 Diseño de Sistemas 2014

K3001

Jueves TM

**Futbol 5 – 4ta Entrega**

**Grupo 4**

|  |  |
| --- | --- |
| Nombre y Apellido | Legajo |
| Julian Fuks | 147.291-4 |
| Axel Suvalski | 147.290-2 |
| Nicolas Orchow | 146.700-1 |
| Joel Melamed | 146.804-2 |
| Eric Lifszyc | 146.655-0 |



Diagrama de clases

****

Análisis de la solución

en la solución, tuvimos que adaptar algunas clases a los cambios propuestos para esta entrega.

en la clase jugador, agregamos los métodos que luego funcionarán como criterios para el ordenado de listas, estos son: obtenerHandicap (para esto, agregamos un atributo hándicap, que va a ser el nivel de juego del jugador), obtenerPromedioUltimoPartido (que, como bien dice su nombre, nos devuelve el promedio de calificaciones del jugador de su último partido), y obtenerPromedioUltimosNPartidos (que nos devuelve las calificaciones de los últimos N partidos, recibiendo como parametro a N). Para evitar la repetición de código, la funcionalidad de obtener el promedio de los ultimos partidos queda definida en el metodo " obtenerPromedioUltimosNPartidos", el cual recibirá un 1 cuando el criterio utilizado sea "obtenerPromedioUltimoPartido", ya que este método llamará al anterior nombrado enviándole como parámetro, un 1.

en la clase Partido, agregamos los siguientes métodos:

- generarEquiposTentativos, el cual recibe como parámetro un criterio de Comparator<Jugador> (ya que el criterio será un mensaje que le enviamos al jugador, más adelante explicaremos el sentido de este parámetro), y una lista de índices del equipo 1, para que el administrador pueda decidir enviándonos como parámetro, cuáles son las posiciones de la lista ordenada de jugadores que obtendremos por el criterio, que ocuparán el equipo 1. Desde ya, las otras ocuparán el equipo 2.

- ordenarJugadoresPorCriterio, el cual es llamado por el método anterior, enviándole el mismo parámetro de criterio. Esto es solo para separar un poco la lógica y que el método no quede tan grande. Nos devuelve la lista de jugadores del partido ordenada por el criterio.

- generarListasDeEquipos, el cual es llamado por el primer método que describimos, recibiendo como parámetro la lista ordenada de jugadores y la lista de indices del equipo 1, y solo se ocupa de la lógica de definir e ir agregando los jugadores según la posición de la lista que ocupan y según la lista de indices lo indica, a sus respectivos equipos.

Además, agregamos dos atributos, equipo1 y equipo2. Desde ya, son listas de tipo jugador que se irán llenando en los métodos descriptos.

Patrones de diseño que utilizamos para la comunicación de componentes:

- Call and return: En este patrón de diseño, A le envía un mensaje a B, esperando una respuesta inmediata, B procesa el pedido y retorna un valor con un resultado. En nuestro caso, podemos ver este patrón aplicado en el componente A (Partido) que se comunica con el componente B (Jugador) para recibir el criterio de comparación para ordenar la lista y así, definir los equipos. B procesa el pedido, es decir, ejecuta el criterio, y lo devuelve, de forma tal que A, ya puede realizar la comparación para ordenar la lista.

- Shared memory: En este patrón de diseño, A coloca el mensaje en C, y B en algún momento se encarga de leer C y procesar el mensaje. Esto lo podemos ver aplicado en el componente Partido, donde A y B serían el partido mismo. En el método generarEquiposTentativos, es donde vemos aplicado el patrón, ya que vacía los equipos, los llena y los almacena ahí hasta que no se confirma que esos son los equipos. Es decir, A (Partido) coloca el mensaje generarEquiposTentativos en C, y B (Partido, devuelta) se encargará de leerlo y procesarlo, llenando los equipos.

Ejemplo de shared memory aplicado:

- Excepciones: esta es otra forma de comunicación entre componentes, en la que se comparte información, como por ejemplo, un mensaje de error. Lo podemos ver en la clase Jugador, donde, por ejemplo, en el método obtenerPromedioUltimosNPartidos, si el jugador no jugó la cantidad de partidos pedida por parámetro, lanzamos una excepción, avisando de este error a quien sea el encargado de atraparla. Si no usáramos este patrón no tendríamos forma de informar los errores entre componentes, si A falla, B nunca se entera.

Ejemplo de call and return y shared memory aplicado:

Ejemplo de call and return aplicado:

**public** **void** generarEquiposTentativos(Comparator<? **super** Jugador> criterio, ArrayList<Integer> indicesEquipo1)

{

equipo1 = equipo2 = **new** ArrayList<Jugador>();

ArrayList<Jugador> jugadoresSegunCriterio = **this**.ordenarJugadoresPorCriterio(criterio);

generarListasDeEquipos(jugadoresSegunCriterio, indicesEquipo1);

}

Vemos como esperamos en el parametro, esperamos un criterio que va a ser un método del jugador. Además, armamos los equipos tentativamente y, hasta que nadie los confirme, se rellenarán siempre y cuando el método sea llamado.

Ejemplo de excepción aplicado:

**if** (calificaciones.size() < cantidadPartidos)

**throw** **new** NoJugoSuficientesPartidosException("No se puede obtener el promedio ya que " + **this**.getNombre() + " no jugó " + cantidadPartidos + " partidos");

Concepto de diseño utilizado:

- Abstracción: definimos en un solo método la generación de equipos, recibiendo como parámetro un criterio, usando el comparator, que puede recibir cualquier tipo de comparador (en nuestro caso, van a ser criterios de comparaciones de jugador), y haciendo que la lista se ordene mediante este criterio recibido. Nos evitamos, de esta forma, repetir código y simplificar la lógica. De esta forma, mejoramos también la mantenibilidad.