Refactor Modelo

Índice:

Introducción: **3**

Objetivos: **3**

Alcance: **4**

Cambios Introducidos: **5**

Comandos: 5

AdminTurnos: 7

Controller 9

Observadores 9

Turno 10

Mejoras: **12**

AdminTurnos: 12

Transfer Objects: 13

1. Introducción:

Durante el sprint 6, debido a la necesidad de adaptar el funcionamiento del juego a los eventos que produce la GUI, hemos refactorizado el modelo, eliminando la mayoría de los bucles que guiaban la ejecución del juego.

De paso, tras leer el feedback proporcionado por Gonzalo, hemos tratado de separar y delegar responsabilidades dentro del modelo para obtener un diseño más limpio y claro.

A pesar de haber hecho los cambios que se comentan a continuación, somos conscientes de los fallos que tiene el diseño y de cómo podemos solucionarlos. Dedicaremos un apartado a estas mejoras más adelante.

2. Objetivos:

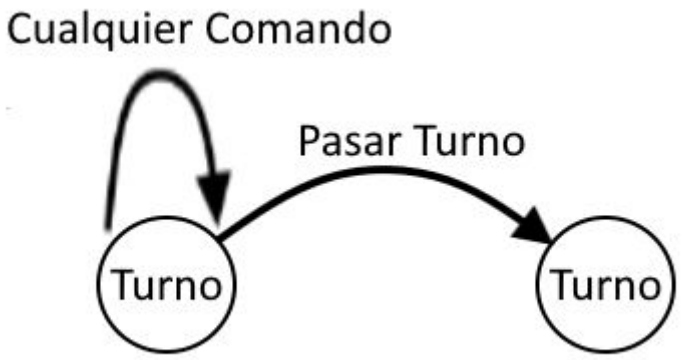
Durante los sprints 5 y 6, cuando estábamos terminando de **adaptar funcionalidades a la vista por la GUI** y nos planteamos **implementar el cambio de turno**, nos encontramos con un **bucle** cuya **condición de salida** era una **variable local** del método run de los turnos que era **imposible de controlar desde la GUI**.

Además, contábamos únicamente con los **conocimientos de TP1 y la primera parte de TP2**. Esto nos hizo **comenzar** a implementar desde un primer momento el juego **empleando bucles** y **no darnos cuenta de que no servían** una vez comenzamos a implementar la GUI. Sin embargo, el **completar la práctica de TP2** nos hizo tener un **punto de referencia sólido** que nos sirvió para orientarnos en muchos aspectos.

Ante esto, nos vimos en la **necesidad de alterar el funcionamiento** del modelo.

Primero nos planteamos **mantener los bucles**, y **alterar** el valor de la variable de **salida** empleando el **comandos**. Esto podría conseguirse si manteníamos esa variable de salida como atributo de la clase Turno, solución que tendría **poco sentido**, ya que estaríamos **manteniendo un bucle para introducir comandos** por el que **no se introducen comandos**.

Después de **desechar la primera solución**, nos dimos cuenta de que el modelo **necesitaba un cambio más severo** para realizar esta adaptación. Además, empleando **GitHub para comentar** los cambios, nos **llegó** una **respuesta a la issue** que teníamos abierta **por parte del profesor** en la que nos **planteaba** algunas **preguntas** que nos fueron **útiles para darnos cuenta de lo que necesitábamos**. Gracias a eso, nos dimos cuenta de que la **solución** que precisábamos era **asimilar el funcionamiento del modelo al de una FSM de tipo Moore** en el que las entradas serían los comandos. De esta manera **logramos que la partida avance exclusivamente conforme ordenen los comandos.**



3. Alcance:

Una vez entendimos lo que necesitábamos implementar, empezamos a plantearnos hasta qué punto nos servían las clases y las interacciones entre ellas que teníamos en ese momento.

Nos encontrábamos en un punto en el que la clase Game agrupaba demasiadas responsabilidades, en el que ya habíamos refactorizado una vez las interfaces de GameObserver y TManagerObserver para reducir el número de implementaciones vacías, y en el que el Controller no era únicamente el intermediario entre el modelo y la GUI, sino que además era el encargado de crear los objetos del modelo y de acceder a los ficheros para cargar y guardar partida.

Por tanto, atendiendo tanto a las necesidades del proyecto para funcionar con la GUI y a las indicaciones del profesor, consideramos que eran necesarios los siguientes cambios:

- Separar la creación de Controller y Modelo
- No hacer intervenir al Game en cada acción del usuario

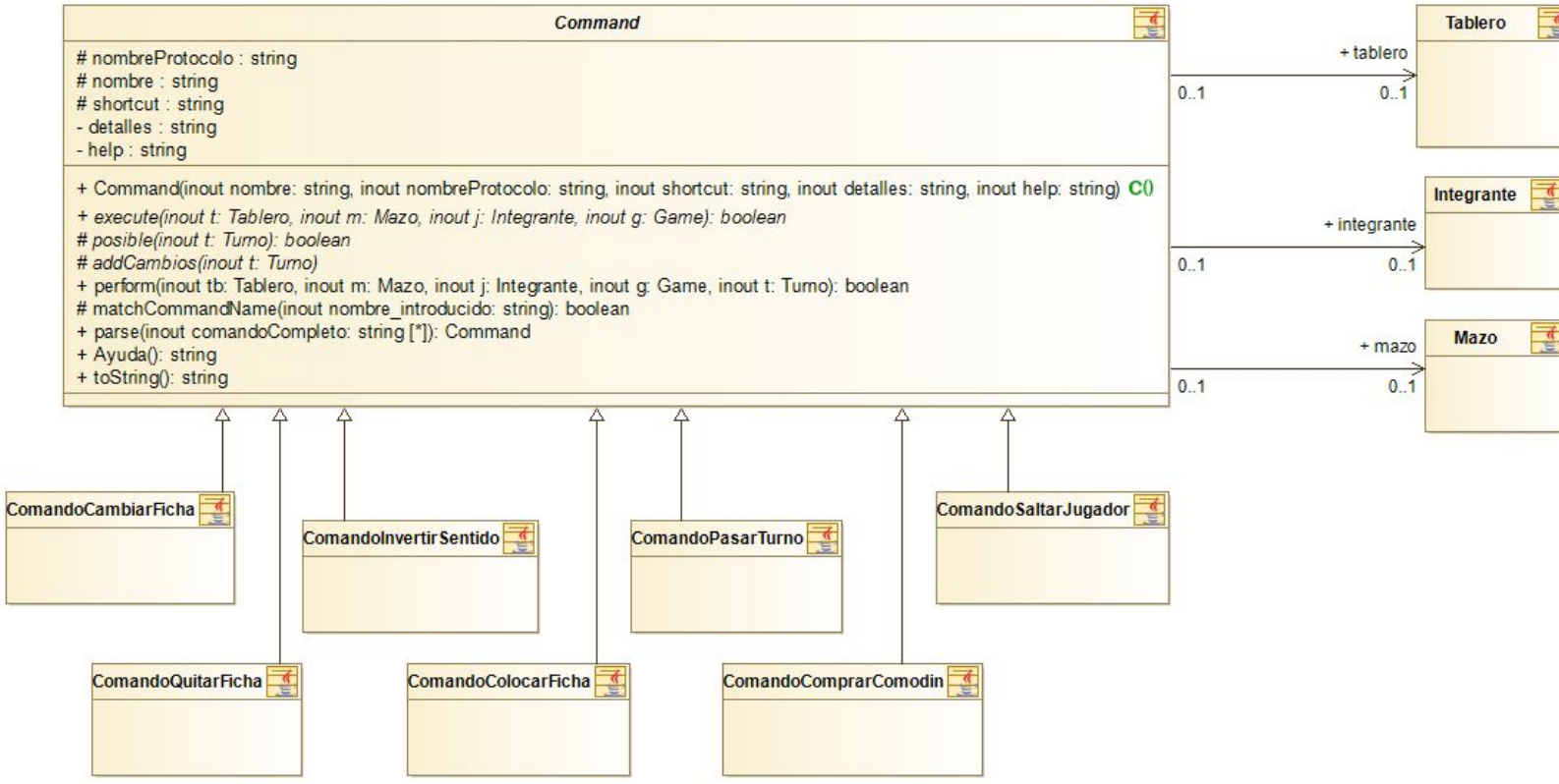
- Refactorizar una vez más los observadores del modelo para distinguir tres tipos: TableroObserver, JugadorObserver, y TManagerObserver
- Dejar como única responsabilidad al Controller de intermediario entre GUI y Modelo

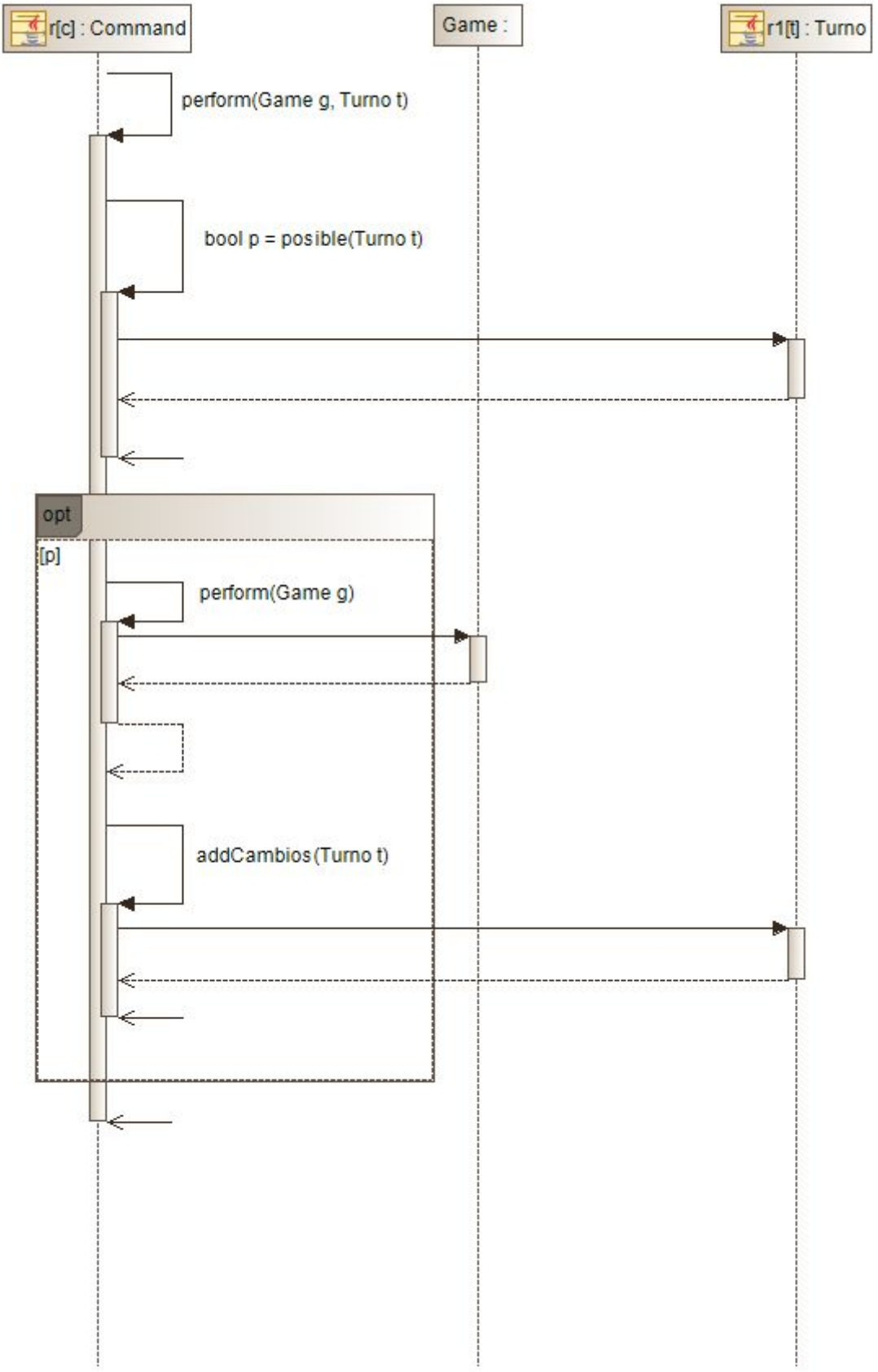
4. Cambios Introducidos:

a. Comandos:

En el diseño anterior, los comandos en su ejecución, llamaban a un método del game que hacía de mediador en la interacción de los diferentes elementos del modelo (Tablero, Jugador, Mazo).

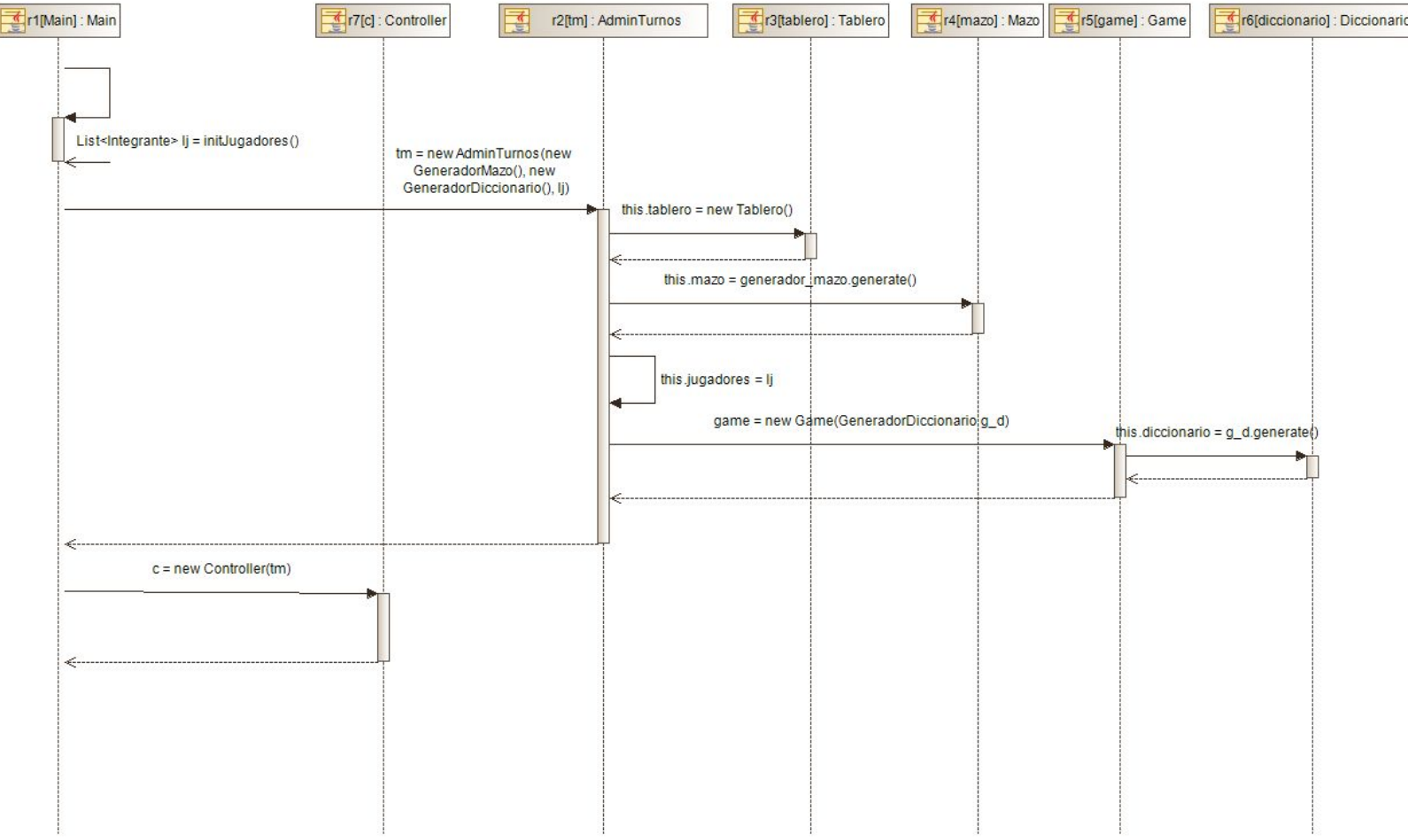
Ahora, esa interacción se produce directamente dentro del método 'execute' de los comandos y el Game únicamente tiene la responsabilidad de verificar palabras usando el diccionario, y de calcular el valor de dichas palabras empleando el tablero.

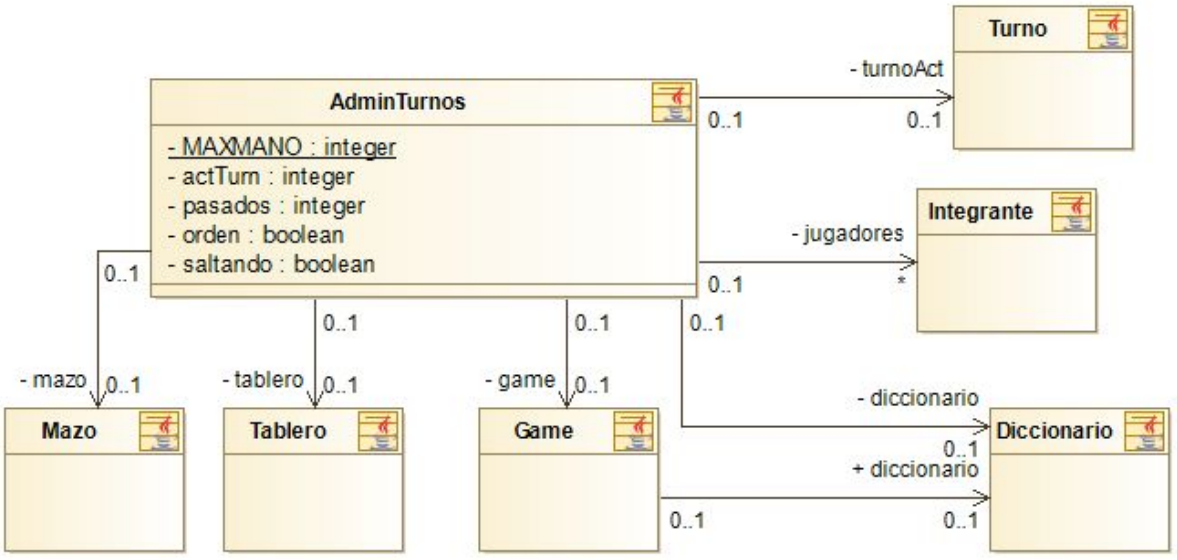


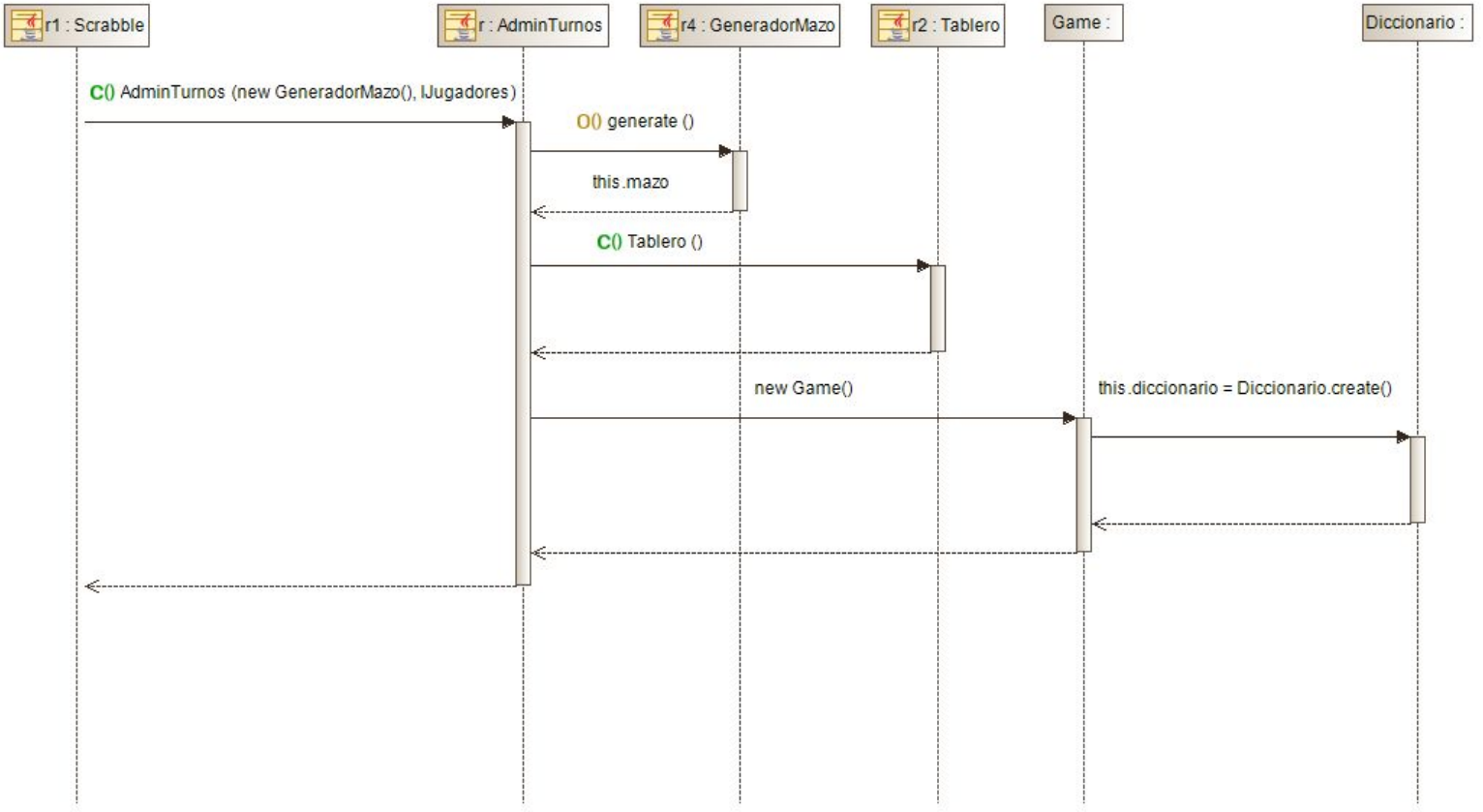

r[c] : Command
Game :
r1[t] : Turno
perform(Game g, Turno t)
bool p = posible(Turno t)
opt
[p]
perform(Game g)
addCambios(Turno t)

b. AdminTurnos:

Anteriormente, el AdminTurnos contenía al Game, que a su vez contenía al Tablero, al Mazo y al Diccionario. Ahora, debido a la modificación anterior del patrón comando, el Game debería dar acceso al Tablero y al Mazo cada vez que se ejecute un comando. Por ello, hemos considerado más sencillo extraer el Tablero y el Mazo y dejarlos como atributos del AdminTurnos.



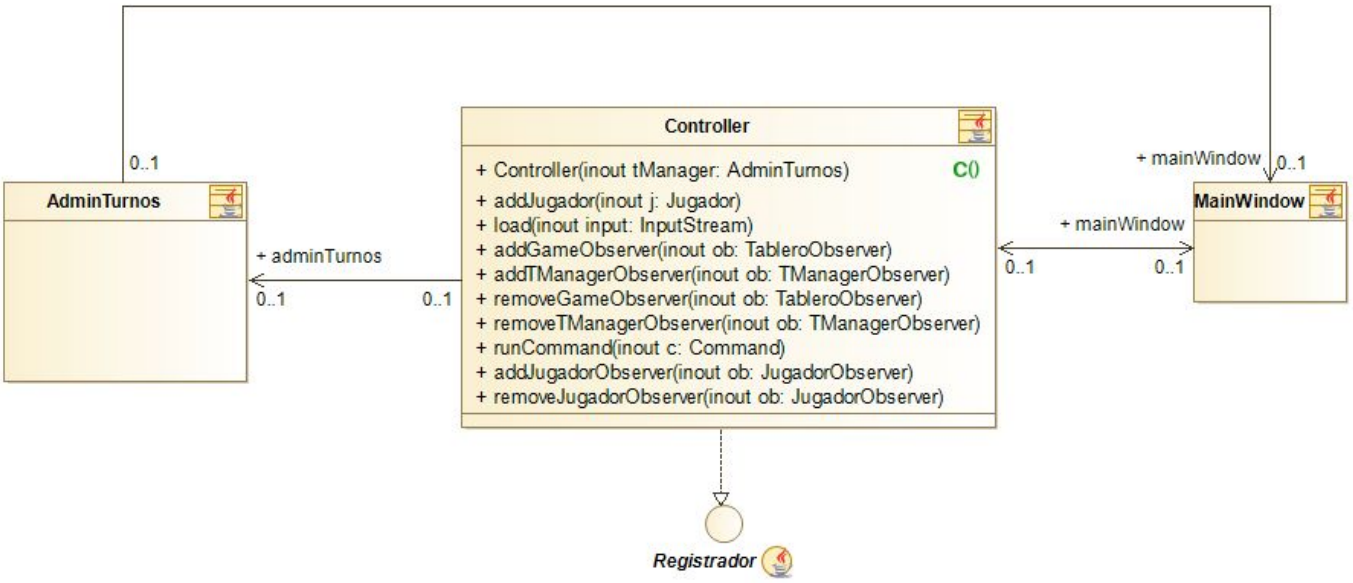

Turno
- turnoAct
0..1
0..1
AdminTurnos
- MAXMANO : integer
- actTurn : integer
- pasados : integer
- orden : boolean
- saltando : boolean
Integrante
- jugadores
0..1
*
0..1
0..1
0..1
0..1
- mazo
0..1
- tablero
0..1
- game
0..1
- diccionario
Mazo
Tablero
Game
Diccionario
0..1
+ diccionario
0..1
0..1


r1 : Scrabble
r : AdminTurnos
r4 : GeneradorMazo
r2 : Tablero
Game :
Diccionario :
C() AdminTurnos (new GeneradorMazo(), lJugadores)
O() generate ()
this.mazo
C() Tablero ()
new Game()
this.diccionario = Diccionario.create()

c. Controller

Anteriormente a este refactor, la responsabilidad de crear el modelo pertenecía al Controller. En esta versión, hemos delegado esa responsabilidad al Main, pudiendo así liberar al Controller de eso, y asignarle únicamente la función de intermediario entre GUI y Modelo. En concreto, es encargado de transmitir las órdenes que envía la GUI al Modelo, y de registrar en el Modelo los elementos de la GUI que observan los cambios en sus diferentes elementos.

Además, el Controller implementa ahora una interfaz registrador, que engloba el comportamiento referente al registro de los observadores en el modelo. Esto nos posibilita un diseño más limpio a la hora de implementar la arquitectura cliente-servidor, pudiendo asignar este comportamiento a otro objeto (Sustituto, ver documento Cliente-Servidor).



d. Observadores

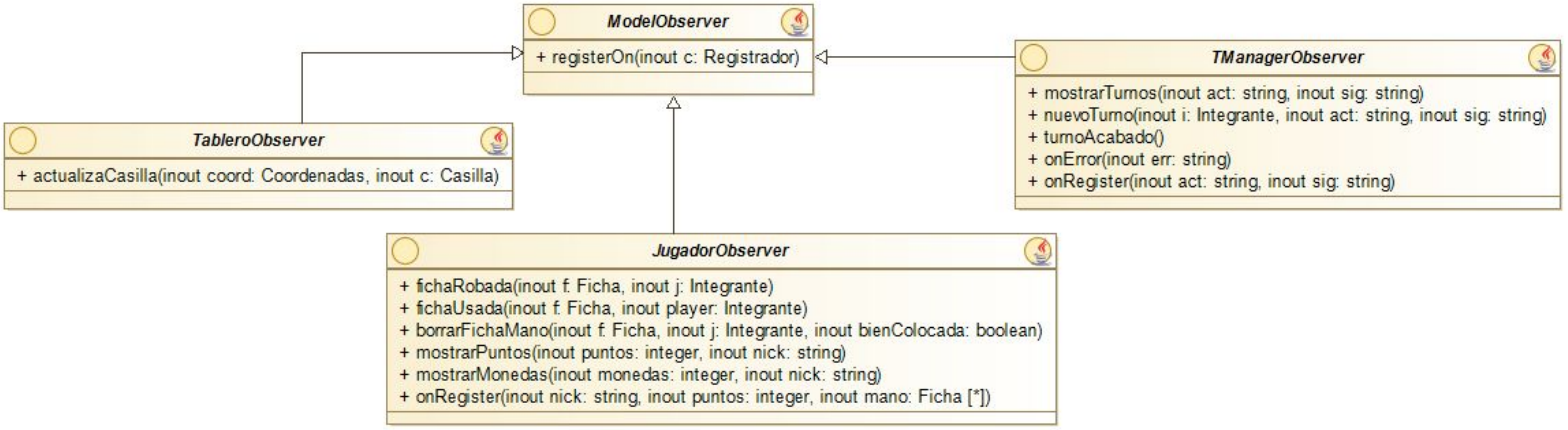
Anteriormente existían dos tipos de observadores TManagerObserver, que se encargaba de observar todos los cambios referentes a los turnos y GameObserver, que observaba los cambios en el Tablero y la Mano del jugador en activo. (se explicaba en 'Nuevos Observadores' que ahora está obsoleto).

Como hemos extraído los Integrantes, el Tablero y el Mazo del Game, ahora hemos tenido que añadir un nuevo tipo de observador más, el Jugador

Observer, que se encarga de observar cambios en la mano del jugador. Además, dados los cambios en los atributos del Game, los GameObservers han pasado a ser TableroObserver.

Model Observer queda como una interfaz general que contiene el comportamiento de cualquier observador, registrarse en un elemento observable del modelo.

De ella, heredan otras 3 interfaces que engloban los métodos que necesita cualquier elemento observador de las diferentes partes del modelo para reaccionar a los diferentes cambios que se produzcan dentro de estas partes. JugadorObserver se encarga de los cambios producidas en cada uno de los integrantes de la partida, TManagerObserver reacciona a los cambios producidos en la secuencia de turnos, y TableroObserver agrupa el comportamiento de los objetos que observan los cambios en el tablero.

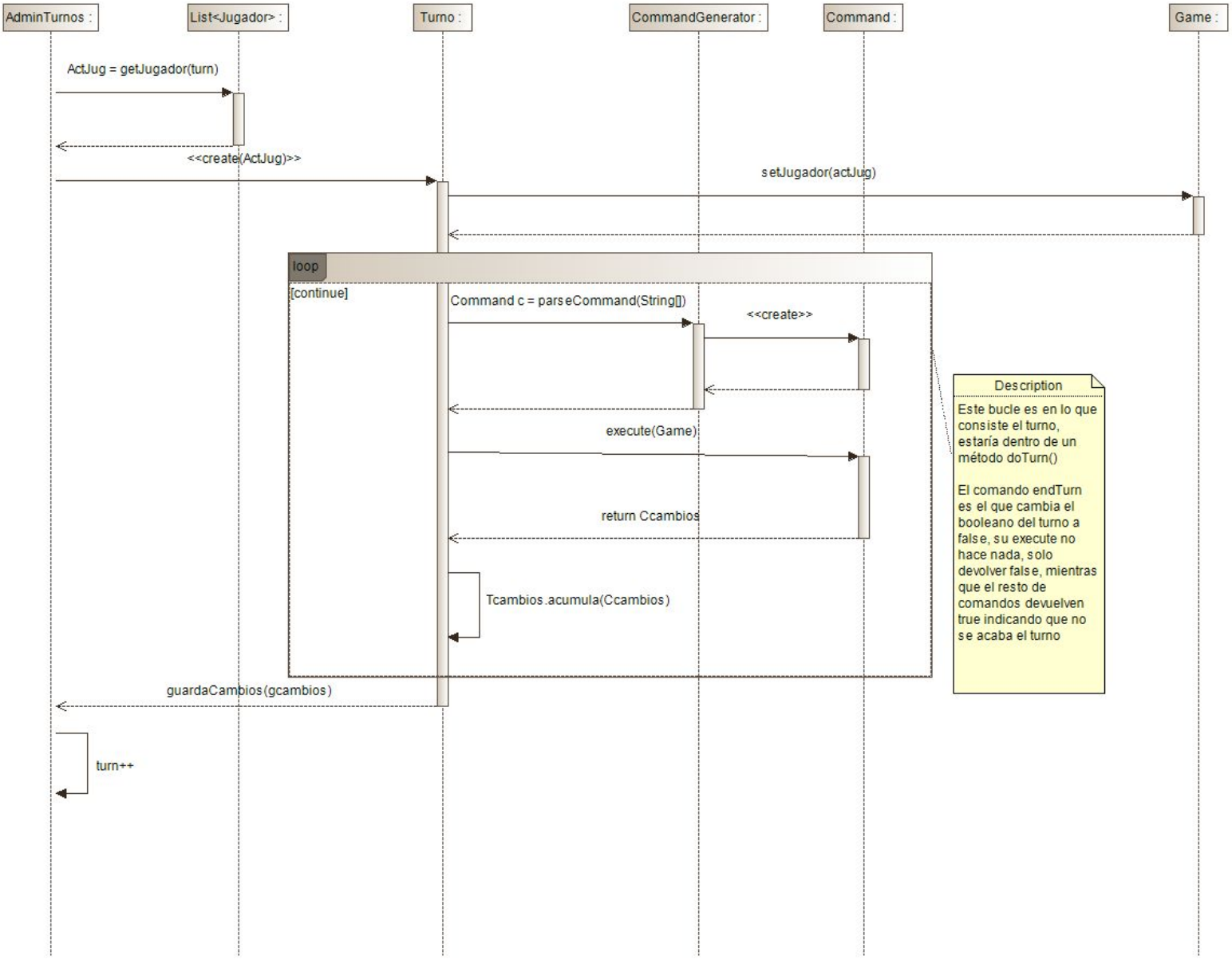


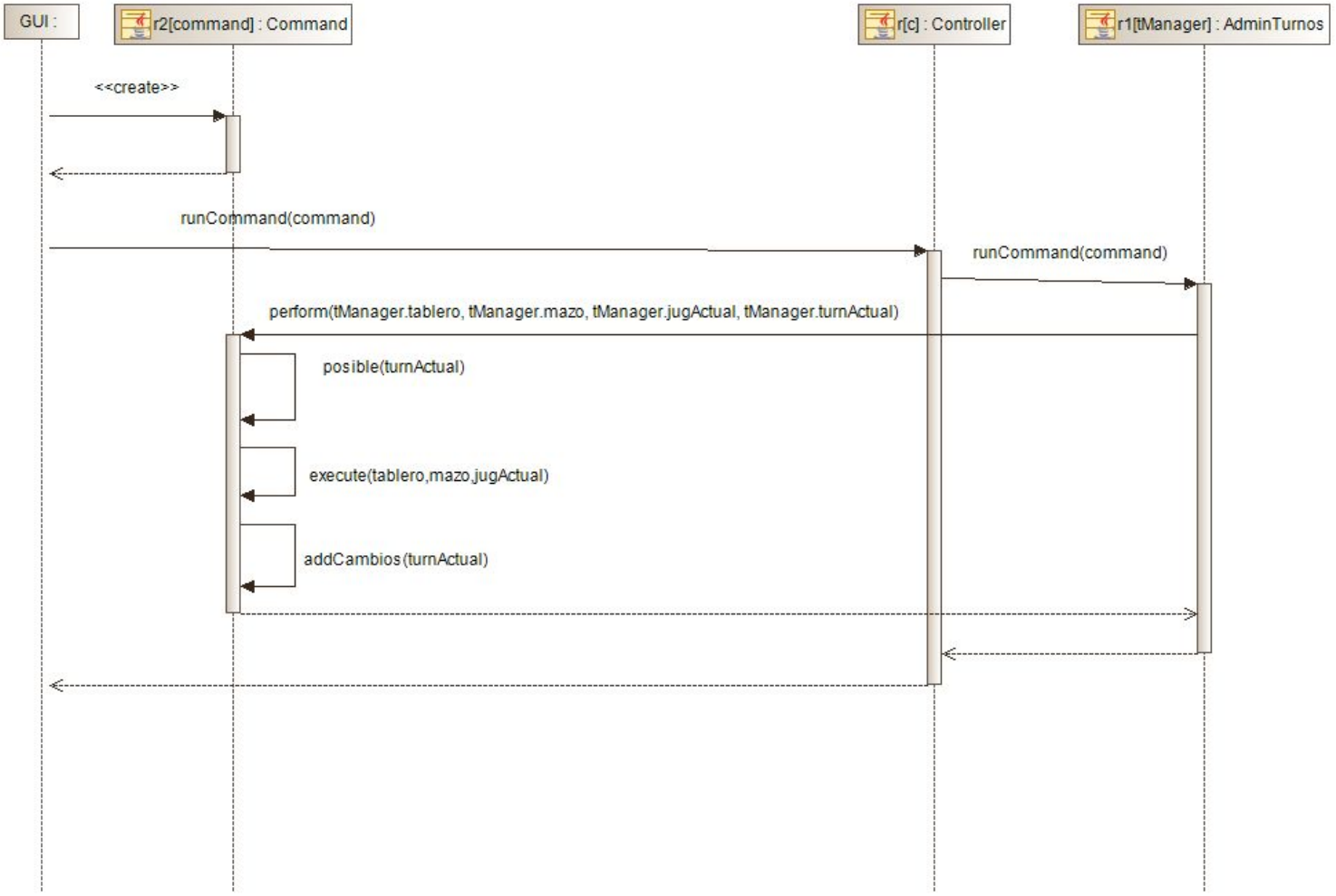
e. Turno

Esta clase ha pasado de ser la que permitía ejecutar comandos a los jugadores sobre el modelo (empleando un bucle que los pedía por consola), a convertirse en un almacén temporal de información para las acciones.

De esta manera, todos los comandos cuya ejecución sea posible o no dependiendo de las acciones que haya realizado el jugador durante el turno, pueden acceder al turno para consultar la información necesaria (por ejemplo, no se pueden hacer más de 7 cambios de una ficha de la mano por otra del mazo durante un turno, es ese tipo información la que se guarda en el turno, y la que es consultada por el comando antes de ejecutarse).

A continuación se presentan diagramas para mostrar cómo era esta clase antes y después del refactor respectivamente.


AdminTurnos :
List<Jugador> :
Turno :
CommandGenerator :
Command :
Game :
ActJug = getJugador(turn)
<<create(ActJug)>>
setJugador(actJug)
loop
[continue]
Command c = parseCommand(String[])
<<create>>
execute(Game)
return Ccambios
Tcambios.acumula(Ccambios)
guardaCambios(gcambios)
turn++
Description
Este bucle es en lo que consiste el turno, estaría dentro de un método doTurn()
El comando endTurn es el que cambia el booleano del turno a false, su execute no hace nada, solo devolver false, mientras que el resto de comandos devuelven true indicando que no se acaba el turno



5. Mejoras:

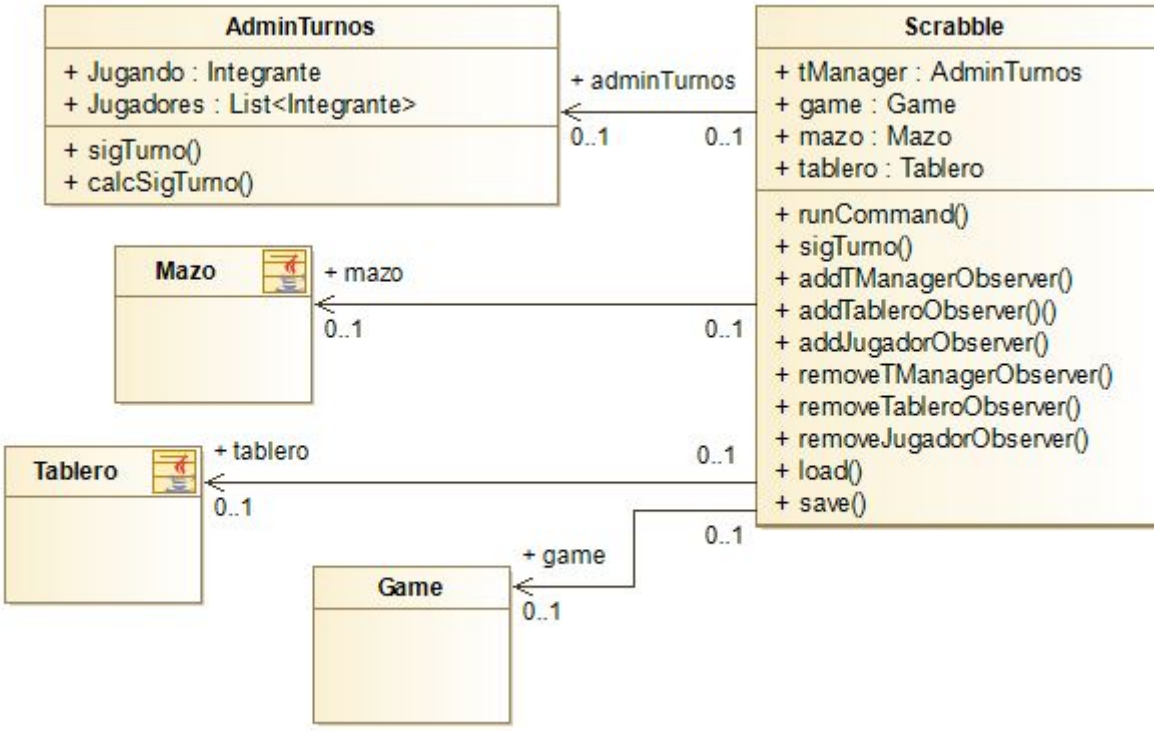
Finalmente, en este apartado se exponen mejoras en el modelo que no se han podido implementar por falta de tiempo.

a. AdminTurnos:

En cuanto al administrador de turnos, esta clase ha acabado responsabilizándose de demasiadas tareas. Para hacer que realmente funcione como un tal, deberíamos extraer los atributos de Tablero, Mazo y Game, dárselos a la clase Scrabble y trasladar todas las funcionalidades que se relacionan con estos también a la clase Scrabble.

Con todo esto, la clase AdminTurnos solo tendría métodos para generar el siguiente turno manejando la lista de jugadores.

El resto de funcionalidades que tendría esta clase, quedarían desplazadas a la clase Scrabble, que ya no solo sería una fachada frente al resto de elementos del programa.



b. Transfer Objects:

Por último, una de las mejoras que se nos ocurre que podríamos implementar sería el uso de transfer objects en la comunicación del Modelo hacia la GUI. De esta manera conseguiríamos, entre otras cosas, desacoplar el tablero que tiene el modelo del tablero que contiene la vista.

Por otro lado, si hablamos de la comunicación en sentido opuesto, es decir, de la GUI al Modelo, encontramos que, sin darnos cuenta, ya habíamos creado transfer objects para desacoplar el Modelo de los mensajes que puede recibir de la GUI. Se trata de los comandos, que funcionan perfectamente como objetos para encapsular cambios que se pueden producir en el modelo y son ideales para viajar desde el Modelo a la GUI.

Además, gracias al CommandGenerator que calcamos de la práctica de TP1, podemos emplear la misma GUI para el servidor que para el cliente. La GUI sigue generando objetos comando que se envían a través de un Controller al Modelo, la única diferencia es que en el caso del servidor, los comandos son traducidos a string por un traductor en el cliente, enviados al servidor mediante un socket, y reconstruidos en el servidor como comandos por el CommandGenerator.

Esto último se puede ver con más detalle en el documento relativo al cliente-servidor.