REFACTORIZACIÓN DEL SPRINT 3

Grupo PMC

1 de abril de 2022

Paquetes Logic y Controll

Controller

La clase *Controller* conocía demasiados detalles de implementación del *Game*. Para evitar que se quedase desactualizado y que no fuese operativo ante cambios hemos decidido delegar la funcionalidad de creación del juego y de sus objetos internos a otra clase nueva, la clase *GameBuilder*.

En *Controller* solo subsisten los métodos para mostrar un menú desde el que se elija jugar, cargar u otras funcionalidades extras que se puedan ir añadiendo, los métodos de jugar un turno y el método *run()* que se encarga de hacer funcionar la aplicación.

GameBuilder

Esta clase hace de factoría del *Game* y se encarga de conocer los detalles de su implementación concreta y de pedir al usuario los datos requeridos para crear el juego.

Por el momento, se debe encargar de:

- Crear los jugadores y pedir el número de los mismos.
- Crear el Board con las características concretas del juego.
- Generar el Game que se desarrollará durante la partida.

Asímismo, implementa un método estático que devuelve un *Game* completamente creado. De este modo, en la clase *Controller* bastaría con llamar a dicho método para asignar el juego creado a su propio atributo juego. Esto nos da una gran ventaja a la hora de introducir distintos modos de juego como ya se va a hacer en ese sprint.

SaveLoadManager

En este sprint, el SaveLoadManager sigue encargándose de gestionar todo lo relativo a entrada y salida del juego y, además, ve aumentadas sus funcionalidades al encargarse también de leer los .txt que contienen la forma del tablero y de cargar y guardar repeticiones.

Para facilitar la manejabilidad de esta clase y futuras posibles ampliaciones hemos decidido refactorizarla por completo, de manera que ahora el formato de guardado en ficheros se lleva a cabo mediante el uso de jsons, a excepción de los que contienen las formas del *Board*.

Así, surge la necesidad de que cada objeto del juego pueda convertirse en un json que contenga toda su información relevante. Para plasmar esta característica en el código se hace uso de una interfaz Reportable, que han de implementar todas aquellas clases que tengan la necesidad de ser guardadas en un fichero json.

```
public interface Reportable {
        public JSONObject report();
}
```

Paquete replay

Para gestionar las repeticiones de partidas se ha creado un nuevo paquete, llamado "replay", que contiene las clases *State* y *Replay* que se detallan a continuación.

State

La clase *State* representa un instante del juego, un momento de la partida, más concretamente una instacia de Game en un ciclo del juego y el comando que generó dicha situación.

Cada *State* debe ser capaz de saber su representación como *String*, por ello será necesario que la clase *Game* sobreescriba el método *toString()*. Dicho método también necesita saber la representación en formato de texto de la clase *Player* y de *Board*, por lo que estas clases también se verán obligadas a implementarlo.

Además, como las repeticiones pueden ser guardadas en ficheros, también surge la necesidad de que *State* implemente la interfaz *Reportable*.

En definitiva, un estado debe poder imprimirse en consola y convertirse en json, pero no requiere ninguna del resto de funcionalidades de *Game* ni del resto de clases. Por ello, se ha creado una interfaz *Replayable* (que extiende de *Reportable*), que ha de implementar *Game* para obtener la funcionalidad de repetición de partidas manteniendo la encapsulación.

```
public interface Replayable extends Reportable {
        public String toString();
}
```

```
public class Game implements Replayable{...}
```

```
public class State implements Reportable{
        private Replayable game;
        private String command;
        ...
        public report(){...}
        public toString(){...}
}
```

Replay

La clase *Replay* tiene la responsabilidad de generar una repetición y de gestionarla una vez que se carga por fichero, de manera que podamos navegar a través de la repetición, avanzado y retrocediendo en sus estados.

Para poder realizar estas gestiones, se necesita una lista de estados, la cual se irá recorriendo según indique el usuario. Por tanto, la clase también necesita un método para añadir estados a la lista de estados.

```
public class Replay implements Reportable{
        private List<State> states;
        ...
        public addState(String command, Replayable game){...}
        public toString(){...}
        public report(){...}
        public startReplay(){...}
        private execute(){...} //Replay tiene comandos propios que
        autogestiona

}
```

Es importante que, al añadir cada estado, el parámetro de tipo *Replayable* de game que recibe la funcion *addState()* sea una nueva instancia, sino todos los estados estarán referenciados a la misma instancia de *Game* y, cuando se avance en la partida, los cambios modificarán los estados anteriores, rompiendo así la funcionalidad deseada. Para solventar este problema ha sido necesario implementar constructores de copia para las clases *Game* y *Board*.

Board

Los tableros ahora pueden tener distintas formas: cuadrado, círculo y rombo, cada una de ellas disponible en tres tamaños diferentes. Para representar estas formas se almacena mediante ficheros .txt las matrices que definen las posiciones válidas de cada una.

Para facilitar la manejabilidad de código, hemos creado una clase enumerada llamada *Shape* que representa a cada tipo de forma y que conoce el nombre del fichero que almacena su forma. No obstante, es el tablero quien guarda la matriz de booleanos una vez cargada del fichero.

```
public class Board implements Reportable{
        private boolean[][] shapeMatrix;
        ...
}
```