REFACTORIZACIÓN DEL SPRINT 4

Grupo PMC

5 de mayo de 2022

Introducción

Para este Sprint hacía falta introducir todo el diseño orientado a que pueda haber distintos modos de juego. Se quedó a medias en el Sprint anterior por la complejidad que suponía ya que requería de hacer *Game* abstracta y hacer instancias concretas de cada modo de juego. El principal problema viene en la creación del mismo, pues para cada uno se necesita pedir al usuario cosas distintas y en *Controller* no se debería saber nada de qué había por debajo del Game abstracto que él tenía para pedir las cosas necesarias para construir el *Game*.

En consecuencia, se ha solventado el problema pasando toda la funcionalidad de creación del juego a una clase factoría que es el *GameBuilder*. Esto permite que antes de iniciar del juego, dicha clase ya sepa qué tiene que preguntar y como tiene que construir el *Game* en función de la opción que haya marcado el usuario en el *Controller*.

Paquete Builders

El principal punto a favor de esta refactorización es que se ha restringido la creación de un *Game* a la creación del mismo a través de un *JSONObject* SIEMPRE, es decir, el método *createGame(JSONObject o)* solo precisa de un JSON para crear el juego y a partir de él se construye. Esto se ha hecho para que llegue como llegue la información del juego (en línea, de carga de fichero, generado por el usuario...) la factoría siempre sepa como crear el juego en cualquier caso, por tanto, se relega la responsabilidad de crear correctamente el *JSONObject* a cualquiera que cargue la información del juego.

Como los juego se guardan a través de un JSONObject, el *SaveLoadManager* no necesita más que, una vez cargado el JSONObject guardado en el fichero, pedirle a la factoría que lo construya y esta por dentro ya sabe como hacerlo. El mismo procedimiento se pretende para el juego en línea, por ejemplo.

GameBuilder

Para encapsular dicha funcionalidad y tal y como se ha explicado en la introducción, se creado una clase abstracta *GameBuilder* con método estáticos para la generación del juego. De dicha clase abstracta, extienden otras que son **factorías específicas** de cada modo de juego concreto y que realmente crean el juego una vez que se recibe el JSONObject.

En general, lo que tenemos es un método *public static Game createGame(JSONObject o)* que genera el juego en GameBuilder que hace lo siguiente:

```
public static Game createGame(JSONObject o){
        String type = o.getString("type");
        GameBuilder gameGen = GameBuilder.parse(type);
        if (gameGen == null)
                throw new IllegalArgumentException("——");
        else
                return gameGen.GenerateGame(o);
}
```

```
public static Game createGame(JSONObject o){
        JSONObject o = GameBuilder.ask();
        return this.createGame(o);
}
```

Es decir, parsea el tipo de juego que se va a generar entre las distintas factorías especializadas en cada juego y cuando alguna hace match con el tipo, entonces esa es la que genera realmente el juego. Sin embargo, no siempre nos llega un JSONObject hecho, si no que necesitamos pedirle al usuario los datos del nuevo juego que quiere generar.

Por el momento y ante la ausencia de envoltura a modo de *Observer* para la consola, el trabajo de preguntar lo hace la propia factoría a través de un método que se llama *private JSONObject ask()* y que se llama cuando se pide generar el juego sin ningún JSON en concreto. Este método genera el JSONObject del juego a través de preguntas generales sobre el mismo (como la creación del tablero o el número de jugadores) y para hacer las preguntas concretas para cada tipo de juego (como el equipo al que puede pertenecer un jugador) se parsea de nuevo el tipo del juego y se llama al método *ask()* de la factoría especializada que terminará de generar el juego correspondiente. Esto permite que la generación del juego siga yendo a través del cuello de botella del JSON y los cambios en el mismo no estropeen el resto de la práctica. El método general quedaría más o menos como:

```
private static JSONObject ask() {
        JSONObject o = new JSONObject();
        // elegimos el modo de juego concreto
        System.out.println(AVAILABLE_MODES_MSG);
        ...
        o.put("type", type);
        // Elegimos el numero de jugadores
        System.out.println(NUMBER_PLAYERS_MSG);
        int nPlayers = input.nextInt();
        input.nextLine();
        while (nPlayers < 2 || nPlayers > Color.size()) {
        ...
        }
        ...
        // Elegimos el tablero concreto
        System.out.print(BOARD_MSG);
        String board_shape;
        ...
        o.put("board", new Board(board_shape);
        // Llamamos al metodo ask especifico de la factoria especializada
        parse(type).whatINeed(nPlayers, o);
        o.put("turn", o.getJSONArray("players").getJSONObject(0).get("color"));

        return o;
}
```

GameClassicBuilder y GameTeamsBuilder

Como se ha mencionado anteriormente, cada modo de juego concreto tiene una factoría especializada que extiende de *GameBuilder* y que sabe crearlo a partir del JSONObject concreto. Estas factorías especializadas a su vez tienen un método *public void whatIneed(JSONObject o)* que reciben el JSONObject generado por el método *ask()* de *GameBuilder* y completan la información del mismo con lo que necesita cada juego en específico cuando se necesita crear el JSONObject para generar el juego. Principalmente solo constan de tres métodos abstractos que tienen que implementar por extender de *GameBuilder* (ya que en esta clase padre son abstractos):

```
protected void whatINeed(int nPlayers, JSONObject o) {
        ...
}

protected abstract boolean match(String type){
        ...
}

protected abstract Game GenerateGame(JSONObject o){
        ...
}
```

Paquete Logic

Como se ha añadido la funcionalidad de poder jugar por equipos, ha sido necesaria una refactorización completa de la parte que atañe a la lógica del juego. Se ha tratado de dejar un diseño que minimice el impacto de la creación de un nuevo modo de juego en el resto de funcionalidades como el cargado o el guardado de replays.

Game, GameClassic y GameTeams

Para poder generar los nuevos modos de juego, la clase *Game* ha pasado a ser una clase abstracta de la que extenderán los futuros modos de juego. Debido a esto, ha sido necesario dejar bien definida la funcionalidad común a todos los juegos (que será la que *Game* se quede como parte de su código) y la parte que corresponde a cada juego en concreto. En consecuencia, se han tenido en cuenta los siguiente criterios:

- Todos los modos de juego siguen teniendo *Players* y *Game* necesita de la lista de jugadores global para tener en cuenta puntuaciones, nombres, etc.
- Se necesitan los booleanos para saber si un juego ha terminado o se ha seleccionado salir de un juego.
- Siempre tenemos que saber cuál es el turno del jugador correspondiente
- Siempre habrá un tablero u otro (porque en el futuro igual puede haber distintos tableros).

Atendiendo a dichos criterios se ha dejado la clase *Game* de la siguiente forma:

```
public abstract class Game implements Replayable {
        protected boolean finished;
        protected List<Player> players;
        protected Board board;
        protected int currentPlayerIndex;
        private boolean exit;
        ...Constructores

        public void setExit() {
                this.exit = true;
        }

        public boolean exited() {
                return this.exit;
        }

        public boolean isFinished() {
```

```
        return this.finished || this.exit;
    }

    public Player getCurrentPlayer() {
        return this.players.get(currentPlayerIndex);
    }

    public abstract boolean play(int x, int y);
    public abstract String toString();
    public abstract Game copyMe();
    public abstract String showRanking();

    @Override
    public JSONObject report() {
        ...
        gameJSONObject.put("board", board.report());
        ...
        gameJSONObject.put("players", playerJSONArray);
        gameJSONObject.put("turn", currentPlayerIndex.getColor());

        return gameJSONObject;
    }
}
```

Claramente, las funcionalidades de jugar un turno (método *play()*), de imprimirse por pantalla, copiarse y mostrar su propio ranking son inherentes al juego concreto seleccionado y por eso se dejan como entidades abstractas propia de cada clase específica. La funcionalidad de *report()* también es inherente a cada tipo de juego, sin embargo, se ha dejado la parte común a todos como método *report()* del *Game* de modo que siempre se llame al de super() y después se añada lo que se requiera en cada clase específica.

La clase nueva, *GameTeams*, únicamente añade una lista de objetos del tipo *Team* puesto que la parte de jugar es la misma que *GameClassic* y la parte específica del report se simplifica gracias a la parte que corresponde a *Team*.

Team

Se ha creado esta nueva clase para el juego por equipos, posee una lista de jugadores que pertenecen a dicho equipo, un nombre y una puntuación. Su principal método que es *update()* actualiza la puntuación del equipo al final de cada turno sumando las puntuaciones de sus jugadores y posee un método estático que dado un jugador, devuelve el equipo al que pertenece.