# Guardar / Cargar partida

## ÍNDICE

# 1. INTRODUCCIÓN

Durante el desarrollo del proyecto estas funcionalidades han ido viéndose modificadas debido principalmente a la inclusión de soluciones más efectivas a medida que íbamos aprendiendo sobre patrones de diseño y en general metodología del diseño.

En este documento se recogen dichos cambios así como las decisiones que han llevado a la implementación final de dichas funcionalidades.

# 2. EVOLUCIÓN

La primera implementación de las **funcionalidades guardar / cargar** partida se realizó usando el **patrón Memento**, dicha implementación fue nuestro primer acercamiento a una funcionalidad de tal magnitud ya que era la primera vez que se nos pedía preservar la información de nuestra aplicación para posteriormente recuperarla, eso supuso un reto técnico en una primera instancia aunque se logró superar de forma satisfactoria.

Posteriormente a medida que nuestros conocimientos sobre diseño fueron aumentando decidimos utilizar el **patrón DAO** para implementar dicha funcionalidad de modo que el tratamiento de ficheros estuviese separado del resto de la lógica de juego.

A continuación se explicarán a fondo ambas implementaciones para dar una visión más precisa de la evolución de estas funcionalidades.

## i. Patrón Memento

### Planteamiento

Durante los primeros días del sprint estuvimos divagando mucho sobre el formato en el que deberíamos guardar los datos de cada clase así como la forma de transportar dicha información al lugar donde sería guardada en Disco.

Al final, tomando la práctica de TP como ejemplo, decidimos usar **JSON** para guardar dicha información y optamos por una **estructura de llamadas encadenadas** (como se verá más claramente en los diagramas de la sección de más adelante) basada también en la que hemos empleado en la práctica de TP.

Gracias a esta decisión pudimos concretar también el número de **Mementos** que necesitaríamos para **preservar el estado de la aplicación**.

En un primer momento se planteó crear un Memento por clase a preservar pero una vez que introducimos el JSON como método de guardado decidimos emplear una sola **clase Memento** que tuviese un **atributo de tipo JSON** (como se muestra más adelante) lo cual nos ayudó a que el código no quedase tan sobrecargado de objetos.

## Clases / Interfaces Añadidas

### Interfaz Originator

Esta interfaz es implementada por todas las clases que tienen que ser preservadas / cargadas y contiene los métodos **CreateMemento** (el cual devuelve el Memento de la clase) y **setMemento** (el cual recibe un Memento con él que se cargan los atributos de la clase).

### Interfaz IMemento

Esta interfaz es implementada por la clase Memento y contiene los métodos **setState** (el cual recibe un JSONObject) y **getState** (el cual devuelve un JSONObject), estos dos métodos se encargan de gestionar el acceso al **atributo JSONObject state** de la clase Memento.

### Clase Memento

Esta clase se encarga de **encapsular la información preservada** de una clase o un conjunto de clases (si por ejemplo guardamos varios Mementos dentro de otro) en un atributo de tipo JSON el cual llamamos state.

Para acceder a dicho atributo se emplean los métodos de la **interfaz IMemento** descrita anteriormente.

### Clase copiaSeguridad

Durante la etapa de desarrollo donde se implementó esta funcionalidad tomamos la decisión de que al final de cada turno se crease un Memento que preservase el estado del juego en ese instante, esta decisión vino motivada por la intención de poder revertir un turno o varios (usando por ejemplo una ventaja que pudiese comprar el usuario para lograr ese efecto).

Para lograrlo esto necesitábamos **una clase que gestionase una lista de Mementos** (los que se generan al finalizar cada turno) y de la cual **pudiésemos extraer un memento concreto**, de esa idea surge la clase copiaSeguridad la cual tiene un **atributo List<Memento>** el cual es gestionado usando getters y setters.

A medida que se fue desarrollando el proyecto y fuimos teniendo que dedicar más tiempo a otras funcionalidades (como la IA o el juego en red) se descartó esta idea.

Como consecuencia de esto, esta clase en la actualidad ya no existe ya que nos resultaba inútil tener una lista de Mementos cuando solo nos interesaba el Memento que quiere guardar el jugador.

### ComandoGuardar

Esta clase se encarga de notificar a la clase Turno de que el usuario había solicitado guardar partida haciendo así que todas las clases a preservar preserven su estado usando los métodos de la **interfaz Originator**.

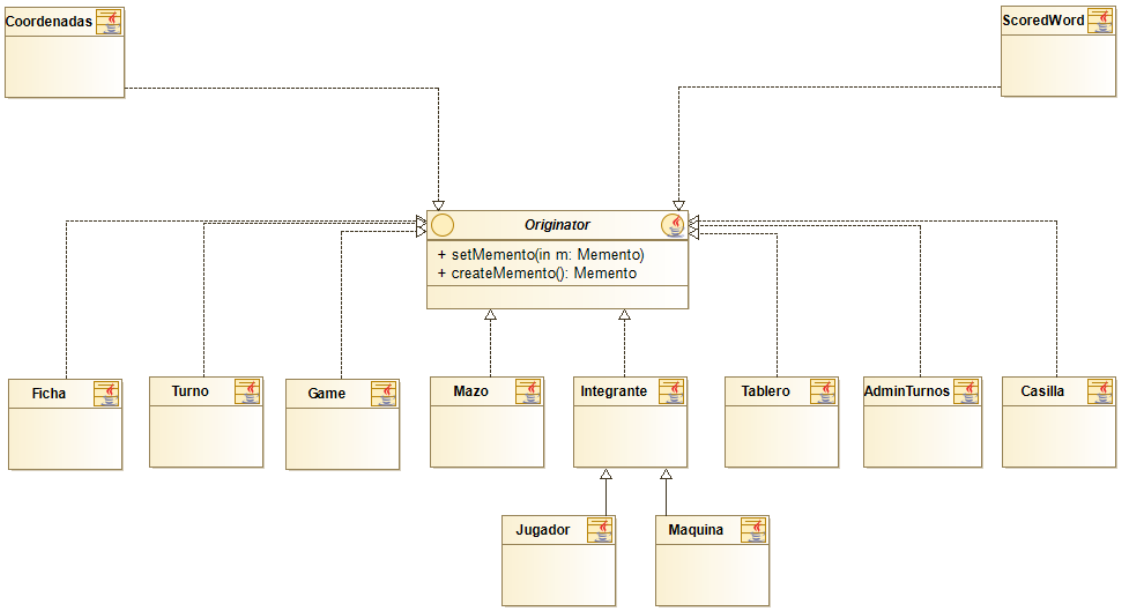
## Cambios en el modelo

Debido a la naturaleza de las funcionalidades guardar / cargar partida el modelo se ha visto poco modificado (quitando las clases e interfaces añadidas que se han mencionado anteriormente), las únicas modificaciones significativas son las siguientes:

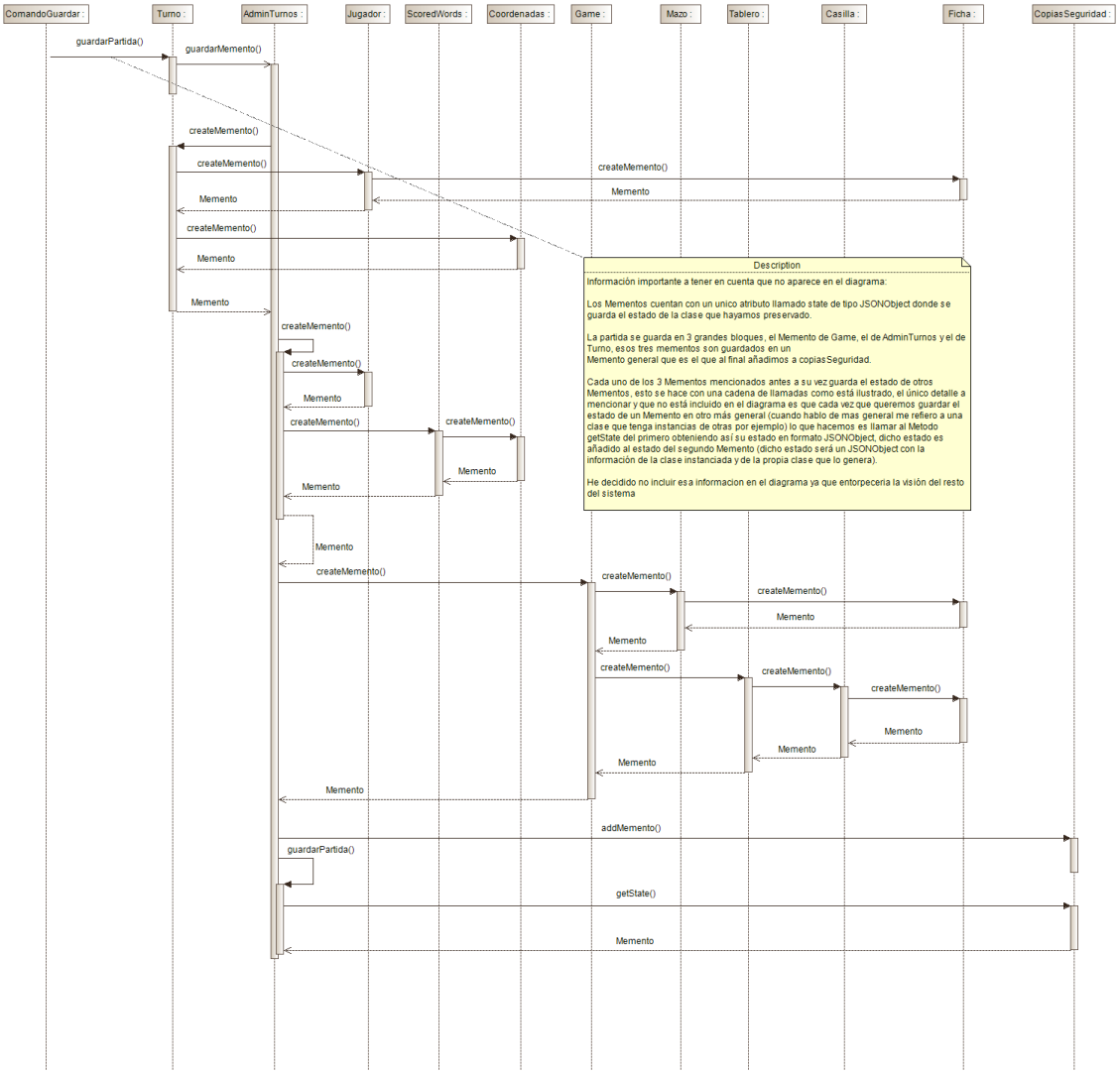
- copiaSeguridad es un atributo de AdminTurnos
- Turno crea un memento al final de cada turno y se guarda en copiaSeguridad

## Diagramas

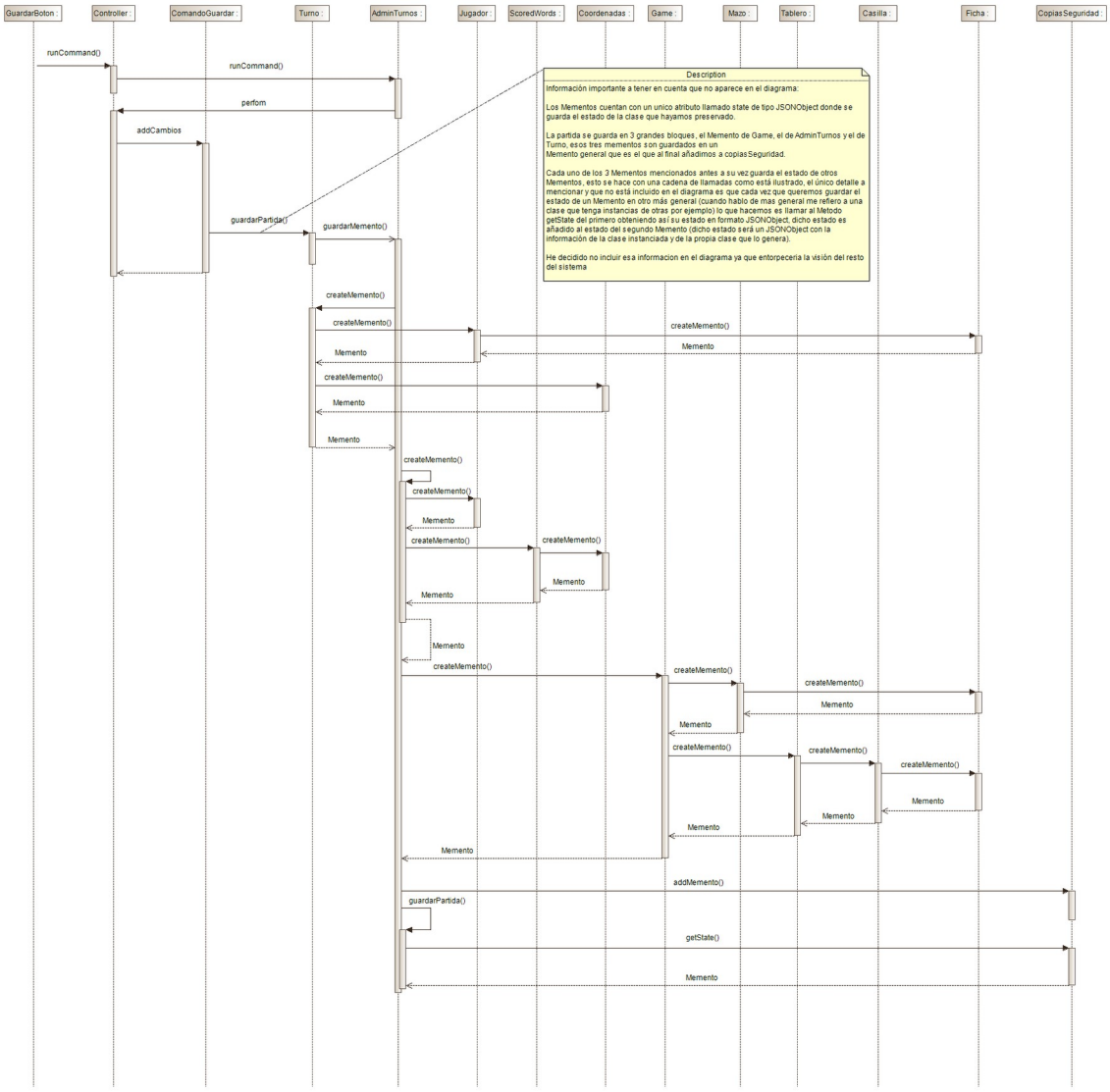
### Diagrama interfaz Originator



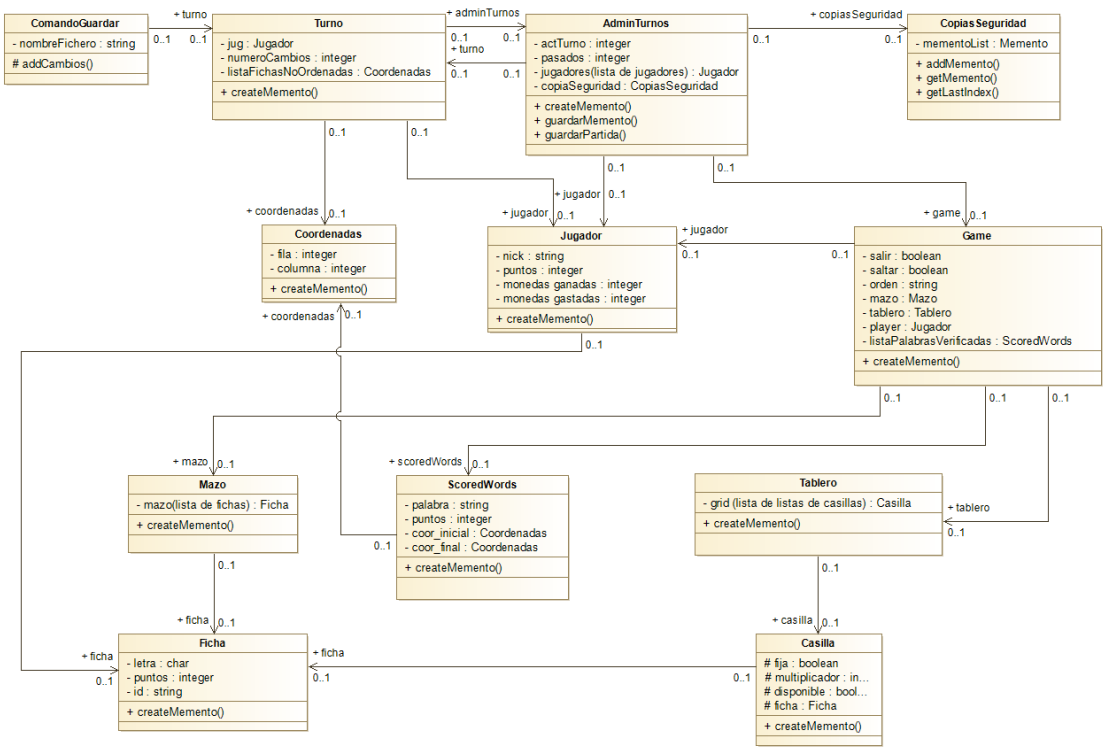
# Diagrama de secuencia pre GUI



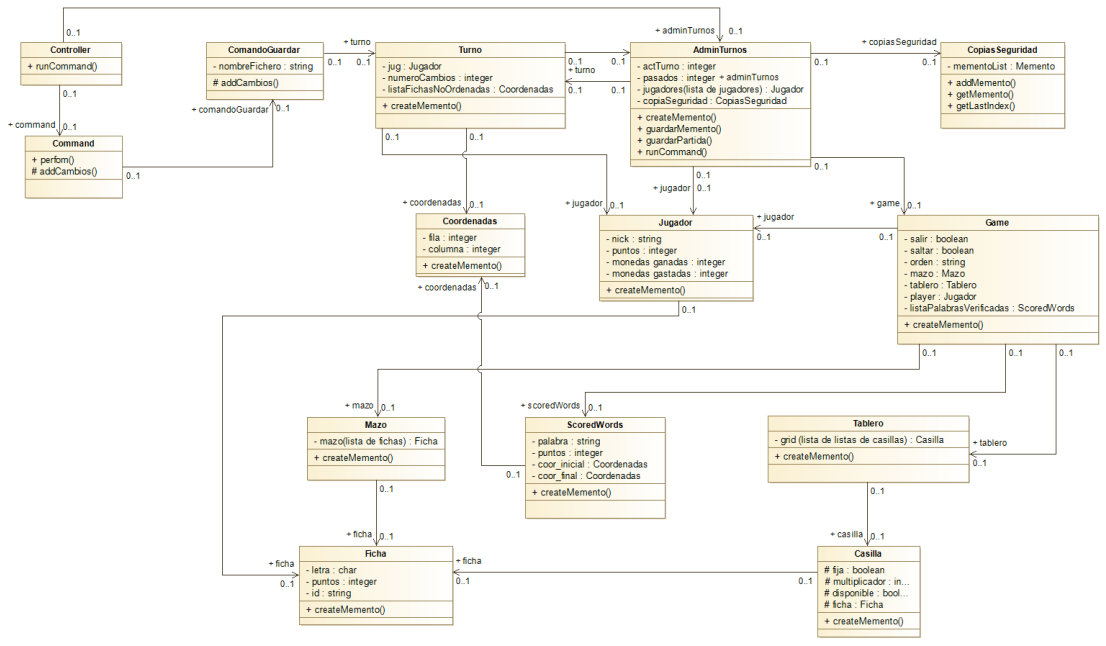
## Diagrama de secuencia desde GUI




## Diagrama de clases pre GUI




## Diagrama de clases desde GUI




## ii. Patrón DAO

## Planteamiento

Durante el sprint anterior se nos explicó la existencia de las **arquitecturas multicapas**, esto hizo que nos diésemos cuenta de algunos errores en nuestro diseño y por lo tanto tuvimos que cambiarlos.

Uno de estos errores era la **gestión de acceso al disco** la cual era realizada por una clase que realmente no debía ser la responsable de esto (como se explicará más adelante).

Otro punto importante que propició la implementación del **patrón DAO** fue que justamente en ese sprint habíamos llevado a cabo una **refactorización general de toda la estructura de la aplicación** y como consecuencia de esto las funcionalidades guardar / cargar partida debían ser modificadas también.

De cara a la implementación, hubo un problema ya que el modelo no tenía una **fachada** y la clase AdminTurnos tenía demasiada importancia (como se explicará más adelante) por lo que tuvimos que buscar una solución a ese problema antes de hacer nada.

Por último, cabe destacar que fue de gran ayuda el sistema anterior fundamentado en el **patrón memento** ya **los métodos para obtener y utilizar los mementos se han mantenido en el diseño final**.

## Clases / Interfaces añadidas

## Clase Scrabble

El propósito de la clase Scrabble es **solucionar uno de los problemas** que llevamos arrastrando desde el comienzo.

Durante todo el desarrollo del proyecto se le ha dado **demasiadas responsabilidades a la clase AdminTurnos** (aparte de administrar turnos también actuaba como fachada de todo el modelo y gestionaba el uso de ficheros).

**La función de la clase Scrabble es la de actuar de fachada entre la capa de presentación** (el controller) **y la capa de negocio** (el modelo) quitándole así a AdminTurnos esta responsabilidad y logrando desacoplar un poco más el diseño que teníamos.

## Clase ModeloDAO

Esta clase se encarga de **gestionar el acceso a disco desde la aplicación** (en nuestro caso a través de ficheros) **logrando así quitarle esta responsabilidad a AdminTurnos** (que era quien anteriormente se ocupaba de esto).

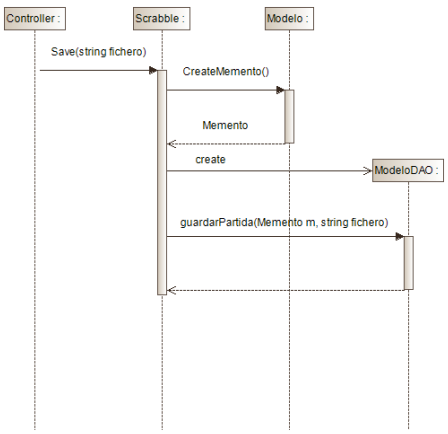
Cuenta con los métodos **guardarPartida** (que recibe un Memento que contiene toda la información del juego que debe ser preservada y un string fichero que indica el nombre con el que el usuario quiere guardar la partida) y **cargarPartida** (que recibe un string fichero que indica el nombre de la partida que quiere cargar el usuario)

Ambos métodos **se comunican con el modelo a través de la clase Scrabble**.

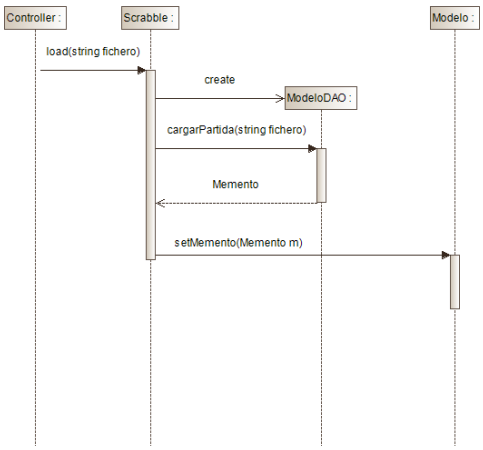
# Cambios en el modelo

Los cambios más significativos en el modelo durante la implementación del patrón DAO en las funcionalidades guardar / cargar partida fueron:

- La clase ComandoGuardar ha sido eliminada ya que la acción **de Guardar no se procesa como un comando**, ahora se hace mediante una serie de llamadas como se muestra a continuación:



- Así  mismo, la acción de cargar partida se realiza de la misma forma:

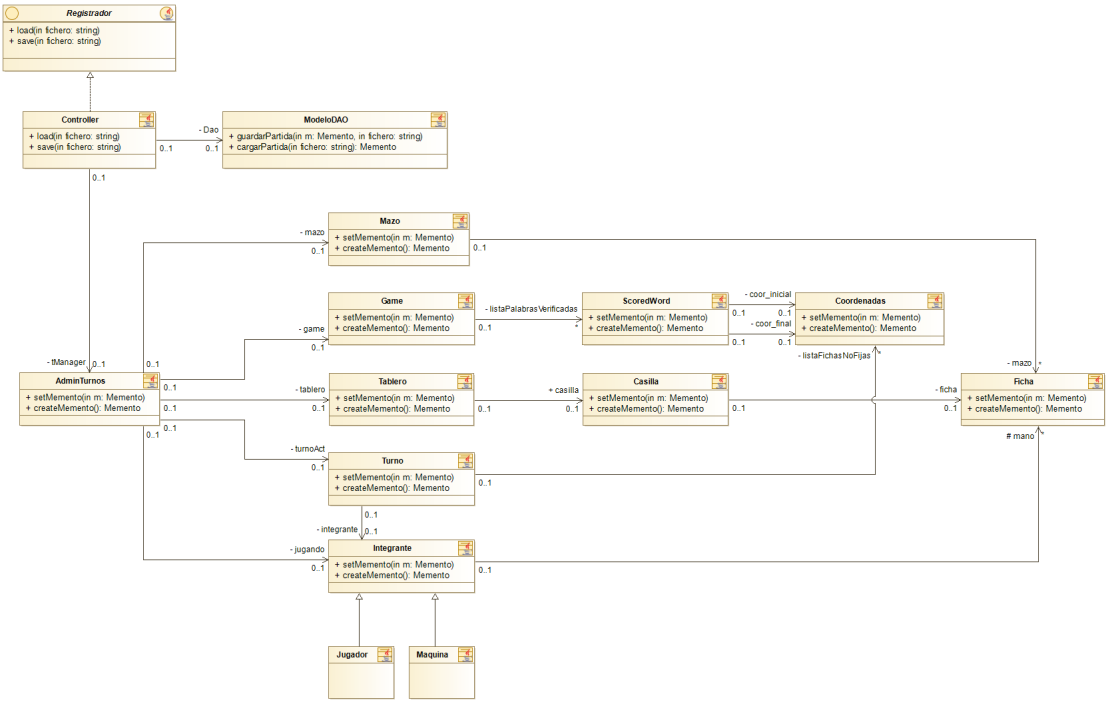



- Los métodos que tenía AdminTurnos para gestionar el acceso a disco han sido trasladados a la clase ModeloDao

- Gracias a la elección del **Memento** como **transfer object** todas las clases que implementan la **interfaz Originator** han podido seguir usando los métodos **createMemento** y **setMemento** logrando así ahorrarnos mucho trabajo y minimizando los cambios en el modelo

## Diagramas

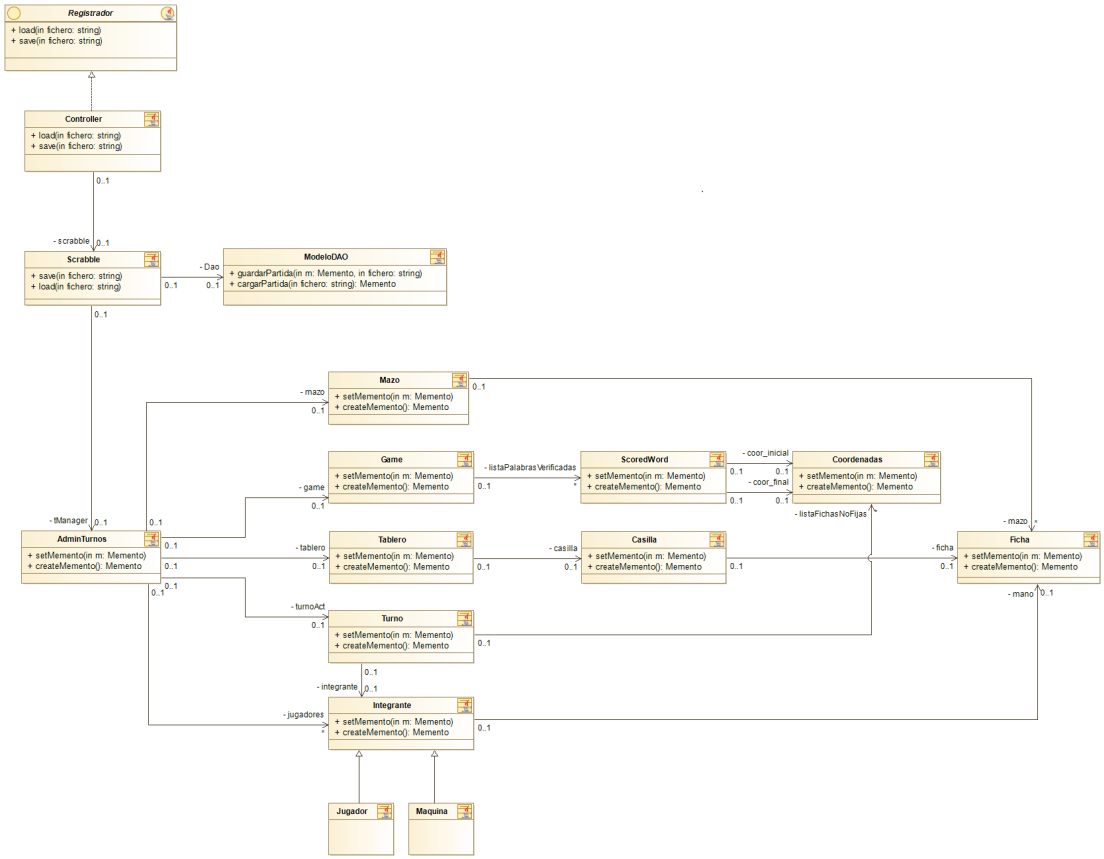
### Diagrama de clases primera implementación




Gracias a este diagrama se puso en evidencia la **necesidad de crear una clase que actuase como fachada del modelo** ya que no tenía sentido que modeloDAO fuese implementado en AdminTurnos (ya que, como se ha mencionado anteriormente, es una de las clases que tienen que guardarse / preservarse y sería raro llamar a un método load de esta que a su vez llame al método setMemento) o en el controlador (el cual pertenece a la capa de presentación).

De este razonamiento surge el diseño final:

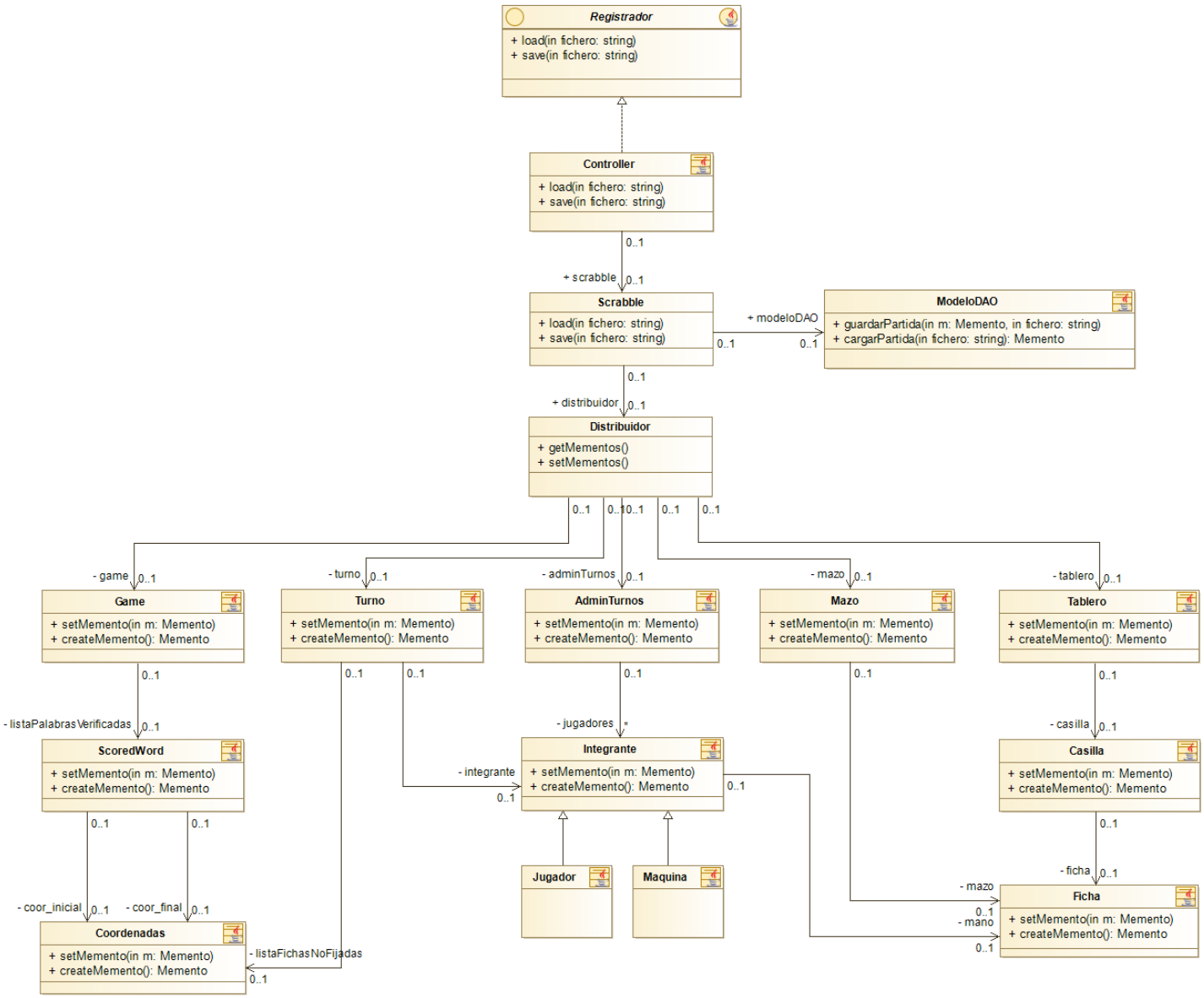
## Diagrama de clases implementación final




## Posibles mejoras

Estuvimos divagando mucho sobre la posible implementación de una **interfaz entre AdminTurnos y el resto de clases** cuya funcionalidad fuese la de **obtener un Memento y distribuirlo en Mementos específicos a cada clase** (y el proceso inverso para guardar un Memento de todas las clases) logrando así una **mayor independencia en los datos y desacoplar un poco AdminTurnos.**

### Diagrama




Esta idea fue descartada debido principalmente a la falta de tiempo y a la baja prioridad que tenía dichos cambio (el diseño realmente no se verían tan modificado)