

# **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 herederas 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)

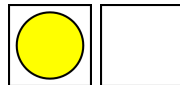
### Console - JUANDI

aaaaa

#### GUI

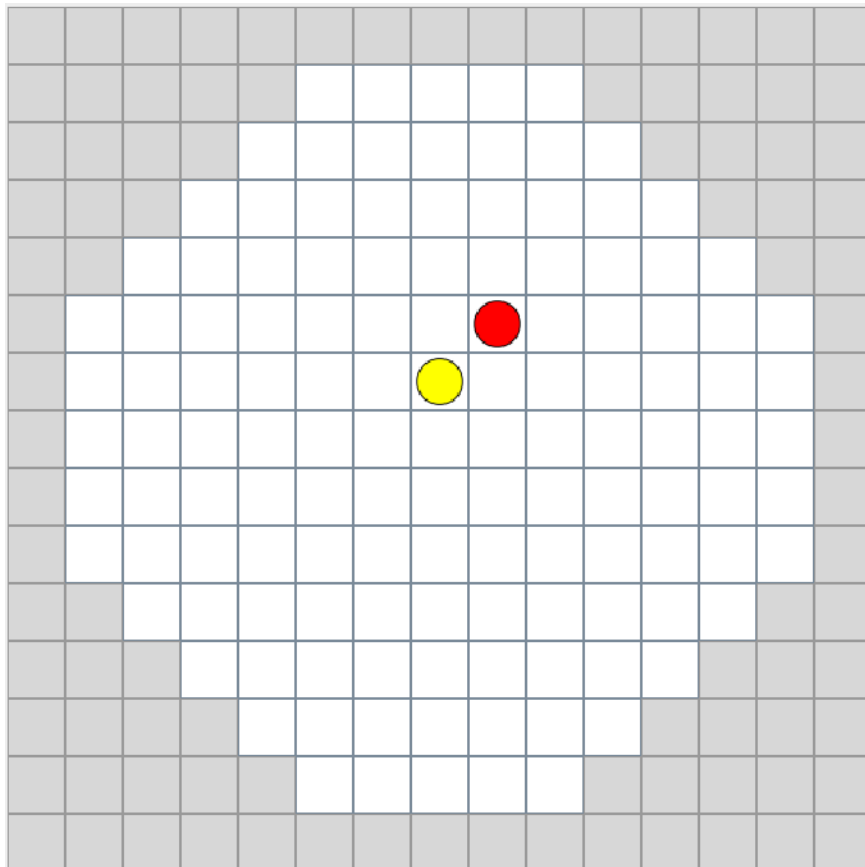
En este sprint se desarrolló una primera versión de la interfaz gráfica, lo cual involucra a un gran número de clases y métodos. Iniciemos un recorrido por cada ventana explicando las clases involucradas y su funcionamiento.

#### 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

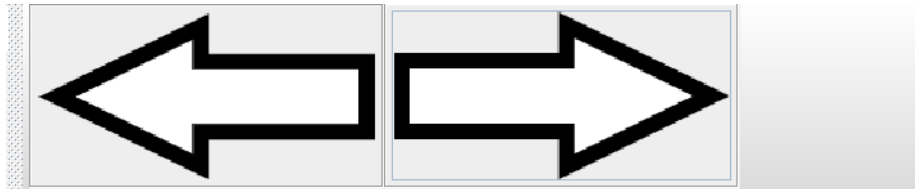


La clase *BoardGUI* extiende de *JPanel* y es la encargada de visualizar el tablero de la partida. Se puede apreciar que el tablero está formado por un conjunto de *CeldaGUI*.

#### ControlPanel

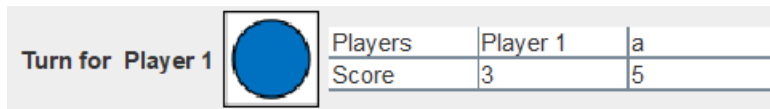


El *ControlPanel* es una *JToolBar* que cuenta con un botón para guardar partidas, que puede ser pulsado en cualquier momento durante la ejecución del juego.



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.

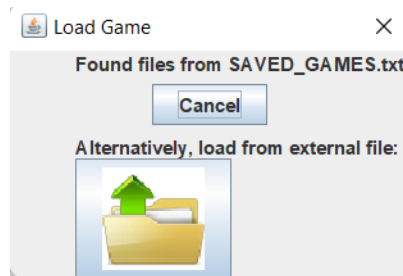
En esta pantalla el usuario elige el modo de juego para la partida (*GameClassic* o *GameTeams*), la forma y tamaño del tablero (cuadrado, círculo o rombo, pequeño, mediano o grande), el número de jugadores (entre 2 y 10, ambos inclusive) y el nombre y color de cada jugador.

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.

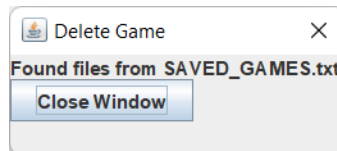
LoadGameDialog



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.

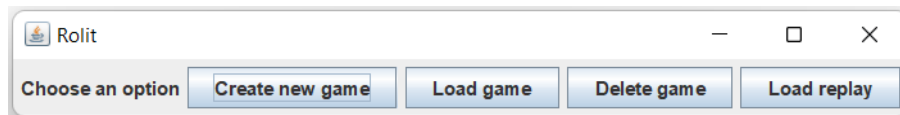
DeleteGameDialog



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.

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*.

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*.

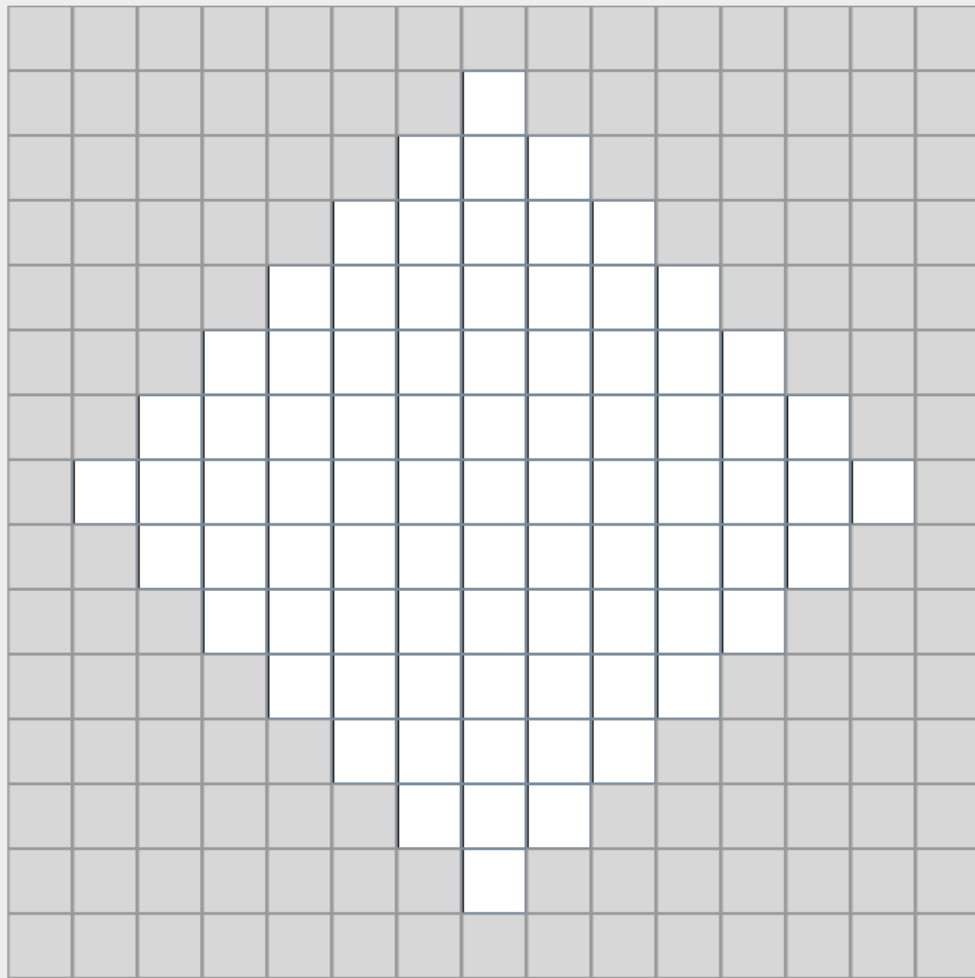
Rolit



Turn for Player1



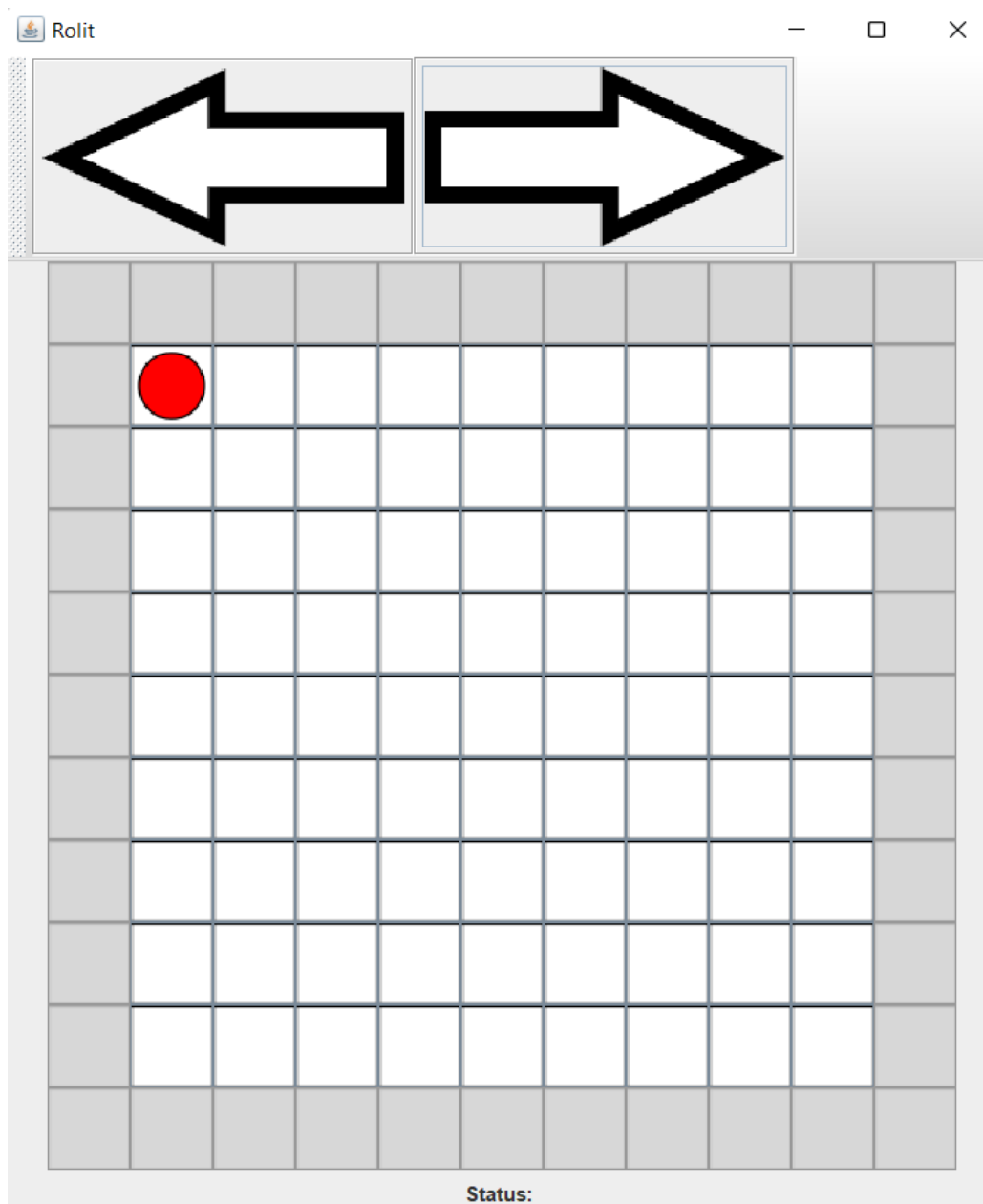
Players	Player1	Player2
Score	0	0



Status:

Jugar una partida





## 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.

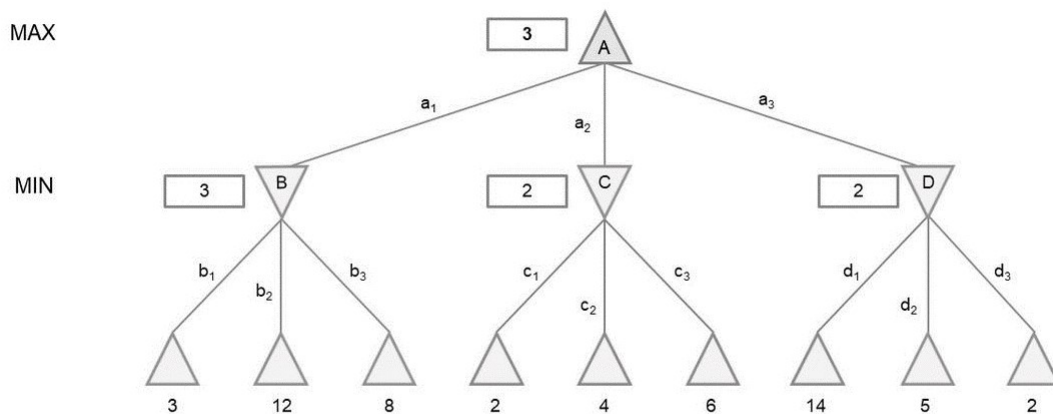
La mejor forma de explicar esto es a través de un ejemplo:

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 GreedyStrategy 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.

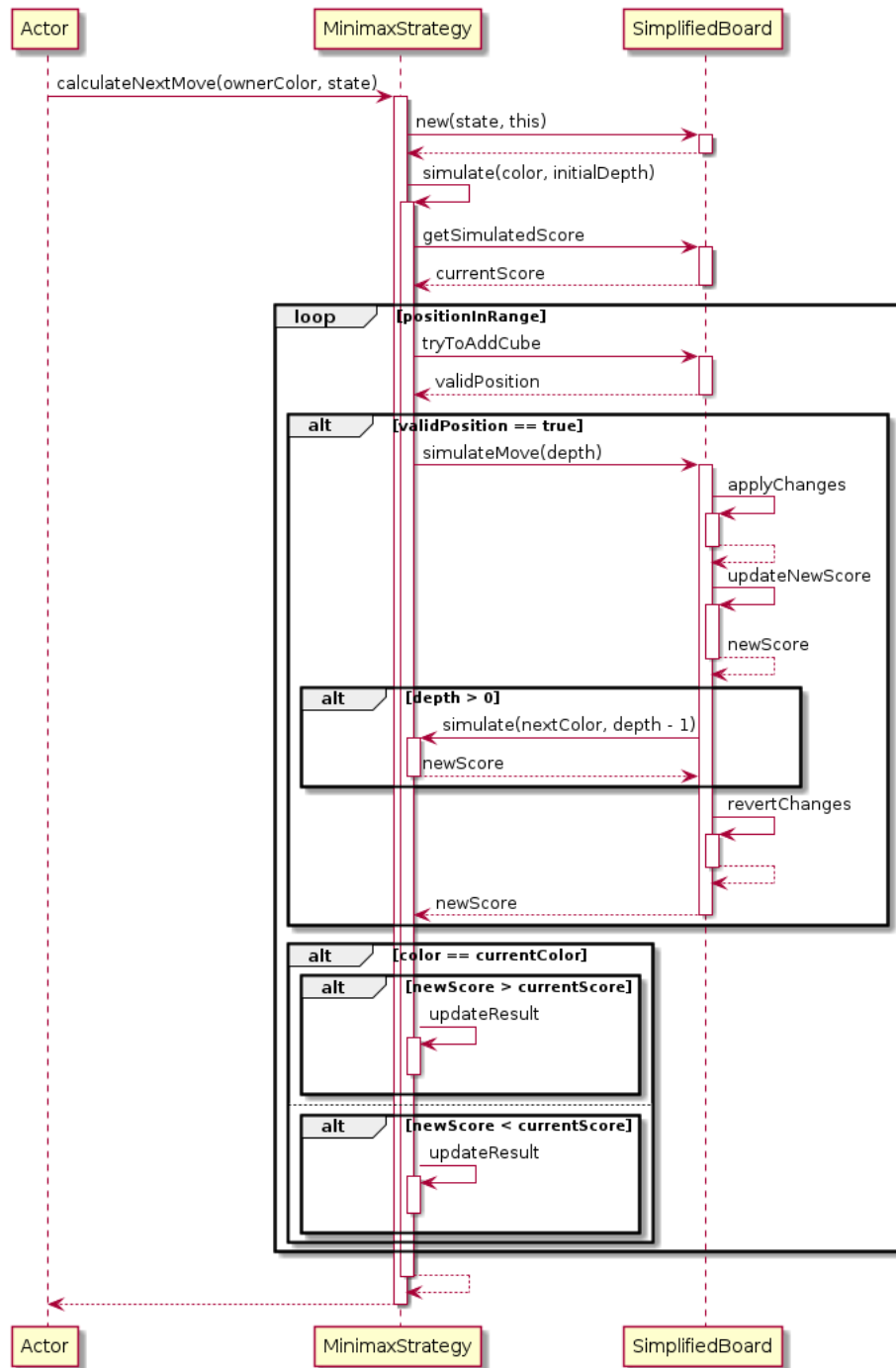
En SimplifiedBoard también se almacenan los puntos de los distintos jugadores, puesto que la idea es consultar los puntos después de simular cada movimiento.

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:



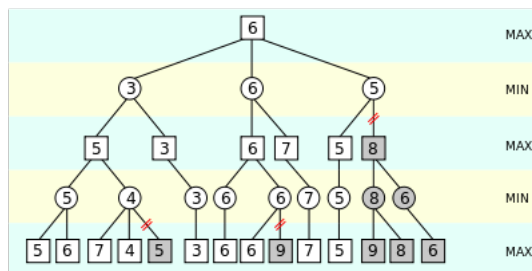
## Sprint (6)

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  $\infty$ .

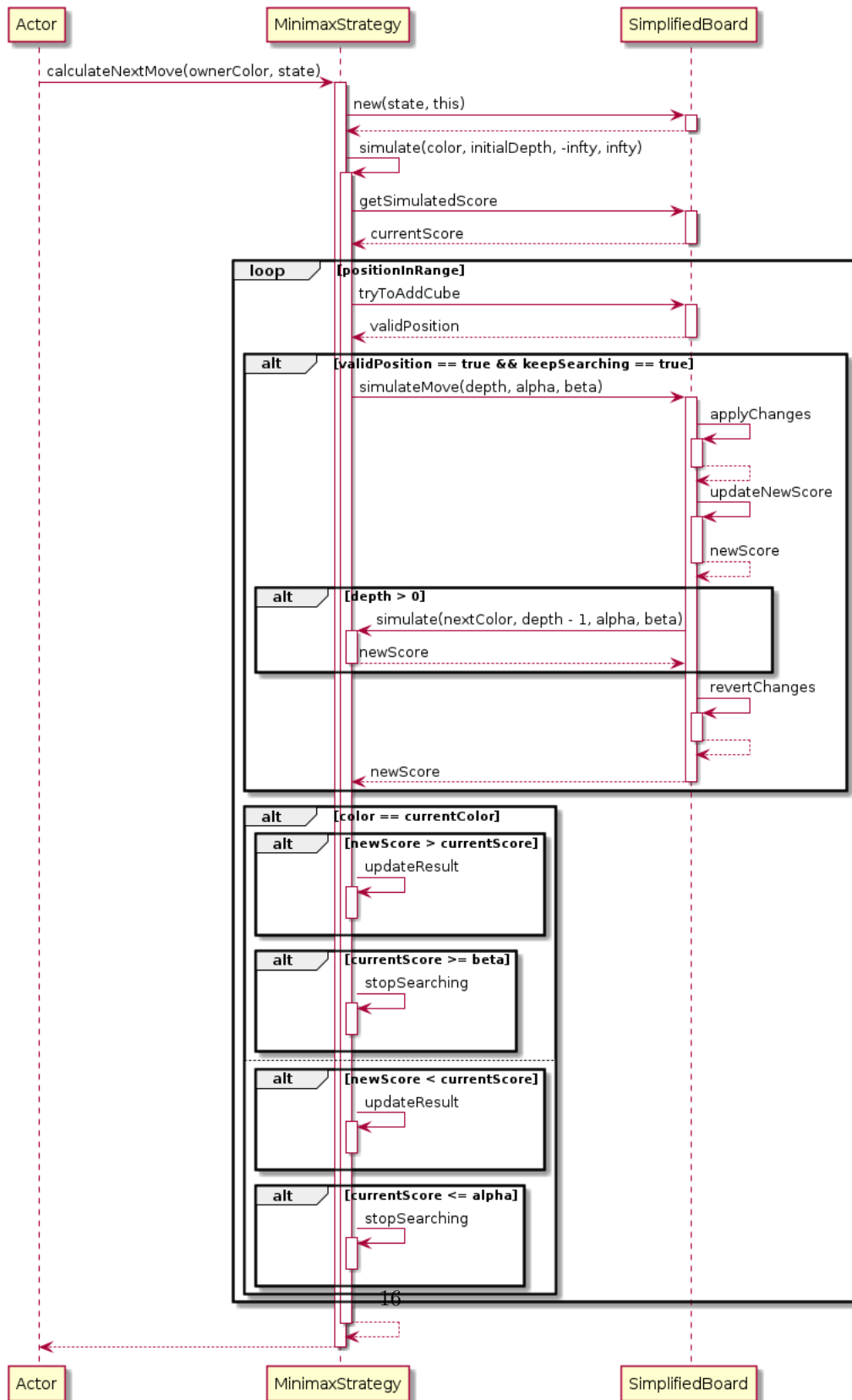
Siempre que la puntuación máxima que se asegura al jugador que minimiza se vuelve menor que la puntuación mínima que se asegura el jugador que maximiza se puede parar de explorar por la rama actual. De forma análoga, se podan ramas en el caso contrario.

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:



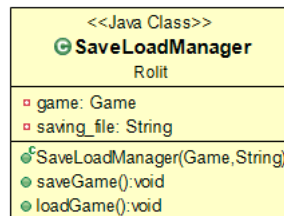


## 6.4 Como usuario quiero que Rolit introduzca características innovadoras siendo intuitivo y cómodo de jugar

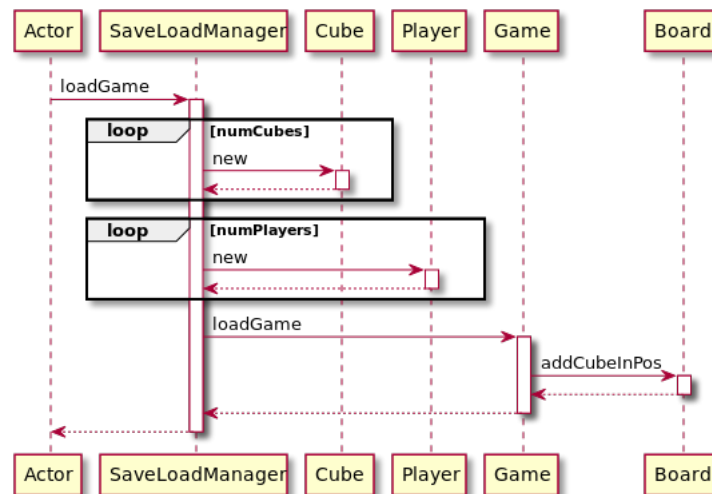
### 6.4.1 Como usuario, me gustaría que se pudiese guardar y cargar partida para continuar más tarde porque permite poner en pausa el juego

#### Sprint (1)

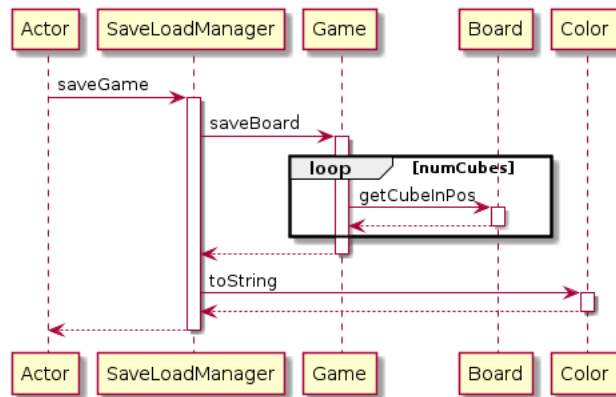
Desde el primer sprint tuvimos clara la necesidad de tener una clase que se encargase de gestionar la comunicación del programa con el exterior para cargar y guardar partidas, el *SaveLoadManager*.



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



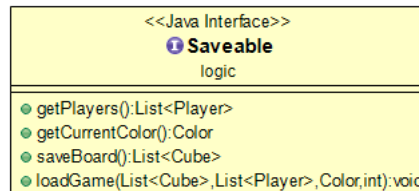
Cargar partida



Guardar partida

## Sprint (2)

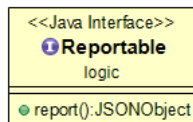
En la versión anterior del *SaveLoadManager*, la clase trabajaba a nivel de *Game*, lo cual no tenía mucho sentido desde el punto de vista de la programación orientada objetos, pues no es necesario acceder a todos sus métodos. Por esta razón, se creó una interfaz *Saveable* que debía implementar *Game*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



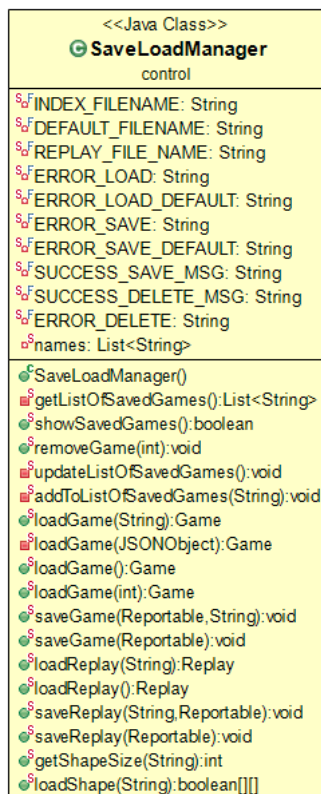
## Sprint (3)

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)

La 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.

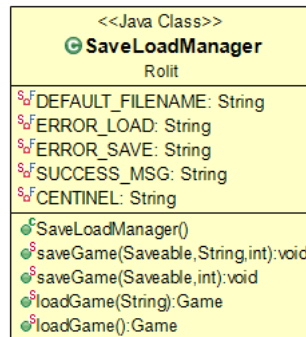
## Sprint (6)

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

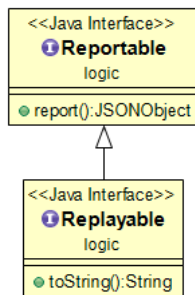
##### Sprint (3)

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.

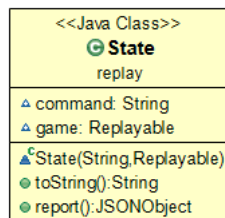
Inicialmente se planteó que los estados almacenaran únicamente el cubo que se añade en su turno, pero esta idea fue descartada debido a que la colocación de un cubo puede llegar a afectar a cualquier parte del tablero, teniendo que incluir la lógica correspondiente para calcular los cambios. Por ello,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

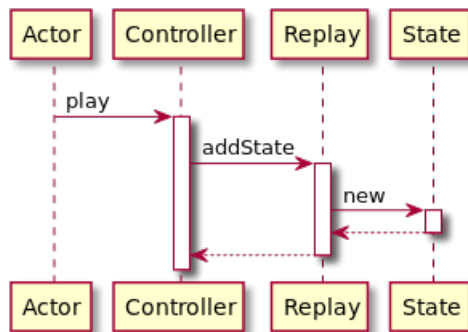
De esta manera surge la interfaz *Replayable*, que extiende de la interfaz *Reportable* y que contiene dos métodos: `toString()`, para facilitar la visualización en la vista de consola, y `report()`, para poder almacenar los estados en formato JSON.



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.

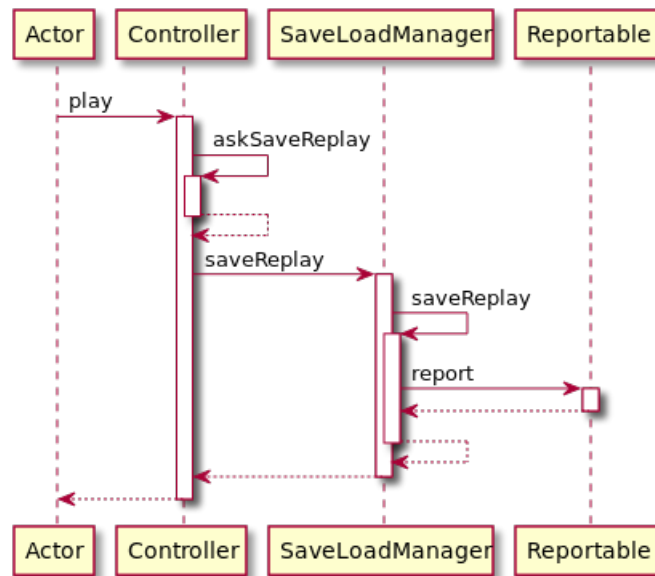


El siguiente paso fue la creación de una clase contenedora de los estados, con la responsabilidad añadida de saber gestionarlos, la clase *Replay*. Esta clase contiene una lista de *State* que se completa durante una partida.



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

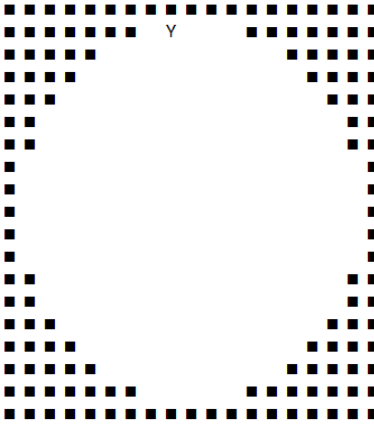
Finalizada la generación de las *replays* era el momento de hacer que pudieran cargarse y ser interpretadas para su correcta visualización. Como ya mencionamos anteriormente, la clase *Replay* es la encargada de gestionar los estados y, por ello, la que contendrá esta lógica.

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)

```



```

> +

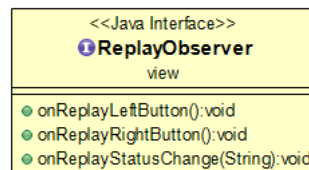
```

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

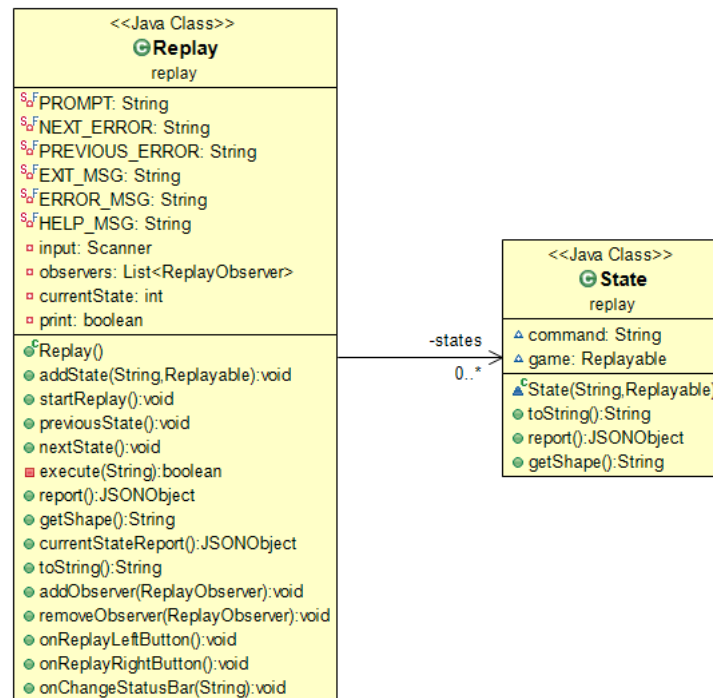
#### Sprint (4)

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)

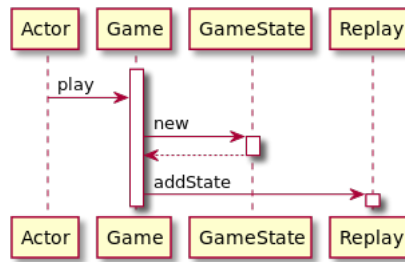
La GUI sufrió una refactorización, siendo necesario para ello la creación nuevos getters. Además, la refactorización de la clase *Controller* trajo consigo problemas con las *replays* que no serían solucionados en este sprint por falta de tiempo.

## 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.

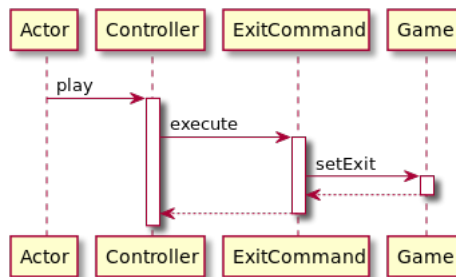




#### 6.4.4 Como usuario, me gustaría poder salir del juego en cualquier momento

##### 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 ...