

1 Diseño y evolución de las clases principales del Modelo

1.1 Diseño del tablero

Toda la lógica que implementa el concepto del tablero queda reflejada a lo largo de todo el proyecto en la clase Board.

1.1.1 Sprint 1

La clase Board es un simple contenedor de cubos, organizados en forma de matriz. No tiene interacción con otros objetos del modelo.

1.1.2 Sprint 2

Ahora la clase Board tiene la funcionalidad de actualizarse a sí mismo una vez se coloca un nuevo cubo, funcionalidad que antes estaba delegada a la clase Game.

1.1.3 Sprint 3

A partir de este Sprint los tableros tienen forma (es decir, la forma no es necesariamente siempre cuadrada) y tamaño elegido por los usuarios. A parte, Board tiene una representación en forma de String y de JSONObject, a través de los métodos toString() y report().

1.1.4 Sprint 5

Ahora, aparte de guardarse los cubos del tablero en forma de matriz, se guardan en forma de lista por ser una representación de los datos muy conveniente en distintas partes del proyecto, entre otras, para la red.

1.2 Diseño de los colores

1.3 Diseño de los cubos

1.4 Diseño del juego

La clase fundamental del juego es la clase Game, la clase principal del modelo.

1.4.1 Sprint 1

La clase Game responde a ejecuciones del PlaceCubeCommand, colocando nuevos cubos en las casillas seleccionadas, actualizando el tablero, manejando los turnos de los jugadores y actualizando sus puntos.

1.4.2 Sprint 2

A partir de este momento, Game no se encarga de actualizar el tablero y las puntuaciones de los jugadores, sino de únicamente pasarle al tablero los cubos nuevos que tiene que insertar. El tablero, a partir de ese cubo, se actualiza a sí mismo, y los cambios de cubos actualizan las puntuaciones de los jugadores.

1.4.3 Sprint 3

Game tiene su propia representación en forma de String y de JSONObject, a través de los métodos toString() y report().

1.4.4 Sprint 4

A partir de este sprint, Game pasa a ser una clase abstracta y se pasa a tener dos modos de juego, el clásico (jugadores individuales) y el modo por equipos, que son implementados por las clases heredadas de Game: GameClassic y GameTeams. También se implementa el patrón MVC, por lo que Game (modelo) pasa a tener una lista de observadores y métodos para el envío de notificaciones a estos.

1.4.5 Sprint 5

Game se adapta con ciertos cambios y nuevos métodos para soportar el juego en línea.

1.4.6 Sprint 6

En este momento Game sufre su mayor refactorización. Esta clase pasa a extender de la clase Thread, de forma que ya no funciona ejecutando comandos en el mismo momento de su creación, sino que estos se ponen en espera y Game, que está en todo momento funcionando y comprobando si hay nuevas peticiones puestas en espera, las ejecuta cuando puede. Esto nos permite evitar problemas de desbordamiento con el cálculo de jugadas por parte de las inteligencias artificiales, aparte de permitir el funcionamiento esperado de la vista sin comprometer su rendimiento. Aparte, Game deja de llevar a cabo el manejo de turnos y se delega esa responsabilidad a la clase TurnManager, que es invocada tras cada jugada para que ejecute (si procede) el siguiente turno.

- 1.5 Diseño de los jugadores**
- 1.6 Diseño de los equipos**
- 1.7 Diseño del gestor de turnos**
- 1.8 Diseño de los estados del juego**
- 1.9 Diseño de las replays**
- 1.10 Diseño de las Inteligencias Artificiales**
- 1.10.1 Sprint 5**

Las estrategias son externas al modelo. El cómputo de movimientos a través de estas estrategias se hace a partir de la vista, a través de la clase `PlayerView`, que representa al `Player` en la vista y se encarga de ejecutar sus acciones. Por razones de encapsulación, las estrategias no tienen acceso al modelo y realizan sus cálculos a través de `GameStates`.

1.10.2 Sprint 6

Las estrategias ahora forman parte del modelo, siendo atributo de aquellos `Players` controlados por la máquina. Como en este punto el manejo de turnos se lleva a cabo por la clase `TurnManager` y el modelo funciona en su propia hebra, la clase `PlayerView` desaparece del proyecto y son los propios `players` quienes ejecutan las estrategias, que a nivel abstracto resulta mucho más intuitivo. Por la hebra del modelo, cuando las estrategias terminan de calcular el siguiente movimiento, este no se ejecuta en ese mismo instante, sino que se deja en espera en el modelo hasta que este pueda ejecutarlo.

2 Diseño del Controlador

3 Diseño de la Vista de GUI

3.1 Diseño del menú principal y pantallas pre-juego

3.2 Diseño de la pantalla de juego

4 Diseño de la Vista de Consola

4.1 Diseño del menú principal y pantallas pre-juego

4.2 Diseño de la pantalla de juego

5 Diseño de la red

5.1 Diseño del servidor

5.2 Diseño de los clientes

6 Diseño y evolución de las Historias de Usuario

6.1 Como usuario quiero poder jugar a Rolit con una interfaz agradable

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

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

Sprint (1 JUANDI)

Sprint (2 JUANDI)

Sprint (3 JUANDI)

Sprint (4)

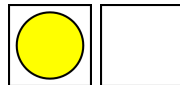
Console - JUANDI

aaaaa

GUI

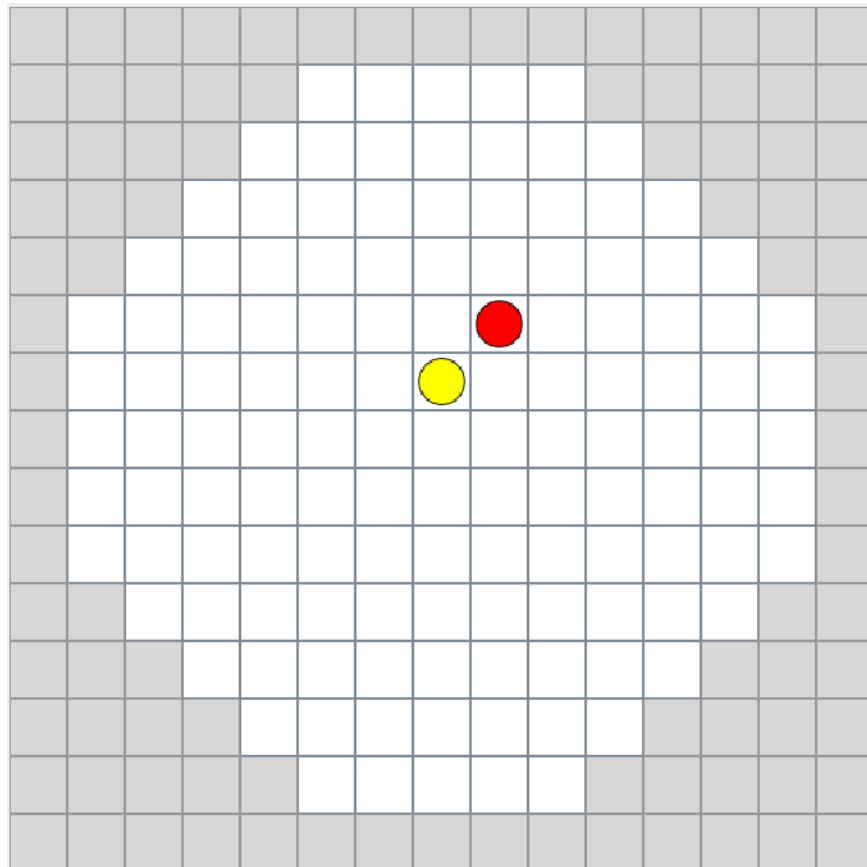
En este sprint se desarrolló una primera versión de la interfaz gráfica, lo cual involucra a un gran número de clases y métodos. Iniciemos un recorrido por cada uno de los componentes visuales explicando su finalidad

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

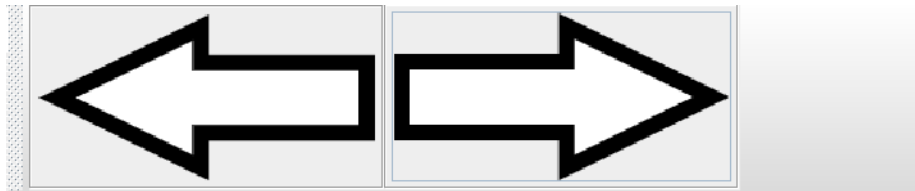


La clase *BoardGUI* extiende de *JPanel* y es la encargada de visualizar el tablero de la partida. Se puede apreciar que el tablero está formado por un conjunto de *CeldaGUI*.

ControlPanel

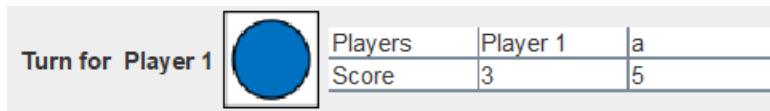


El *ControlPanel* es una *JToolBar* que cuenta con un botón para guardar partidas, que puede ser pulsado en cualquier momento durante la ejecución del juego.



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

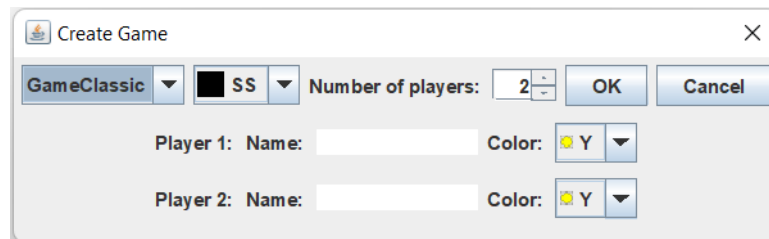
TurnAndRankingBar



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.

StatusBar

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.

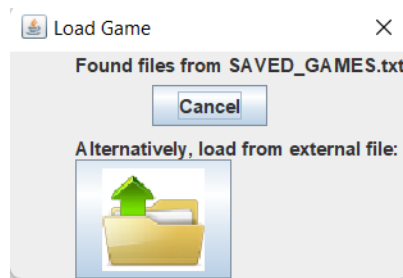
En esta pantalla el usuario elige el modo de juego para la partida (*GameClassic* o *GameTeams*), la forma y tamaño del tablero (cuadrado, círculo o rombo, pequeño, mediano o grande), el número de jugadores (entre 2 y 10, ambos inclusive) y el nombre y color de cada jugador.

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.

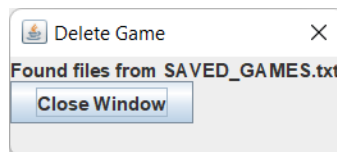
LoadGameDialog



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



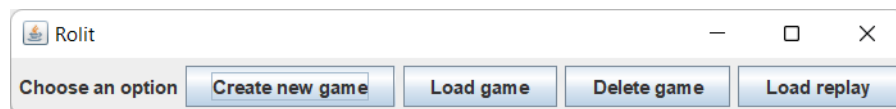
En esta pantalla se muestra la lista de partidas guardadas y un botón para confirmar el borrado. Con esto, si se selecciona una de las partidas guardadas y se presiona el botón inferior, la lista se elimina de la lista de partidas guardadas.

StatusBar



La *StatusBar* 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*.

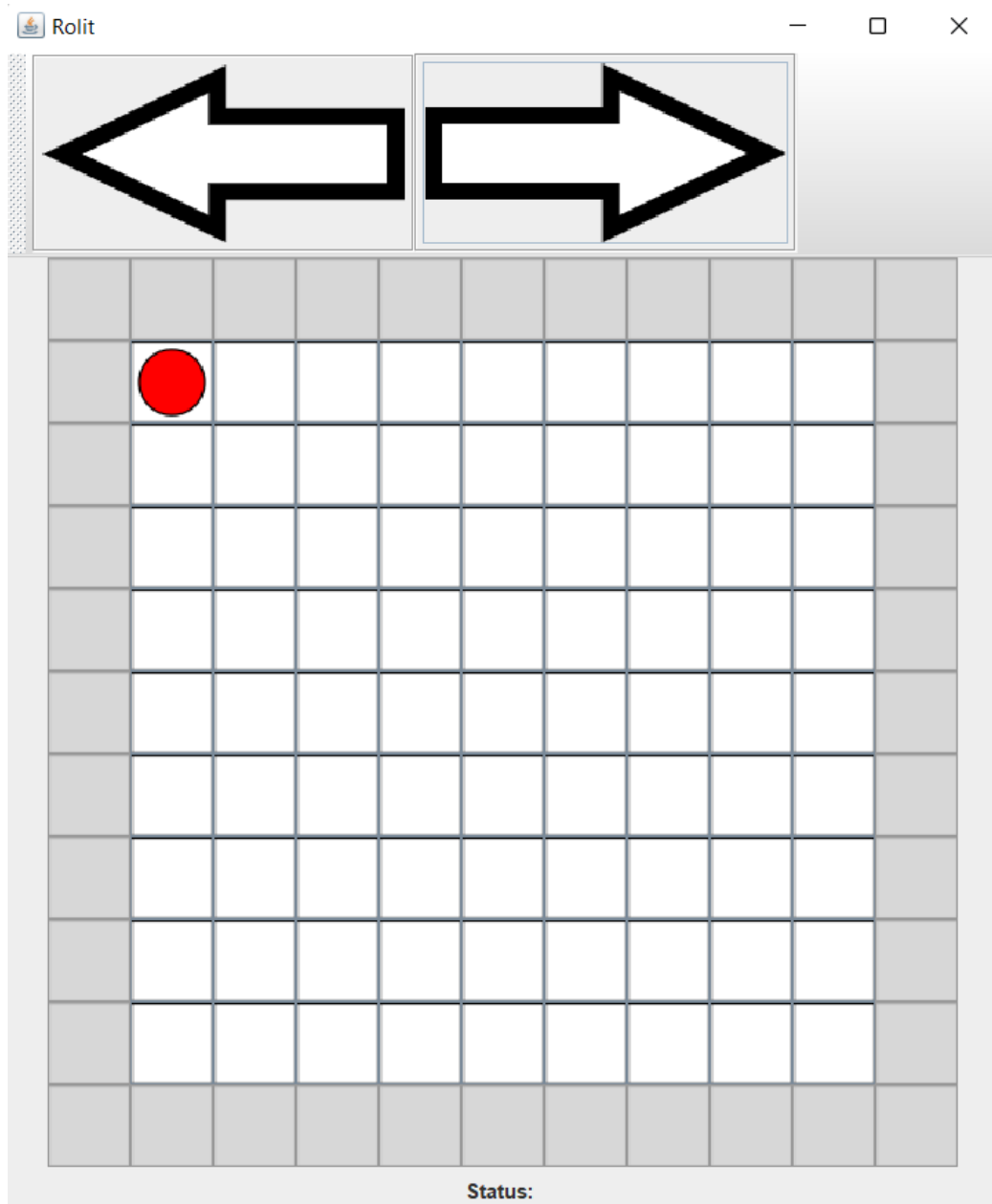
Rolit

Turn for Player1

| | | |
|---------|---------|---------|
| Players | Player1 | Player2 |
| Score | 0 | 0 |

Status:

Jugar una partida



Reproducir una *replay*

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)

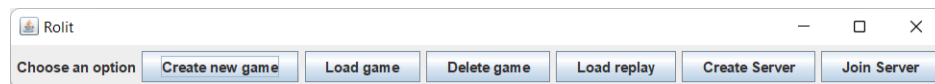
Console - JUANDI

aaaaa

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.



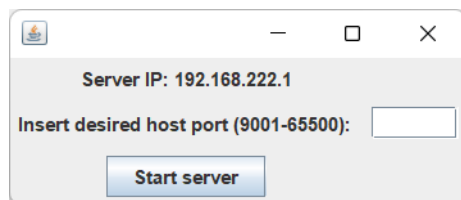
CreateGameDialog



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.

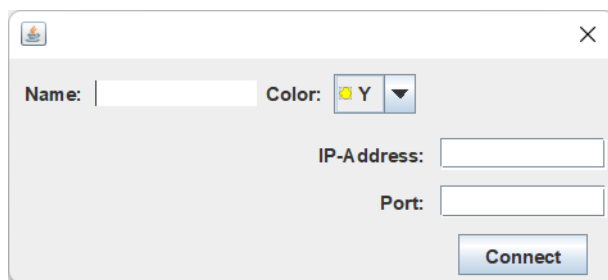
CreateServerDialog



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.

Funcionamiento 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.

| <<Java Class>> BoardGUI view |
|--|
| <ul style="list-style-type: none"> ▪ nFilas: int ▪ nColumnas: int ▪ celdas: CeldaGUI[][] ▪ game: Game ▪ replay: Replay ▪ lastCubeAdded: JSONObject |
| <ul style="list-style-type: none"> • BoardGUI(Game) • BoardGUI(Replay) ▪ shapeMatrixToSideButtonLength(boolean[][]):int • crearTablero(JPanel):void • update(Game, Board):void • onError(String):void • onCommandIntroduced(Game, Board, Command):void • onReplayLeftButton():void • onReplayRightButton():void ▪ updateReplay():void • onRegister(Game, Board, Command):void • onGameFinished():void • onTurnPlayed(String, Color):void • onGameStatusChange(String):void • onFirstPlay(String, Color):void • onReplayStatusChange(String):void |

| <<Java Class>> BoardGUI view.GUIView |
|--|
| <ul style="list-style-type: none"> ▪ nFilas: int ▪ nColumnas: int ▪ celdas: CeldaGUI[][] ▪ ctrl: Controller ▪ state: State ▪ replay: Replay ▪ lastCubeAdded: JSONObject |
| <ul style="list-style-type: none"> • BoardGUI(Controller, State) • BoardGUI(Replay) ▪ shapeMatrixToSideButtonLength(boolean[][]):int • crearTablero(JPanel):void • update():void • onError(String):void • onReplayLeftButton():void • onReplayRightButton():void ▪ updateReplay():void • onRegister(State):void • onTurnPlayed(State):void • onGameStatusChange(State):void • onReplayStatusChange(String):void • onGameFinished(List<Rival>, String):void |

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*.

| <<Java Class>> State replay |
|---|
| <ul style="list-style-type: none"> △ command: String △ game: Replayable |
| <ul style="list-style-type: none"> • State(String, Replayable) • State(Replayable) • toString():String • report():JSONObject • getCubes():JSONArray • getTurnColorShortcut():char ▪ findTurnPlayer():int • getShape():String • getRivals():JSONArray • getType():String • getCommand():String • getFirstPlayerName():String • getFirstPlayerColorShortcut():char • getTurnName():String |

Sprint (6)

Console - JUANDI

aaaaa

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 utilizaba los componentes visuales de Java poder 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 re-estructuraciones en los Layouts y la inclusión de nuevas imágenes e iconos MainWindow



 Rolit




—

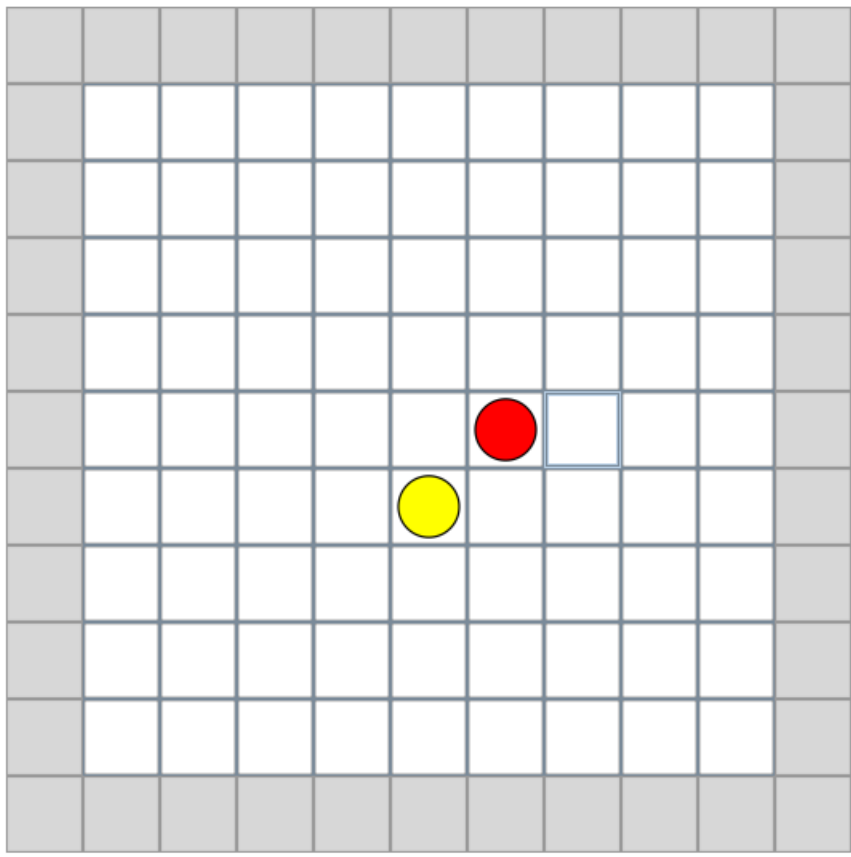
□

×

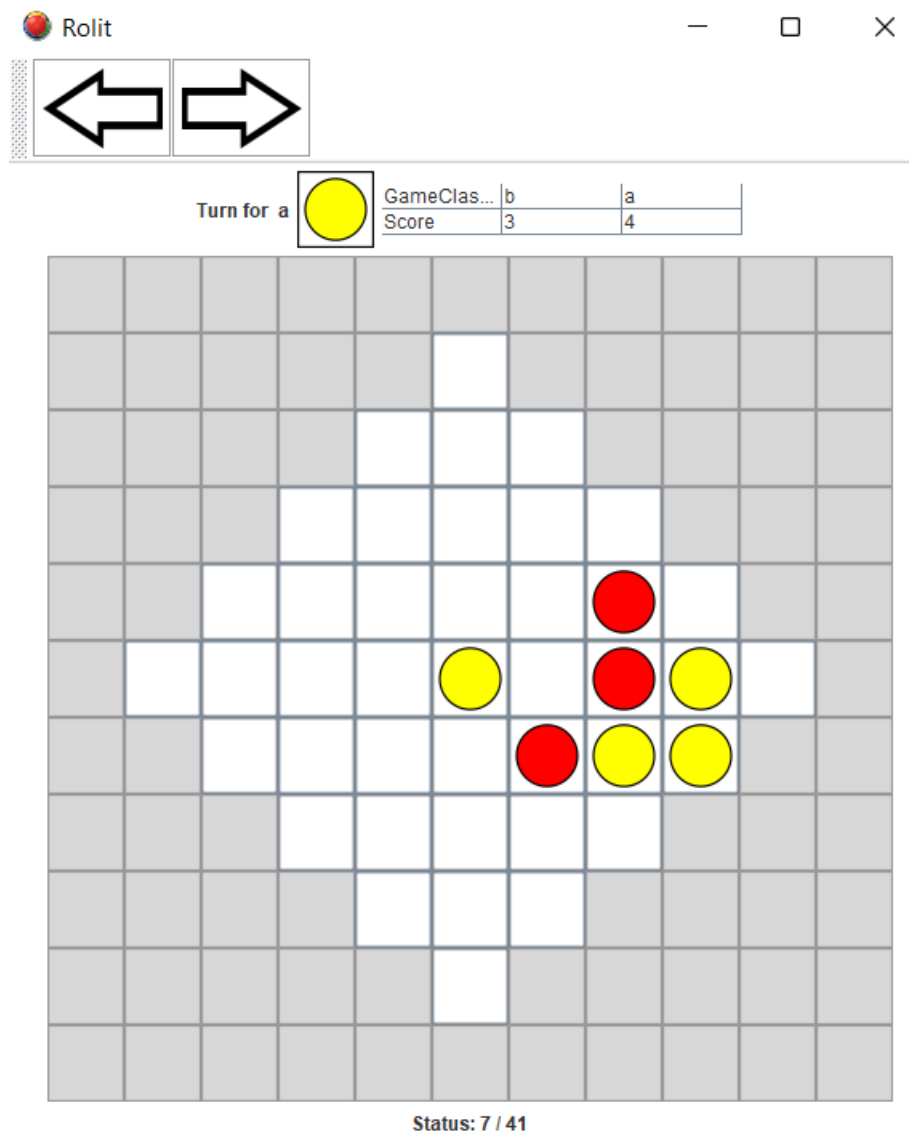
Turn for b



| | | |
|-------------|---|---|
| GameClas... | 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: ■ Yellow Alt:

- *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...

GameTeams Shape: ■ Small Number of players: Number of teams:

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

 Create Game ×

GameClassic Shape: ■ Small Number of players:

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


 Create Game ×

GameClassic Shape: ■ Small Number of players:

Player 1: Name: Color: ■ Yellow Alt:

Player 2: Name: Color: ■ Yellow Alt:

LoadFileDialog

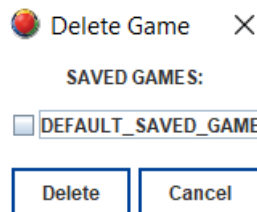
 Load File ×

SAVED FILES:

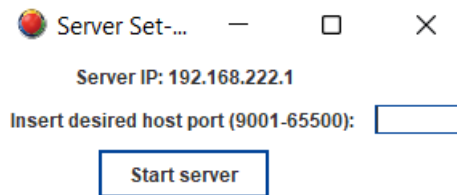
☐ DEFAULT_SAVED_GAME.json

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

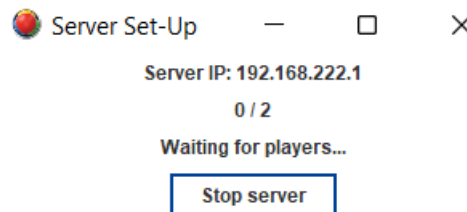
DeleteGameDialog



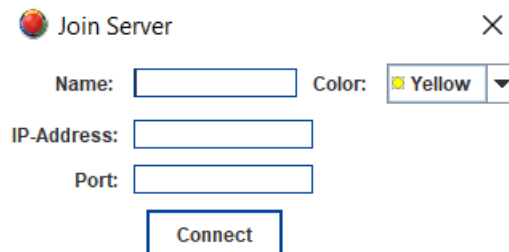
CreateServerDialog



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.

Waiting for all players to be connected...

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

6.3.1 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 ejecute inmediatamente después de su cálculo.

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:

Antes de explicar la implementación de esta estrategia en el caso de Rolit, debemos explicar primero en qué consiste la estrategia Minimax en teoría de juegos:

Estrategia Minimax en teoría de juegos:

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

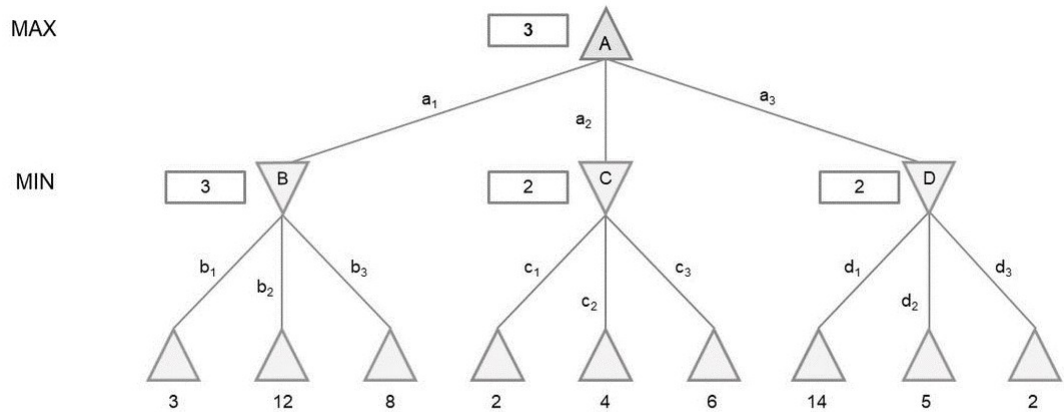
suponga, el resultado real será siempre igual o mejor al calculado, de forma que el cálculo es fiable.

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 cual 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.

Debido a lo exhaustivos que resultan estos cálculos (y tras comprobación empírica simulando numerosas partidas) la estrategia `MinimaxStrategy` representa el nivel difícil de las inteligencias artificiales.

Ahora, antes de la explicación más técnica, observemos que la estrategia `GreedyS-`

strategy se puede implementar aplicando una MinimaxStrategy en la que solo se explora un nivel de profundidad, puesto que en este nivel se busca la jugada que más puntos garantice, y no se sigue buscando más allá. Por tanto, a nivel de clases, la clase GreedyStrategy es heredera de MinimaxStrategy, y el atributo de profundidad máxima pasa a valer 0.

Pasamos ahora a explicar la implementación de estas estrategias:

Para la simulación de movimientos pensamos originalmente en usar la clase Board para colocar cubos y evaluar resultados, pero no tardamos en darnos cuenta de que esto era inviable, puesto que los métodos de Board tienen una comunicación con otras clases que no deseamos para esto, puesto que nosotros simplemente queremos hacer simulaciones, y no cambios reales.

Es por esto que fue necesario crear otra representación del tablero puramente funcional y adaptada a la simulación de movimientos, de donde surgió la siguiente clase:

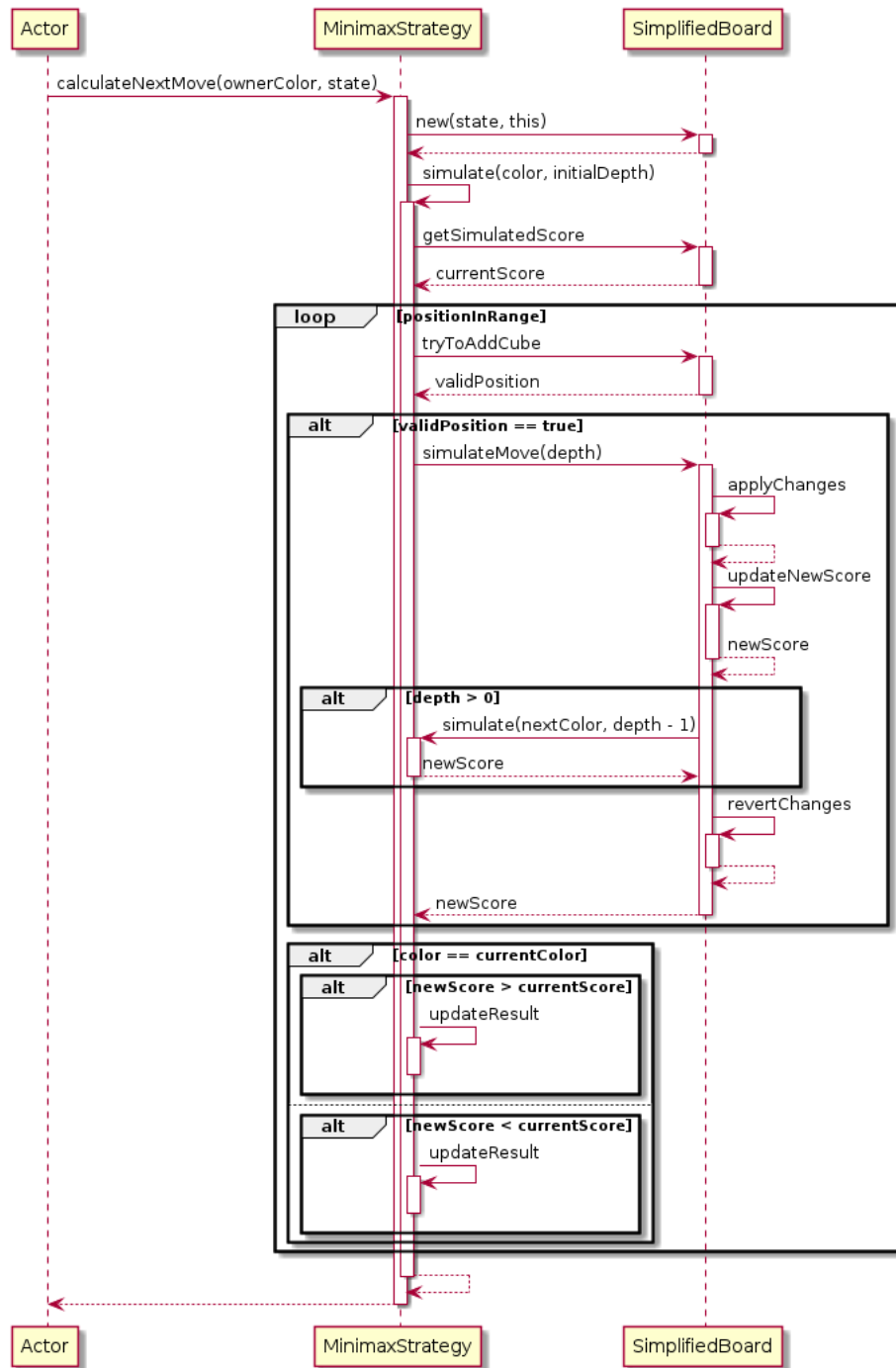
SimplifiedBoard:

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 esta 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.

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:

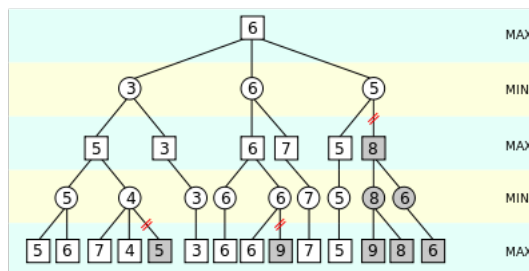


Sprint (6)

Por motivos de eficiencia de MinimaxStrategy, se ha implementado de forma complementaria la poda alfa-beta, que se explica a continuación:

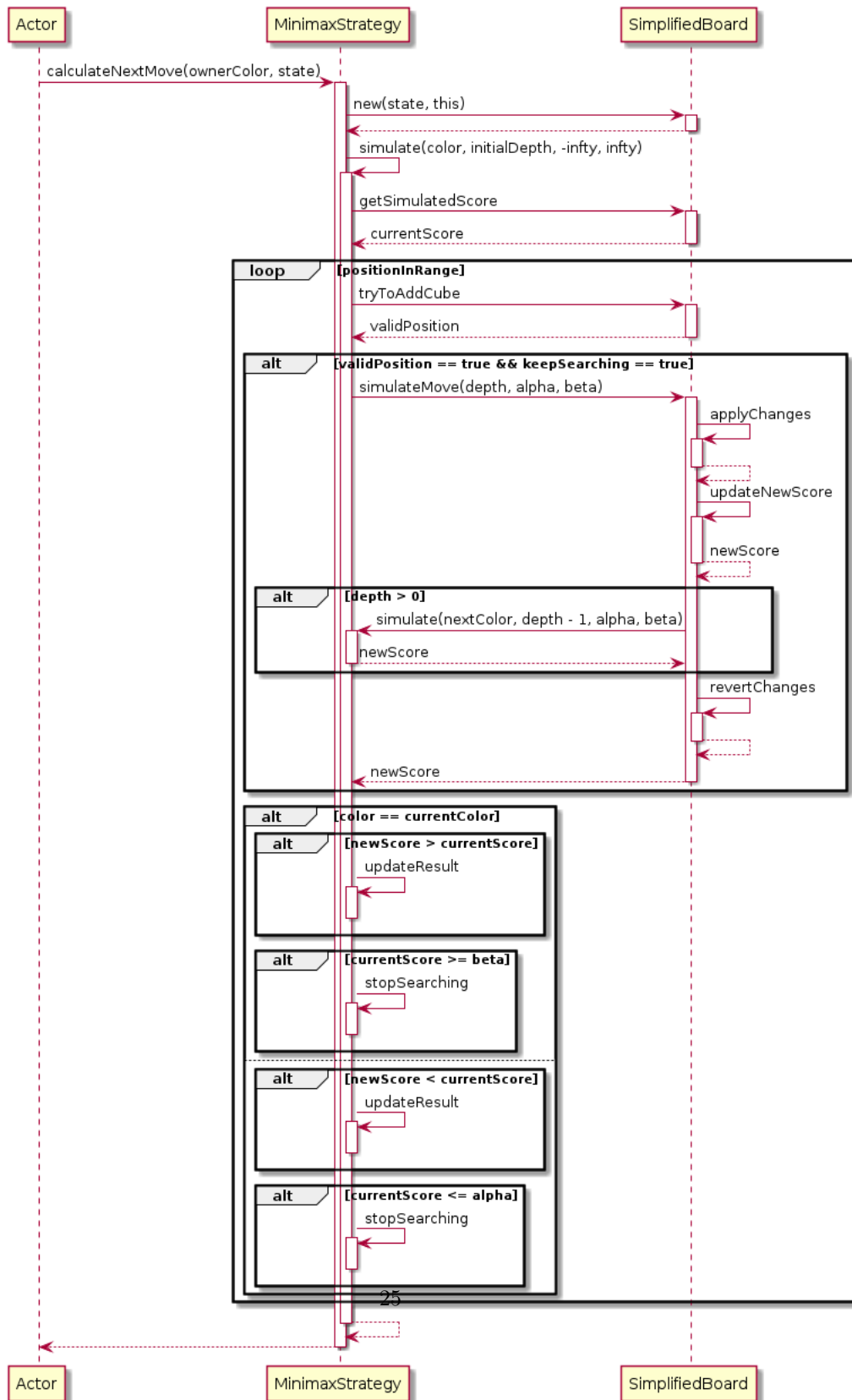
Poda alfa-beta: La poda alfa-beta es una mejora del algoritmo Minimax. Se mantienen dos valores, alfa y beta, que representan respectivamente la puntuación mínima que se llevará el jugador maximizador y la puntuación máxima que se asegura el jugador minimizados. Inicialmente, alfa es $-\infty$ y beta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

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

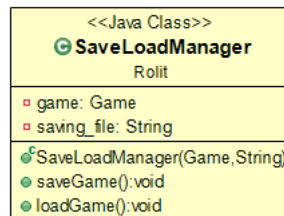


6.4 Como usuario quiero que Rolit introduzca características innovadoras siendo intuitivo y cómodo de jugar

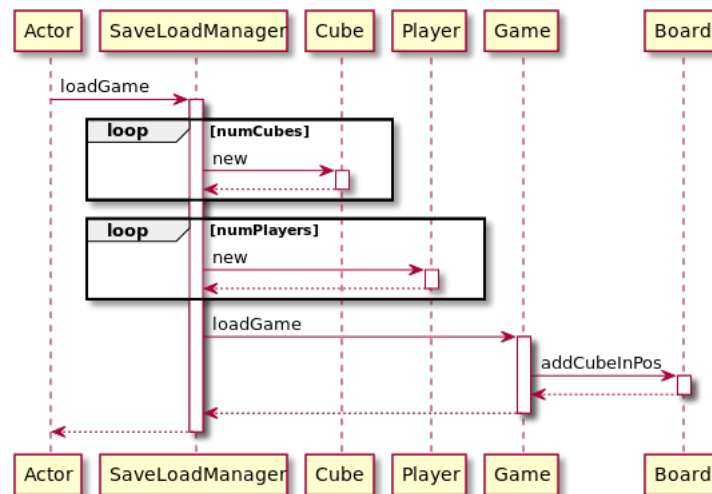
6.4.1 Como usuario, me gustaría que se pudiese guardar y cargar partida para continuar más tarde porque permite poner en pausa el juego

Sprint (1)

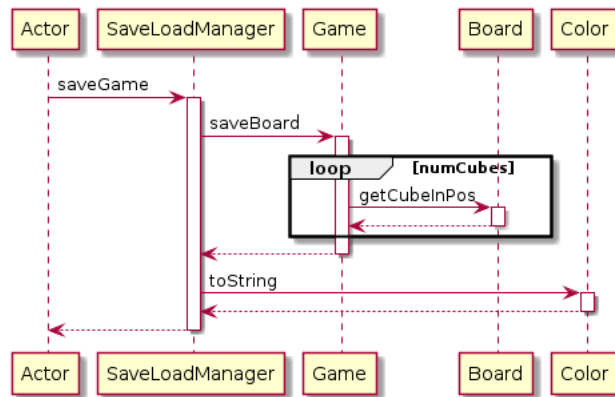
Desde el primer sprint tuvimos clara la necesidad de tener una clase que se encargase de gestionar la comunicación del programa con el exterior para cargar y guardar partidas, el *SaveLoadManager*.



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



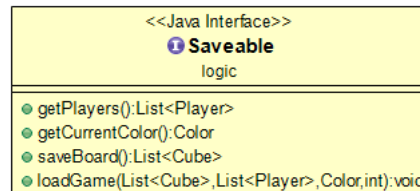
Cargar partida



Guardar partida

Sprint (2)

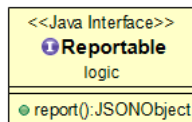
En la versión anterior del *SaveLoadManager*, la clase trabajaba a nivel de *Game*, lo cual no tenía mucho sentido desde el punto de vista de la programación orientada objetos, pues no es necesario acceder a todos sus métodos. Por esta razón, se creó una interfaz *Saveable* que debía implementar *Game*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



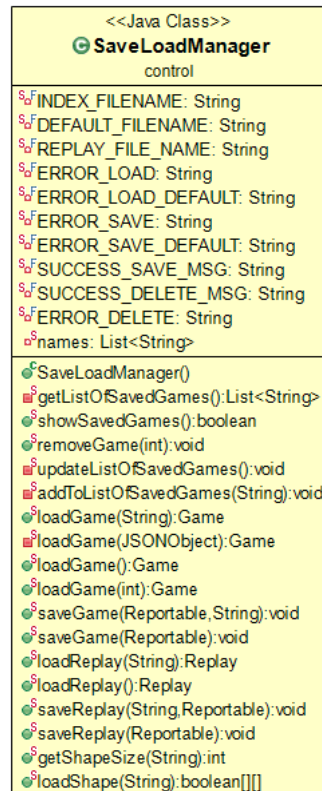
Sprint (3)

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 JSONObjects.



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.

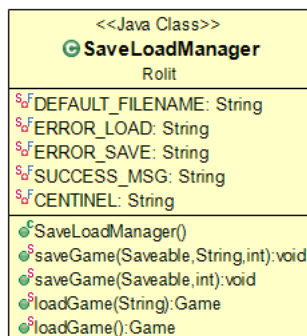
Sprint (6)

La similitud 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.

6.4.2 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.



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

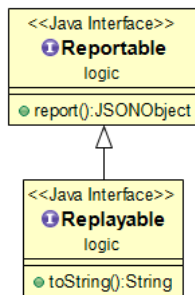
Sprint (3)

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.

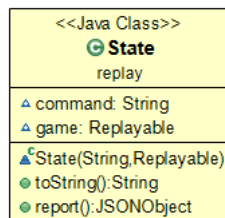
Inicialmente se planteó que los estados almacenaran únicamente el cubo que se añade en su turno, pero esta idea fue descartada debido a que la colocación de un cubo puede llegar a afectar a cualquier parte del tablero, teniendo que incluir la lógica correspondiente para calcular los cambios. Por ello,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

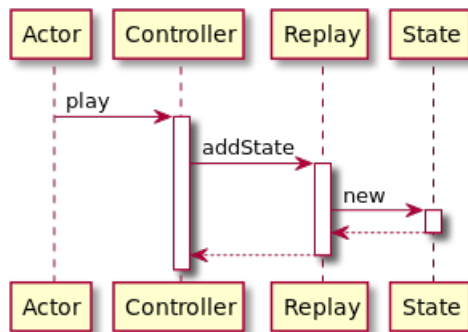
De esta manera surge la interfaz `Replayable`, que extiende de la interfaz `Reportable` y que contiene dos métodos: `toString()`, para facilitar la visualización en la vista de consola, y `report()`, para poder almacenar los estados en formato JSON.



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

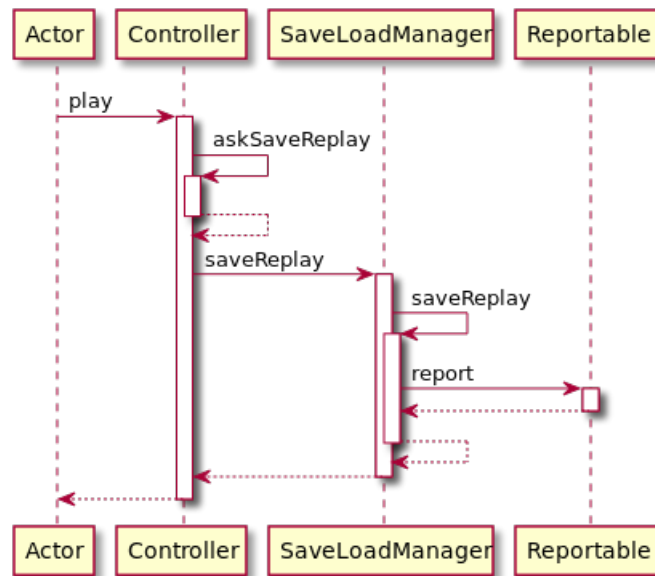


El siguiente paso fue la creación de una clase contenedora de los estados, con la responsabilidad añadida de saber gestionarlos, la clase *Replay*. Esta clase contiene una lista de *State* que se completa durante una partida.



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

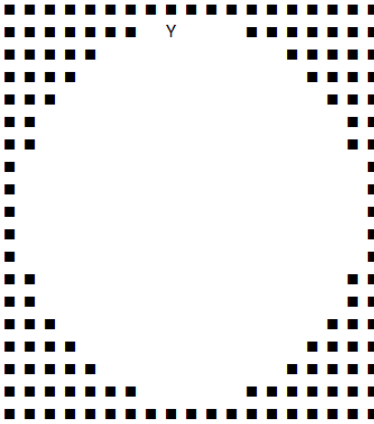
Finalizada la generación de las *replays* era el momento de hacer que pudieran cargarse y ser interpretadas para su correcta visualización. Como ya mencionamos anteriormente, la clase *Replay* es la encargada de gestionar los estados y, por ello, la que contendrá esta lógica.

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 “+” y “-”.

```

State: 1/5
Command > p 1 8
Turno: leo (L)

```



```

> +

```

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

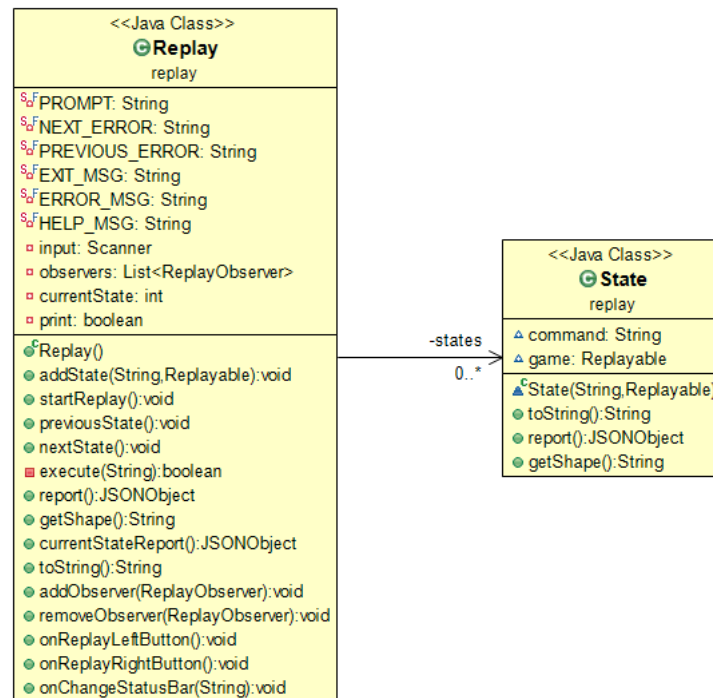
Sprint (4)

Este sprint se caracterizó por la implementación de una GUI funcional, lo que conlleva la adaptación de las *replays* a este nuevo formato. Decidimos que la comunicación entre modelo y vista se llevara 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*.



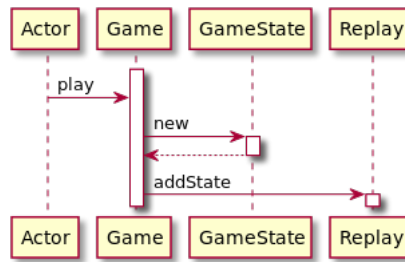
Sprint (5)

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*.

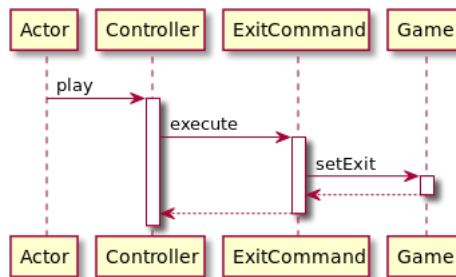
Además, la necesidad de notificar a la vista con el *Replay* para que sea guardada en caso de que así lo desee el usuario, tiene como consecuencia que la clase *Game* sea la encargada de generar sus propias repeticiones. Este proceso puede verse reflejado en el siguiente diagrama de secuencias.



6.4.4 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 ...

6.4.5 Como usuario, me gustaría jugar a Rolit pensando en las posibilidades que brinda el multijugador (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. 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

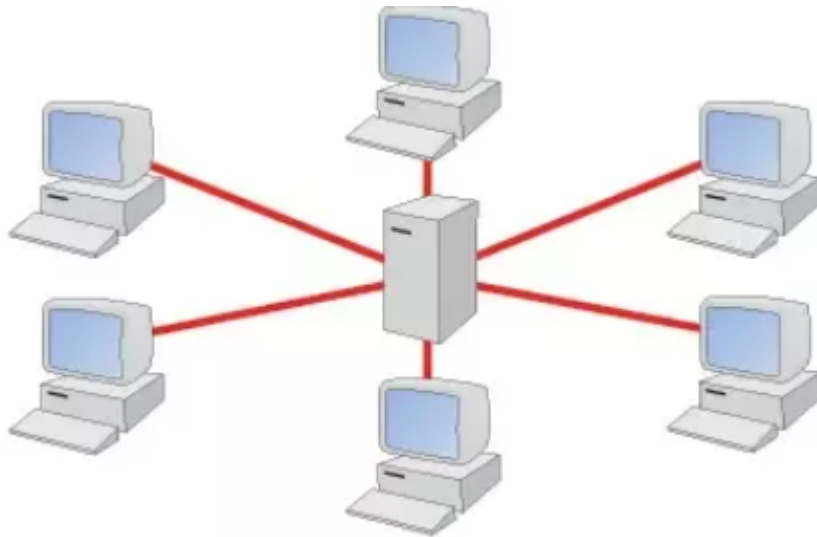
El diseño y evolución pormenorizados de estos diálogos se encuentran en el apartado de la historia de usuario dedicada a la interfaz.

- Reutilizamiento 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.

Fundamentalmente, la conexión en red se basa en una estructura en la 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 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 nuevo Socket 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.

```
//Perspectiva del servidor  
  
ServerSocket serverSocket = new ServerSocket(port);  
Socket socketCliente1 = serverSocket.accept();  
...  
...  
...  
Socket socketClienteN = serverSocket.accept();
```

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

```
//Perspectiva del cliente  
  
Socket socket = new 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 `Strings` se volverán a construir en `JSONObject` por medio de la constructora de `JSONObject` 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.

```
//Perspectiva del servidor
BufferedReader inCliente1 = new BufferedReader(new
InputStreamReader(socketCliente1.getInputStream()));
...
...
...
PrintWriter outClienteN = new PrintWriter(
socketClienteN.getOutputStream(), true);
```

```
//Perspectiva del cliente
BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

PrintWriter out = new PrintWriter(
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:

```
//Envio de mensaje ClienteK Servidor

//Perspectiva del cliente k
out.println(msg);

—>

//Perspectiva del servidor
String msg = inClienteK.readLine();
```

```

//Envio de mensaje Servidor-ClienteK

//Perspectiva del servidor
outK.println(msg);

—>

//Perspectiva del cliente k
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 las 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.

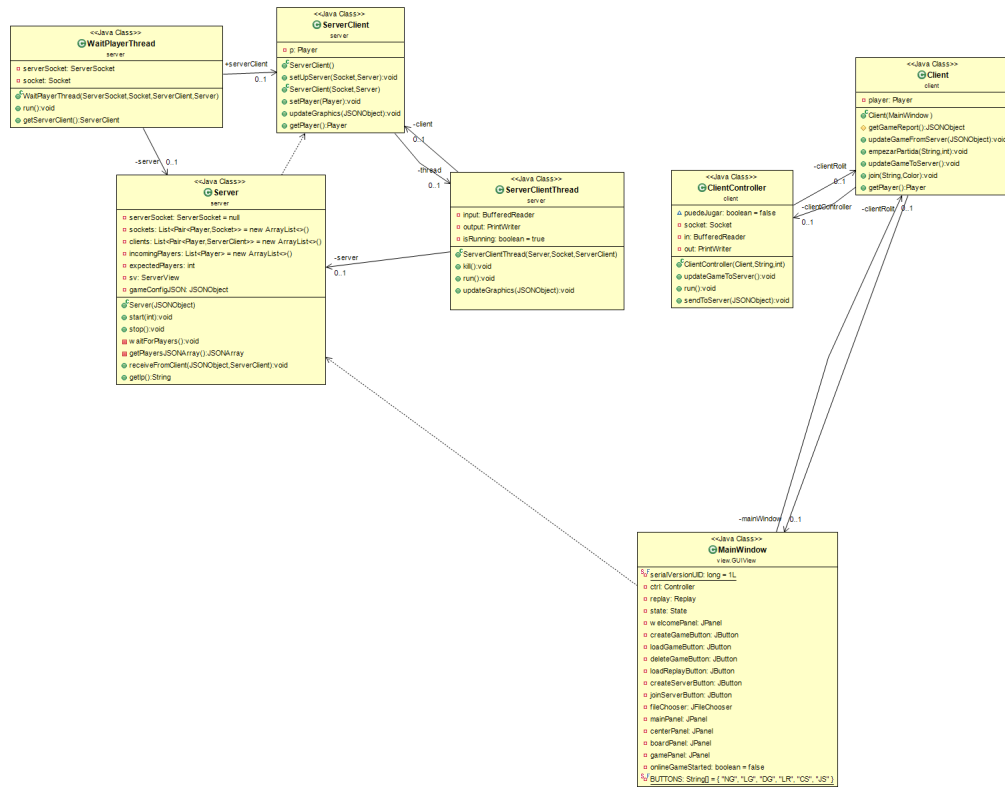
Por ejemplo, podría ocurrir que en el modo por equipos, dos clientes se conecten. Reciben ambos del servidor una lista de equipos en la que pueden conectarse. Se ve que es absurdo imponer un orden de quién envía la información del equipo elegido, pues esta es imposible de anticipar. Puede darse que el 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 programación 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 procesamiento de forma secuencial para evitar errores imprevistos), que pertenece a la clase `Server`, que de forma sincronizada procesa este mensaje.

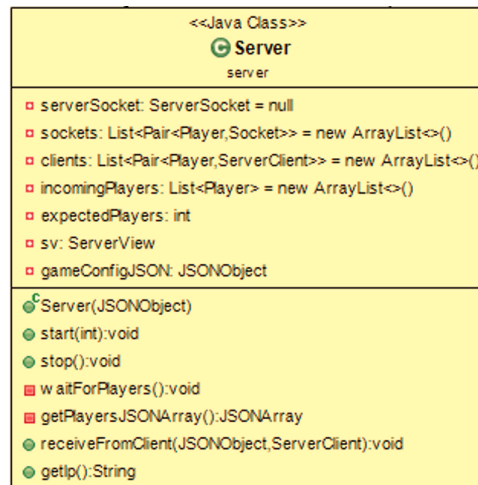
El cliente se encuentra en una situación parecida. ¿Qué ocurre si recibe una información del servidor, la procesa y mientras se da este procesamiento, recibe otro mensaje del servidor? Necesitará el cliente, por tanto, dos hilos: un hilo que se dedique a recoger mensajes, y otro hilo que se dedique a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

Una vez el modelo de red ha sido completamente explicado, procedemos a de-

tallar los detalles de implementación relativos al juego en específico. Es preciso introducir el diagrama de clases relativo a este Sprint.



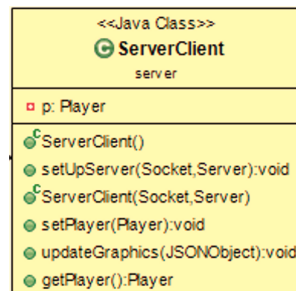
Empezamos a detallar la estructura del servidor. ServerView, especificado en el apartado de diseño correspondiente, tiene el papel de pasar la información pertinente para el funcionamiento de la clase Server. Partimos de la clase Server, clase que gestiona los clientes desde la perspectiva del servidor y que procesa los mensajes emitidos desde los clientes.



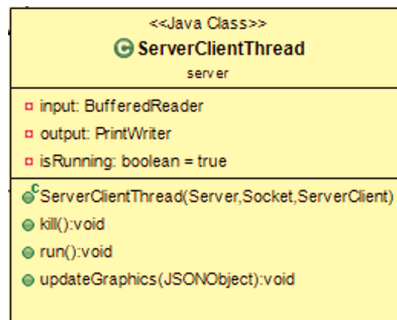
Como hemos anticipado, `Server` debe poseer un único `ServerSocket` a través del cual se abre el servidor. Posee el atributo `expectedPlayers` recibido desde la GUI, en el que el usuario que ha abierto el servidor especifica qué número de jugadores se conectarán al servidor. Es de vital importancia conocer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

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.

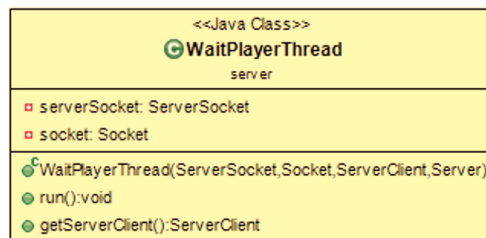


ServerClient posee una referencia a su ServerClientThread y viceversa. Esto es porque:

- 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.

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



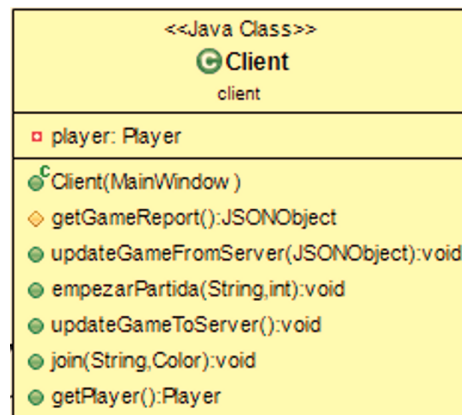
Se crean un número "expectedPlayers" (atributo de Server) de instancias de esta clase, cada una de ellas con un ServerClient asociado (por tanto, ServerClient es atributo de WaitPlayerThread). En su método run se encarga de ejecutar el método serverSocket.accept() que especificamos anteriormente. En cuanto se acepta la conexión, se procede a llamar al método setUpServer de ServerClient para que este cree su ServerClientThread.

La necesidad de hacer esto de forma paralela justifica la creación del hilo WaitPlayerThread. Para evitar problemas de concurrencia, en esta clase los atributos Server y ServerClient son volátiles para que todos los hilos WaitPlayerThread conozcan en tiempo real el estado de Server y ServerClient.

En particular, Server debe ser volatile pues debe ir actualizando su lista de Player-Socket en tiempo real y de forma organizada de 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.

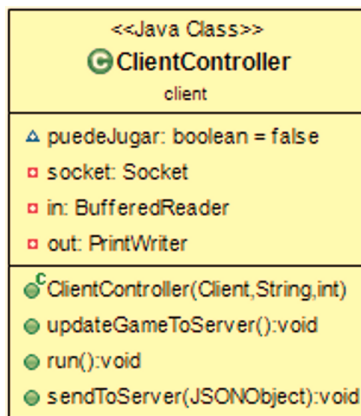
El punto de partida es la clase Client, que será un intermediario entre su thread de recepción de mensajes y la GUI.



Como Client es un intermediario, MainWindow precisa una referencia de Client y Client una de MainWindow; es decir, la comunicación es bidireccional. La razón de esto es:

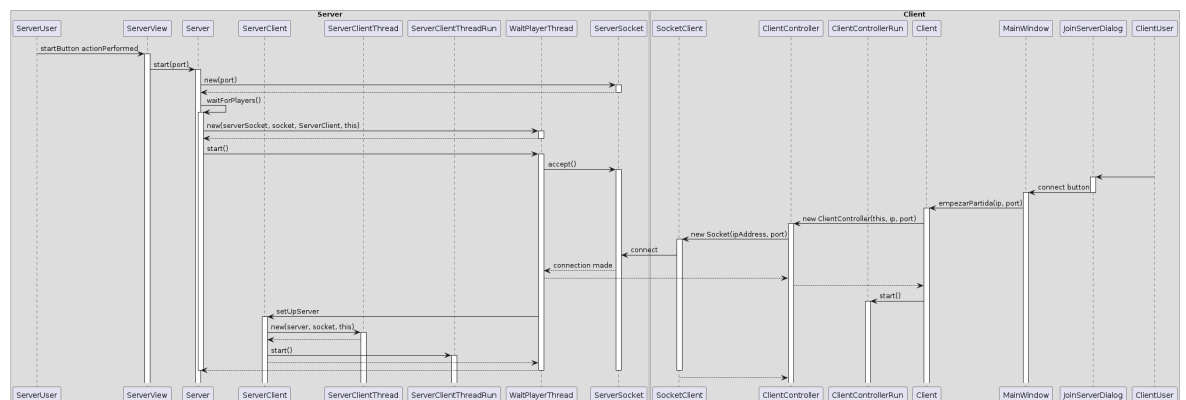
- Al hacerse una jugada en el modelo, como se ha especificado, necesitamos pasar el nuevo estado del juego al servidor para que este, a su vez, se la pase al resto de clientes. Por tanto, MainWindow pasa el estado del juego a su Client. MainWindow, a su vez, obtiene este estado del juego desde el controlador. 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.



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.

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.



Observamos que al pulsar el botón de Start Server en el diálogo de abrir un servidor, se llama crea Server pasándole como parámetro un JSON (gameConfigJSON) correspondiente a una configuración del juego básica especificada desde la GUI, que el servidor rellenará añadiendo los jugadores pertinentes. En el constructor de Server se almacena este atributo JSON y se llama al método start(port) que crea el ServerSocket. Posteriormente, llamada al método wait-ForPlayers 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 WaitPlayerThread y se llama a start(). El WaitPlayerThread, en su método run() llamado desde start(), espera en el método de serverSocket.accept() que se conecte un cliente.

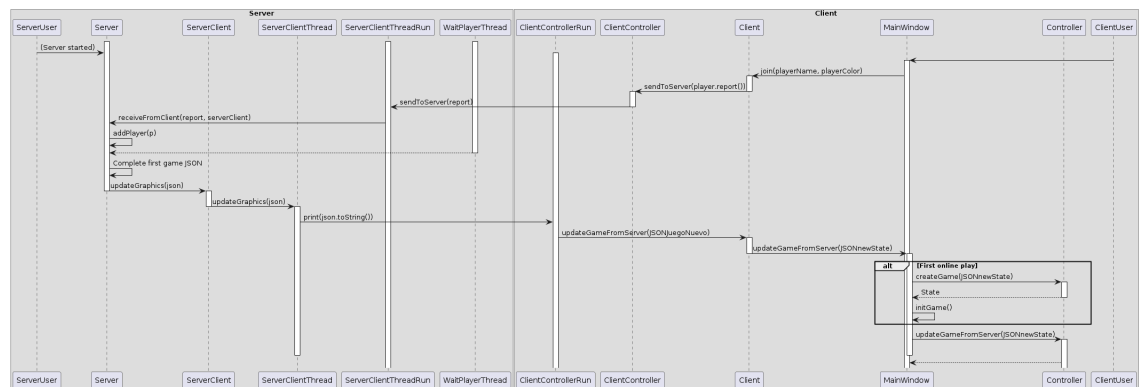
Un cliente, por medio de JoinServerDialog, se conecta al servidor desde la IP y puerto requeridos. Pasa por el MainWindow quien a su vez llega al cliente (Client) con el método empezarPartida, en base a las relaciones ya descritas. El cliente crea su thread ClientController. En la constructora de ClientController 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 WaitPlayerThread 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 WaitPlayerThread espera. El mecanismo pormenorizado es el siguiente:



1. Tras ejecutarse el hilo ClientController, el método empezarPartida 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
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 o información respecto a un nuevo jugador añadido. 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 WaitPlayerThread; se deja un segundo de cortesía para dar un amplio margen a los equipos más lentos.

El método run() de WaitPlayerThread 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 waitForPlayers procede a crear relleno 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 players. Se rellenan 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 crea 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.

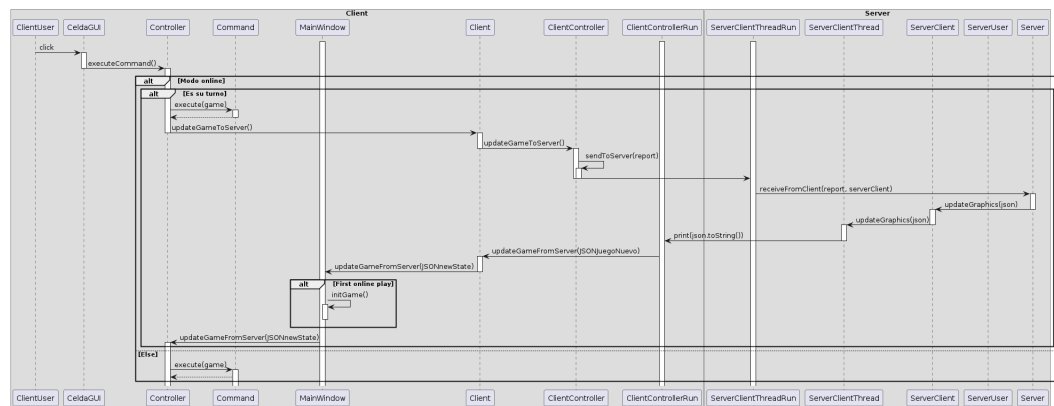
```
public void updateGameFromServer(JSONObject o) {
```

```

Game newGame = GameBuilder.createGame(o);
newGame.updateGameFromServer(game.getObserverList());
game = newGame;
}

```

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.

```

if (onlineMode) {
    if (game.getCurrentPlayer().getColor()
        .equals(clientRolit.getPlayer().getColor())) {
        command.execute(game);
        clientRolit.updateGameToServer();
    }
}
else
    command.execute(game);

```

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.

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

- Una clase creada para tal efecto es `ChooseTeamFromServerDialog`.



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.
(INSERTAR DIAGRAMAS DE SECUENCIA UML)

Sprint (7)

En cuanto a la red, si bien está concluida 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.

Los diagramas UML finales son los siguientes: