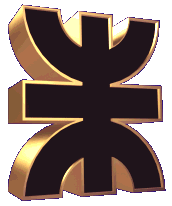
|  |
| --- |
| **Nombre y Apellido** |
| Cappellini Darío |
| Carella Germán |
| Centurion Damian |
| Fage Victor |
| Gonzalez Santiago |

|  |  |
| --- | --- |
| **Fecha De Presentación:** | 16/07/2014 |
| **Fecha de Devolución:** |  |
| **Calificación** |  |
| **Firma Profesor** |  |



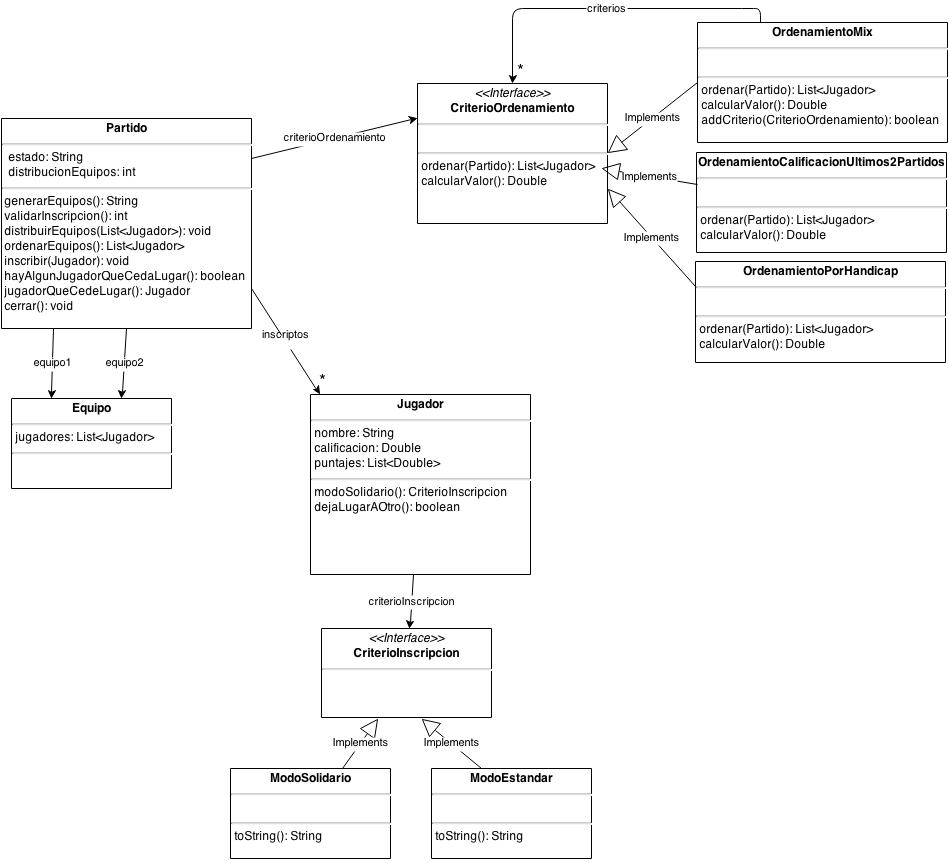
**GRUPO N° 7**

TP Anual: 5ta Entrega

***Diseño de Sistemas 2014***

***Miércoles mañana***

**Introducción**

Se recibió una solución con la siguiente estructura:

De acuerdo a lo solicitado, sólo se pasa a analizar la parte del diseño que tiene que ver con el caso de uso “Generar equipos”. Para generar los equipos, el partido primero le pide a su criterio de ordenamiento que ordene la lista de jugadores, pasándola por parámetro y recibiéndola como valor de retorno. Esta lista de jugadores ordenada se la pasa a sí mismo como parámetro para generarEquipos. Este método generarEquipos llama otro método, que realiza validaciones, y pregunta por el valor de retorno, si es -1 es porque hubo un error y entonces levanta una excepcion, sino puede continuar. Parte de estas validaciones es preguntar si el estado del partido permite la generación de equipos, el estado es un string y el partido sabe qué string corresponde a qué comportamiento.Luego pregunta por el tipo de distribución propio (un int), y si es 5 distribuye los jugadores en par-impar y si no es 5 asume que es 16 y los distribuye de la otra forma propuesta. Notar que el partido aparece con algunas responsabilidades de más: preguntar por un código de error, preguntar por el tipo de distribución y actuar según la respuesta, preguntar por su estado. También se observa directamente en el diagrama que la clase Equipo no tiene comportamiento.

**Puntos a mejorar:**

String estado

En la solución propuesta, el partido se guarda un string "estado". Según ciertos eventos, setea este string y después al momento de hacer validaciones pregunta con condicionales por su valor. Esto es un ejemplo del bad smell conocido como "primitive obsession", ya que estoy utilizando un tipo de datos del lenguaje donde podría tener una abstracción útil para el modelo.

La consecuencia principal es que el partido está teniendo responsabilidad de más, ya que debe preocuparse por implementar distintas acciones según el valor de "estado". Esto hace que el código sea menos modular y se concentre en una sola clase: si tengo que cambiar cómo se comporta algún estado o agregar algún estado nuevo, tengo que encontrar la línea específica en el partido. Además, la solución propuesta es poco expresiva y para poder mantener el código hay que acordarse que "A" es abierto, "C" es cerrado, etc etc.

El refactor propuesto es reificar el estado del partido y cambiar todos los "if estado == ..." por mensajes al estado. Esto nos permite eliminar algunos condicionales, reemplazándolos por una sola sentencia, y nos da la posibilidad de poder usar el partido sin que se entere cuál es su estado. El estado pasa a ser un objeto aparte, y se puede agregar cualquier cantidad de nuevos estados sin que eso tenga consecuencias en el partido.

Códigos de error mediante valores de retorno

En el partido se utilizan valores de retorno para representar condiciones de error. Esto nos hace un poco de ruido porque sugiere que no se está aprovechando bien el mecanismo de excepciones que ofrece Java.

En efecto, generarEquipos llama a validarInscripción y, si devolvió -1, lanza una BusinessException. Esto es mala idea porque, si tuviera que manejar otro tipo de errores, tendría que agrandar ese if, lo que quita mantenibilidad. Además, el mensaje de la excepción es muy poco informativo ("hubo un error").

Aprovechando que validarInscripción es un método nuestro, el refactor lógico es reemplazar todos los "return -1" por "throw new BusinesException(msj)" donde, para cada caso, se incluya un mensaje un poquito más útil de lo que pasó. Así, no necesito preguntar qué retorno validarInscripción, sino que dejo que levante la excepción que requiera. Pasa a ser un método que retorna void. Si se incluyeran nuevas validaciones que el "hubo un error" que está representando el -1, el partido no se entera. Observar también que el partido tampoco necesita saber que se está levantando una excepción, pero por cómo funciona el mecanismo de excepciones, la funcionalidad no cambia: los objetos que llamen al partido van a recibir la excepción por parte del partido como antes del refactor. El código del método generarEquipos() queda mucho más corto y va directo a lo que le importa.

Código repetido en las implementaciones de CriterioOrdenamiento

En todas las implementaciones de "CriterioOrdenamiento" aparece el mismo código para el método "ordenar". Es decir, el código está repetido en las tres clases que lo implementan.

Esto demuestra que tenemos un comportamiento común a todos los criterios de ordenamiento y no lo estamos reconociendo como tal. En este caso, si quisiéramos cambiar la forma de ordenar, tendríamos que tocar las N clases que implementen esta interfaz, dificultando la modificación del código. Además, la posibilidad de olvidarse de alguna clase al hacer tal cambio pone en peligro la consistencia del comportamiento esperado.

El refactor propuesto es convertir la interfaz "CriterioOrdenamiento" en clase abstracta, para poder ubicar ahí el código que comparten las implementaciones (que pasan ahora a ser subclases).

Clase vacía: Equipo

La clase Equipo no tiene comportamiento. Se le está dando clase propia a una idea que bien puede ser representada con un ArrayList. Se propone eliminar la clase Equipo ya que no aporta nada al diseño.

Int distribucionEquipos

El int distribucionEquipos está representando, con un entero, una forma de resolver un problema. Esto es un problema porque, por una parte, los ints asignados son arbitrarios y por lo tanto dificultan la legibilidad del código (los comentarios explicando el significado del int son un ejemplo de comentarios como code smell), y por otra parte se está agregando comportamiento dentro del partido que podría estar delegado en otro objeto.

Esto es un problema porque se tiene un switch (en este caso un if pero la idea es la misma) en el partido que puede crecer indefinidamente si se agregan N formas de distribuir los equipos. Además, por cómo está en el código ahora, se está infiriendo que si no es 5 es 16 por descarte, el valor 16 no se usa ni siquiera como selector.

Se propone crear un objeto que cumpla la tarea de ordenar los equipos. Notar que para ordenar los equipos hay que setear las variables equipo1 y equipo2 de partido, por lo que este objeto debe poder hablar con el partido. Elegimos usar una interfaz con N implementaciones para los distintos modos de distribución y pasarles el partido como parámetro (solución sin estado). Se observa que al hacer este cambio el código del partido queda muchísimo más limpio y legible.