las matrices definidas de Shape

5.1.1 Builders

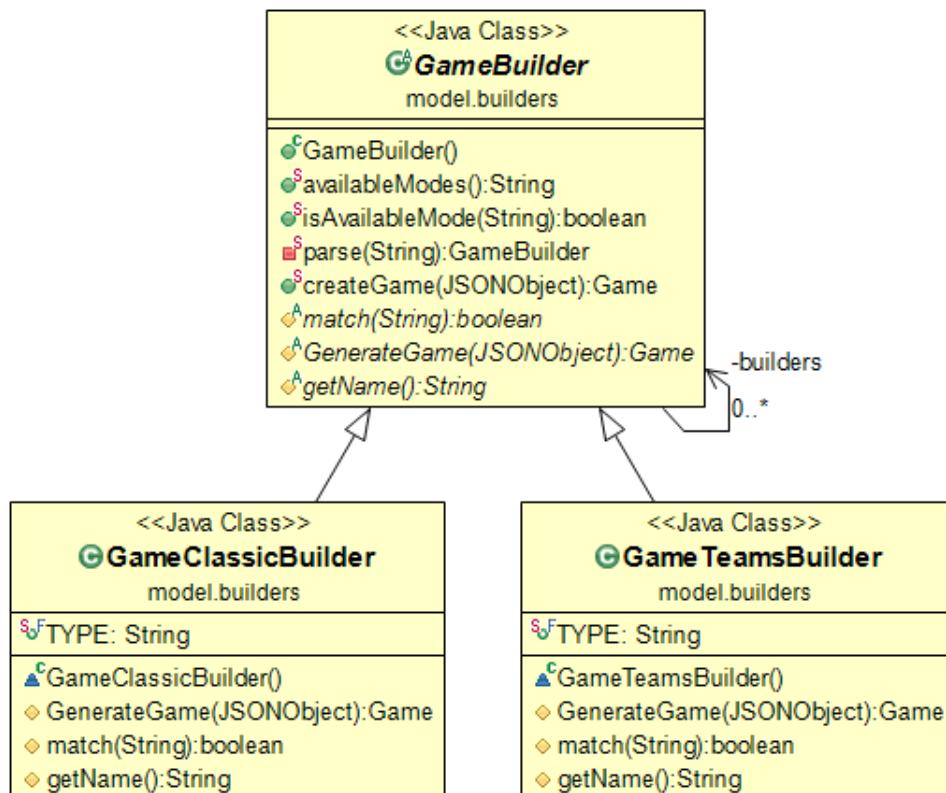
Este paquete agrupa las clases cuya responsabilidad es saber construir un objeto del tipo **Game** a través de un conglomerado de información. Para unificar dicha funcionalidad se han tomado las siguiente decisiones:

- La factoría abstracta que genera el **Game** es una clase abstracta de la que extienden las factorías concretas que saben generar cada tipo de **Game**.
- La factoría abstracta tiene un listado de todas las posibles factorías concretas que hay, de modo que cuando se le pide crear un objeto **Game** ella internamente llama a la factoría correspondiente para que complete la creación en los detalles específicos.
- Precisamente por el hecho anterior y por tener un lugar concreto donde crear el juego (puesto que el **Game** se crea tras iniciar la aplicación, ya que en las pantallas sucesivas se puede decidir crear un juego nuevo o en el modo red no se puede crear el juego hasta tener los datos de todos los jugadores), el método para generar **Game** es un método estático que puede ser llamada desde los

lugares donde se requiere una creación de **Game**, puesto que la responsabilidad de la factoría es dado un conjunto de datos, generar el **Game** correspondiente.

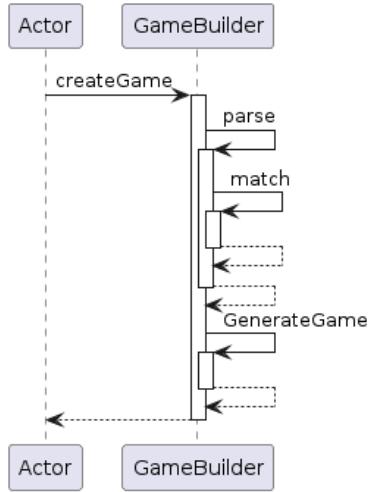
- El formato para la información de construcción del juego es **JSONObject** y es necesario que se pase a la factoría para tener un proceso unificado de creación del juego. Que las clases que pueden crear un **Game** tengan que saber cómo generar el **JSONObject** no rompe la encapsulación, pues si poseen la habilidad para crear un **Game** están íntimamente ligadas a la estructura del mismo (puesto que tienen que pedir al usuario los datos para crearlo) y, por tanto, los cambios en ella deben afectar a los métodos donde construyan dicho **JSONObject**.

Observadas las anteriores decisiones de diseño, se presenta el conjunto de clases implicadas en la construcción del juego siguiendo el patrón de *Abstract Factory* y *Builder*:



* GameBuilder *

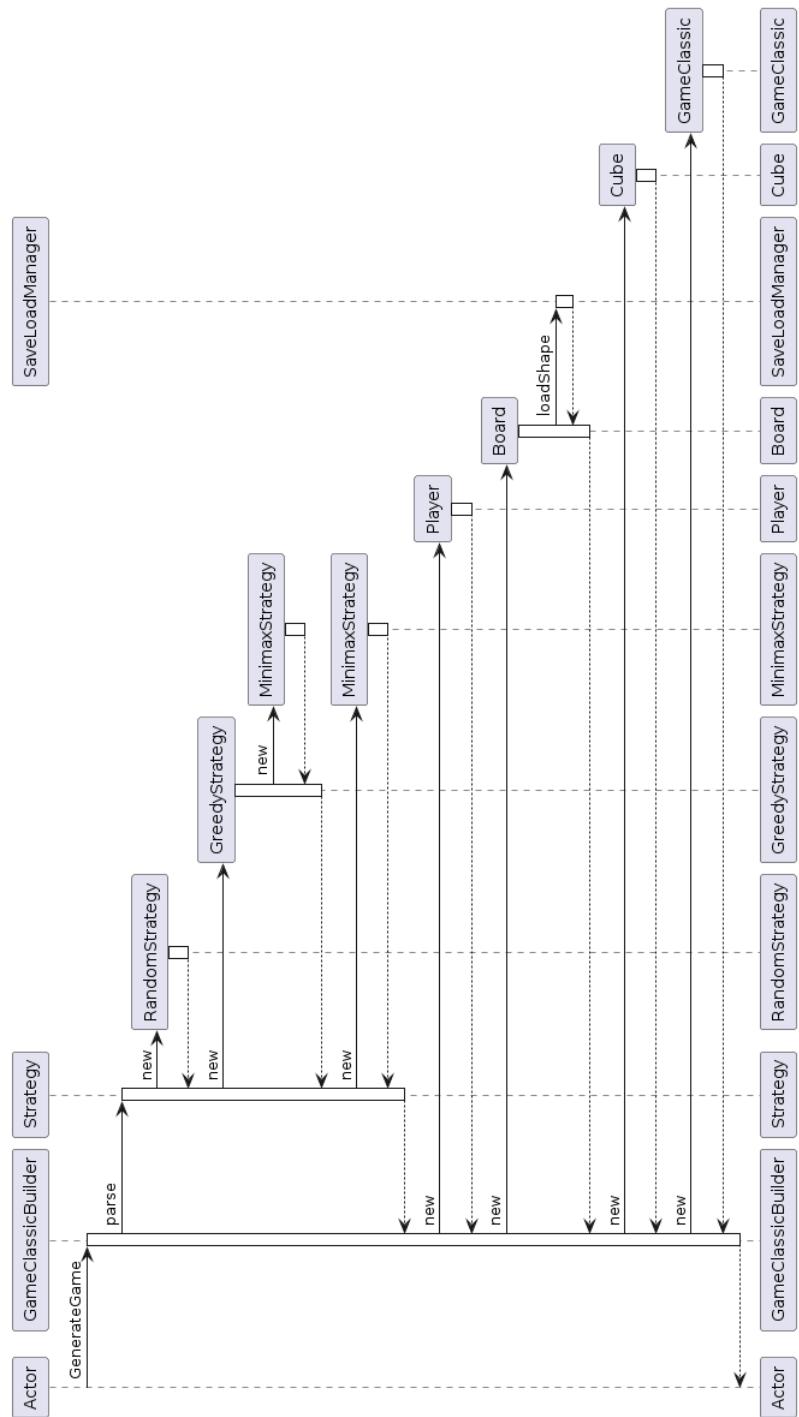
La clase principal es **GameBuilder** cuyo método principal para la creación es `public static Game createGame(JSONObject o)`. Dicho método se encarga recoger el tipo de juego que se desea generar del **JSONObject** y parsear el Builder concreto que sabe construir dicho juego. Una vez lo encuentra se pasa la responsabilidad de crearlo al mismo llamando al método de los Builder `Game GenerateGame(JSONObject o)`. Es decir, la interacción a la hora de crear un **Game** sería:

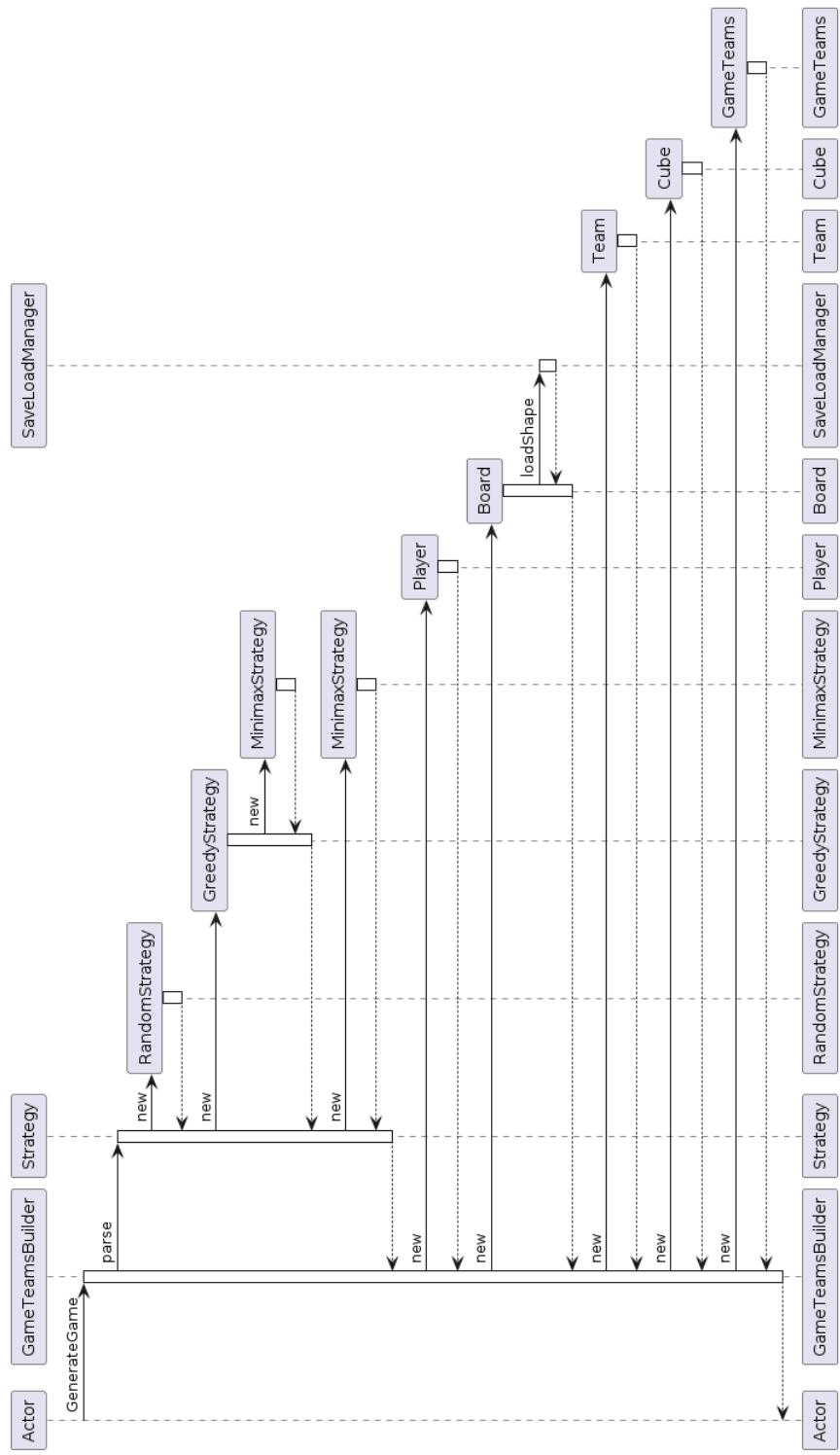


Esta clase posee una serie de métodos protegidos que utilizan sus hijas, los Builders específicos, que sirven para verificar en tiempo de construcción del juego que la información recibida va siendo consistente y válida (los jugadores tienen nombres distintos, colores distintos, escogemos colores válidos...).

* GameClassicBuilder y GameTeamsBuilder *

Ambas son una subclase de `GameBuilder` y son las responsables de la lógica de construcción de los `GameClassic` y los `GameTeams`. El flujo de ejecución es idéntico para ambas, se llama al método `Game GenerateGame(JSONObject o)` que devuelve el juego listo para empezar la ejecución, aunque la forma de construir el juego sí es distinta:





5.1.2 Commands

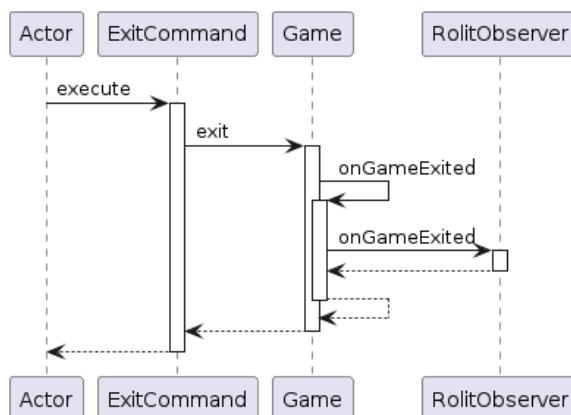
Este paquete agrupa las clases que se encargan de los comandos del juego a partir de una solicitud. Dicha solicitud es convertida en un objeto independiente que contiene toda la información sobre la misma, lo cual permite la separación de responsabilidades y la división del código en capas. Para ello se han tomado las siguientes decisiones:

- Existe una clase abstracta `Command` de la cual extienden todos los comandos específicos que saben crear cada comando concreto. Además, `Command` implementa a la interfaz `Replayable`.
- La clase `Command` contiene una lista de todos los comandos disponibles, de modo que cuando se pide crear un objeto `Command`, esta clase se encarga de llamar al comando concreto correspondiente para completar la creación con la información específica de este.
- Cada comando se identifica por un nombre, unos detalles, una abreviatura y un mensaje de ayuda (explicación de lo que hace el comando).
- Cada subclase de `Command` implementa su propio método `execute()`, en el que se ejecuta lo que hace cada uno de los comandos.
- Existen métodos comunes a todos los comandos: un método para parsear los argumentos del comando, otro para mostrar un mensaje de ayuda con el comando, o el report (común a todas las clases que implementan la interfaz `Replayable`).

Teniendo en cuenta lo anterior, se presenta el conjunto de subclases implicadas en la construcción de objetos comando, siguiendo el patrón de diseño *Command*.

* ExitCommand *

Este comando sirve para salir del juego a través del método `exit()` de `Game`, el cual realiza las notificaciones correspondientes siguiendo el patrón *Observer*.



*** HelpCommand ***

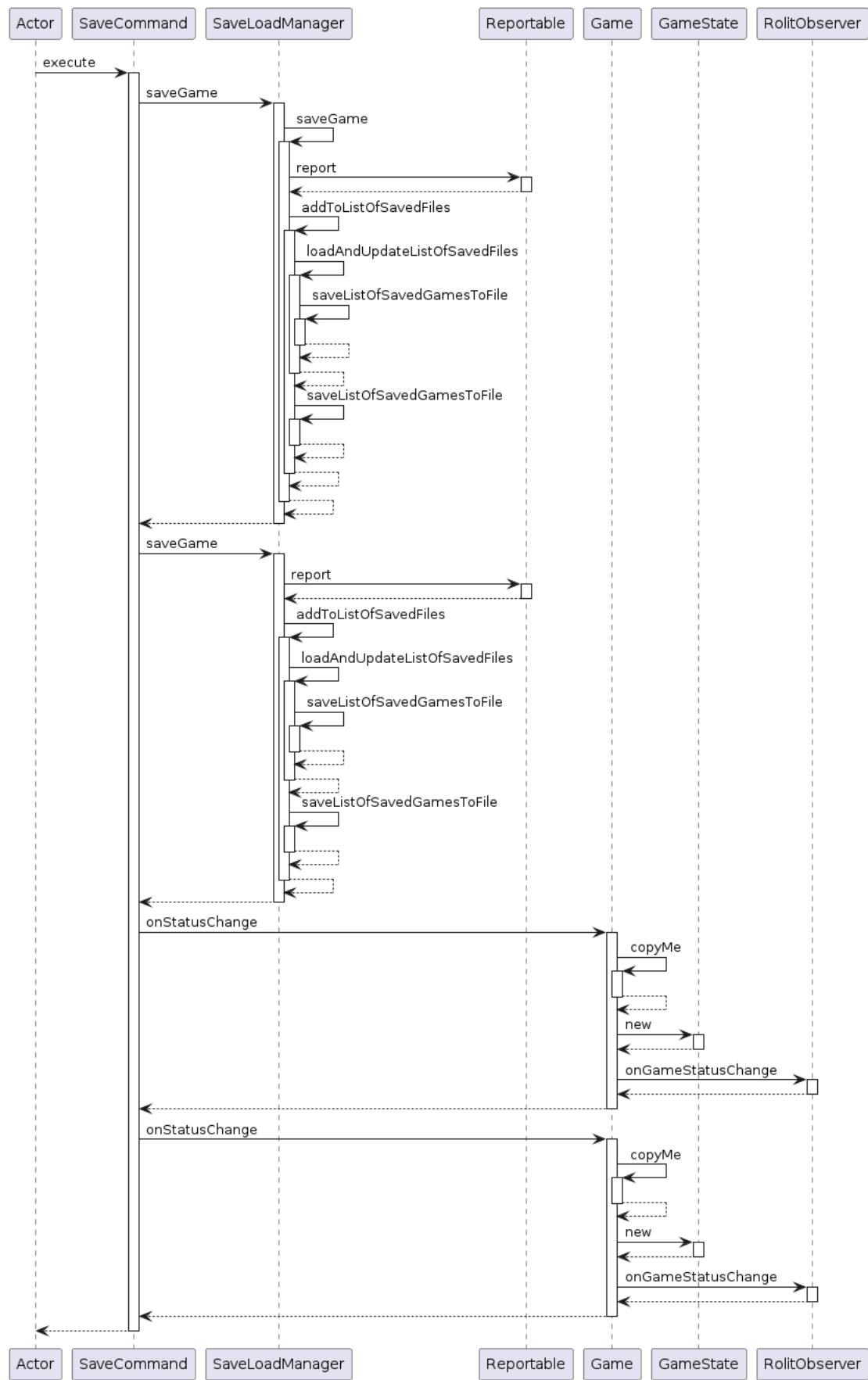
Este comando sirve para mostrar un mensaje de ayuda para que el jugador entienda para qué sirve cada comando.

*** PlaceCubeCommand ***

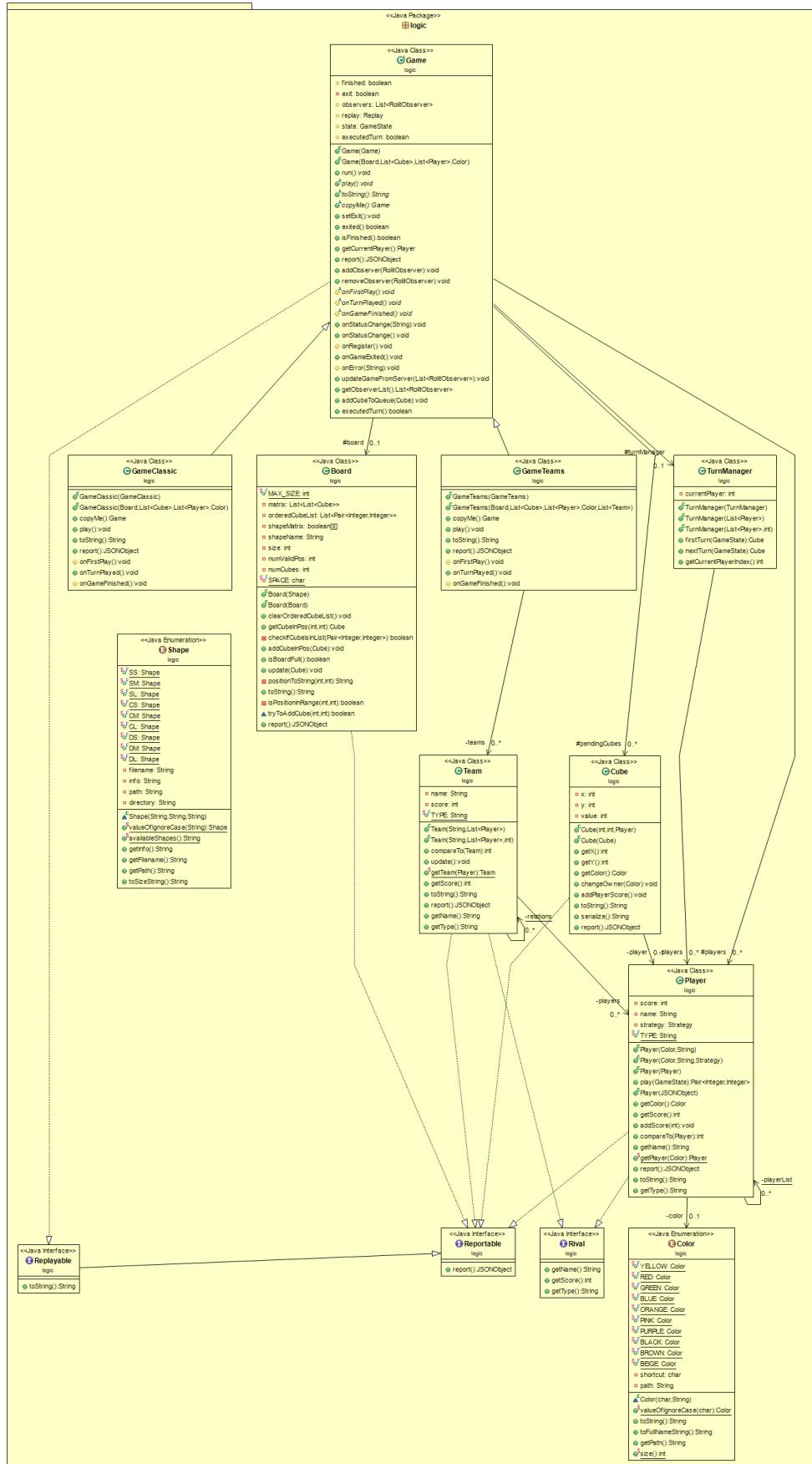
Este comando sirve para colocar un cubo en una posición del tablero añadiéndolo a la cola de cubos pendientes de colocar de la clase `Game` mediante el método `addCubeToQueue()`. Además, en esta clase se sobrescribe el método `parse` ya que se necesitan 2 argumentos (posición en x y en y) para colocar el cubo en el tablero.

*** SaveCommand ***

Este comando sirve para guardar una partida en un fichero a través del método `saveGame()` de la clase `SaveLoadManager`, distinguiendo los casos en los que se dispone de un nombre de fichero de los casos en los que no. Esta clase añade el juego a la lista de ficheros de partida guardados y también se encarga de notificar a las clases correspondientes siguiendo el patrón *Observer*.

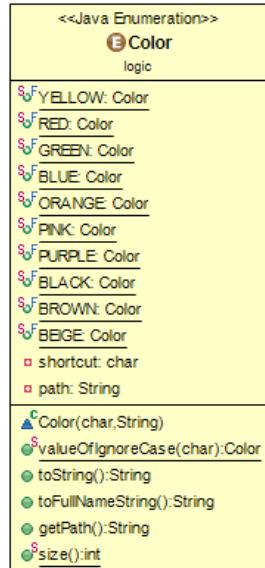


5.1.3 Logic



Este paquete agrupa la lógica principal encargada del funcionamiento de los elementos que componen el juego.

* Color *

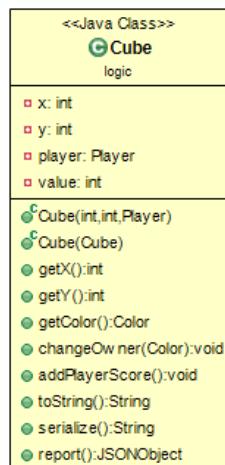


La clase **Color** es la que recoge la lógica necesaria para los colores de los cubos. Vemos en la imagen que los colores disponibles en el juego son: Amarillo, rojo, verde, azul, naranja, rosa, morado, negro, marrón y beige.

De cara a la representación de los colores los dos atributos más relevantes de la clase son **shortcut**, para su representación por consola, y **path**, ruta con la imagen para su representación por interfaz gráfica.

* Cube *

La lógica referente a los cubos se recoge en la clase **Cube**. Vemos a continuación el estado final de la clase en cuestión:

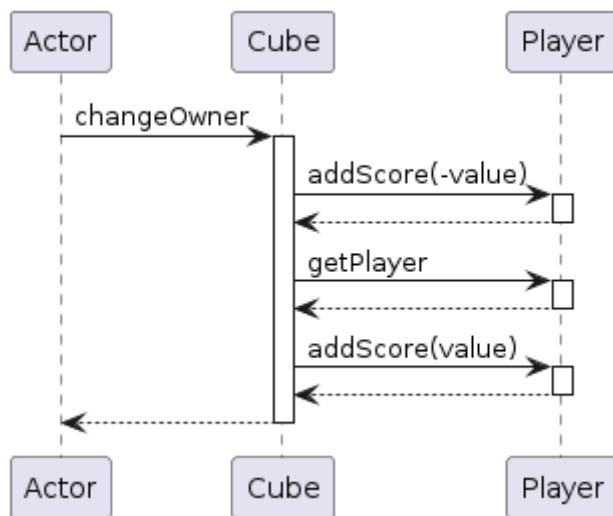


Los atributos necesarios son los siguientes:

- x: Coordenada x en el tablero.
- y: Coordenada y en el tablero.
- player: Jugador al que pertenece el cubo.
- value: Cantidad de puntos que otorga al jugador la posesión del cubo.

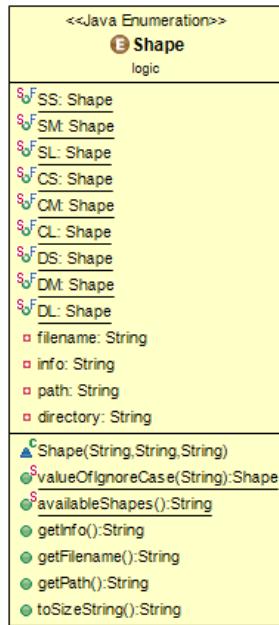
Como vemos, esta clase no tiene ningún atributo de tipo **Color**. Esto es porque el color del cubo es el color del jugador al cual está asociado (player). Por tanto, cuando el tablero se actualiza, a nivel de representación interna, los cubos no cambian de color, sino de jugador asociado. Esto se hace con el método `changeOwner()`, a través del cual se decrementa la puntuación del antiguo jugador asociado y se incrementa la del nuevo.

Este funcionamiento se refleja en el siguiente diagrama:



* Shape *

La lógica interna de las formas de tableros se recoger en la clase **Shape**, clase enumerada que tiene como valores todas las posibles formas (esto incluye los diferentes tamaños). La siguiente ilustración muestra el estado final de la clase:



Las posibles formas son las siguientes:

- SS: Small Square
- SM: Medium Square
- SL: Large Square
- CS: Small Circle
- CM: Medium Circle
- CL: Large Circle
- DS: Small Diamond
- DM: Medium Diamond
- DL: Large Diamond

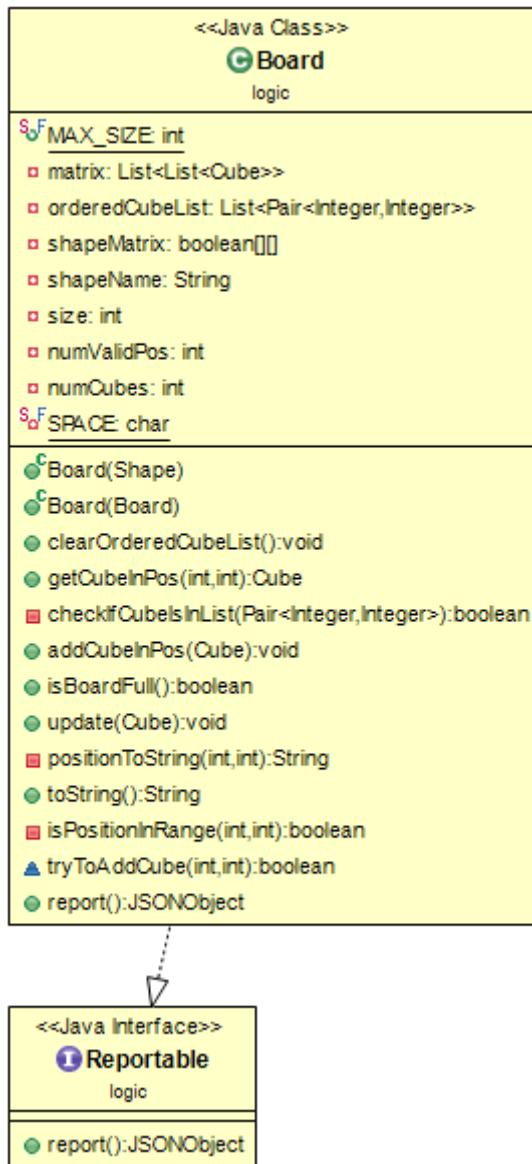
Cada una de estas formas tiene asociada el nombre de un archivo, en el cual se recoge toda la información relevante acerca de cada forma (véanse el tamaño y la representación de las casillas válidas).

* Board *

La parte fundamental de la lógica que encapsula el concepto del tablero se recoge en la clase **Board**.

Se puede pensar en la clase **Board** como el contenedor de cubos para el juego. Esta clase implementa la interfaz **Reportable**, puesto que es de gran interés tener una representación de esta clase en forma de **JSONObject**.

El estado final de la clase se ve en la siguiente imagen:



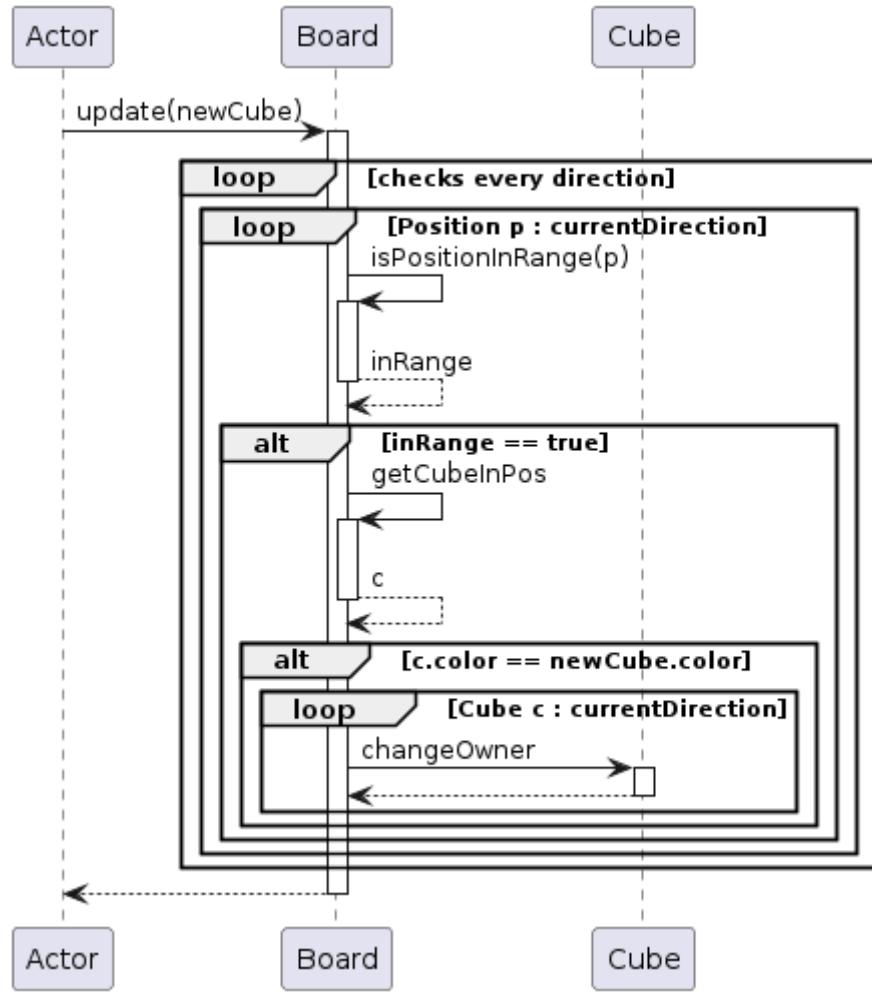
Para conseguir que esta clase cumpla con la idea de su creación, hace de contenedor de cubos a través de varias representaciones.

Vemos en la imagen que tenemos un atributo llamado **matrix**. Esta es la representación fundamental de los cubos en el tablero. Consiste en guardar en cada posición el cubo que la ocupa, o un valor nulo en caso de no estar ocupada. Es la representación más intuitiva y de mayor uso.

Otra representación útil de los cubos es en forma de lista ordenada por el momento de creación de los cubos, para lo que tenemos el atributo **orderedCubeList**. Esta es una representación fundamental para el funcionamiento del juego en red.

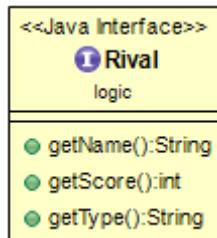
Como contenedor de los cubos del juego, **Board** tiene una responsabilidad fundamental, y esa es la de actualizarse conforme se añaden cubos en el juego. Para esto tenemos el método **update(Cube)**, cuyo parámetro de entrada es el cubo que ha sido añadido. Para actualizarse, al colocar el nuevo cubo se hace un recorrido en todas las direcciones, comprobando qué cubos han de ser cambiados de color.

El funcionamiento de este método se recoge en el siguiente diagrama:



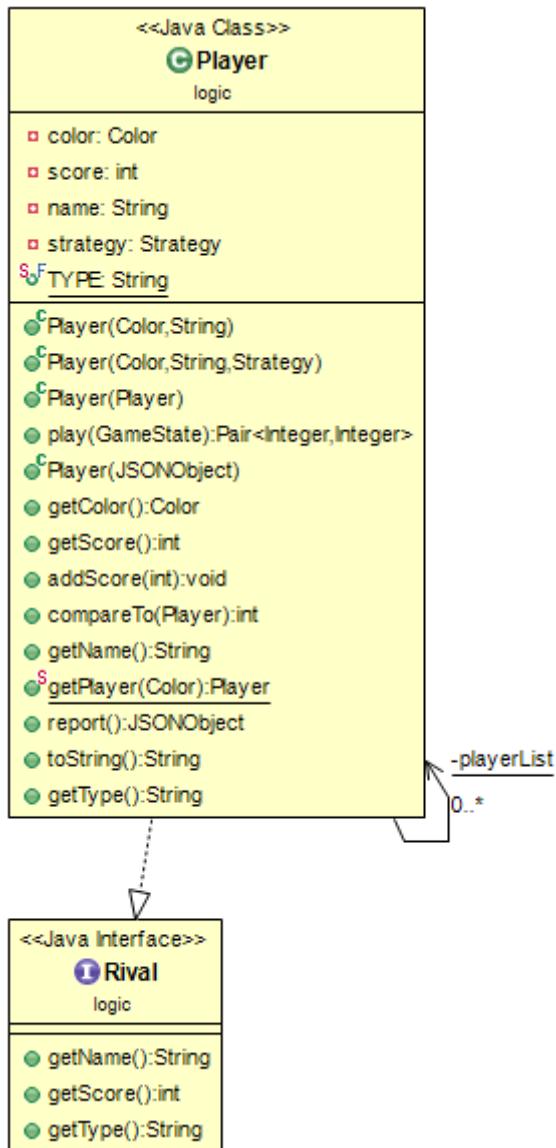
Como el tablero puede tener distintas formas, vemos la existencia del constructor `Board(Shape)`, que crea un tablero vacío con las restricciones impuestas por la forma que recibe como parámetro.

* Rival *



La interfaz `Rival` sirve para recoger comportamientos comunes entre los equipos y jugadores individuales, de cara a sus representaciones de forma abstracta y uniforme.

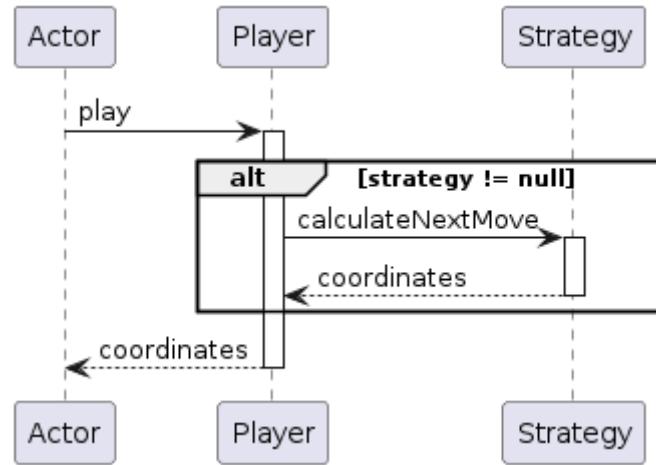
* Player *



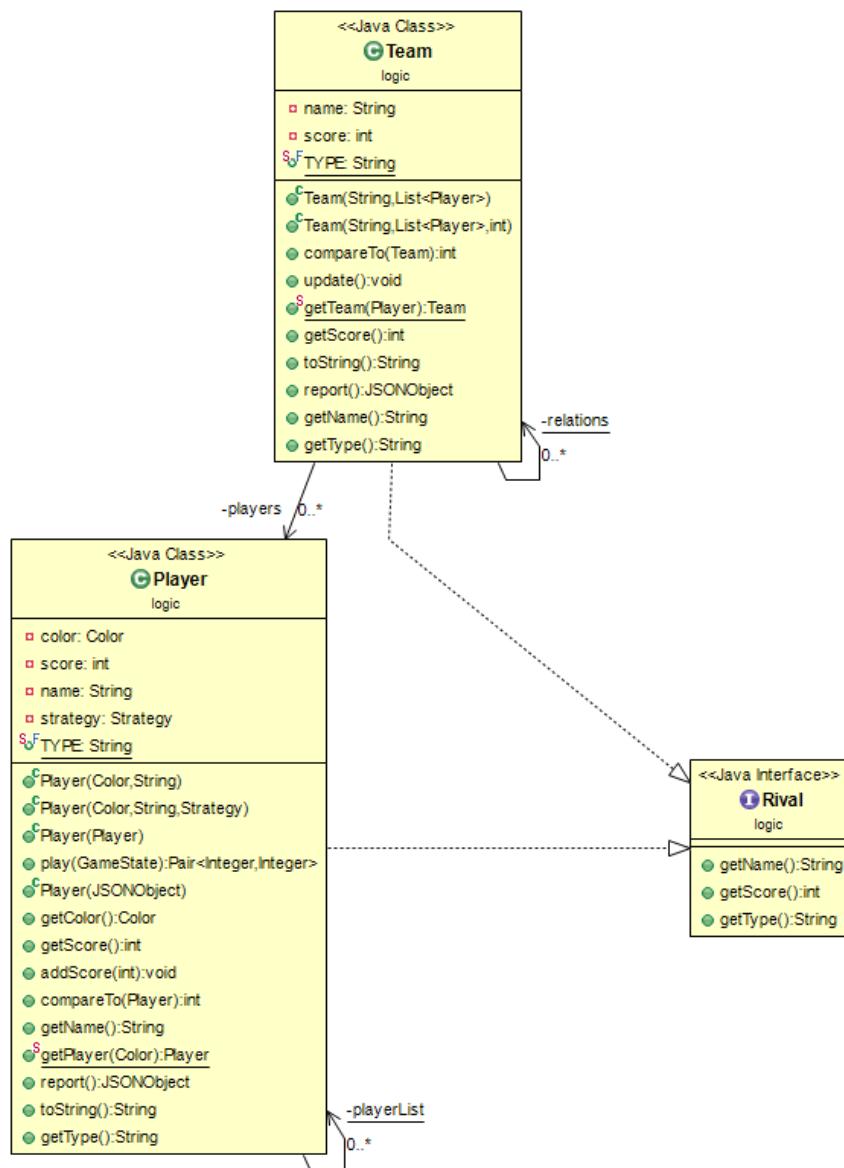
Esta es la clase que recoge la lógica necesaria para el funcionamiento de los jugadores. Vemos que tiene un atributo de tipo **Strategy**, el cual, en caso de no ser nulo, indica que el jugador es una inteligencia artificial, y su comportamiento está definido por dicha estrategia.

El método fundamental para el funcionamiento de los jugadores y sus turnos es el método **play(GameState)**. Si el jugador es una inteligencia artificial, este método será el encargado de hacer el cálculo de la posición en la que colocará un cubo. Por el contrario, si el jugador es real (no es una inteligencia artificial) este método no hará nada, puesto que la ejecución de su turno se llevará a cabo cuando el usuario en cuestión introduzca los datos necesarios.

El funcionamiento de este método queda reflejado en el siguiente diagrama:



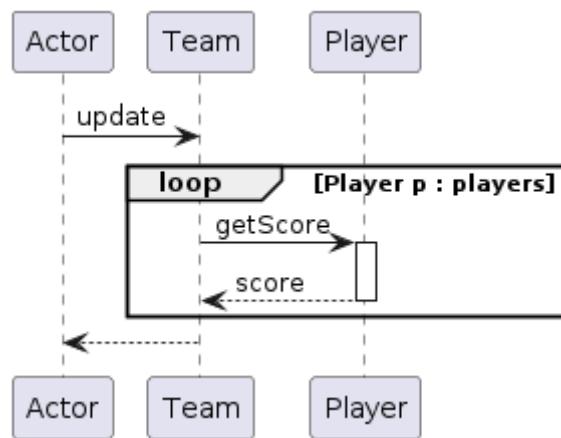
* Team *



La clase `Team` recoge la lógica encargada del funcionamiento de los equipos en el modo de juego por equipos. Su función se resume en hacer de contenedor de los jugadores.

El método principal de esta clase de cara a mantener datos coherentes es el método `update()`, en el cual recalcula la puntuación del equipo a partir de sus integrantes.

Vemos este comportamiento en el siguiente diagrama:

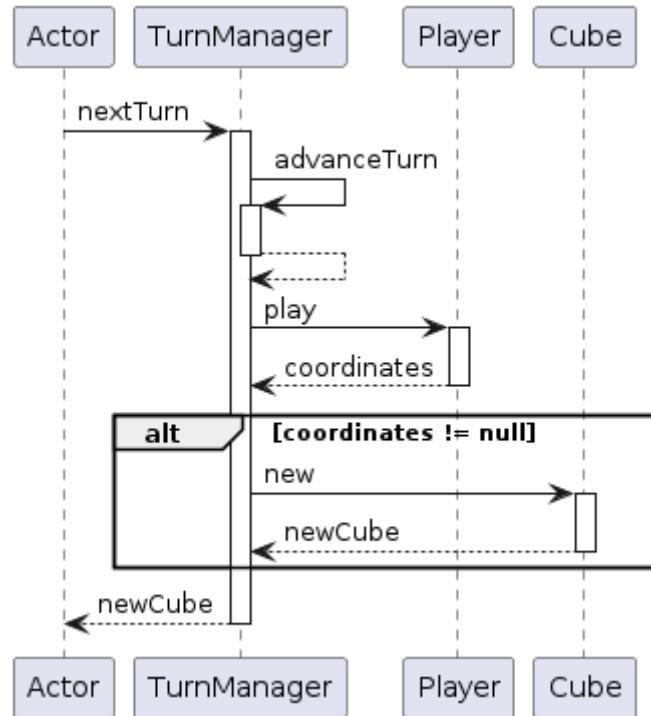


* TurnManager *



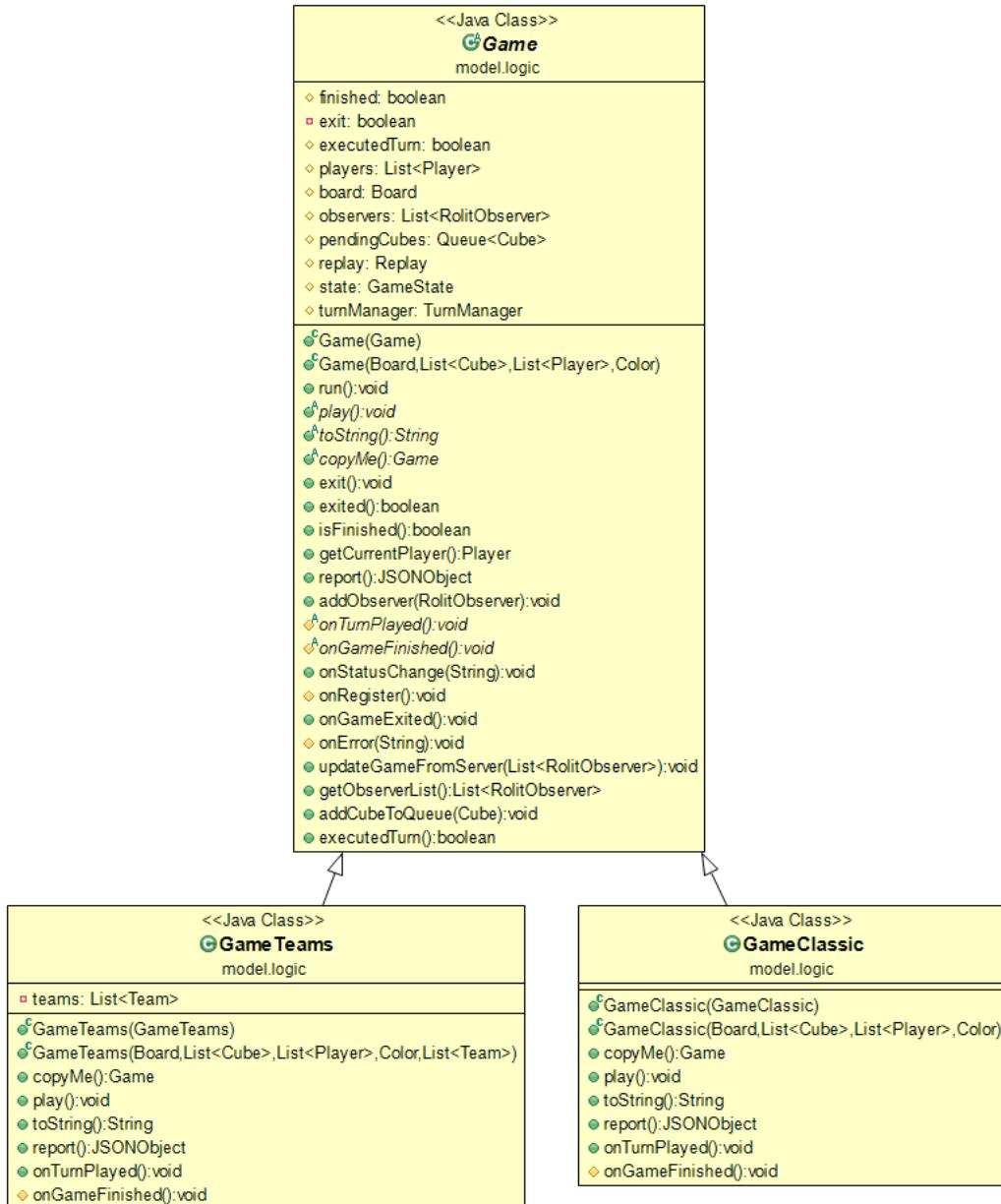
La clase `TurnManager` hace de gestor de turnos. Tras cada turno avisa al siguiente jugador para que ejecute su turno, y en caso de recibir las coordenadas correspondientes al siguiente movimiento, crea la nueva instancia de `Cube` para que sea manejada por el juego.

Esto queda reflejado en el siguiente diagrama del método `nextMove()`:



* Game, GameClassic y GameTeams *

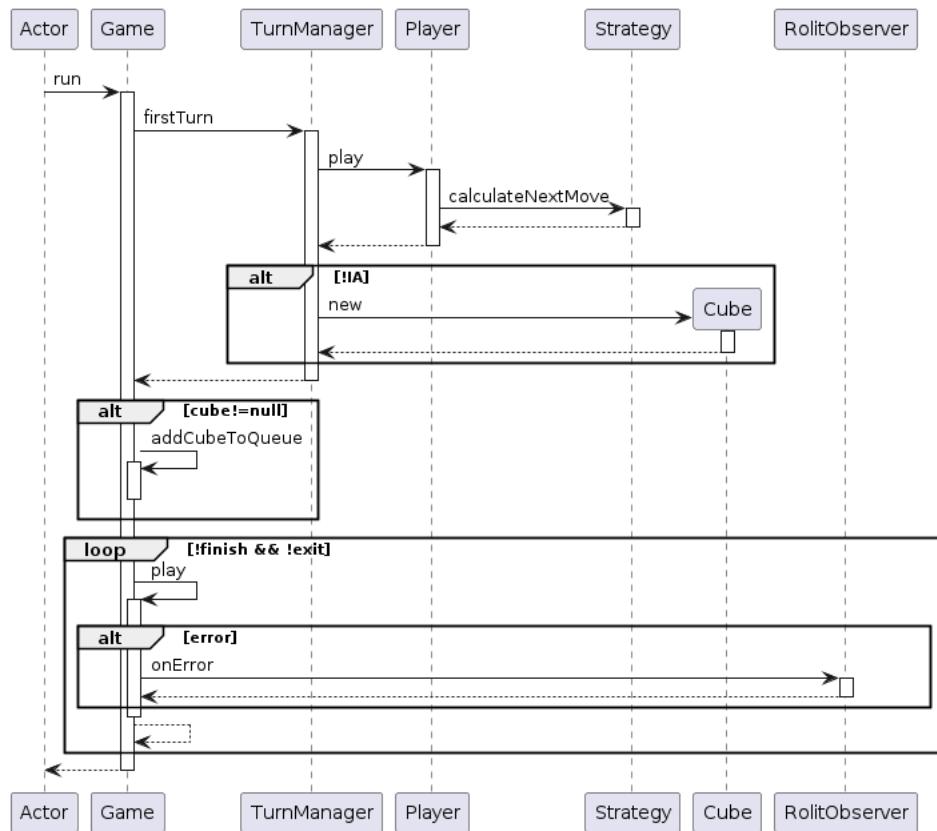
La clase **Game** es la clase fundamental del modelo y es la encargada de representar el concepto del juego completo, es decir, de la misma manera que el **Board** representa el juego físico (tablero, cubos, etc.) el **Game** representa todo el juego en su totalidad (jugadores, puntos, tablero, normas, etc.).



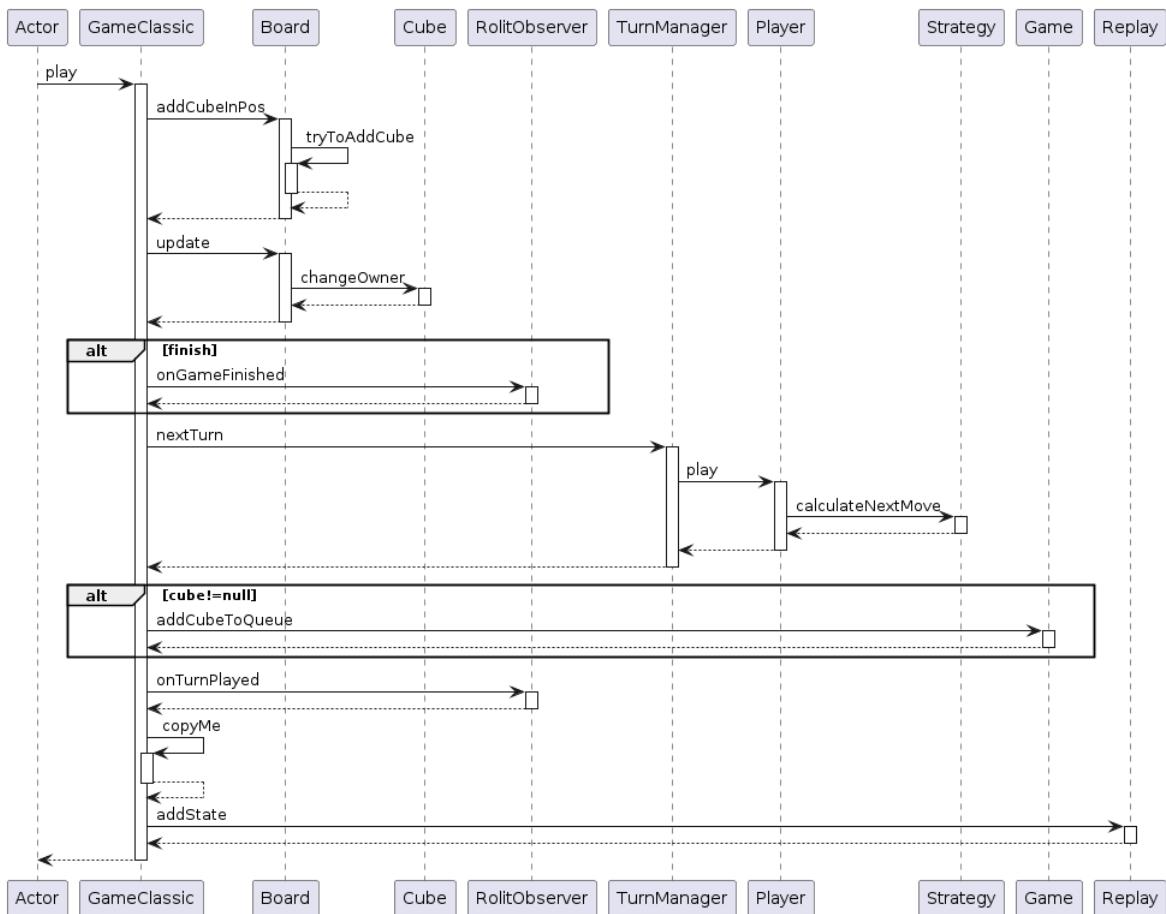
En primer lugar, la decisión que se ha tomado en cuanto al modelo es que tenga su propio hilo, es decir, continuamente el **Game** está ejecutando el bucle de juego hasta que finaliza la partida. También es necesario saber para el funcionamiento del bucle de juego que la estructura de datos del juego es una cola, pues cualquiera que haga jugadas simplemente encola dicho cubo para que el modelo, cuando pueda, procese dicha jugada y sitúe los cubos en el orden en el que se han ido haciendo las jugadas.

Principalmente la actividad del hilo de **Game** y de la funcionalidad de la clase reside en el método `void run()`. Tras ser creado en algún sitio, el modelo comienza la

ejecución de su método `void run()` que pone en marcha un bucle de juego que no termina hasta que el juego finalice o se seleccione salir. Dentro de dicho bucle, llamamos a la función `void play()` abstracta y relativa a cada subclase de `Game` que encapsula toda la lógica de un turno en cada tipo de juego.

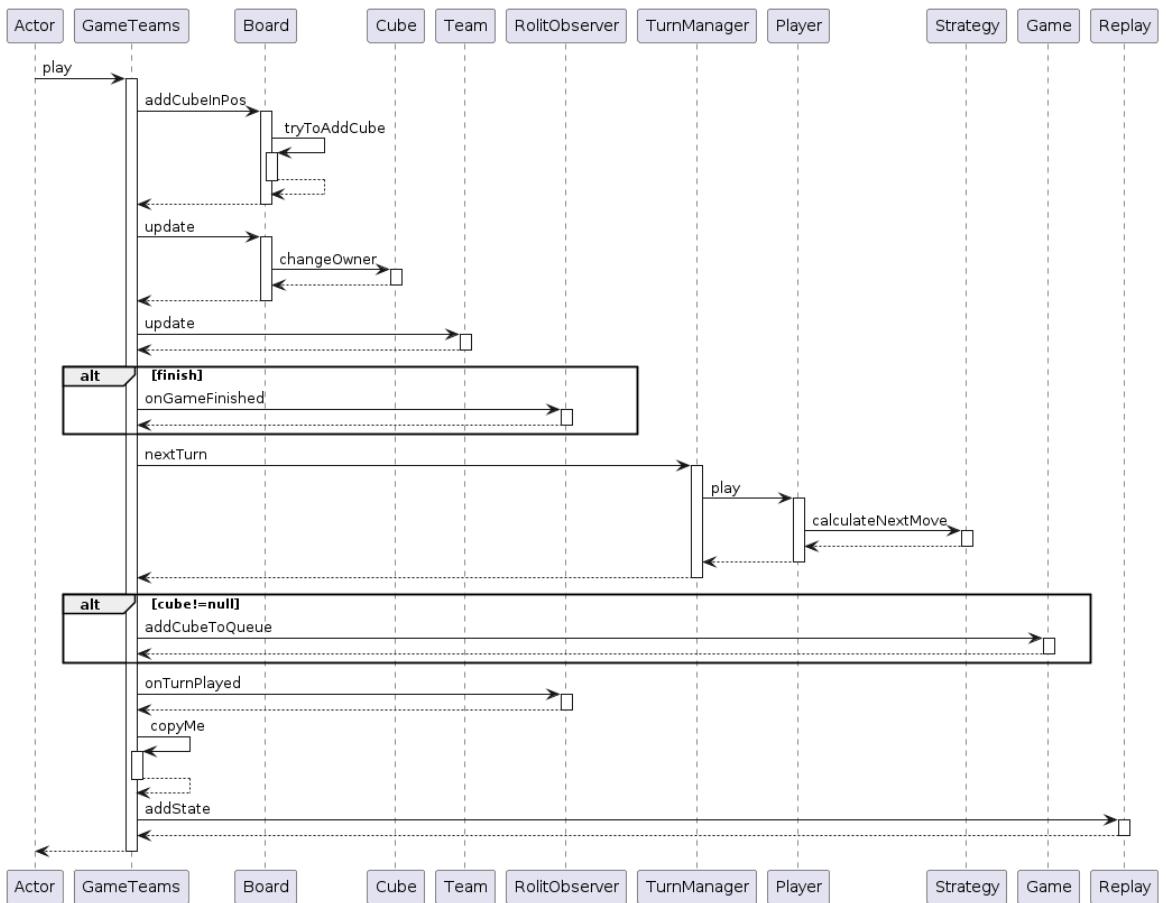


La decisión de que el juego tenga su propio hilo viene de la implementación de las inteligencias artificiales en el proyecto. Por un lado, si todo el juego tiene que esperar a que la IA calcule el movimiento que desea la ejecución se ralentiza (mucho cuando se busca precisión en los movimientos). Pero por otro, y más importante, si no se tuviese el hilo, los jugadores automáticos colocarían un cubo y harían que la ejecución del juego siguiese sin esperar a que se notificara a todos los observadores que el juego ha cambiado. Por tanto, lo mejor es dejar que cada uno haga la jugada cuando se lo solicite y lo que mande sea una solicitud de poner un cubo. El modelo cuando pueda ejecutará dicha solicitud y pedirá al siguiente que mande solicitud si procede, etc.



La ejecución del juego clásico es sencilla:

- Desencolamos el primer cubo pendiente y tratamos de añadirlo al tablero.
- En caso de que el tablero esté completo termina el juego.
- En caso contrario, cambiamos de turno y en todos los casos notificamos los cambios.



Tal y como se ve, la única diferencia notable (al menos para estos dos tipos de juego) es que en el por equipos se actualizan las puntuaciones de los equipos tal colocar un nuevo grupo.

5.1.4 Online

Mencionamos y agradecemos especialmente a la aplicación de código abierto [Chess](#), de Oliver Ekberg, sobre el cual nos hemos inspirado a la hora de elaborar el boceto de la estructura del juego online.

Como explicamos en el apartado de Arquitectura, el juego en red se basa en el **modelo cliente-servidor**, donde los clientes poseen el modelo, realizan cambios en el mismo y lo notifican al servidor, el cual procede a enviar al resto de clientes la información nueva según la cual deben actualizar sus modelos.

Para implementar esta funcionalidad de red en Java operamos según el modelo **ServerSocket-Socket**. Primero, desde la perspectiva del usuario que abre el servidor, se debe crear una instancia de **ServerSocket** pasándole como parámetro en el constructor el puerto en el que localmente debe operar el servidor. Posteriormente, se crea un **newSocket** por medio de llamar al método **accept()** de **ServerSocket**. De esta forma, tenemos un socket asociado a un cliente en específico. Para n clientes el servidor necesitará n sockets. Cada cliente solo necesita un socket, pues solo tiene conexión con el servidor y no con el resto de clientes.

```
1 //Perspectiva del servidor
2
3 \texttt{ServerSocket} serverSocket = new \texttt{ServerSocket}(port);
4 \texttt{Socket} socketCliente1 = serverSocket.accept();
5 ...
6 ...
7 ...
8 \texttt{Socket} socketClienteN = serverSocket.accept();
```

Este método **accept()** se queda "bloqueado." en espera, hasta que un cliente se conecta por medio de la creación de un **Socket** desde su aplicación de la siguiente manera:

```
1 //Perspectiva del cliente
2
3 \texttt{Socket} socket = new \texttt{Socket}(ip, port);
```

donde **ip** es la dirección IP donde opera el servidor (ya sea local, o pública con el puerto especificado abierto para permitir conexiones desde fuera de su red), y **port** el puerto donde esta opera.

Una vez que las creaciones del **Socket** de servidor y cliente han sido creadas de forma satisfactoria, la conexión ha sido realizada con éxito. Por tanto, podemos proceder al envío de mensajes entre cliente-servidor y viceversa.

Para ello, Java nos ofrece realizar este cambio de información por medio de pares de instancias **BufferedReader-PrintWriter**, llamémosle **in** e **out** respectivamente. Estas instancias leen y reciben **String**, respectivamente. Nuestro modelo procederá a enviar **JSONObject** pasados a formatos **String**, que después al ser recibidos como **String** se volverán a construir en **JSONObject** por medio de la constructora **deJSONObject** que admite como parámetro un **String**.

Cada servidor posee un número de parejas **in-out** equivalente al número de clientes conectados, y cada cliente posee una pareja.

```
1 //Perspectiva del servidor
2 BufferedReader inCliente1 = new BufferedReader(new
```

```

3 | InputStreamReader(socketCliente1.getInputStream()));
4 | ...
5 | ...
6 | ...
7 | PrintWriter outClienteN = new PrintWriter(
8 |   socketClienteN.getOutputStream(), true);

```

```

1 | //Perspectiva del cliente
2 | BufferedReader in = new BufferedReader(new
3 | InputStreamReader(socket.getInputStream()));
4 |
5 | PrintWriter out = new PrintWriter(
6 |   socket.getOutputStream(), true);

```

De esta forma, cada vez que se pretende enviar un mensaje desde el cliente al servidor, el cliente k envía un mensaje `msg` por medio del método `out.println(msg)`. Este mensaje es recogido desde el servidor por el método `inClienteK.readLine()`. Asimismo, si el servidor pretende enviar un mensaje al cliente k, este debe llamar a `outK.println(msg)`. El cliente k recoge el mensaje desde `in.readLine()`. En resumen:

```

1 | //Envio de mensaje ClienteK-Servidor
2 |
3 |
4 |
5 | //Perspectiva del cliente k
6 | out.println(msg);
7 |
8 | --->
9 |
10 | //Perspectiva del servidor
11 | String msg = inClienteK.readLine();
12 |
13 |
14 | //Envio de mensaje Servidor-ClienteK
15 |
16 |
17 |
18 | //Perspectiva del servidor
19 | outK.println(msg);
20 |
21 | --->
22 |
23 | //Perspectiva del cliente k
24 | String msg = in.readLine();

```

Sin embargo, surgen una serie de problemas con respecto a este modelo. El servidor debería recibir de manera independiente, paralela y en cualquier momento los peticiones de cada cliente, de lo contrario habría que definir un orden completamente arbitrario de llegada de mensajes según cliente. Si bien esto pudiera hacerse para realizar juegos por turnos en el que un mensaje enviado por un cliente al que no le corresponde el turno no sea procesado por el servidor, otras funcionalidades serían imposibles de implementar.

Por ejemplo, podría ocurrir que en el modo por equipos, dos clientes se conecten. Reciben ambos del servidor una lista de equipos en la que pueden conectarse. Se ve que es absurdo imponer un orden de quién envía la información del equipo elegido, pues esta es imposible de anticipar. Puede darse que el primer cliente que se conecte sea el segundo que especifique en qué equipo quiere conectarse.

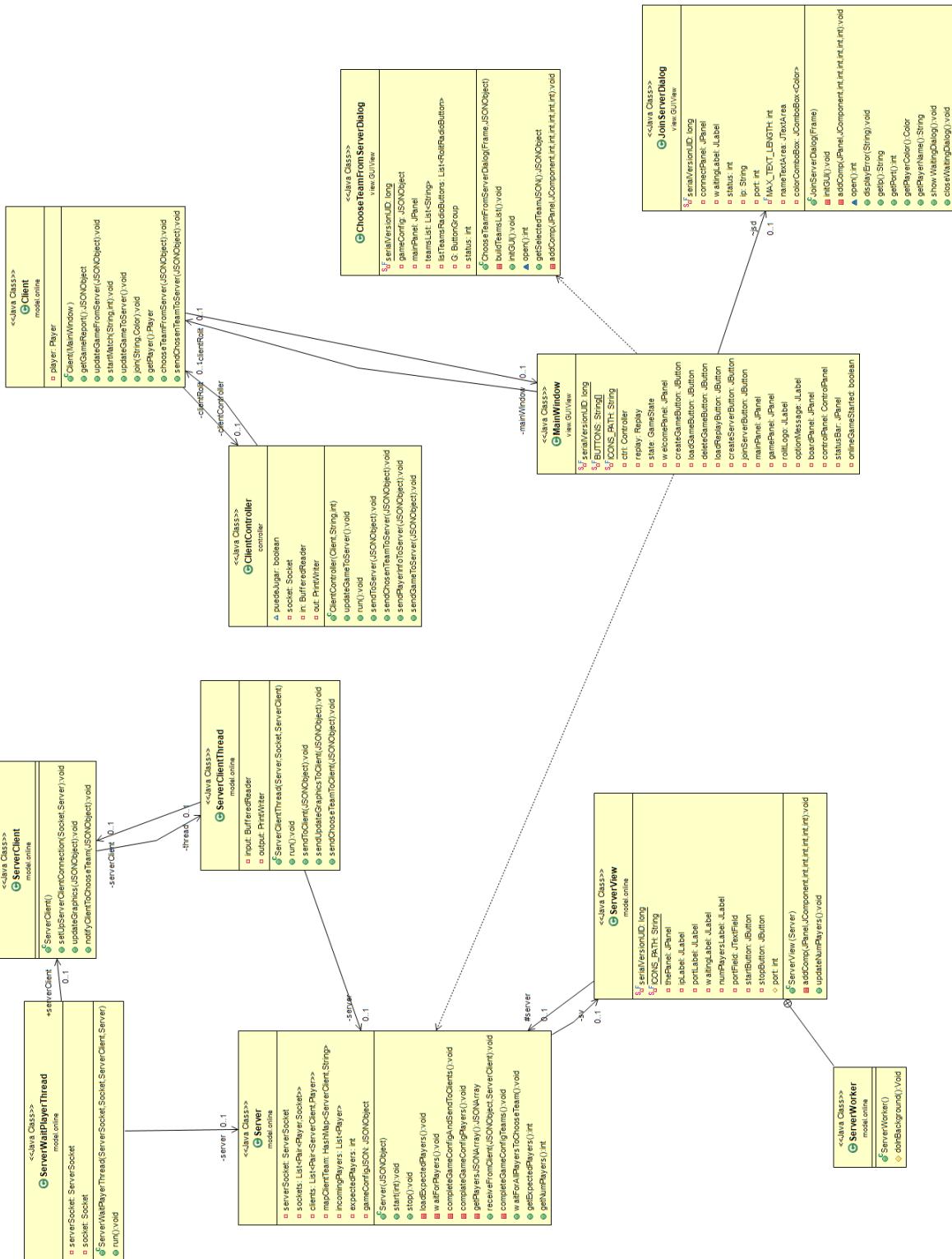
En suma, nos interesa que la llegada de mensajes no deba ser regida por un orden definido y arbitrario. Para ello, nos interesa utilizar una herramienta de la progra-

mación concurrente y paralela, los hilos; en el caso de Java, proporcionados por la clase `Thread`.

De esta manera, si el servidor tiene un hilo por cada cliente, podrá recibir mensajes de forma paralela. Así, ningún mensaje enviado desde un cliente es perdido; todos llegan al servidor con independencia de cuándo se emitan desde el cliente. Desde la perspectiva del servidor, este es el cometido de la clase `ServerClientThread` implementada en el proyecto, que extiende de `Thread`. En su método `run()`, recibe mensajes con el método `readLine()` anteriormente descrito **de forma periódica y constantemente (en un while), hasta que se haya decidido cerrar el juego**. Posteriormente, se envía este mensaje al método `receiveFromClient()`, `synchronized` (pretendemos ejecutar múltiples procesamientos que de forma secuencial para evitar errores imprevistos), que pertenece a la clase `Server`, que de forma sincronizada procesa este mensaje atendiendo al tipo de mensaje enviado (comprueba el campo `notification` del JSON, que puede albergar `"playerInfo"`, `"chooseTeam"`, `"updateGraphics"`).

El cliente se encuentra en una situación parecida. ¿Qué ocurre si recibe una información del servidor, la procesa y mientras se da este procesamiento, recibe otro mensaje del servidor? Necesitará el cliente, por tanto, dos hilos: un hilo que se dedique a recoger mensajes, y otro hilo que se dedique a procesarlos. De esta forma, ninguna información emitida desde el servidor es perdida.

Una vez el modelo de red ha sido completamente explicado, procedemos a detallar los detalles de implementación relativos al juego en específico.



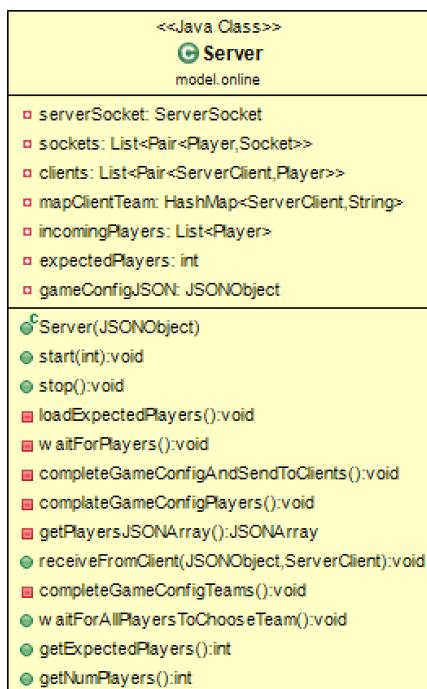
Empezamos a detallar la estructura del servidor.

ServerView, especificado en el apartado de diseño correspondiente, tiene el papel de pasar la información pertinente para el funcionamiento de la clase **Server**.

Partimos de la clase **Server**, clase que gestiona los clientes desde la perspectiva del servidor y que procesa los mensajes emitidos desde los clientes.

* Server *

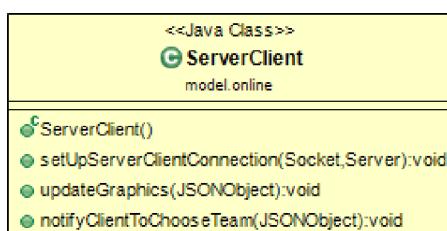
Como hemos anticipado, **Server** debe poseer un único **ServerSocket** a través del cual se abre el servidor. Posee el atributo **expectedPlayers** recibido desde la GUI, en el que el usuario que ha abierto el servidor especifica qué número de jugadores se conectarán al servidor. Es de vital importancia conocer este dato, pues de ello dependerá el número de **ServerWaitPlayerThread** creados, clase que pasaremos a comentar después.



Asimismo, **Server** posee dos listas: una correspondiente a pares Player-Socket, de modo que a cada jugador se le asocia el Socket a través del cuál puede comunicarse con el cliente en específico que juega bajo su identidad; y otra correspondiente a pares Player-**ServerClient**, donde a cada jugador se le asocia el **ServerClient** específico. Pasaremos a describir posteriormente qué es la clase **ServerClient**. En resumen, tenemos asociaciones biunívocas jugador-cliente las cuales aprovecharemos para el envío y la recepción de mensajes.

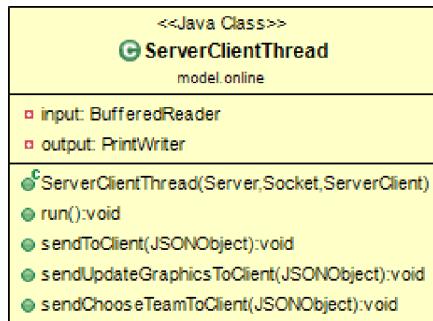
* ServerClient *

El **ServerClient** constituye una representación de un cliente en específico desde la perspectiva del servidor.



Lo fundamental de la clase `ServerClient` es que posee el *thread* encargado de la recepción directa de mensajes desde el cliente en específico; es decir, posee una única instancia de clase `ServerClientThread`, anteriormente mencionada.

* ServerClientThread *

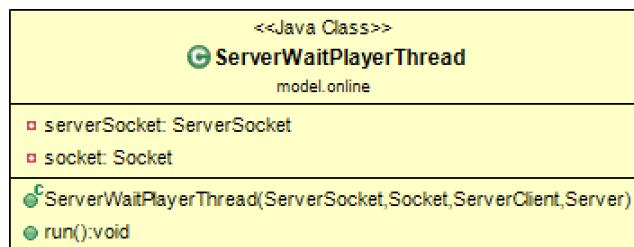


`ServerClient` posee una referencia a su `ServerClientThread` y viceversa. Esto es porque:

- `ServerClient` posee a `ServerClientThread` porque el servidor, al enviar mensajes al cliente, pasa el mensaje por `ServerClient` (recordemos la lista `Player-ServerClient`) quien pasa a enviárselo a a `ServerClientThread`.
- `ServerClientThread` posee a `ServerClient` porque al recibir mensajes desde el cliente, `ServerClientThread` envía el mensaje a `Server` para procesarlo. En esta función `server.receiveFromClient`, se necesitan dos parámetros: el mensaje, y el `ServerClient` asociado. Es por esto que para este segundo parámetro, `ServerClientThread` necesite una referencia de `ServerClient`.

Como vemos, la finalidad es encapsular el código de forma que `Server` no conozca de `ServerClientThread` sino solo de `ServerClient`.

* ServerWaitPlayerThread *



`ServerWaitPlayerThread`, por otra parte, es un hilo encargado de ir recibiendo los jugadores.

Se crean un número `expectedPlayers` (atributo de `Server`) de instancias de esta clase, cada una de ellas con un `ServerClient` asociado (por tanto, `ServerClient`

es atributo de `ServerWaitPlayerThread`). En su método `run` se encarga de ejecutar el método `serverSocket.accept()` que especificamos anteriormente. En cuanto se acepta la conexión, se procede a llamar al método `setUpServer()` de `ServerClient` para que este cree su `ServerClientThread`.

La necesidad de realizar este proceso de forma paralela justifica en gran medida la creación del hilo `ServerWaitPlayerThread`. Para evitar problemas de concurrencia, en esta clase los atributos `Server` y `ServerClient` son volátiles para que todos los hilos `ServerWaitPlayerThread` conozcan en tiempo real el estado de `Server` y `ServerClient`.

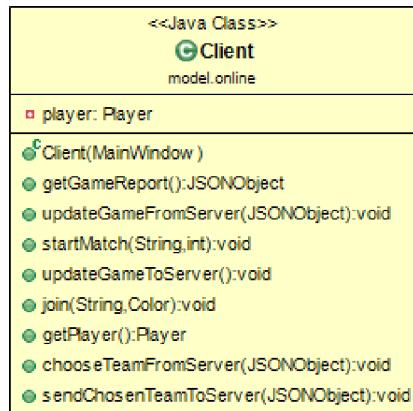
En particular, `Server` debe ser volatile pues debe ir actualizando su lista de `PlayerSocket` en tiempo real y de forma organizada a fin de evitar *bugs*. Esto es porque el método sincronizado `waitForPlayers()` de `Server` recoge periódicamente el número de conexiones aceptadas. En cuanto estas conexiones igualan el número de conexiones aceptadas, el método deja de ejecutarse y se procede a las gestiones que tiene que realizar el servidor una vez todos los usuarios esperados se han conectado.

Una vez detallados los cometidos de todas las clases que posee el servidor, pasamos a detallar las del cliente.

El punto de partida es la clase `Client`, que será un intermediario entre su *thread* de recepción de mensajes y la GUI.

* Client *

Como `Client` es un intermediario entre el *thread* de recepción de mensajes y la GUI, `MainWindow` precisa una referencia de `Client` y `Client` una de `MainWindow`; es decir, la comunicación es bidireccional. La razón de esto es:



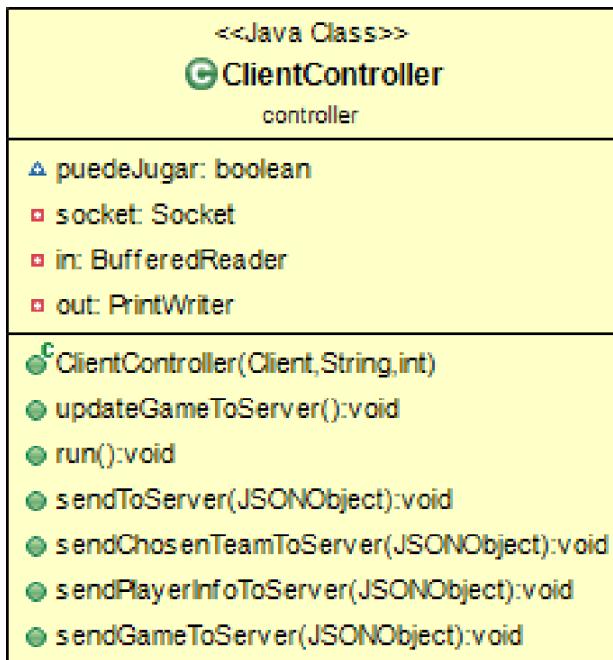
- Al hacerse una jugada en el modelo, como se ha especificado, necesitamos pasar el nuevo estado del juego al servidor para que este, a su vez, se lo pase al resto de clientes. Por tanto, `MainWindow` pasa el estado del juego a su `Client`. `MainWindow`, a su vez, obtiene este estado del juego desde el controlador. Todos estos pasos de información se realizan a través de los métodos `updateGameToServer()` que poseen estas clases.

- Al recibirse un nuevo estado del juego desde el servidor, Client recibe esta información desde su *thread*. Necesita a MainWindow para que este, a su vez, envíe el nuevo estado del juego al controlador con el fin de actualizar el modelo al nuevo juego requerido. Todos estos pasos de información se realizan a través de los métodos updateGameFromServer() que poseen estas clases.

Finalmente, pasamos a describir ClientController, el *thread* de Client.

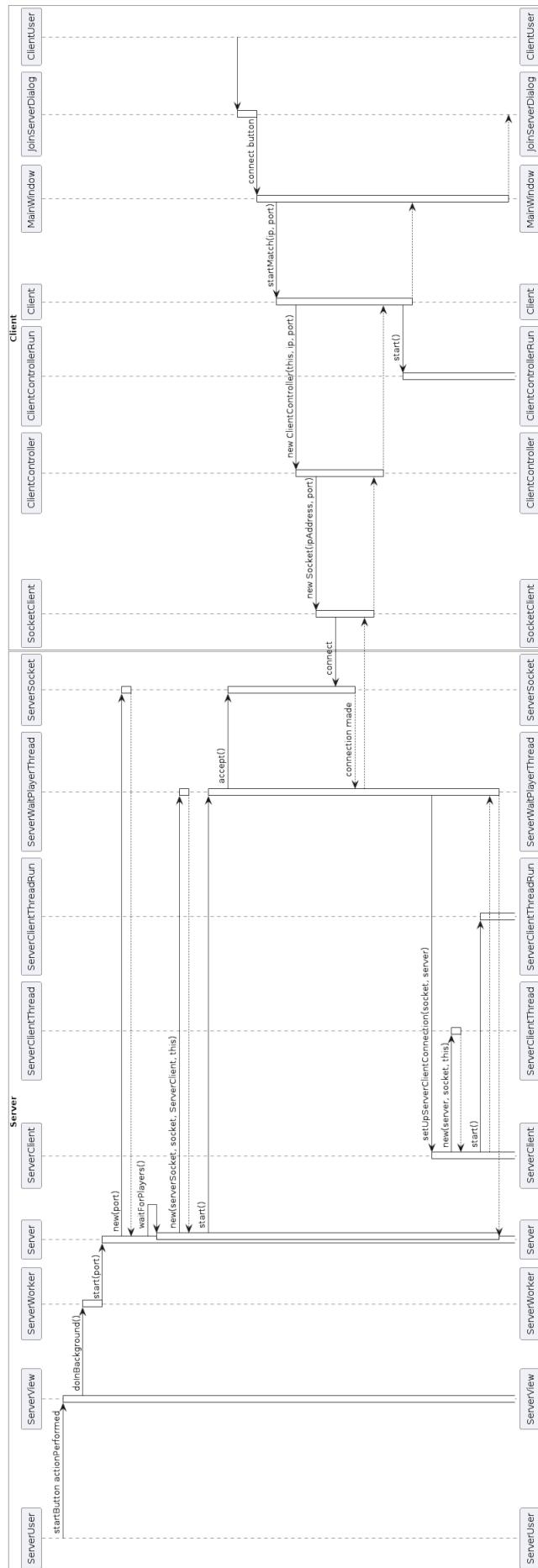
* ClientController *

Nuevamente la comunicación entre Client y ClientController es bidireccional: para enviar información al servidor, Client notifica a ClientController; para recibir información del servidor, ClientController notifica a Client.



Como hemos anticipado, ClientController extiende de Thread, y en su método run() recibe información del servidor que procede a enviar al cliente. El método sendToServer envía la información directamente al servidor por medio de out.println() como hemos descrito; este método es llamado por una serie de métodos -sendChosenTeamToServer, sendPlayerInfoToServer, sendGameToServer-, los cuales introducen el campo notification en el JSON, con el valor chooseTeam, playerInfo y updateGraphics respectivamente. El objetivo es que el servidor pueda distinguir la naturaleza del mensaje enviado y hacer su tratamiento específico en base a ello.

Una vez especificadas todas las clases, describiremos el hilo típico de la ejecución para comprender cómo operan estas y en el orden en el que lo hacen.



Cuando se pulsa a crear un servidor, el usuario rellena un `CreateGameDialog` para especificar una configuración del juego básica especificada. Una vez especificada, se crea el objeto `Server`, que tendrá en su atributo `gameConfig` dicha información. Finalmente, en este constructor se abre el diálogo de crear un servidor; el usuario *host* del servidor debe especificar el puerto en el que el servidor debe operar. Finalmente, se pulsa el botón `Start Server`.

Observamos que al pulsar el botón de `Start Server` en el diálogo de abrir un servidor, se crea un `ServerWorker`, que es un `SwingWorker` (encargado de correr un *thread*, el del servidor, de forma concurrente al de *Swing* -diálogo de abrir servidor). Dicho `ServerWorker` llama al método `start(port)` de `Server`, que crea el `ServerSocket` y posteriormente, llama al método `waitForPlayers()` para realizar las conexiones oportunas.

Este método se compone de un bucle `for` que recorre un número de vueltas equivalente al del número de jugadores a conectarse esperados. por cada vuelta, se crea su `ServerClient` y `Client` asociados, y se crea el *thread* `ServerWaitPlayerThread` y se llama a `start()`. El `ServerWaitPlayerThread`, en su método `run()` llamado desde `start()`, espera en el método de `serverSocket.accept()` que se conecte un cliente.

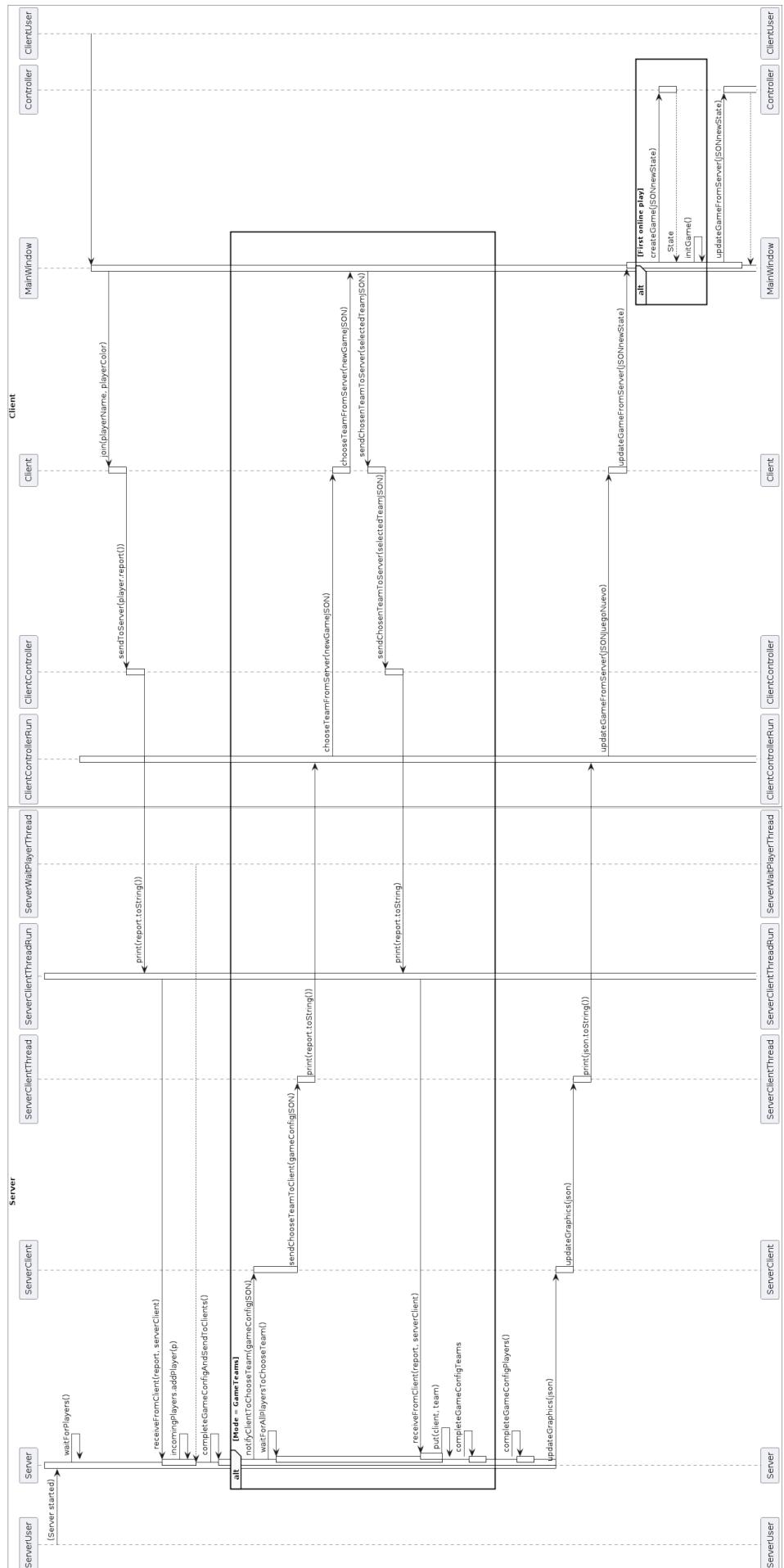
Un cliente, por medio de `JoinServerDialog`, se conecta al servidor desde la IP y puerto requeridos. Pasa por el `MainWindow` quien a su vez llega al cliente (`Client`) con el método `startMatch`, en base a las relaciones ya descritas. El cliente crea su *thread* `ClientController`. En la constructora de `ClientController` se realiza la conexión al servidor mediante la creación del `Socket`.

En cuanto esta se realiza con éxito, ocurren dos procesos de forma paralela:

- En el servidor, el método `serverSocket.accept()` es ejecutado con éxito. Se procede a llamar al método `setUpServer()` del `ServerClient` específico asociado a esta conexión; se crea después un *thread* `ServerClientThread` el cual crea los `BufferedReader` y `PrintWriter` oportunos, y `ServerClient` procede a llamar al `run()` de este *thread* (representado en la lifeline `ServerClientThreadRun`)
- En el cliente, una vez se ha creado el `Socket` con éxito en la constructora de `ClientController`, se crean los `BufferedReader` y `PrintWriter` oportunos. Se termina la constructora, y acto seguido `Client` llama al `start` de `ClientController` para que empiece a recibir mensajes en su método `run()`.

Si la conexión no se ha realizado bien, se generan las excepciones oportunas que cierran las ventanas de tanto cliente como servidor.

El *thread* `ServerWaitPlayerThread` procede a esperar un segundo antes de cerrarse. En este segundo, se espera que el cliente envíe al servidor la información relativa al `Player` (qué color y qué nombre ha escogido). Esta información pasa a registrarse en la lista de `incomingPlayers` de forma síncrona mientras `ServerWaitPlayerThread` espera. El mecanismo pormenorizado es el siguiente:



1. Tras ejecutarse el hilo `ClientController`, el método `startMatch` de `Client` se finaliza. `MainWindow` procede a ejecutar el método `join` de `Client`, pasándole como argumentos el nombre y color escogidos.
2. `Client` pide al thread `ClientController` que envíe al servidor esta información.
3. `ClientController` envía la información por medio del método `sendToServer`, adjuntándole previamente en el `JSON` la notificación `playerInfo` para que el servidor conozca la naturaleza de esta información (proporcionar la información del jugador cliente)
4. El `ServerClientThread` específico asociado al cliente que envía esta información recibe en su método `run()` el mensaje. Procede a llamar al método asíncrono `receiveFromClient` de `Server`.
5. `receiveFromClient` distingue, observando el `JSONObject` enviado, si se trata de información de actualización de juego, información respecto a un nuevo jugador añadido, o información sobre el equipo escogido observando el campo notificación del `JSON`. En el caso que nos ocupa, es esta segunda opción; `receiveFromClient()` procede a crear una instancia de `Player` y añadirla a `incomingPlayers` (`ArrayList` de `Player`).

Todo este proceso tarda (bastante) menos del segundo que espera `ServerWaitPlayerThread`; se deja un segundo de cortesía para dar un amplio margen a los equipos más lentos.

El método `run()` de `ServerWaitPlayerThread` se finaliza y se ejecuta desde `waitForPlayers` (`Server`) el método `join()` para que concluya el *thread*.

Dado la información obtenida de `Socket`, `ServerClient` y `Player` (este último con `incomingPlayers`, cuyo último elemento añadido es el `Player` correspondiente al cliente), se rellenan las dos listas `Player-Socket` y `ServerClient-Player` de `Server` con una nueva posición.

Todo esto es una única vuelta del bucle de `waitForPlayers()`. Como hemos comentado, se ejecutarían un número de vueltas equivalente al número de jugadores esperados.

Una vez todos los jugadores se han conectados y las dos listas completadas, se sale del bucle y el método `waitForPlayers()` llama después a `completeGameConfigAndSendToClients()`, encargado de llenar el `gameConfig`. Este último método primero tiene que verificar si se ha escogido el modo por equipos, mirando en el `gameConfig`. De ser así, se hace una operación adicional antes de proseguir.

* Modo por equipos *

En caso de que se haya especificado el modo por equipos, se procede a llamar a una serie de métodos cuyo fin es notificar a los usuarios para que escojan un equipo; una vez todos los usuarios han escogido equipo, se añade esta información al `gameConfigJSON`; posteriormente, como en la implementación de sprints anteriores, procede a completarse el `gameConfigJSON` con la información de los jugadores.

Primero, como observamos, se llama al método `notifyClientToChooseTeam` de todos los `Client`; este, por medio de `ServerClientThread` envía el `gameConfigJSON`, con el campo `chooseTeam` como notificación (explicaremos posteriormente cómo funcionan las notificaciones), al cliente.

Cada cliente en su respectivo `ClientController` recibe esta información. Observa la naturaleza de la notificación; como esta es `chooseTeam`, procede a abrir un diálogo `ChooseTeamFromServerDialog` para escoger uno de los equipos ofrecidos desde el servidor. Nótese que la información de los equipos viene proporcionada en el `gameConfigJSON` enviado desde el servidor.

Una vez todos los clientes han sido notificados, se llama a `waitForAllPlayersToChooseTeam()`. Su cometido es el de detener el hilo de ejecución hasta que todos los jugadores han escogido equipo y el servidor conoce sus elecciones.

Además, como puede verse, se ha introducido como nuevo atributo del servidor un `HashMap` de `ServerClient` - Nombre del equipo escogido. Este mapa se actualiza en tiempo real conforme los clientes envían al servidor su elección de equipo. Cada medio segundo se comprueba si todos los jugadores han elegido equipo; de ser así, finaliza el método y se sigue con la ejecución de `completeGameConfigAndSendToClients()`.

¿Cómo los clientes notifican al servidor su elección? Una vez un cliente elige equipo, manda al servidor un `JSONObject` con el nombre del equipo elegido en el campo `team`. Antes de enviarlo, pone en el `JSON` un campo `notification` con el valor `chooseTeam`; para que así el servidor pueda conocer las intenciones del envío de la información por parte del cliente (notificar que se ha escogido un equipo, y cuál).

El servidor recibe un `JSON` con la notificación `chooseTeam`; en `receiveFromClient (Server)` observa que al ser la notificación `chooseTeam`, debe añadir una nueva entrada al mapa `ServerClient` - Nombre del equipo escogido.

Una vez todos los equipos han sido escogidos, finaliza el método `waitForAllPlayersToChooseTeam()`. Se procede a llamar al método `completeGameConfigTeams()` para añadir al `gameConfigJSON` toda la información pertinente de los equipos, ahora que se posee toda.

Ahora, independientemente de que se haya escogido el modo por equipos o no, se procede a crear rellenar el incompleto `gameConfigJSON` que fue pasado con anterioridad por el constructor de `Server`; ahora es el momento correcto para hacerlo pues ya se conoce la información de todos los `Player`. Se llenan los campos pertinentes previamente incompletos de `gameConfigJSON`; posteriormente, se envía este primer estado del juego a todos los clientes por medio del método `updateGraphics()`.

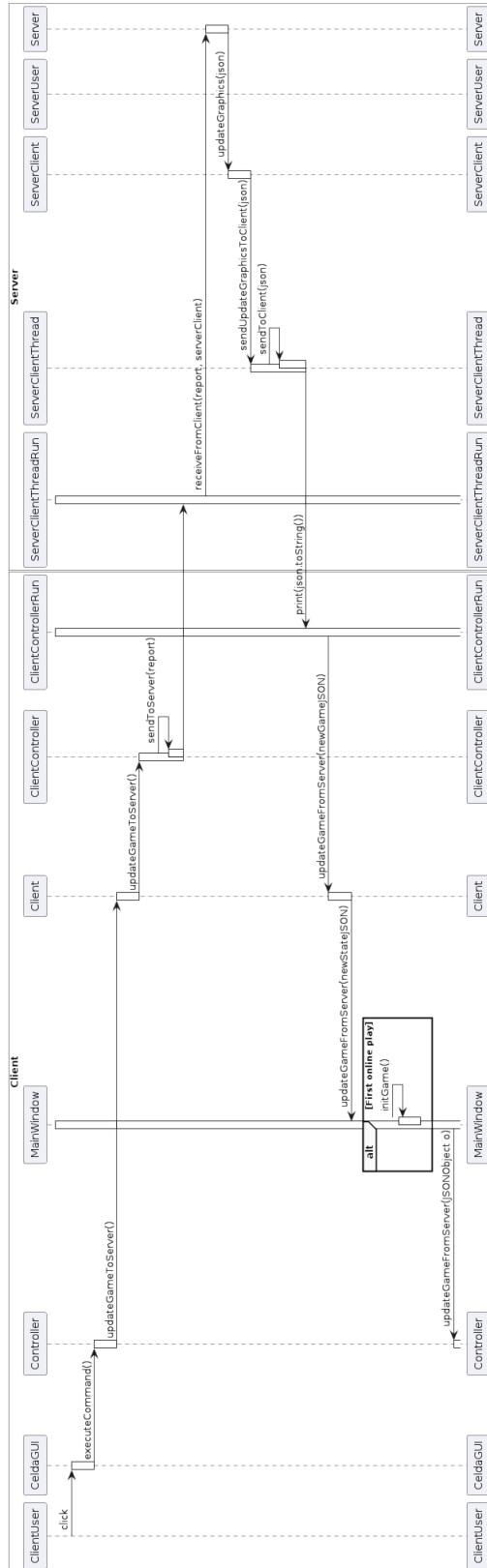
El cliente recibe esta configuración del juego en su *thread*, se llama al método `updateGameFromServer()` de `Client`, quien a su vez llama a `updateGameFromServer()` de `MainWindow`.

En este último método, si es la primera jugada, se llama a crear el juego en el controlador, y después se llama a `initGame()` para inicializar las componentes visuales relativas al juego.

En cualquier caso, se llama después al método `updateGameFromServer()` de `Controller` para actualizar el juego tal y como ordena el `JSON` pasado como parámetro. Los mecanismos de este método son crear un nuevo juego a partir del nuevo `JSON` estado

del juego, pasar los observadores del Game antiguo al nuevo, y adjudicar al atributo Game de Controller este nuevo juego.

Falta por especificar cómo se produce una jugada en el modo online y cómo evitar que los clientes jueguen cuando no es su turno.



Se pulsa una celda en el tablero y se pide al controlador que ejecute el comando de poner cubo en la posición pedida, como de costumbre. En caso de que se juegue el modo online, se comprueba si el jugador está legitimado para jugar (es decir, es su turno). Para ello, se comprueba que el jugador turno del juego es el mismo que el jugador del cliente (recordemos que `Client` tiene el atributo `Player` del jugador al que corresponde).

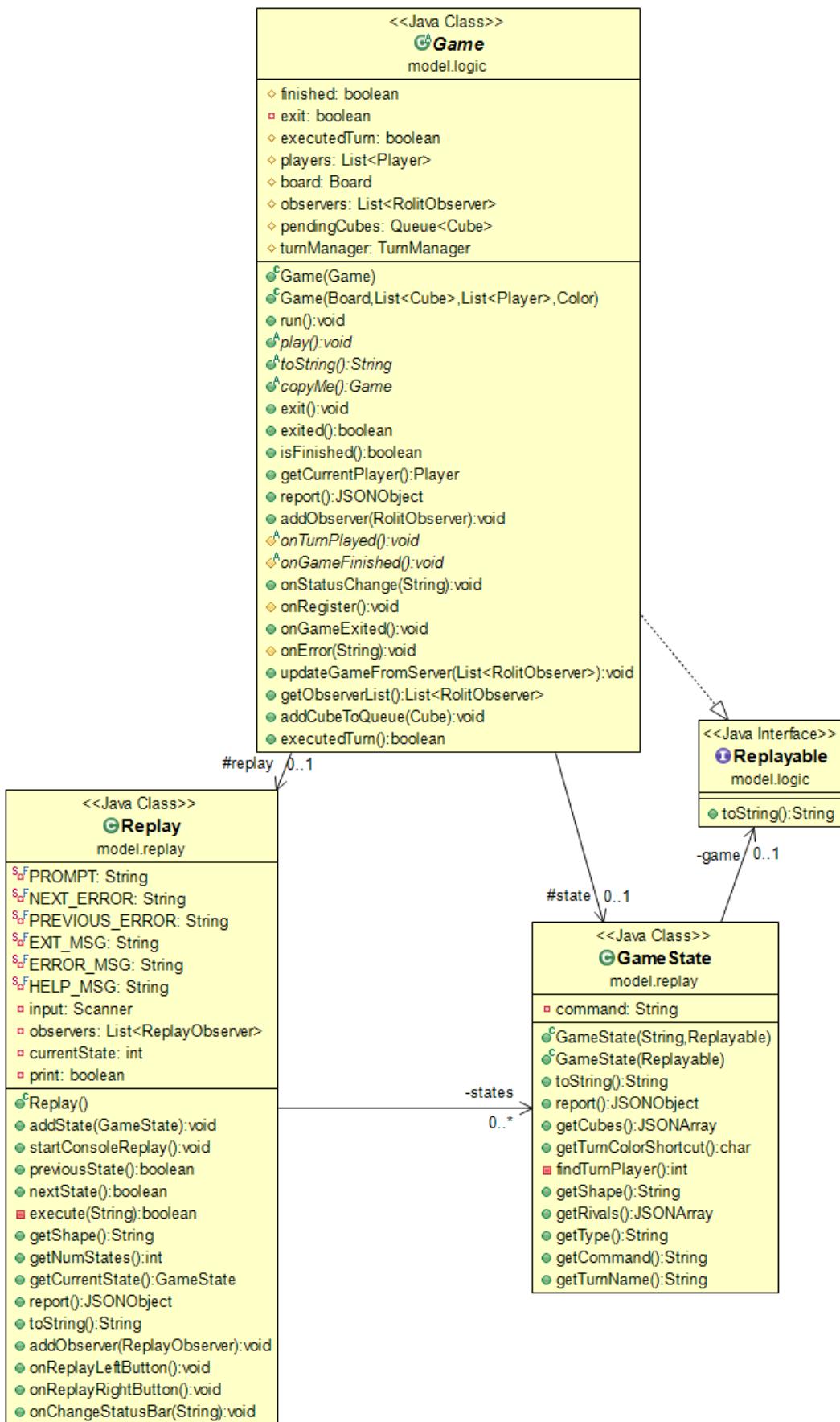
De verificarse estas dos condiciones, se ejecuta el comando y se envía el nuevo estado del juego al servidor, quien enviará este nuevo estado al resto de clientes.

De no jugarse el modo online, se ejecuta el comando como de costumbre.

Nótese que este último diagrama de secuencia se representa a `Client` como entidad abstracta emisora y receptora, realmente un `Client` sería emisor y el resto de `Client` serían receptores.

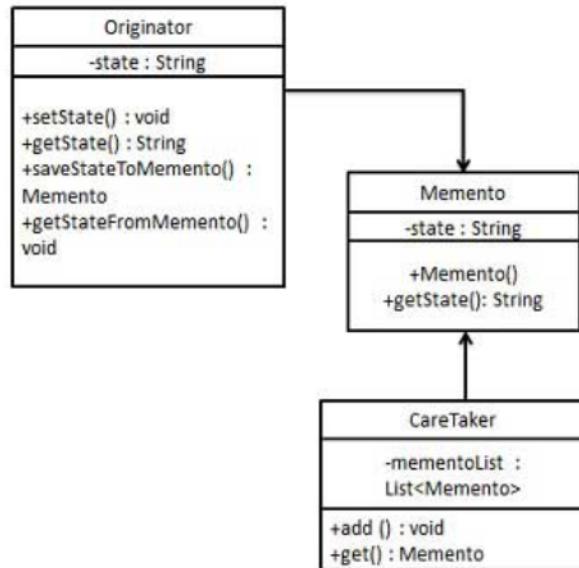
Si se escoge el modo por equipos cuando se crea una partida, el `gameConfigJSON` pasado como parámetro ahora adoptará la estructura de report de juego por equipos, tal y como especificaremos en el apartado de reports.

5.1.5 Replay



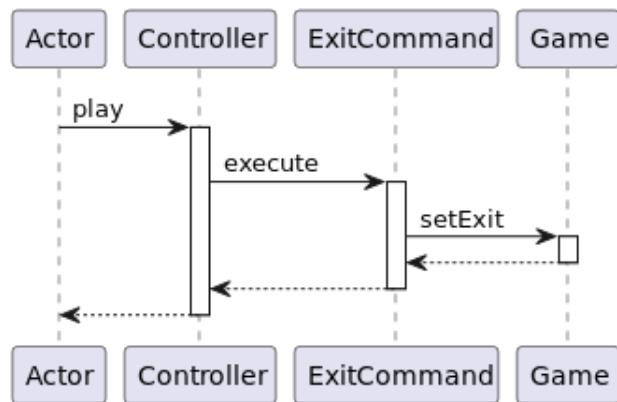
El paquete *replay* contiene a las principales clases involucradas en el funcionamiento de las repeticiones de partidas, que son *Replay* y *GameState*. No obstante, la clase *Game* juega un papel fundamental en la creación de una repetición.

La implicación de *Game* en esta funcionalidad es una consecuencia directa de haber empleado una adaptación del patrón *Memento* en su diseño.



Patrón *Memento*

Así pues, la clase *Game* es el *Originator*, *GameState* es el *Memento* y *Replay* el *CareTaker*. Para cada momento del juego, la clase *Game* genera un *GameState* y lo añade a la lista que contiene *Replay*.



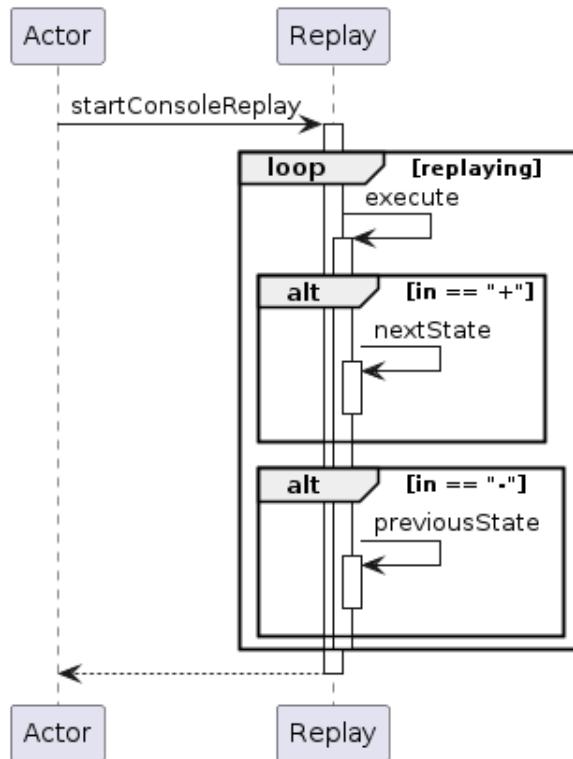
* Replay *

La clase *Replay*, además de funcionar como contenedora de los *GameState*, es la responsable de navegar a través de ellos, convirtiéndose así en un modelo independiente de *Game* para reproducir repeticiones, pues cuando se inicia el programa *Rolit*, o bien se juega una partida o se visualiza una *replay*.

Como todo modelo, debe comunicarse con la vista de alguna manera. En este caso se ha optado por utilizar el Patrón *Observador*. Aquellas clases que necesiten recibir notificaciones de **Replay** tendrán que implementar la interfaz **ReplayObserver**.



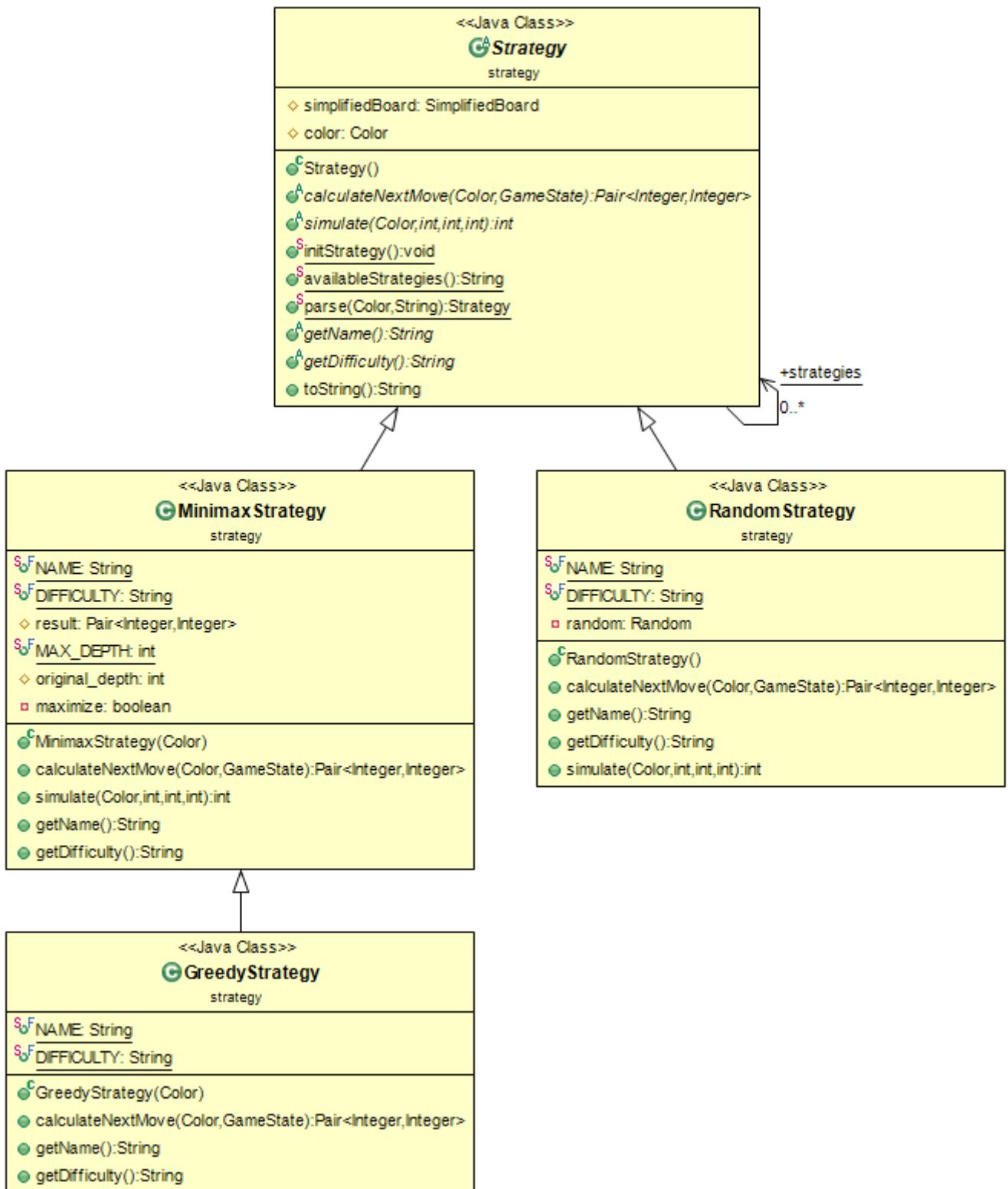
Si bien para la Interfaz Gráfica se utiliza este patrón, la clase autogestiona su vista de Consola, para ello cuenta con el método **startConsoleReplay()**.



* GameState *

La clase **GameState** representa un instante de **Game**. Esto se consigue mediante dos atributos. El primero de ellos se llama **game** y es del tipo **Replayable**. La interfaz **Replayable** cuenta con los métodos **toString()** y **report()**, mediante los cuales puede representarse un momento al completo, pues permiten imprimir un instante del juego en consola y tener acceso a la serialización del juego, respectivamente.

5.1.6 Strategy



* Strategy *

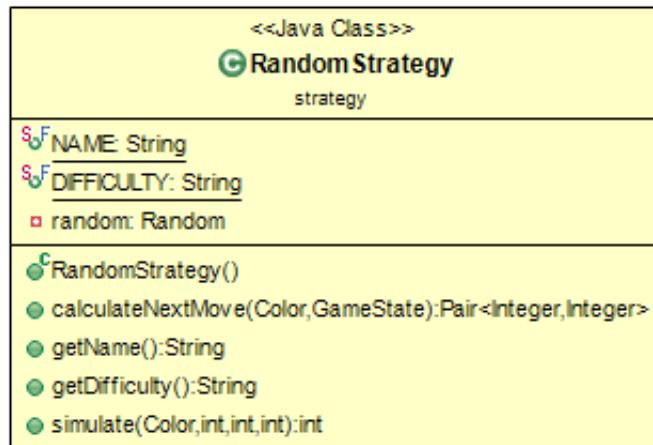
Para las estrategias de las inteligencias artificiales se emplea el **patrón estrategia**.

Las estrategias disponibles son las siguientes:

- RandomStrategy: Estrategia donde el jugador elige una posición aleatoria del tablero para colocar un cubo (estrategia de nivel fácil).
- GreedyStrategy: Estrategia en la cual el jugador busca la posición que maximiza sus puntos en el turno actual (estrategia de nivel medio).
- MinimaxStrategy: Estrategia basada en el algoritmo Minimax y la poda Alfa-Beta (estrategia de nivel alto, explicaciones detalladas en los anexos [Anexo III](#) y [Anexo IV](#))

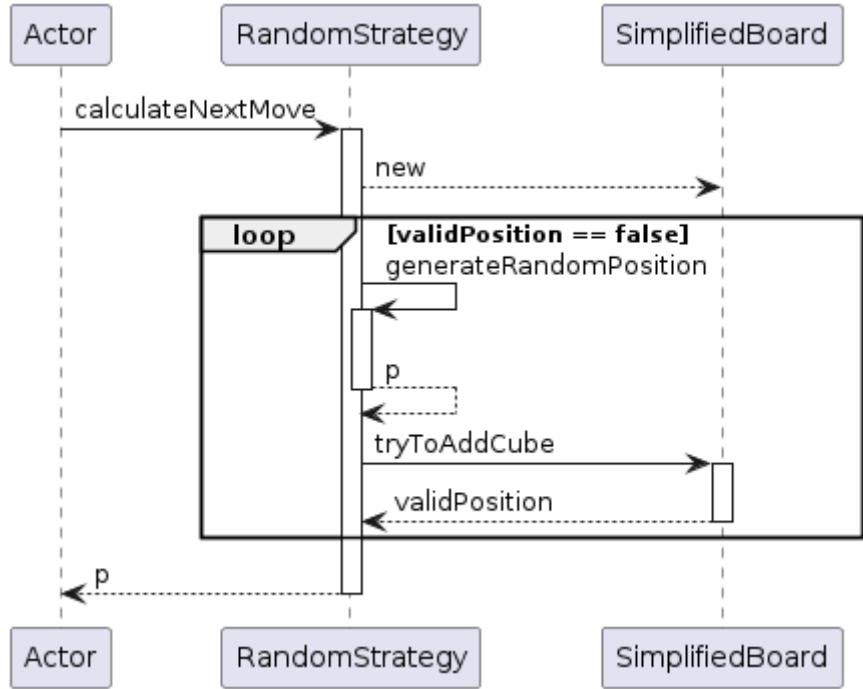
Como se ve en los parámetros de diversas funciones, todas las clases de este paquete obtienen la información necesaria a través de instancias de la clase `GameState`, de forma que se preserva la encapsulación en todo momento.

* RandomStrategy *

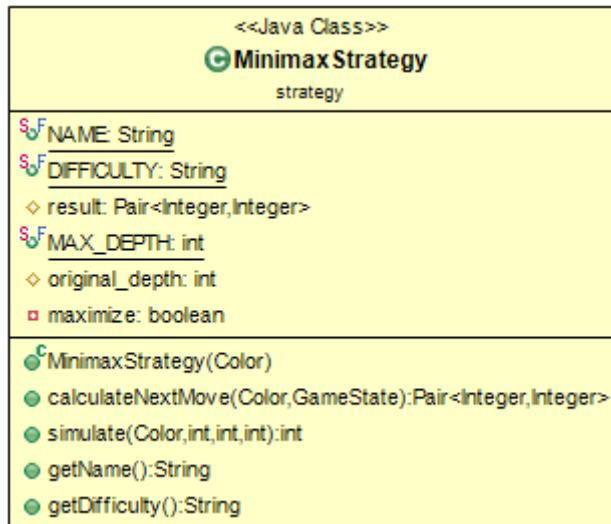


El método principal de esta clase es `calculateNextMove()`, en el cual se inicia la simulación con el método `simulate()`. En este, se generan aleatoriamente posiciones hasta que se da con una posición válida acorde al estado actual del tablero. En el momento en que se da con una posición válida se devuelve esa posición como resultado de la simulación.

Este funcionamiento se ve en el siguiente diagrama:



* MinimaxStrategy *



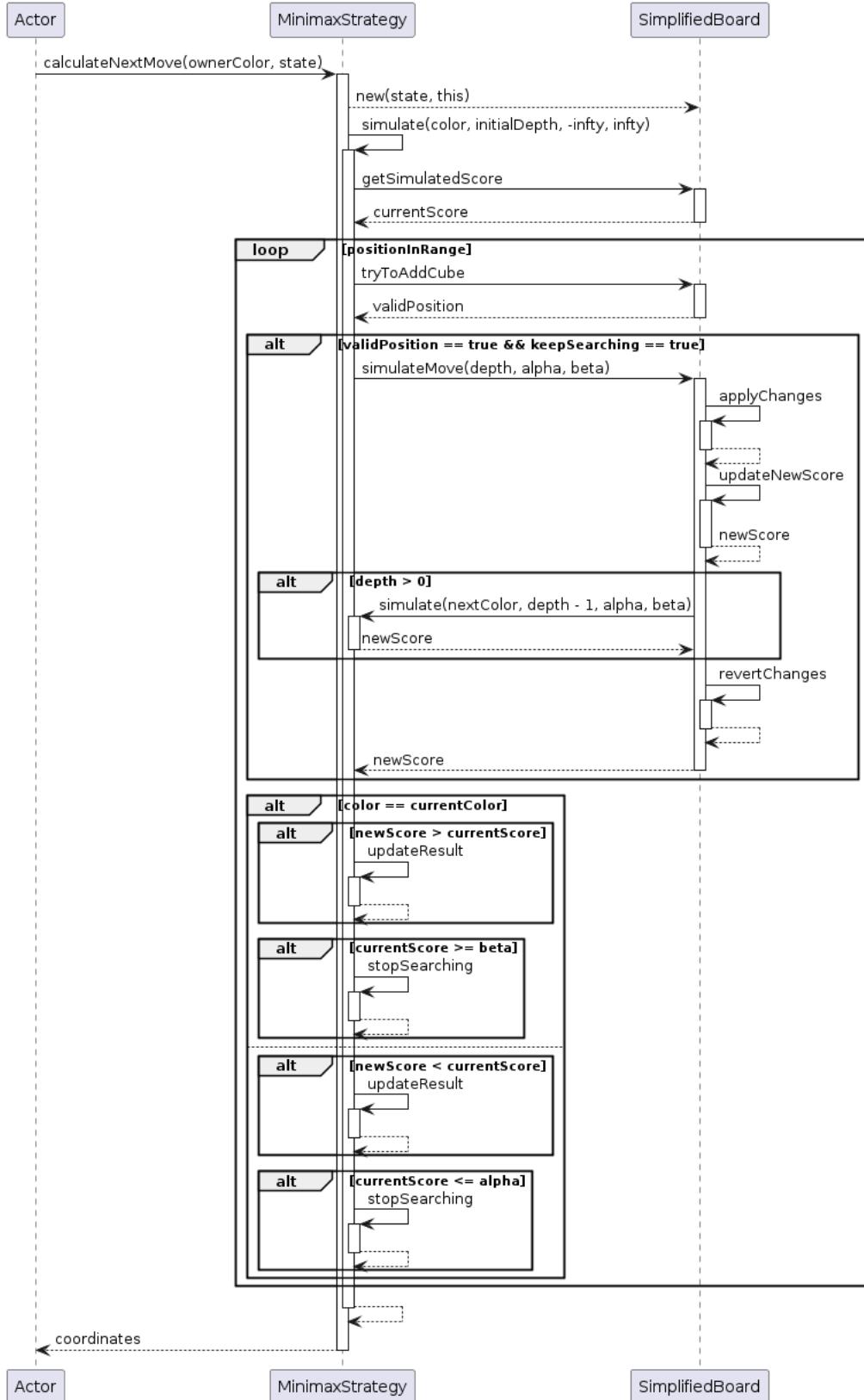
Para entender el propósito y funcionamiento de esta clase, es importante leer antes los anexos [Anexo III](#) y [Anexo IV](#).

Cada instancia de la clase **MinimaxStrategy** está asociada a un jugador (el propietario) para el cual intenta maximizar la puntuación al hacer simulaciones.

El método fundamental de esta clase es **calculateNextMove()**. En este método se inicia la simulación, con el método **simulate()**, en el que se hacen recorridos sobre las casillas del tablero, haciendo simulaciones de movimientos en ellas hasta una profundidad de búsqueda determinada y valorando los resultados según si el turno que está siendo simulado es del jugador propietario de la estrategia o de otro.

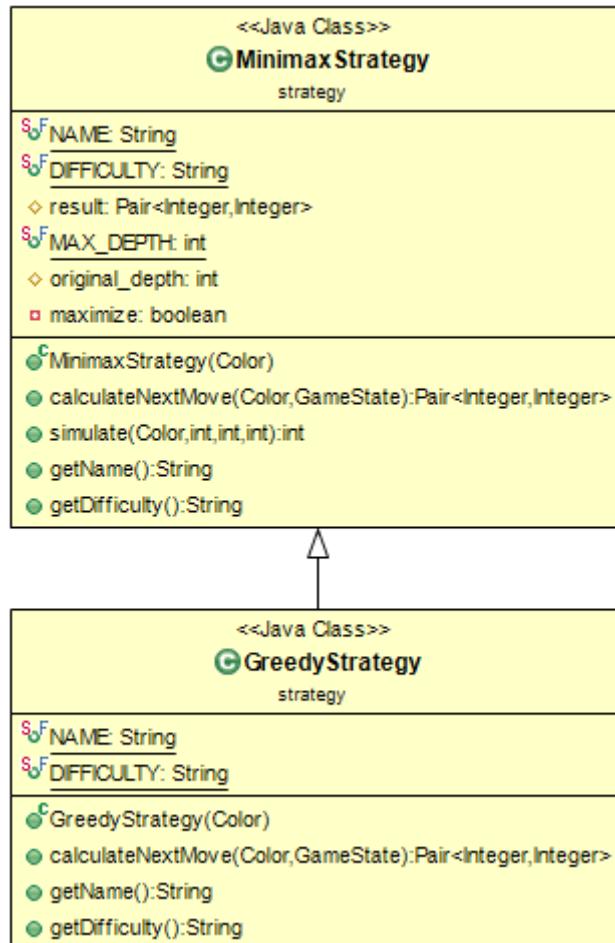
A parte, se evalúan los resultados tras cada simulación para comprobar si seguir o no con las simulaciones acorde a la poda Alfa-Beta.

El funcionamiento de este método se refleja en el siguiente diagrama:



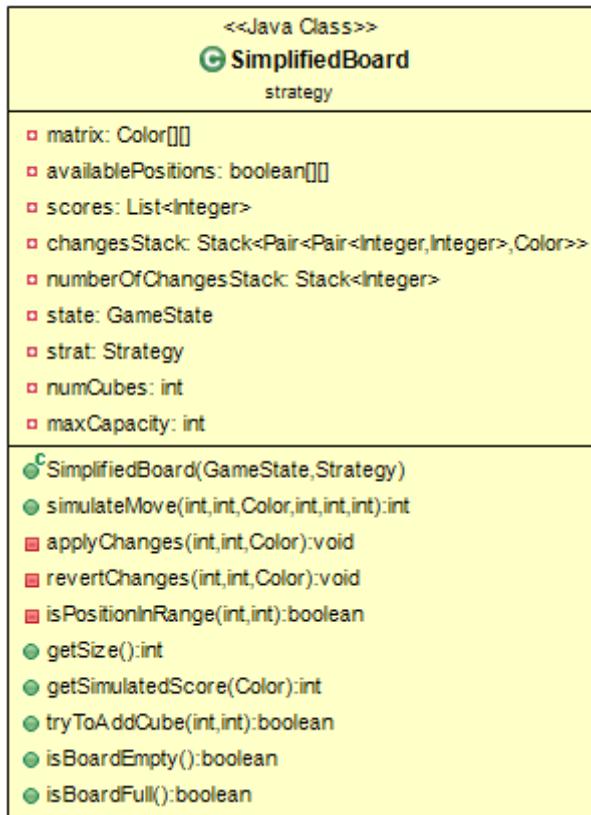
El único caso en el que no se aplica el mismo criterio de cálculo es si el movimiento a simular es el primero de la partida, en cuyo caso se genera una posición aleatoria.

* GreedyStrategy *



Dada la clase **MinimaxStrategy**, el funcionamiento de la clase **GreedyStrategy** se limita al definido por la clase anterior, pero fijando la profundidad máxima de simulación a 1. Esto funciona dado que en el método **calculateNextMove** se empieza simulando un movimiento para el jugador propietario, y dado que la puntuación de este es la que se busca maximizar, al limitar la simulación a un único nivel de profundidad de búsqueda, el resultado de la búsqueda es la posición que garantiza más puntos al jugador en el turno actual.

* SimplifiedBoard *



SimplifiedBoard surge por la necesidad de una representación puramente funcional del tablero para hacer simulaciones en los cálculos de las estrategias.

Esta clase consta de una matriz en la que almacena el color de los cubos del tablero real. Para disminuir costes y evitar tener que hacer copias del tablero tras cada movimiento simulado, se lleva una pila con los cambios que se realizan al simular un movimiento, de forma que cuando se quiere dejar el tablero en el estado previo a la simulación para realizar otra simulación, en vez de realizar una copia se revierten los cambios aplicados, lo cual resulta mucho menos costoso.

En **SimplifiedBoard** también se almacenan los puntos de los distintos jugadores, puesto que la idea es consultar los puntos después de simular cada movimiento.

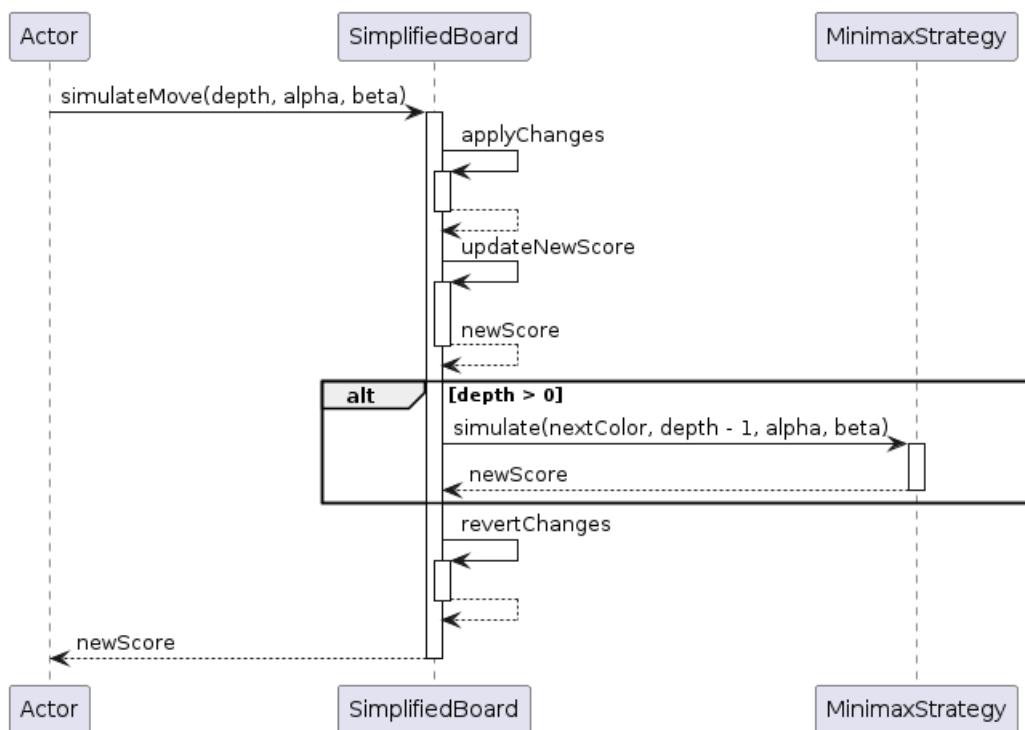
Ahora, para calcular el mejor movimiento para ejecutar, en la clase de la estrategia se realiza un bucle en el que se recorren todas las posiciones del tablero, consultando si cada posición es válida o no, y en caso de dar con una posición válida, se simula ese movimiento.

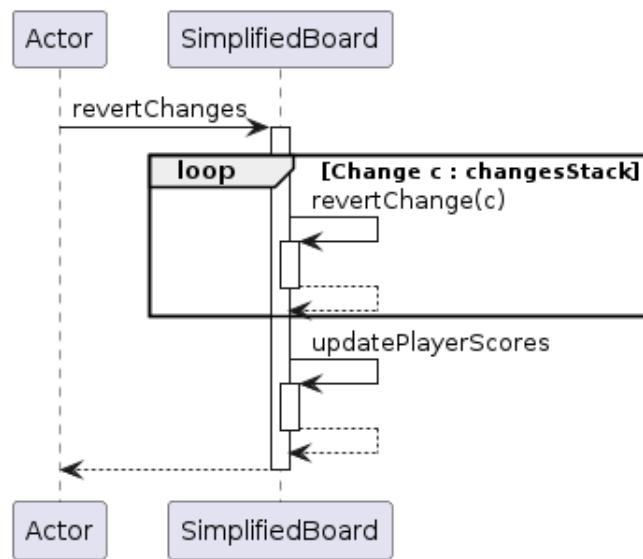
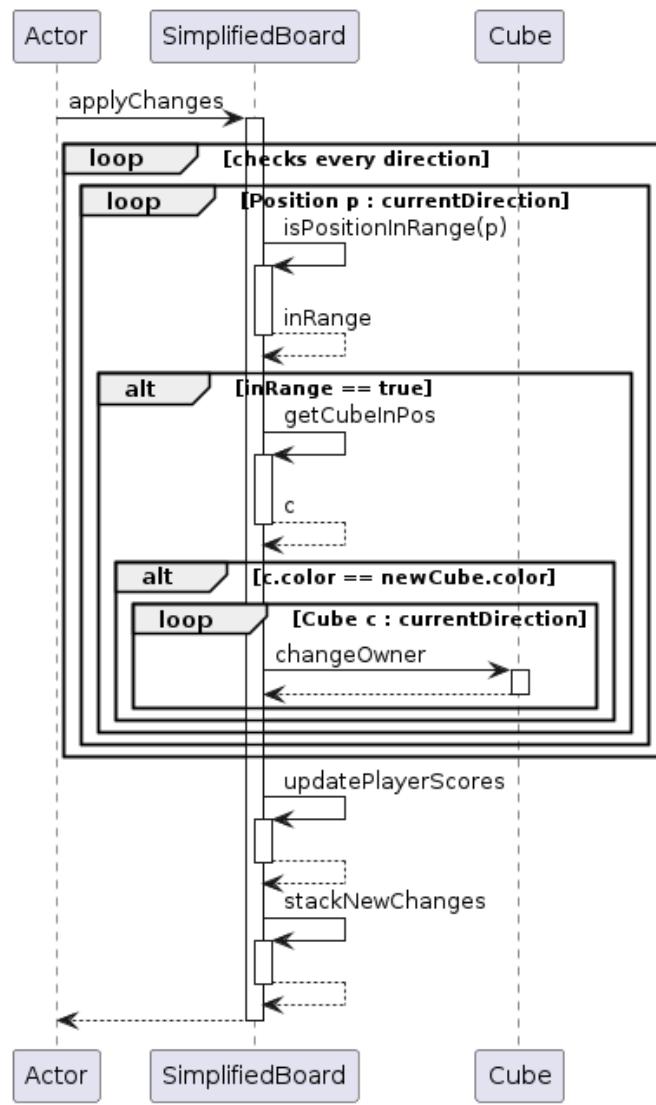
Dentro de la simulación, en **SimplifiedBoard**, si la profundidad a explorar es mayor que 0, antes de revertir los cambios se vuelve a realizar el bucle de las posiciones, pero simulando esta vez para el siguiente jugador, y así hasta que la profundidad a explorar es 0. Hay que tener en cuenta que el jugador propietario de la estrategia busca maximizar sus puntos, mientras que el resto de jugadores buscan minimizarlos. Por tanto, en los bucles de recorrido de posiciones, la estrategia es conocedora de para qué jugador está simulando el siguiente movimiento, de forma que si está simulando para el jugador propietario devolverá el resultado más favorable, y si está simulando

para cualquier otro jugador devolverá el resultado más perjudicial posible para el propietario. De esta forma, se podrá conocer el resultado final realista de cada jugada posible, y así elegir la mejor jugada para el jugador propietario.

La simulación en general se hace en el método `simulateMove()`, en el cual se llama al método `applyChanges()` para aplicar los cambios resultantes de colocar un nuevo cubo, y a `revertChanges()` para revertirlos una vez se han hecho todos los cálculos resultantes de la simulación actual.

Mostramos el funcionamiento de los métodos mencionados en los siguientes diagramas:





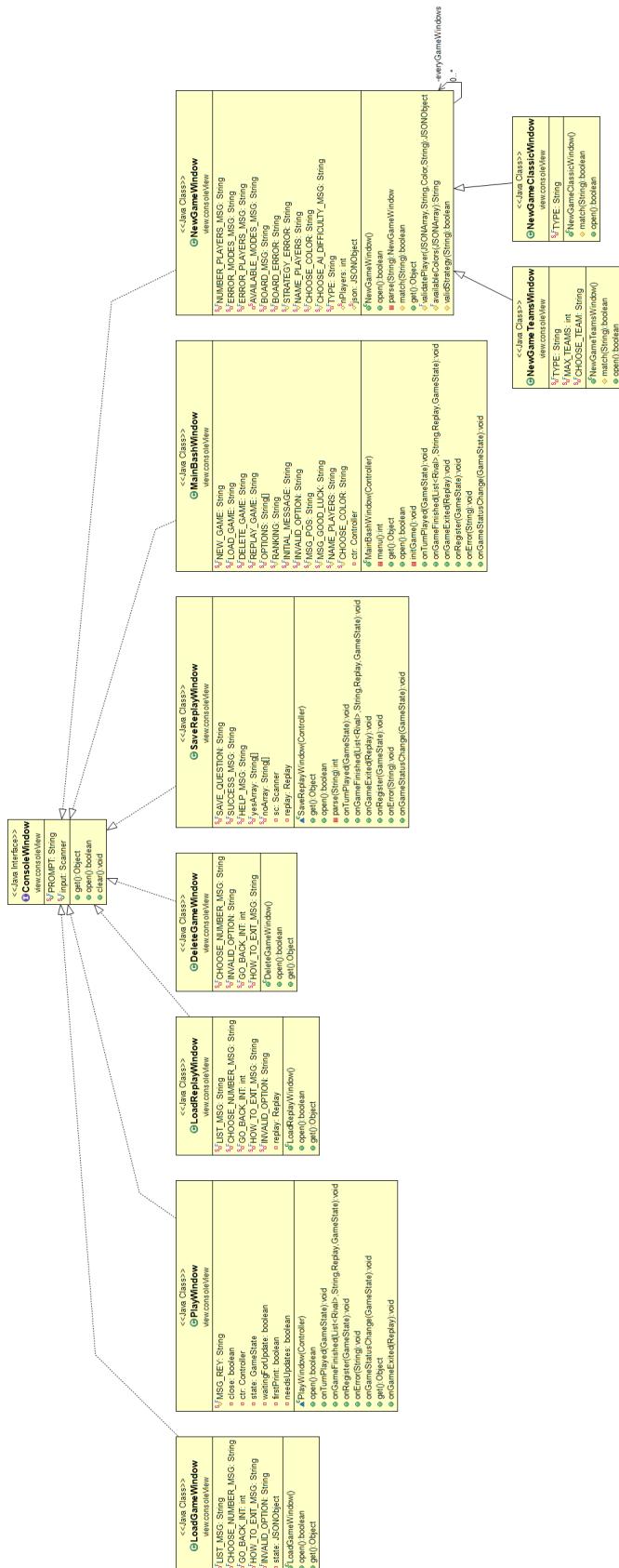
5.2 Paquete Controller

5.3 Paquete View

5.3.1 ConsoleWindow

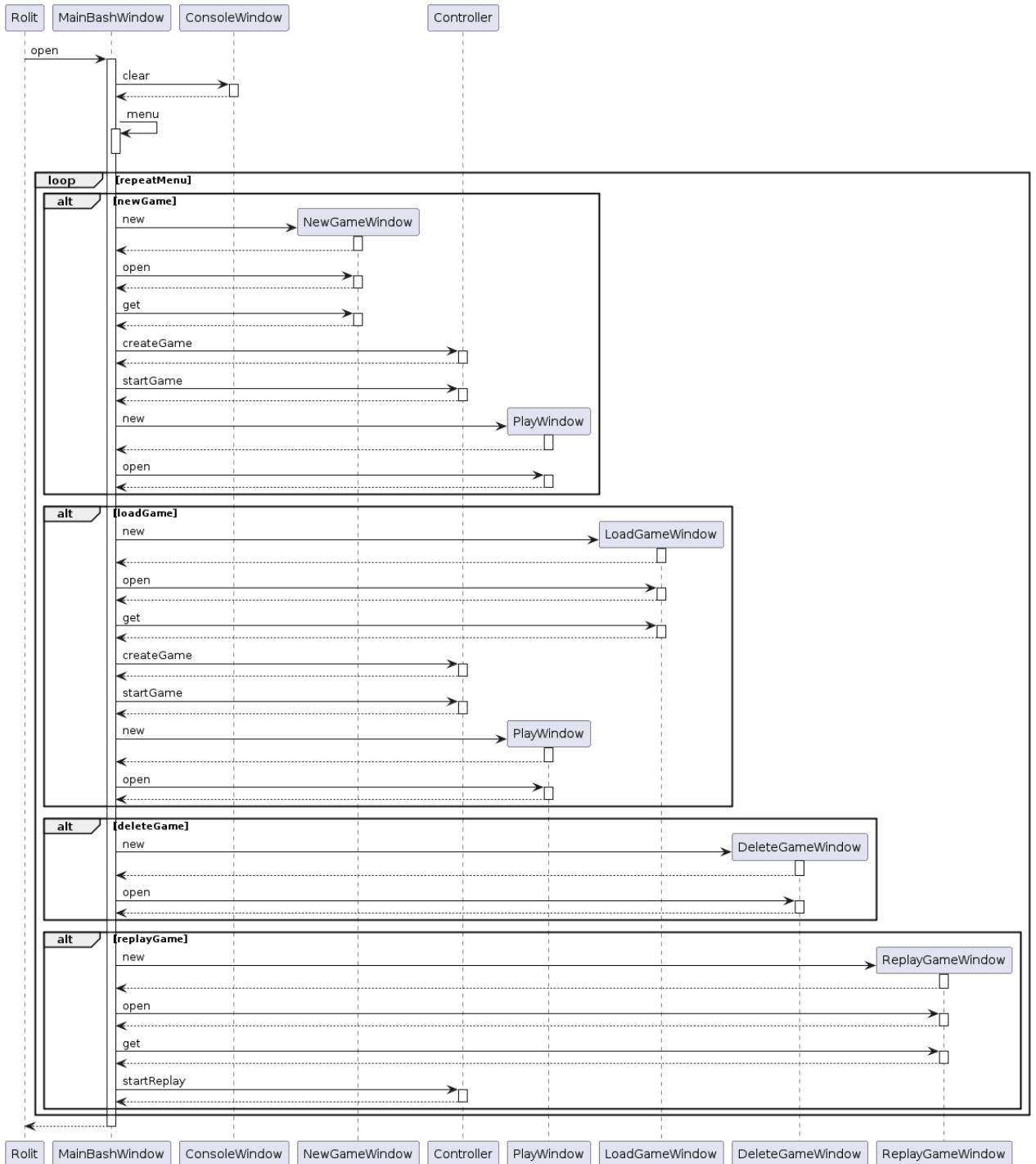
La interfaz de consola que hemos creado ha tratado de ser una copia de la lógica de componentes a modo de ventanas de *Swing*. La implementación de dicha lógica de forma similar ha permitido que apenas haya diferencias ni discrepancias en lo que la notificación eventos y de errores supone a cada vista. En general, ha permitido que al pensar todo en términos de las mismas precondiciones supuestas para la vista que iba a estar en la ejecución las funcionalidades que metíamos en una de las vistas era muy fácilmente extensible a la otra.

Para lograr este objetivo, se ha definido la interfaz `ConsoleWindow` que serán los objetos que se manejen en la vista de consola como homólogos de los `JFrame` de *Swing*. Dicha interfaz nos permite poder abrir una ventana, recoger un dato de ella, y limpiar la consola. Además de esta forma unificamos el flujo de salida para todos los componentes de la vista de consola, pues todos tienen como flujo de salida el `Scanner input` definido como variable estática de la interfaz. Una vez hecho esto, se ha definido una clase concreta para cada tipo de ventana de forma que añadir nuevas ventanas simplemente se traduce en añadir nuevas clases.



* MainBashWindow *

Es la ventana principal del juego y la que se encarga de gestionar que ventanas abrir o cerrar en función de la interacción del usuario. Actúa a modo de `MainWindow` tal y como está en la GUI. Al llamar al método `bool open()` de dicha ventana se nos muestra un menú recogido en la función privada `int menu()` que nos pide indicar cuál es la acción que queremos realizar de las siguientes: new game, load game, delete game, replay game.

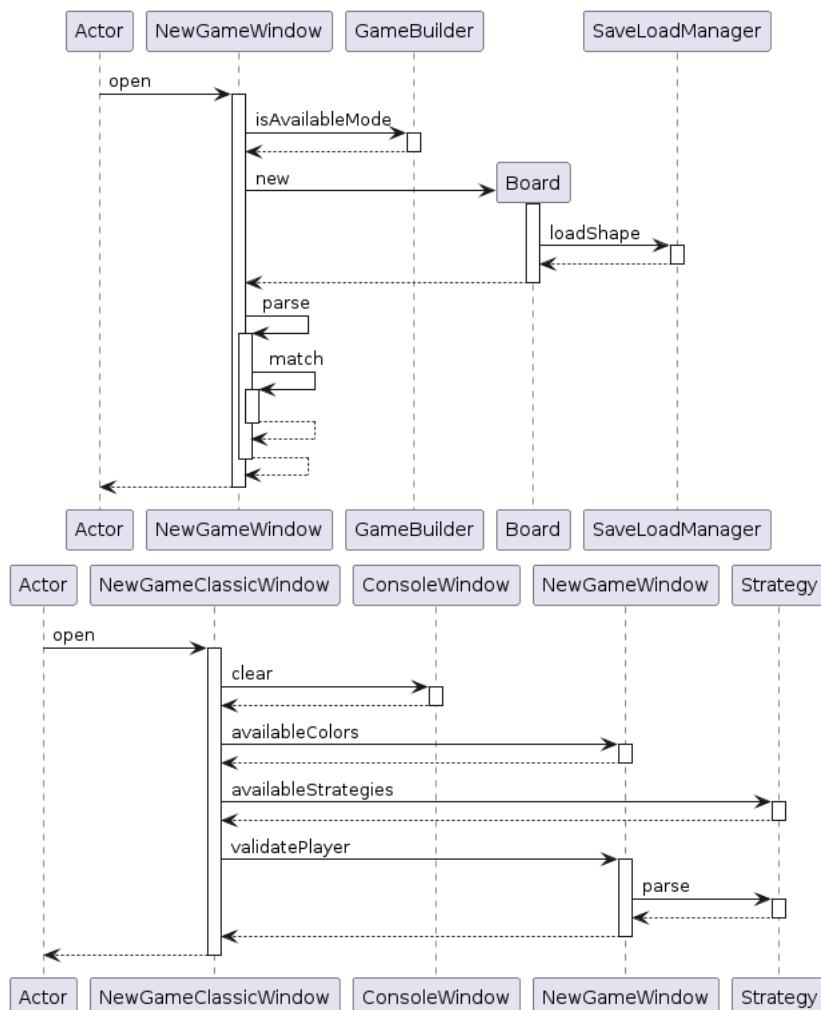


Una vez elegida la opción del menú, esta ventana abre la siguiente ventana correspondiente a cada una de las opciones y en caso de ser un opción que sigue con una nueva partida, inicia la ventana de juego PlayWindow.

* NewGameWindow, NewGameClassicWindow y NewGameTeamsWindow *

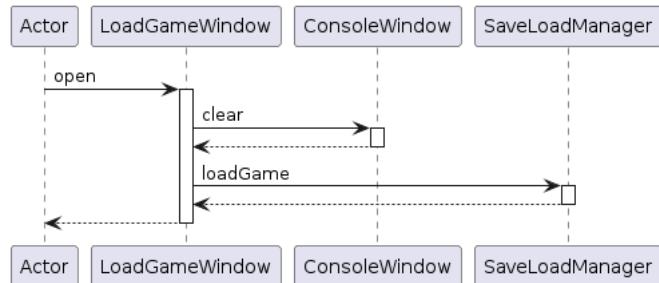
La ventana NewGameWindow y sus hijas NewGameClassicWindow y NewGameTeamsWindow están íntimamente ligadas con los Builder, pues son clases que se dedican a crear los `JSONObject` que utilizan dichas clases para generar el juego.

La ventana NewGameWindow ejecuta por pantalla todas las preguntas necesarias para poder crear un Game genérico. También tiene los métodos de `availableColors()` y de `validatePlayer()` que son llamados por sus hijas para comprobar si el jugador con el color introducido se puede agregar al `JSONObject`.



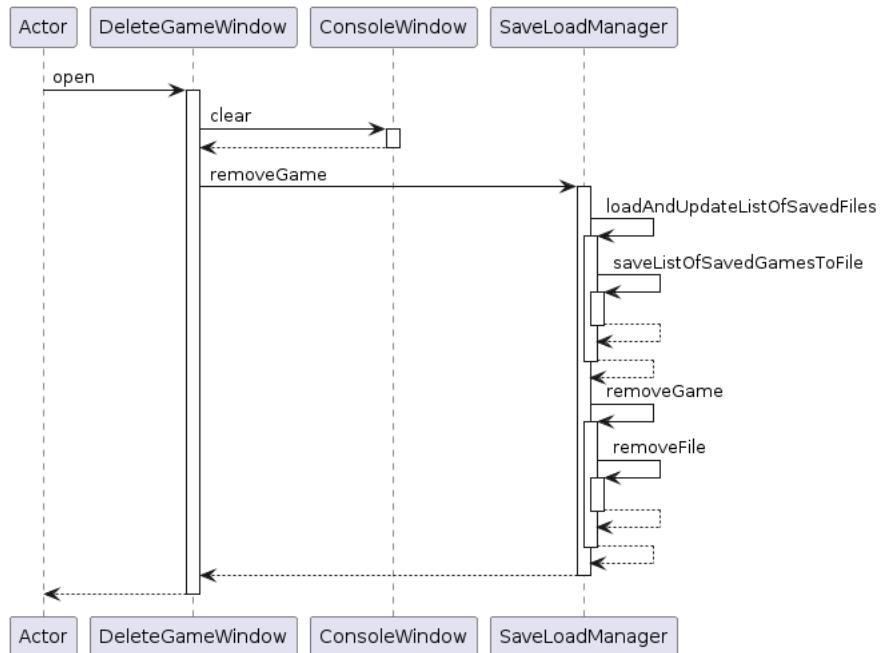
* LoadGameWindow *

La única funcionalidad de esta clase es presentar de forma amable la lista de juegos que se pueden cargar y hacer las llamadas correspondiente a la clase experta en guardado y carga desde ficheros que es **SaveLoadManager**.



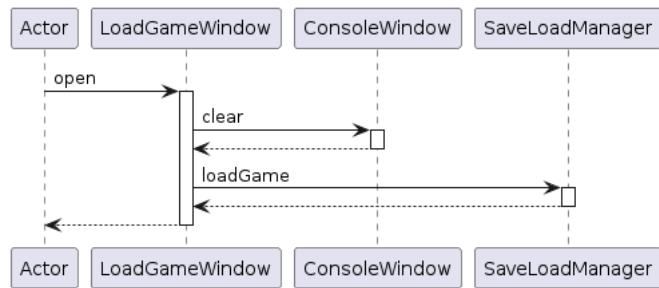
* DeleteGameWindow *

Al igual que la clase **LoadGameWindow**, esta clase principalmente presenta de una forma amable el listado de los juego guardados disponible para ser cargado y permite eliminar uno haciendo las llamas adecuadas a la clase **SaveLoadManager**



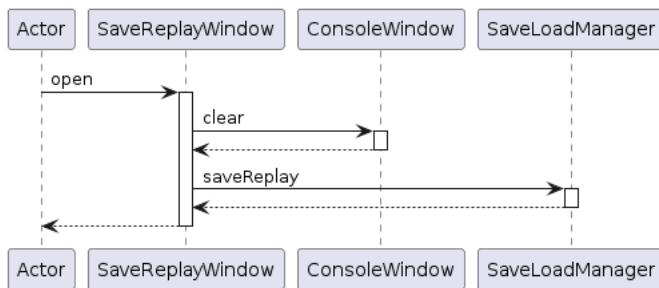
* LoadReplayWindow *

Es la clase homóloga a la clase `LoadGameWindow`, solo que su principal función es cargar una `Replay` en vez de un `Game` concreto:



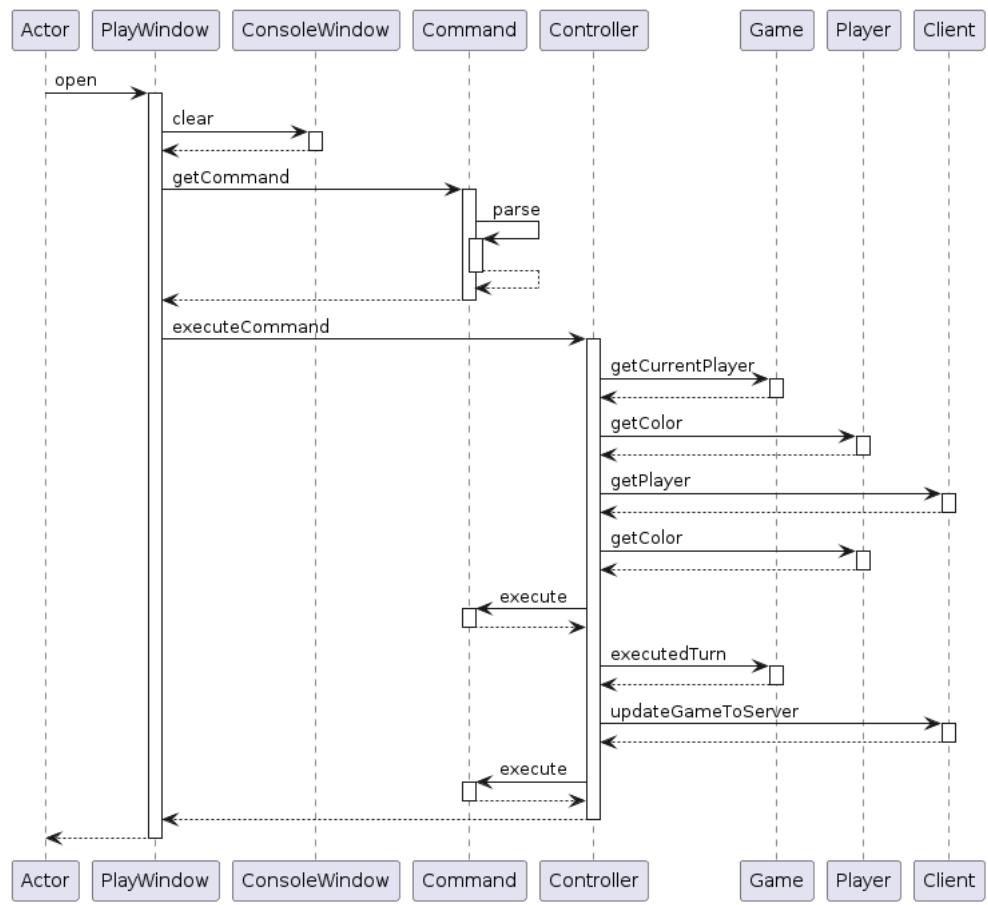
* SaveReplayWindow *

Del mismo modo que la clase `LoadReplayWindow` presentaba las posibles `Replay` para cargar, esta clase se encarga manejar la entrada/salida para poder guardar una `Replay`.



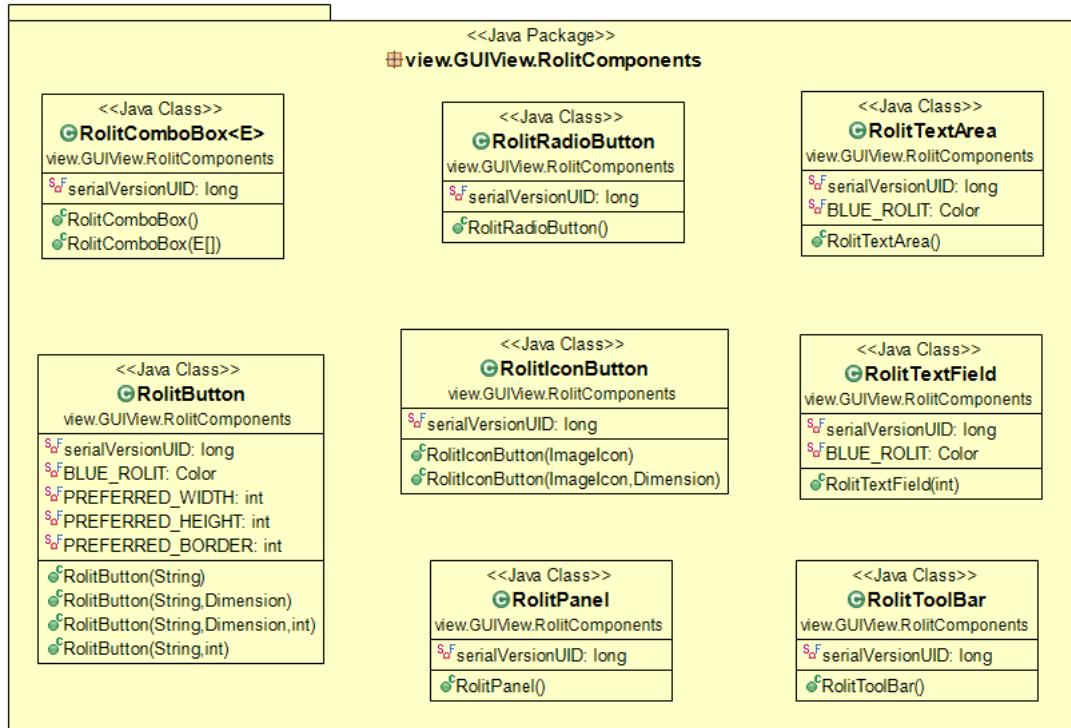
* PlayWindow *

Esta clase, que implementa `ConsoleWindow` y que se presenta en modo consola, muestra en pantalla el hilo de ejecución de una partida, y es el encargado de recoger las peticiones del usuario.



5.3.2 GUIView

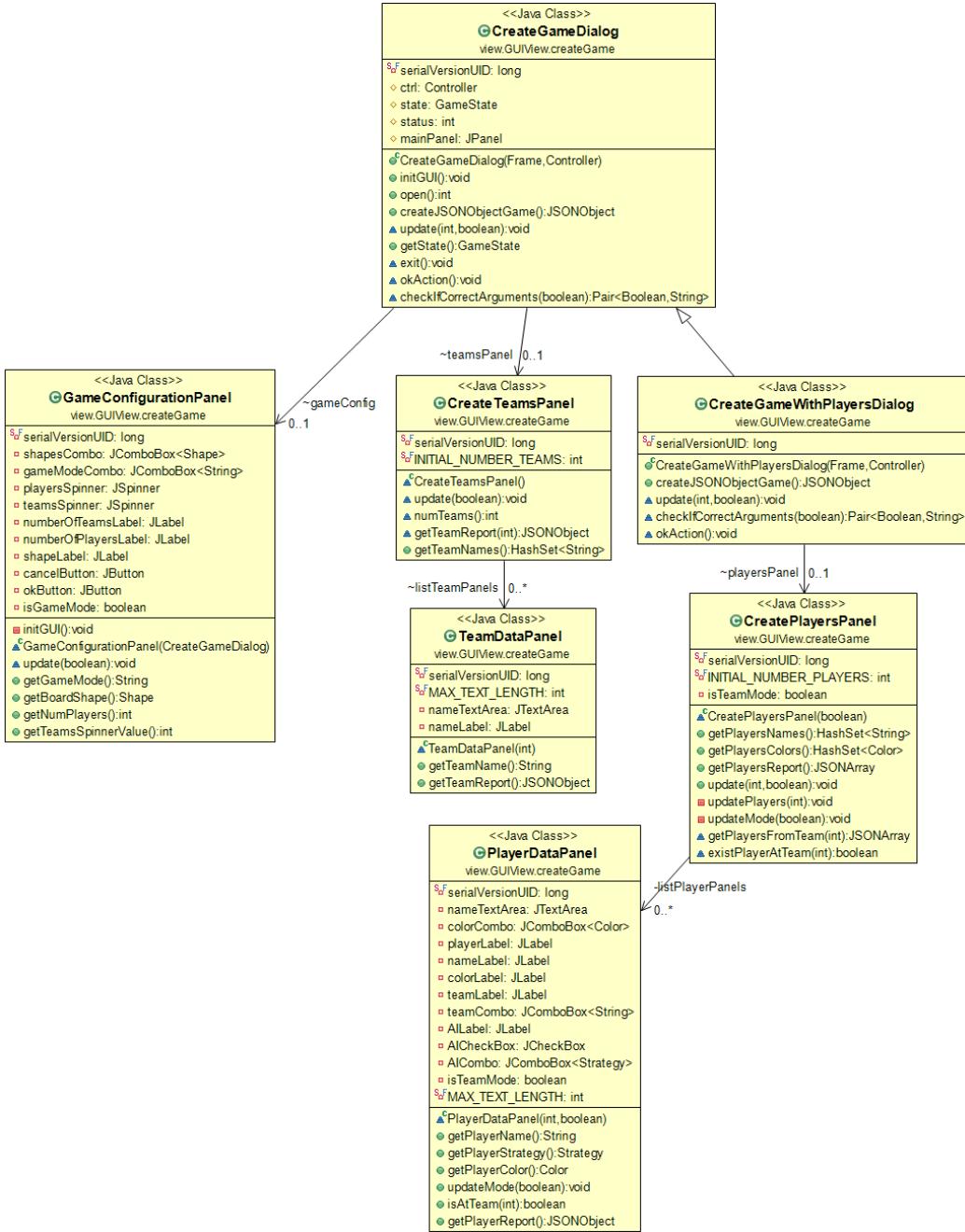
* Paquete RolitComponents *



El paquete **RolitComponents** está formado por un conjunto de clase que extienden a los que Java proporciona por defecto. La definición de componentes propios aporta uniformidad visual al juego, evita la repetición de código para configurar los componentes y facilita la adaptabilidad a futuros cambios.

Estos componentes siguen un mismo código visual basados en el color blanco y en el BlueRolid (#004398). Además, algunos de ellos contienen tamaños predeterminados, aunque puede modificarse si es necesario. Este es el caso de **RolitButton**, que cuenta con varios constructores para aumentar su versatilidad.

* Paquete CreateGame *

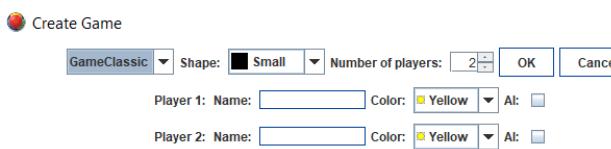


Esta agrupación de clases son las encargadas de visualizar y recopilar los datos necesarios para crear una partida, adaptándose a las necesidades y requerimientos del usuario. Así, tenemos dos tipos de cuadros de diálogo para pedirle al usuario la configuración de su partida.

En ocasiones, el usuario optará por jugar una partida en red. La persona que hostea el servidor tan solo puede elegir las propiedades del juego, pues es cada jugador al unirse al servidor quién decide su nombre y color. En estos casos, la responsabilidad para reunir los ajustes del juego recae sobre la clase **CreateGameDialog**.

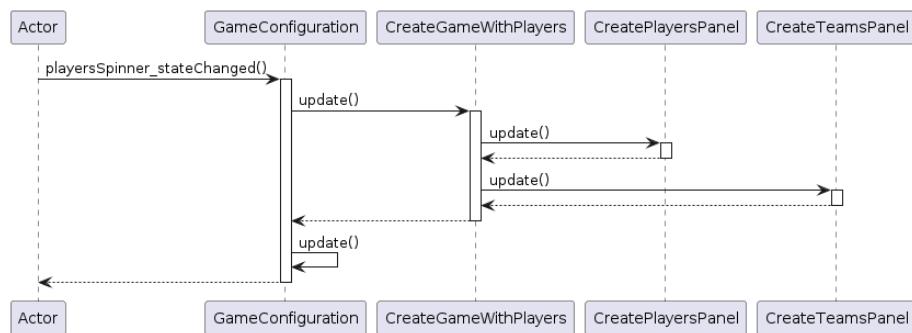
Sin embargo, cuando se juega en local, los jugadores deben introducir sus datos antes de iniciar la partida, de ahí la existencia de **CreateGameWithPlayersDialog**, que

extiende a `CreateGameDialog` e incluye un panel en la parte inferior para añadir los jugadores. Dicho panel se trata de un `CreatePlayersPanel`, que es un conjunto de `PlayerDataPanel` dispuestos verticalmente.



`CreateGameDialog` (arriba) vs `CreateGameWithPlayersDialog` (abajo)

Nótese que en el caso del cuadro de diálogo con jugadores, el número de participantes se decide mediante un `JSpinner` contenido en `CreateGameWithPlayersDialog`, pero el numero de paneles con la información de los jugadores se gestiona en `CreatePlayersPanel`. Para poder actualizar el número de `PlayerDataPanel` en tiempo de ejecución se utilizan los métodos `update()`.

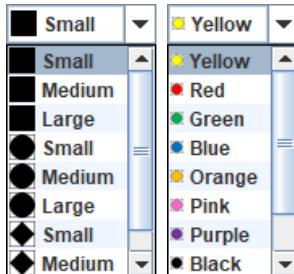


De forma análoga, ocurre con los equipos, pues el `CreateTeamsPanel` ha de mostrarse o no en pantalla dependiendo del modo de juego elegido en el `RoletComboBox`.

* BoardRenderer y ColorRenderer *

<pre><<Java Class>> BoardRenderer view.GUIMew \$serialVersionUID: long iconMap: Map<Shape,ImageIcon> background: Color defaultBackground: Color \$SIDE_LENGTH: int BoardRenderer() getListCellRendererComponent(JList<?>,Object,int,boolean,boolean):Component</pre>
<pre><<Java Class>> ColorRenderer view.GUIMew \$serialVersionUID: long iconMap: Map<Color,ImageIcon> background: Color defaultBackground: Color \$SIDE_LENGTH: int ColorRenderer() getListCellRendererComponent(JList<?>,Object,int,boolean,boolean):Component</pre>

Estas clases se encargan de renderizar correctamente los `RoletComboBox` que se utilizan en las clases de `CreateGameDialog` y `CreateGameWithPlayersDialog`. Más concretamente, la labor de estas clases es que se muestre un el ícono de la forma del tablero o del color del jugador junto a su nombre asociado.

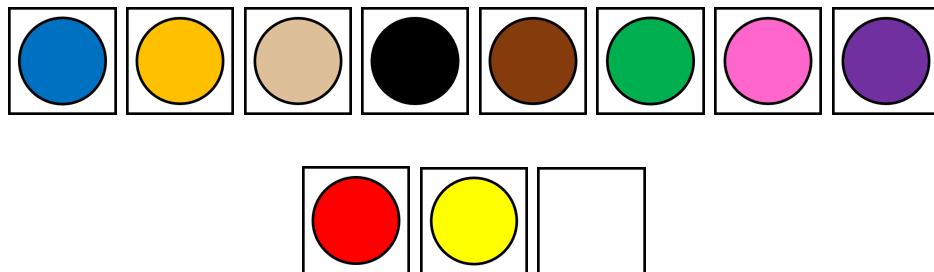


Resultado visual del uso de los renderers

* CellGUI *

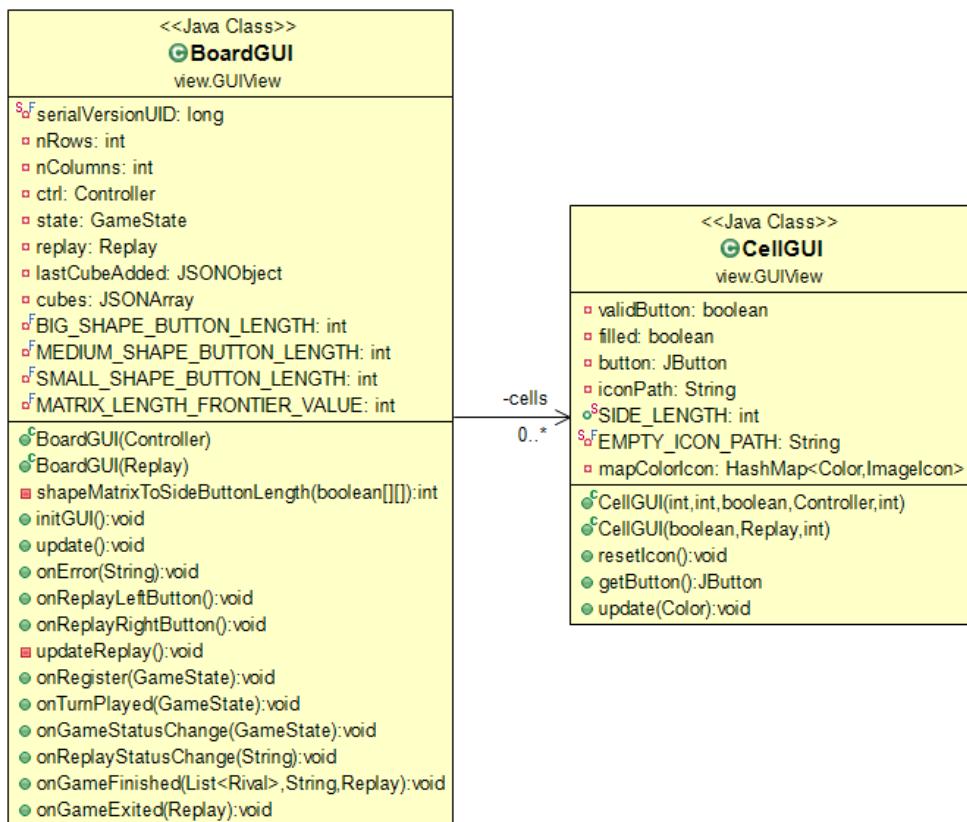
<pre><<Java Class>> CellGUI view.GUIMew validButton: boolean filled: boolean button: JButton iconPath: String \$SIDE_LENGTH: int \$EMPTY_ICON_PATH: String mapColorIcon: HashMap<Color,ImageIcon> CellGUI(int,int,boolean,Controller,int) CellGUI(boolean,Replay,int) resetIcon():void getButton():JButton update(Color):void</pre>
--

Esta clase representa una casilla del tablero, la cual puede actualizarse para dejar de estar vacía y contener un cubo de cierto color. Así, el método más relevante de esta clase es `update(Color)`, que implementa dicha funcionalidad.



Todas las posibles representaciones gráficas de `CellGUI`

* BoardGUI *

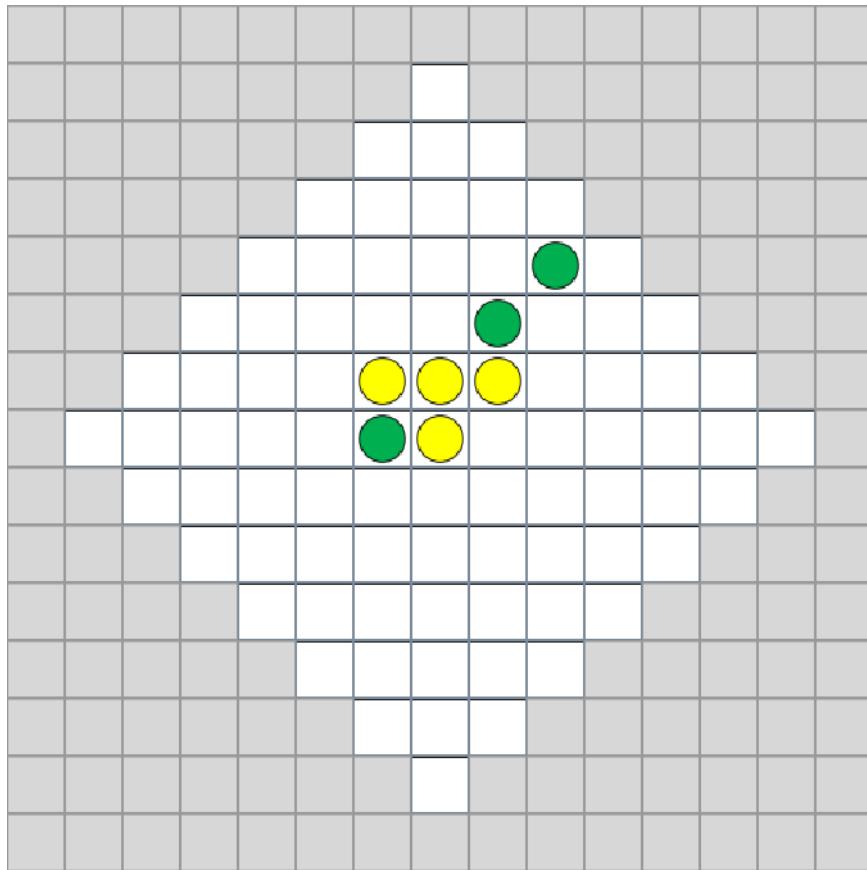


`BoardGUI` es la clase encargada de representar el tablero visualmente y consta de un conjunto de `CellGUI` dispuestos en forma de cuadrado, aunque su tamaño dependerá de la forma que elija el usuario.

Para poder mostrar dicha forma gráficamente, se deshabilitan ciertas celdas de manera estratégica para obtener la apariencia deseada. Además, el tamaño de las celdas se adapta al tablero para evitar problemas con las dimensiones de la ventana.

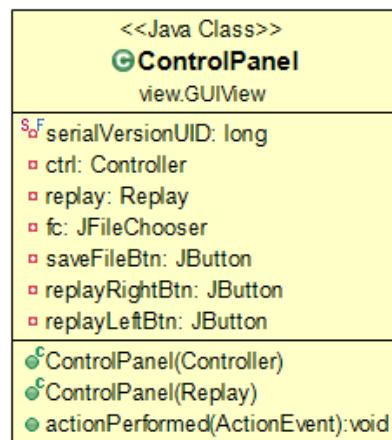
La comunicación con los modelos se lleva a cabo a través del *Patrón Observador*. De esta manera, nótese que en el diagrama se muestran métodos asociados a las *replays*

y otros a las partidas que permiten la correcta actualización del tablero durante la ejecución del programa.

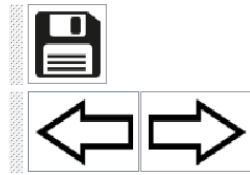


Representación gráfica de BoardGUI con forma diamante y tamaño mediano

* ControlPanel *

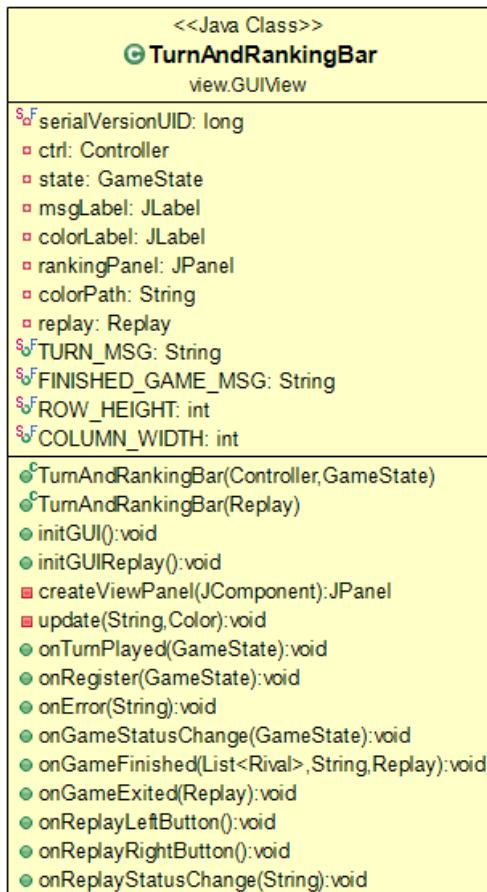


Esta clase representa una barra con herramientas básicas que se pueden utilizar durante una partida o una repetición. En el caso de las partidas contiene un botón para poder guardarla, mientras que durante las *replays* permite avanzar o retroceder en las jugadas.



ControlPanel durante una partida (arriba) vs durante una *replay* (abajo)

* TurnAndRankingBar *



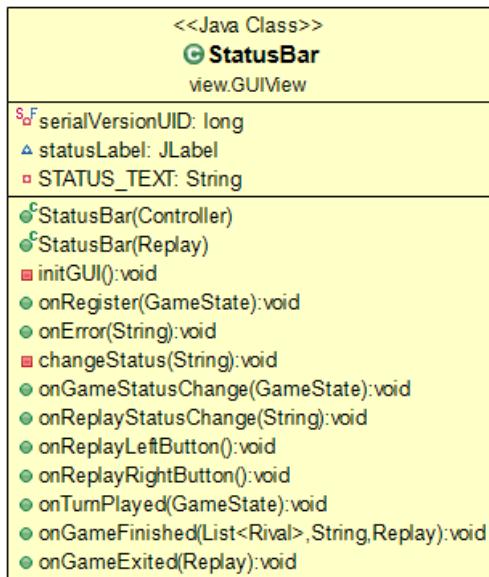
Tal y como su nombre sugiere, esta clase es un **RollitPanel** que se encarga de mostrar el turno y el ranking en cada instante de una partida o repetición. La información se obtiene del **GameState** que notifica el modelo cuando se ha jugado un turno.

Turn for Player1

	GameClassic	Player1	Player2	Player3	Player4
Score	5	2	4	1	

Representación gráfica de TurnAndRankingBar durante una partida

* StatusBar *

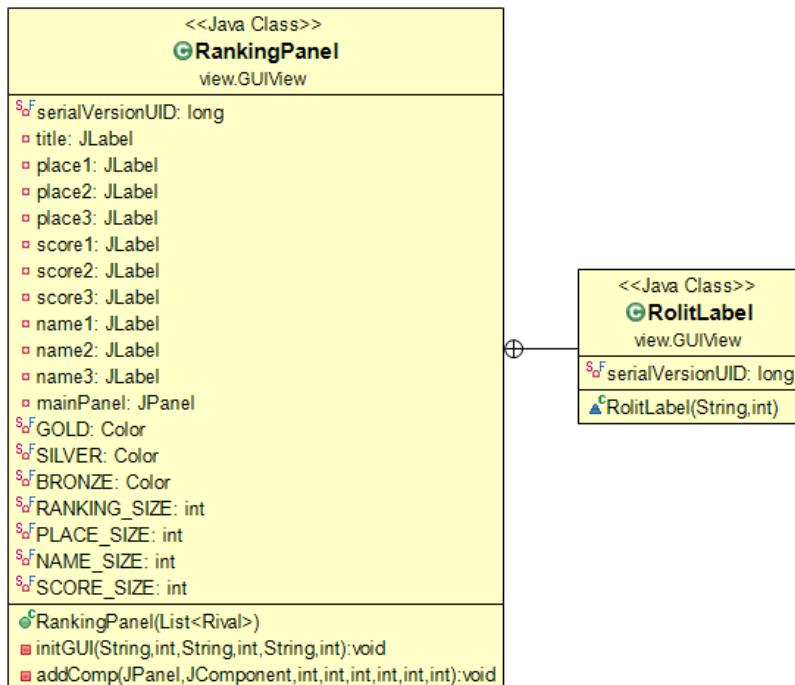


La **StatusBar** es responsable de comunicar información relevante al usuario durante la ejecución del programa. Así, el modelo notifica a esta clase con un mensaje y esta se encarga de visualizarlo.

Status: Non valid position for a cube

Ejemplo de la **StatusBar** mostrando un mensaje

* RankingPanel *

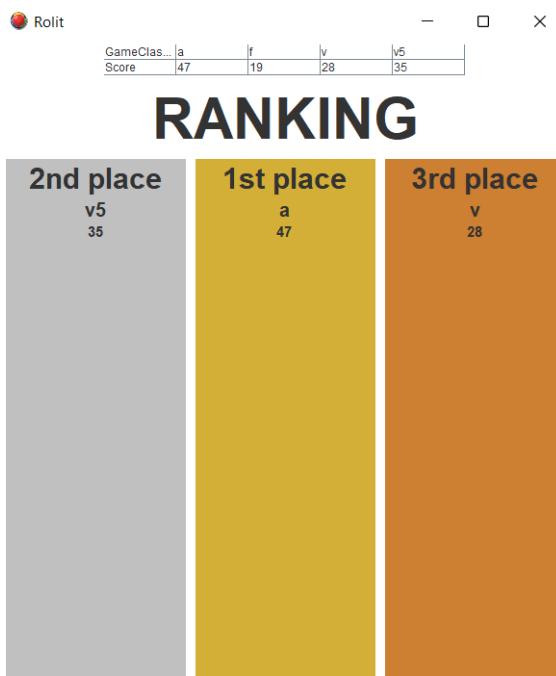


Cuando finaliza un juego, la clase `RankingPanel` muestra en pantalla una clasificación con los tres jugadores que obtuvieron la mejor puntuación durante la partida. En aquellos juegos donde solo haya dos jugadores, siempre se mostrarán en el ranking.

Además, el `RankingPanel` cuenta con la peculiaridad de tener un componente interno, la clase `RolitLabel`. En este caso se optó por trabajar con este componente de manera privada porque es el único lugar de todo el código donde se utiliza.

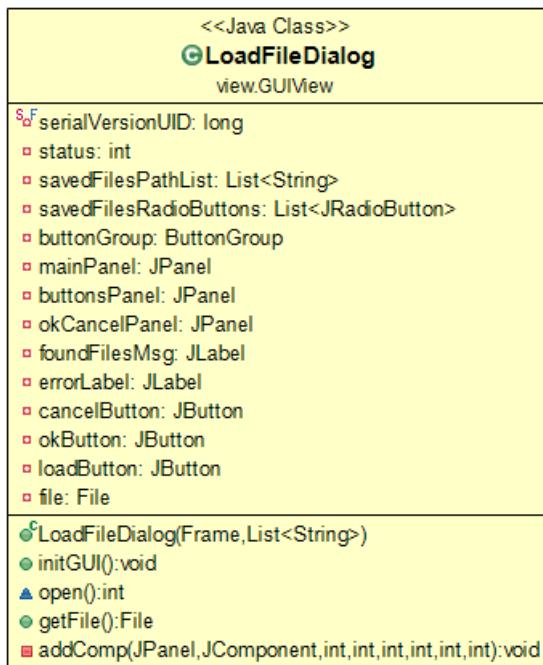


Ranking para dos jugadores



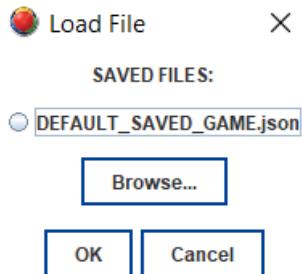
Ranking para más de dos jugadores

* LoadFileDialog *



LoadFileDialog es un cuadro de diálogo que permite cargar una partida o una *replay* ya guardada. Esta adaptabilidad se consigue pasándole a la clase a través del constructor una lista con los archivos disponibles, independientemente de su contenido.

Además, esta clase abre la posibilidad de cargar archivos que no están en la lista (porque no se generaron en el ordenador donde se ejecuta el programa) mediante un **JFileChooser**.

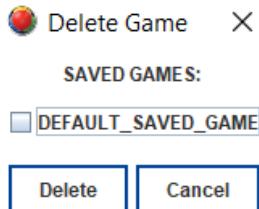


Representación visual de **LoadFileDialog**

* DeleteDialog *

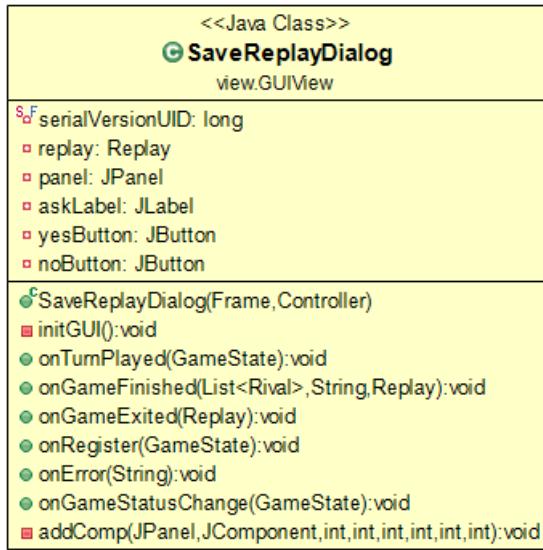


Esta clase es un cuadro de diálogo que permite al usuario eliminar los juegos que ya ha guardado. Para ello, cuenta con una lista de JCheckBox, donde cada uno representa un juego guardado. Al seleccionar alguna partida y hacer click en “Delete”, se borra dicho archivo tanto de la lista interna del juego como del ordenador.

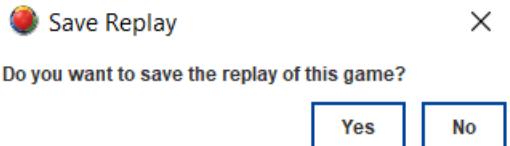


Representación visual de DeleteDialog

* SaveReplayDialog *

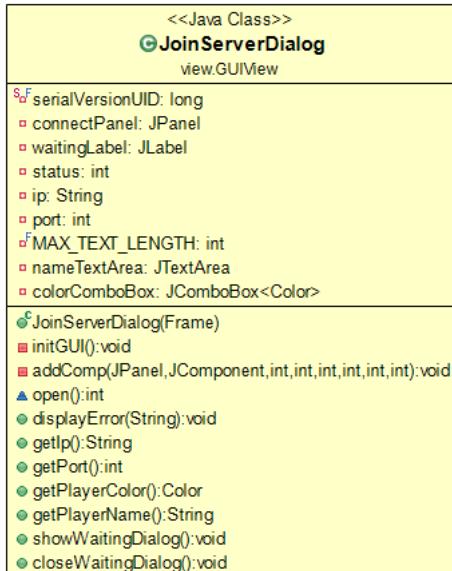


Esta clase se trata de una ventana emergente que tiene la responsabilidad de preguntarle al jugador al finalizar la partida si desea guardar su repetición. Si la respuesta es afirmativa, abre un `JFileChooser` para que el usuario elija su ruta.

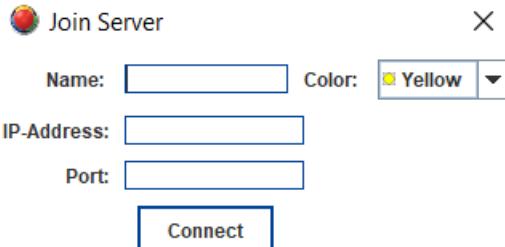


Representación visual de SaveReplayDialog

* JoinServerDialog *

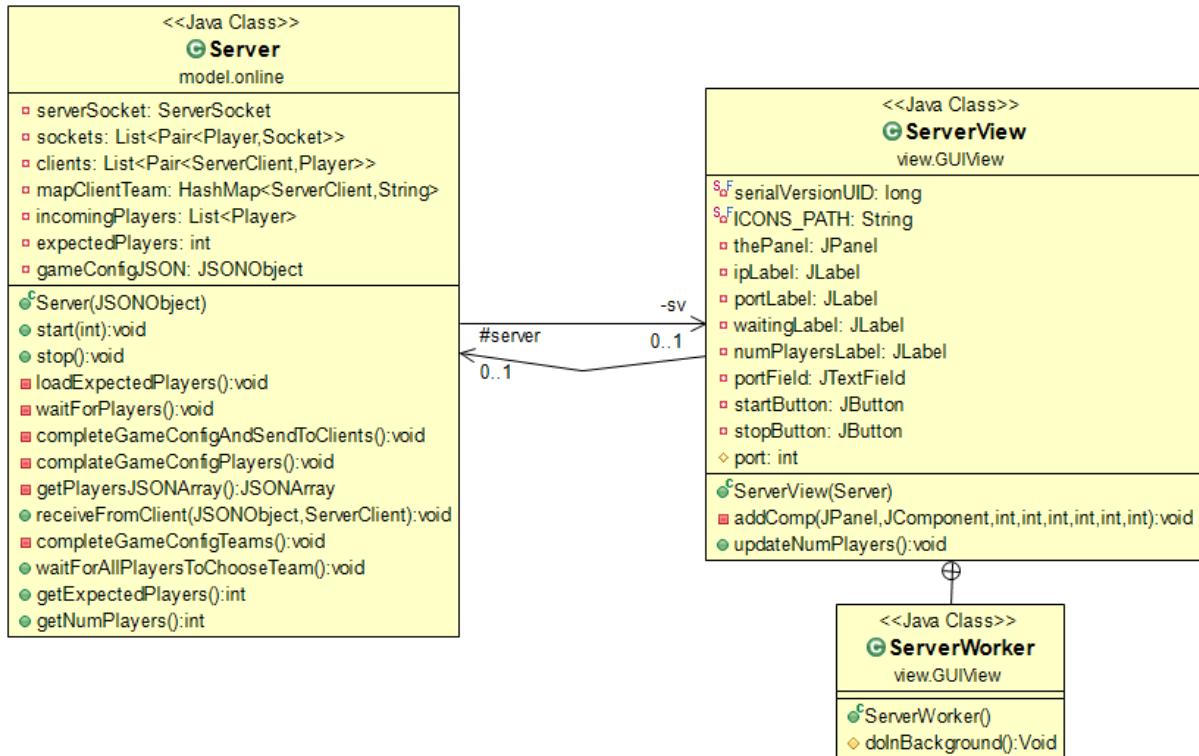


La clase `JoinServerDialog` extiende de `JDialog` y se encarga de recopilar la información necesaria del usuario para que pueda unirse a un servidor. Esto es, los datos del jugador y los del servidor.



Representación visual de `JoinServerDialog`

* ServerView *



La clase `ServerView` extiende de `JDialog` y su cometido es el de recoger el puerto en el que el usuario host desea que el servidor opere.

Al botón de Start Server en el diálogo de abrir un servidor, se crea un `ServerWorker`, que es un `SwingWorker` (encargado de correr un thread, el del servidor, de forma concurrente al de Swing -diálogo de abrir servidor). Dicho `ServerWorker` llama al método `start(port)` de `Server` que se encarga de abrir el servidor.

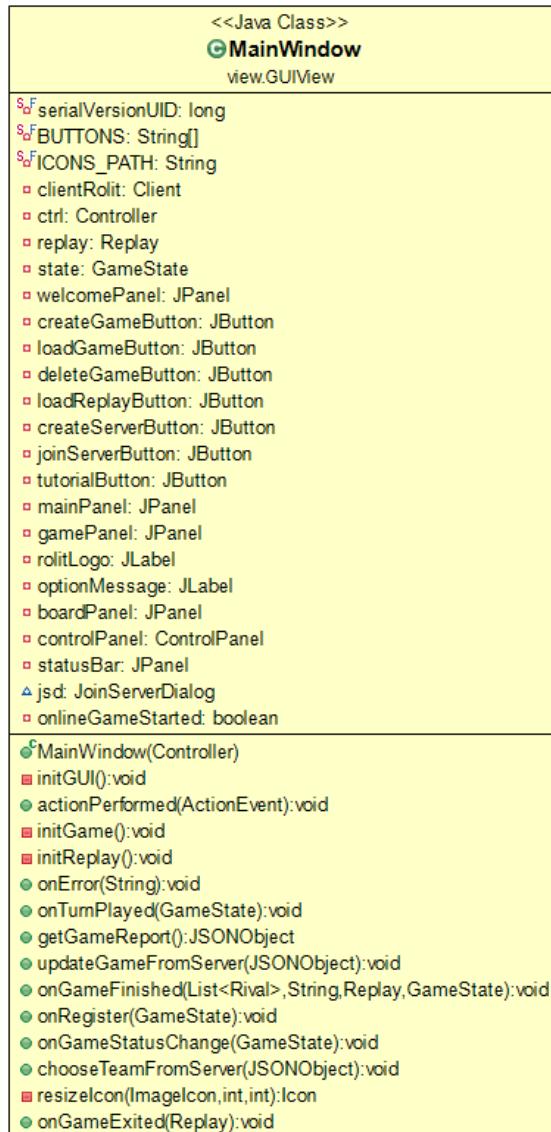


Representación gráfica de **ServerView** antes de crear el servidor

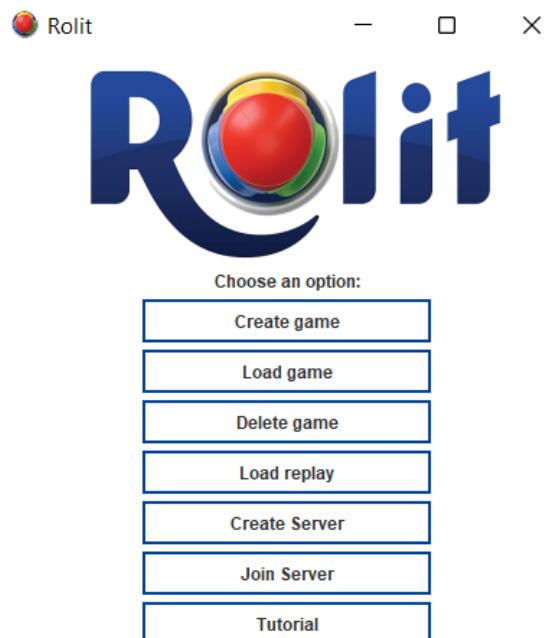


Representación gráfica de **ServerView** una vez creado el servidor

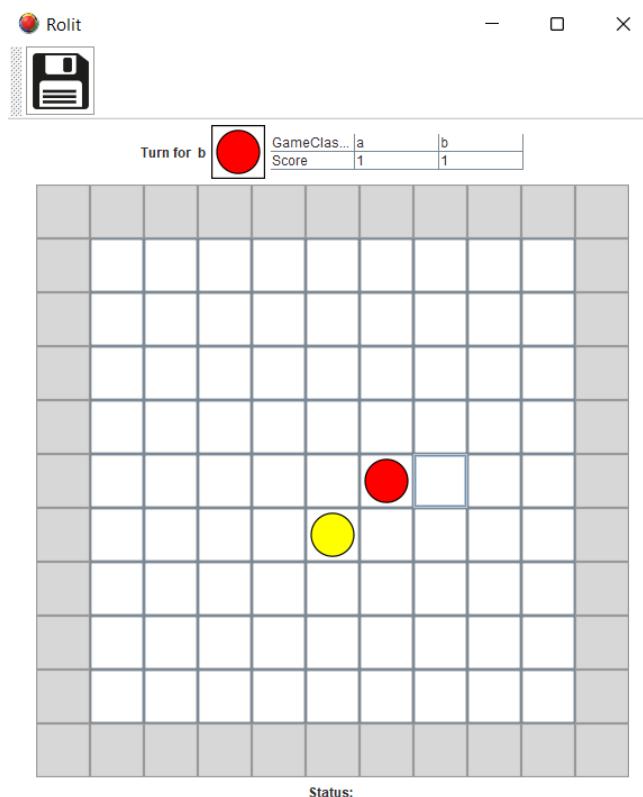
* MainWindow *



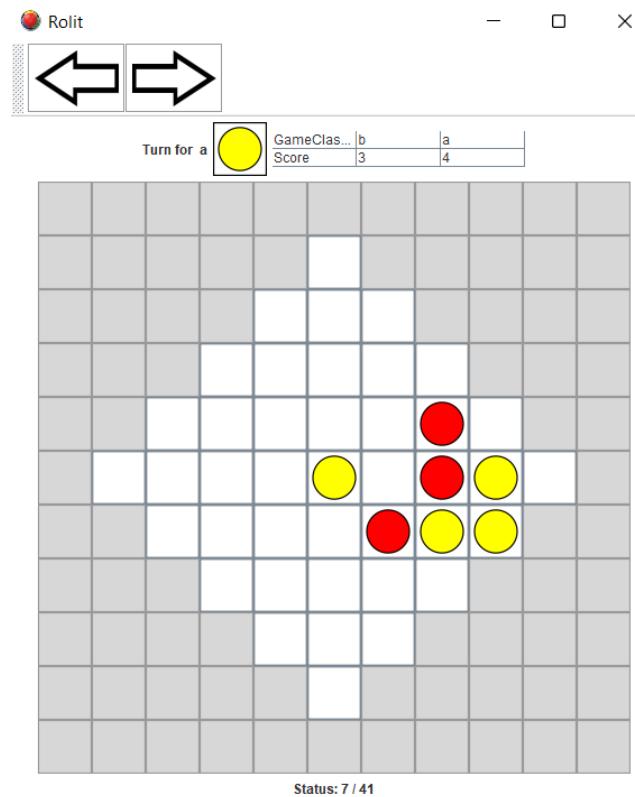
La MainWindow es la ventana principal del programa, que inicialmente permite al usuario decidir entre las opciones que brinda Rolit: crear una partida, cargar una replay, crear un servidor... Una vez seleccionada alguna de las alternativas, la ventana se actualiza combinando todos los elementos explicados anteriormente, de manera ordenada, para poder dar lugar a los requerimientos del usuario.



Apariencia inicial de la MainWindow

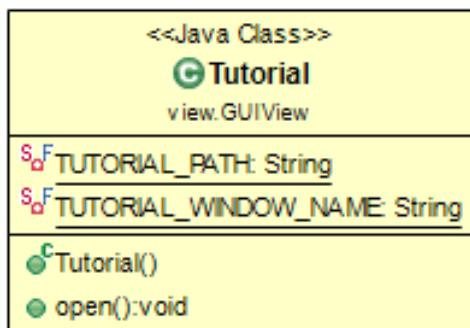


MainWindow durante una partida



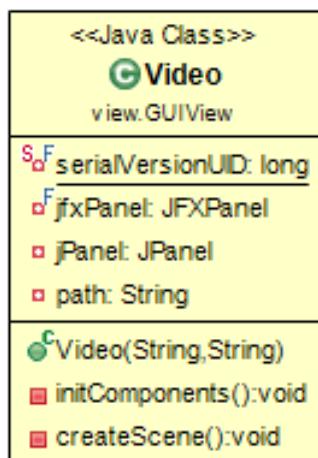
MainWindow durante una *replay*

* Tutorial *



Esta clase contiene la funcionalidad de mostrar el tutorial de juego. Su método principal es el método `open()`, en el que se abre en otro hilo de ejecución el vídeo en cuestión.

* Video *



Esta clase es la que permite la ejecución de videos en una nueva ventana durante la ejecución del programa. Se usa únicamente para mostrar el tutorial. Tiene un atributo path, que es la ruta relativa del video a reproducir.

A. Reports

Cube

```
1 {  
2     "color" = String,  
3     "pos" = [int x, int y]  
4 }
```

Board

```
1 {  
2     "shape" : String,  
3     "cubes": [{"color": String, "pos": [int x, int y]},  
4                 ...  
5                 ,{"color": String, "pos": [int x, int y]}]  
6  
7 }
```

Player

```
1 {  
2     "name" = String,  
3     "score" = int,  
4     "color" = String  
5     "strategy" = String (opcional)  
6 }
```

Team

```
1 {  
2     "name" = String,  
3     "score" = int,  
4     "players" = [{"name" = String, "score" = int, "color" = String},  
5                   ...  
6                   ,{"name" = String, "score" = int, "color" = String}]  
7 }
```

GameClassic

```
1 {
2   "type" = "GameClassic",
3   "board" = {
4     "shape" : String,
5     "cubes": [{"color": String, "pos": [int x, int y]}, ,
6       ...
7       , {"color": String, "pos": [int x, int y]}]
8   },
9
10  "players" = [{"name" = String, "score" = int, "color" = String},
11    ...
12    , {"name" = String, "score" = int, "color" = String}],
13
14  "turn" = String
15 }
```

GameTeams

```
1 {
2   "type" = "GameTeams",
3   "board" = {
4     "shape" : String,
5     "cubes": [{"color": String, "pos": [int x, int y]}, ,
6       ...
7       , {"color": String, "pos": [int x, int y]}]
8   },
9
10  "players" = [{"name" = String, "score" = int, "color" = String},
11    ...
12    , {"name" = String, "score" = int, "color" = String}],
13
14  "teams" = [{"name" = String, "score" = int, "players" =
15    [{"name" = String, "score" = int, "color" = String},
16      ...
17      , {"name" = String, "score" = int, "color" = String}]}],
18
19  ...
20
21  , {"name" = String, "score" = int, "players" =
22    [{"name" = String, "score" = int, "color" = String},
23      ...
24      , {"name" = String, "score" = int, "color" = String}]}
25  ],
26
27  "players" = [{"name" = String, "score" = int, "color" = String},
28    ...
29    , {"name" = String, "score" = int, "color" = String}],
30
31  "turn" = String
32 }
```

State

```
1 {
2     "command" = String,
3     "game" = game.report() //consultar arriba
4 }
```

Replay

```
1 {
2     "states" = [{"command" = String,
3                 "game" = game.report(), //consultar arriba
4                 ...
5                 ,{"command" = String,
6                 "game" = game.report()}] //consultar arriba
7 }
```


B. Listado de clases

Las clases del proyecto con las siguientes:

GameBuilder: Builder genérico de Game. Se encarga de generar GameClassic o GameTeams con sendas llamadas a GameClassicBuilder y GameTeamsBuilder.

GameClassicBuilder: Builder de GameClassic.

GameTeamsBuilder: Builder de GameTeams.

Client: Clase intermediaria entre el ClientController y la vista.

ClientController: clase desde la perspectiva del cliente. Es el controlador envía y recoge información de y al servidor. El controlador es único por cliente.

Command: Clase genérica de comando que parsea los comandos introducidos para poder llamar a las clases específicas que gestionan dichos comandos.

ExitCommand: Clase que, dado el comando exit, termina el juego.

HelpCommand: Clase que, dado el comando help, muestra los comandos disponibles a ejecutar.

PlaceCubeCommand: Clase que, dado el comando de poner cubo, pide a Game que se coloque un cubo en la posición indicada.

SaveCommand: Clase que, dado el comando save, llama a SaveLoadManager para que guarde la partida en el instante en el que se llama al comando.

Controller: El controlador (en el sentido del MVC), intermediario de la vista y el modelo.

SaveLoadManager: Clase encargada de la carga, guardado y borrado de partidas y replays.

Board: Clase que define la estructura y lógica del tablero del juego.

Color: Clase de enumerados con métodos auxiliares para la representación de los colores.

Cube: Clase que representa un cubo, con su posición y jugador al que pertenece.

Game: clase abstracta que representa una visión general del juego, que luego se particulariza en GameClassic y GameTeams. Es un modelo (en el sentido del MVC).

GameClassic: Clase que implementa la lógica del juego en el modo Classic (jugadores individuales).

Gameteams: Clase que implementa la lógica del juego en el modo Teams (jugadores por equipos).

Player: Clase que representa a un jugador humano o IA y sus características (color, nombre, puntuación...).

Replayable: Interfaz que extiende de Reportable y que incluye un método `toString()` con el fin de encapsular Game para generar estados, que son usados por multitud de clases.

Reportable: Interfaz a través de la cual, implementada en clases, permite ejecutar un `report()`, serialización del objeto en formato `.json` a través de unos estándares definidos.

Rival: Interfaz que permite unificar si se juega contra un jugador o contra un equipo mediante el concepto de rival.

Shape: Clase de enumerados con métodos auxiliares para la representación de las formas del tablero.

Team: Clase que representa a un equipo con el modo de juego GameTeams.

TurnManager: Clase que gestiona el paso de turnos.

GameState: Clase que representa el estado del juego en un momento determinado.

Replay: Clase a través de la cual se gestionan las replays, pudiendo por ejemplo avanzar y retroceder en las mismas.

Rolit: Clase de punto de partida de la aplicación. En esta se muestra el menú principal y se decide si acceder al modo GUI o al modo consola.

HelpException: Clase que representa la excepción ejecutada desde el comando HelpCommand.

Server: Clase que gestiona los clientes desde la perspectiva del servidores y que procesa los mensajes emitidos desde los clientes.

ServerClient: Clase que representa cada cliente desde la perspectiva del servidor.

ServerClientThread: Clase que interactúa con el cliente enviando y recibiendo mensajes.

ServerView: Clase que abre el diálogo a través del cual el usuario puede abrir y detener un servidor.

WaitPlayerThread: Thread a través del cual el servidor va recibiendo las conexiones entrantes de los distintos clientes que se van conectando.

GreedyStrategy Estrategia IA consistente en una búsqueda superficial con el algoritmo Minimax.

MinimaxStrategy: Estrategia IA consistente en una búsqueda en profundidad con el algoritmo Minimax y la poda alfa-beta.

RandomStrategy: Estrategia IA consistente en introducir un cubo de forma aleatoria siempre que este esté en una posición válida.

SimplifiedBoard: Encapsulación del tablero a través del cual la IA decide sus movimientos.

Strategy: Clase genérica estrategia IA de la cual heredan las estrategias específicas.

Pair: Clase que representa un par, utilidad básica para varias partes del código.

StringUtils: Clase que recopila distintas utilidades para trabajar con cadenas, básico para varias partes del código.

RolitObserver: Interfaz que declara las notificaciones a través de las cuales se trabaja en el patrón observador implementado para la estructura modelo-vista-controlador para particularidades del juego.

ConsoleWindow: Interfaz de ventana genérica para el modo consola

DeleteGameWindow: Ventana para la consola a través de la cual se puede borrar un juego guardado.

LoadGameWindow: Ventana para la consola a través de la cual se puede cargar un juego guardado.

LoadReplayWindow: Ventana para la consola a través de la cual se puede cargar un replay.

MainBashWindow: Ventana para la consola a través de la cual se muestra el menú principal.

NewGameClassicWindow: Ventana para la consola a través de la cual se crea un nuevo juego con modo GameClassic.

NewGameTeamsWindow: Ventana para la consola a través de la cual se crea un nuevo juego con modo GameTeams.

NewGameWindow: Ventana para la consola a través de la cual se crea un nuevo juego.

PlayWindow: Ventana para la consola a través de la cual se muestra el hilo de ejecución de una partida.

SaveReplayWindow: Ventana para la consola a través de la cual se puede guardar la partida en el estado actual.

BoardGUI: Clase que gestiona la representación del board en modo GUI

BoardRenderer: Renderer Swing que diseña los ComboBox que muestran las distintas formas de tablero que el usuario puede escoger.

CeldaGUI: Clase que representa una celda del tablero en modo GUI, responsabilizándose de su diseño y de recoger los clic del usuario.

ChooseTeamFromServerDialog: Diálogo GUI que, desde el cliente, se escoge en qué equipo desea unirse el usuario a la hora de conectarse a una partida en red con modo GameTeams.

ColorRenderer: Renderer Swing que diseña los ComboBox que muestran los distintos colores el usuario puede escoger.

ControlPanel: Panel Swing que muestra, según el caso, guardar partidas y el avance y retroceso en los replays. Se coloca en la parte superior de la ventana de la aplicación.

DeleteGameDialog: Diálogo GUI a través del cual el usuario puede eliminar partidas guardadas.

JoinServerDialog: Diálogo GUI a través del cual el cliente puede conectarse a un servidor introduciendo los datos de este.

LoadFileDialog: Diálogo GUI a través del cual el usuario puede cargar partidas guardadas.

MainWindow: Ventana principal del modo GUI sobre la cuál se colocan todos los componentes y nacen todos los diálogos.

Observable: Interfaz que declara el método de añadir observador para la implementación del patrón Observador.

RankingTableModel: Tabla Swing (GUI) que muestra la tabla de las puntuaciones en tiempo real de los usuarios de una partida.

ReplayObserver: Interfaz que declara las notificaciones a través de las cuales se trabaja en el patrón observador implementado para la estructura modelo-vista-controlador para particularidades del Replay.

SaveReplayDialog: Diálogo GUI a través del cual el usuario puede guardar la repetición de una partida finalizada.

StatusBar: Componente GUI situado en la parte inferior de la aplicación que muestra las notificaciones lanzadas por el modelo.

TurnAndRankingBar: Panel Swing (GUI) que junta tanto el ranking como el turno del usuario actual en la partida.

CreateGameDialog: Diálogo GUI a través del cual el usuario puede crear una partida.

CreateGameWithPlayersDialog: Clase que se encarga de los jugadores dentro del CreateGameDialog (GUI).

CreatePlayersPanel: Panel Swing (GUI) encargado de gestionar los jugadores introducidos en el CreateGameDialog.

CreateTeamsPanel: Panel Swing (GUI) encargado de gestionar los equipos introducidos en el CreateGameDialog.

GameConfigurationPanel: Panel Swing (GUI) que recoge la configuración de juego especificada en el CreateGameDialog.

PlayerDataPanel: Panel Swing (GUI) que recoge los datos introducidos de jugadores especificados en el CreateGameDialog.

TeamDataPanel: Panel Swing (GUI) que recoge los datos introducidos de equipos especificados en el CreateGameDialog.

RolitBorder: Extensión de Border (Swing, GUI) adaptada para el diseño particular decidido para la aplicación.

RolitButton: Extensión de JButton (Swing, GUI) adaptada para el diseño particular decidido para la aplicación.

RolitCheckBox: Extensión de JCheckBox (Swing, GUI) adaptada para el diseño particular decidido para la aplicación.

RolitComboBox: Extensión de JComboBox (Swing, GUI) adaptada para el diseño particular decidido para la aplicación.

RolitIconButton: Extensión de JButton con icono (Swing, GUI) adaptada para el diseño particular decidido para la aplicación.

RolitPanel: Extensión de JPanel (Swing, GUI) adaptada para el diseño particular decidido para la aplicación.

RolitRadioButton: Extensión de JRadioButton (Swing, GUI) adaptada para el diseño particular decidido para la aplicación.

RolitTextArea: Extensión de JTextArea (Swing, GUI) adaptada para el diseño particular decidido para la aplicación.

RolitTextField: Extensión de JTextField (Swing, GUI) adaptada para el diseño particular decidido para la aplicación.

RolitToolBar: Extensión de JToolBar (Swing, GUI) adaptada para el diseño particular decidido para la aplicación.

C. Estrategia Minimax adaptada a Rolit

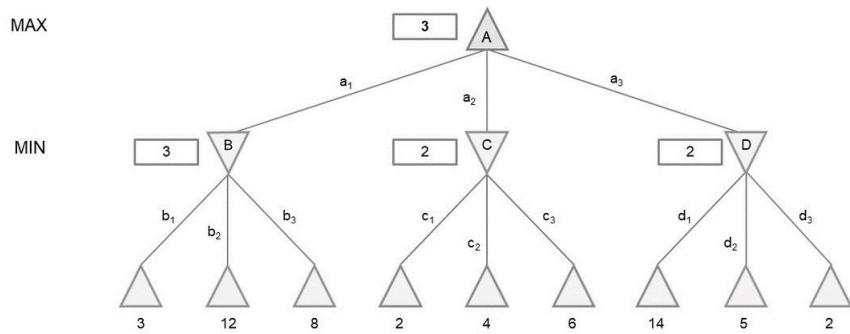
En teoría de juegos, el Minimax busca minimizar la pérdida esperada. La aproximación que se toma es asumir que el oponente va a tomar las decisiones que más te perjudiquen. De esta manera, al encontrar la decisión que menor pérdida suponga, el resultado real será siempre igual o mejor al calculado, de forma que el cálculo es fiable.

La mejor forma de explicar esto es a través de un ejemplo:

Imaginemos que estamos en un juego de dos jugadores, uno contra el otro, basado en turnos, en el cual ambos jugadores conocen en todo momento el estado actual de la partida en su totalidad. Un buen ejemplo de esto es el ajedrez. Supongamos pues que juegas con las piezas blancas, y tu adversario juega con las piezas negras. En cada uno de tus movimientos vas a jugar el movimiento que consideres que más te favorece. Por el otro lado, bajo nuestra aproximación, suponemos que el otro jugador va a jugar el movimiento que más te perjudique. Podemos hacer una representación de esto en forma de árbol:

Imaginemos que cada nodo contiene un número que representa el estado actual de la partida (para esto hace falta tener un criterio de valoración del estado actual de la partida, en el cual no entraremos en detalle en el caso del ajedrez, pero más adelante sí en el caso del Rolit), y cada arista representa un movimiento jugado, por el cuál se desciende en el árbol de un estado de la partida al siguiente. En cuanto a la valoración de los estados del juego, si el número de un nodo es positivo va ganando el jugador blanco (a mayor mejor); si el número es negativo, va ganando el jugador negro (a menor peor). Como el juego va por turnos, si un nivel del árbol se corresponde con el turno de un jugador, el siguiente nivel se corresponde con el siguiente jugador. De esta forma, volviendo al ejemplo propuesto, si desde un nodo se conoce el valor de todos sus descendientes pueden pasar dos cosas: si es el turno del jugador blanco, tomará la decisión que le lleve al mayor valor; si es el turno del jugador negro, tomará la decisión que le lleve al menor valor.

Se ilustra el funcionamiento del algoritmo en la siguiente imagen:



En el caso del ajedrez (y de la mayoría de juegos por turnos, como Rolit) hay siempre muchos posibles movimientos a jugar. De esta forma, en el árbol de decisión, de cada nodo salen muchos descendientes, resultando en un algoritmo con coste aproximadamente exponencial en el promedio de jugadas disponibles. Esto hace que la búsqueda del mejor posible movimiento se convierta en un problema intratable en no demasiados niveles de profundidad de búsqueda. Por tanto, surge la obligación de limitar la profundidad hasta la que se quiere hacer la búsqueda.

Volviendo ahora al Rolit, no estamos en un juego de dos jugadores (o al menos no necesariamente). Afortunadamente, en este juego el criterio de valoración de jugadas es fácil: La mejor jugada es la que te lleve a acabar con el mayor número de puntos.

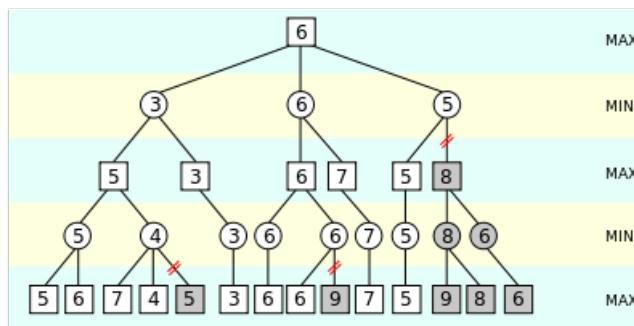
Ahora, como hay más de dos jugadores, para conseguir un cálculo de puntos fiable, el cálculo se lleva a cabo considerando que el jugador propietario de la estrategia quiere hacer aquella jugada que más puntos le otorgue a la larga, mientras que el resto de jugadores en sus turnos hacen la jugada que más puntos le quite al jugador propietario. De esta forma se calculan puntos para el jugador propietario situándonos en la situación más desfavorable posible, de forma que todo aquello que se calcule va a derivar siempre en un resultado igual o mejor al calculado, de forma que los resultados de los cálculos son fiables.

D. Poda Alfa-Beta

La poda Alfa-Beta es una mejora del algoritmo Minimax. Se mantienen dos valores, alfa y beta, que representan respectivamente la puntuación mínima que se llevará el jugador maximizador y la puntuación máxima que se asegura el jugador minimizador. Inicialmente, alfa es $-\infty$ y beta es ∞ .

Siempre que la puntuación máxima que se asegura al jugador que minimiza se vuelve menor que la puntuación mínima que se asegura el jugador que maximiza se puede parar de explorar por la rama actual. De forma análoga, se podan ramas en el caso contrario.

La mejor forma de visualizar esta poda es a través de una ilustración:



Como vemos en este ejemplo, si exploramos en el árbol de izquierda a derecha, una vez llegamos a la rama derecha vemos que el jugador minimizador encuentra una rama por la que logra llegar al valor 5. Como minimiza, se sabe que el valor de ese nodo va a ser, como mucho, 5. Al ver esto el jugador maximizador, teniendo en cuenta que en una rama anterior ha llegado al valor 6, sabe que no tiene que seguir explorando esa rama, porque de ninguna manera va a encontrar un valor mejor que 6, y por tanto, en esta situación, el mejor resultado es el que le brinda empezar explorando la rama del centro.

De esta forma, este método permite podar en muchas ocasiones el árbol de exploración, resultando en menores tiempos de ejecución.

E. Aspectos mejorables del diseño

El equipo de desarrollo ha recopilado un conjunto de posibles mejoras que realizar al proyecto en caso de haber habido más sprints hábiles para poder hacer modificaciones y tras conocer con mayor exactitud cuáles son las ventajas de aplicar algunos patrones nuevos y otros ya utilizados pero de forma más efectiva.

* Builders *

Realmente el diseño de estas clases es bueno en cuanto tener un proceso unificado y extensible a través de la creación de nuevos Builder específicos para generar nuevos tipos de juegos. Sin embargo, la ejecución en la implementación podría haber sido mejor dejando la clase `GameBuilder` tal y como está (exceptuando el quitar los métodos protegidos a los que acceden los Builder concretos) implementando el patrón *Abstract Factory* y haber hecho una nueva clase abstracta llamada `Builder` que fuera la clase padre de todos los Builder en específico. De este modo, encapsulamos todos los métodos privados de la clase `GameBuilder` como `AvailableColors()` y la responsabilidad de saber construir juegos en este grupo de clases y dejamos que la clase factoría general simplemente sea una experta en identificar si los modos de juego que se piden son válidos, si la información que llega no es inválida, etc.

* Replay *

La comunicación de la clase `Replay` con la GUI se realiza de manera adecuada mediante el uso del Patrón *Observador*. Sin embargo, no existe comunicación con la vista de consola, pues es la propia clase la que se encarga de imprimirse. Inicialmente se llevó a cabo esta implementación porque no existía una vista de consola, pero cuando fue incluida en el proyecto, no se adaptaron las *replays*, convirtiéndose en una deuda técnica que ha llegado hasta el día de hoy. La clase es completamente funcional, pero desde el punto de vista de la POO, la clase `Replay` no debería de tener esta responsabilidad.