Un refresco a la memoria

Empezamos este trabajo reuniendonos con el cliente para que nos comente un poco sobre el sistema que quiere que le diseñemos y las funcionalidades esperadas.

Entre otras cosas nos comentó que tenía un problema con sus amigos para organizar las partidos de fútbol 5 y que quería que lo ayudemos a automatizar este proceso.

Entre varias cosas nos habló de que quería que poder organizar un partido y que los jugadores se anoten a este de diferentes maneras posibles. También contó que los partidos generalmente no eran parejos y que nuestro sistema pudiera armar las partidos lo más equitativamente posible. Por último, nos precisó que estaría bueno que este sistema pudiera castigar a los jugadores que no fueran al partido para que nunca falten jugadores.

Ante esta situación, le propusimos fraccionar este nuevo sistema en varias interacciones, donde cada una de ellas agregue más funcionalidad.

Como primera iteración acordamos que el sistema pudiera organizar partidos y permitirle a los jugadores inscribirse a ellos de la forma que ellos quisiesen.

Bajado a idioma desarrollador, lo que hicimos fue abstraer la idea de partido, jugador y tipo de inscripción en clases. La interfaz de entrada de nuestro sistemita era inscribirJugador(Jugador, TipoDeInscripcion).

Cometimos un error a propósito: le dimos entidad a Jugador cuando este, en realidad, no hace nada, ni tampoco tiene atributos propios. Sin embargo, consideramos que en el futuro iba a ser necesario.

A la la entidad tipo de inscripción le dimos una definición de interfaz y a cada tipo de inscripción las hicimos implementar esta interfaz.

Segunda iteración

Nos reunimos de nuevo con el cliente para mostrarle los avances de nuestro sistema y quedó encantado. Después de felicitarnos, nos pusimos a hablar de las nuevas funcionalidades a incorporar.

El cliente nos solicitó varias mejoras al sistema. Uno de los problemas que tenía era que no se sabía cuándo había 10 jugadores confirmados para jugar y cuándo no. Además muchas veces pasaba que había 10 confirmados y uno se bajaba y no avisaba, por lo que nadie se enteraba y no podían buscar un reemplazante. Por último, nos comentó que muchas veces él se olvidaba de avisarle a sus amigos de que había un partido y después se enojaban con él.

Básicamente los problemas que tenían eran de comunicación y esa fue la gran motivación de esta interacción. Sin olvidarse que también había que darle la posibilidad a los jugadores de bajarse de un partido. Y en eso nos pusimos a trabajar.

Resumiendo

Nuevas funcionalidades a incorporara:

1. Caso de uso de baja de un jugador de un partido
2. Notificar cuando se completó un partido
3. Notificar cuando se descompletó un partido
4. Notificar cuando se inscribió un amigo

Para seguir con la idea de desarrollo incremental nos propusimos implementar cada una de estas funcionalidades en diferentes iteraciones.

Caso de uso de baja de un jugador

La idea de este caso de uno es que se pueda dar de baja de un partido un jugador con la posibilidad de indicar un reemplazante. Si no se especifica reemplazante, el jugador adquiere una infracción.

Lo que pensamos en este caso es que la clase partido maneje la baja de los jugadores ya que ella sabe quiénes son los inscriptos a ese partido. Pensamos en cohesividad en este caso, ya que nos pareció correcto que la clase Partido maneje la situación del jugador en el partido (si está inscripto, si no lo esta, si se baja).

Para desarrollar esto le agregamos un método que actuaría de interfaz de entrada a la clase Partido. Lo llamamos darDeBaja para ser expresivos y darDeBaja ConReemplazo si es que indica un reemplazo.

Parece trivial la implementación de este método pero no lo es tanto.

Planteamos dos posibilidades de resolución:

1- Buscar en la lista el jugador, eliminarlo y en caso de reemplazo agregarlo.

2- Usar la potencia del lenguaje Java para resolverlo (el diseño no es independiente de la tecnologia).

Optamos por la segunda ya que le agrega mucha más declaratividad a nuestro sistema, propiedad que nos gusta mucho. Adiós al for, buen día a los métodos otorgados por la clase Stream.

Por último nos falta atacar el tema de las infracciones. Acá se nos complicó un poco el tema. ¿Por qué? Bueno, el tema es que pensamos en que una infracción debería ser algo en nuestro sistema, teniamos que abstraer la idea. Pero, ¿de qué

forma la abstraemos? Por lo pronto en Java existen tres formas: una clase común, una clase abstracta o interfaz.

El análisis hecho nos solicita que una infracción tenga una fecha y un motivo, por lo tanto descartamos a la interfaz porque ésta no puede tener atributos. Por lo que nos queda una clase común o una clase abstracta. Pero he aquí un problemón, el análisis no indica que esta clase tiene comportamiento y si no tiene comportamiento va en contra de lo que debería ser una clase, un conjunto de comportamiento cohesivo con la posibilidad de incorporar atributos.

No nos pudimos convencer con ninguna de las soluciones. Optamos por abstraerla en una clase con los atributos fecha y motivo a sabiendas de que está mal, pero con la intención de solucionar este problema en la próxima interacción. Lo anotamos en nuestro TODO.

Notificar cuando se completó un partido

La idea de esta funcionalidad es, básicamente, que en el momento que se inscriba un jugador a un partido y con él se complete el partido con jugadores confirmados, se le mande un mensaje al administrador.

Dos ideas/entidades nuevas florecen de este requerimiento. La idea de que un partido tiene un administrador o de alguna manera desde el partido se puede llegar a él. La idea de un sistema de mensajes o de envío de mensajes.

Al atacarla lo primero que se nos ocurrió fue proveer a la clase de un atributo administrador y un método nuevo llamado avisarAlAdmin. Este método se dispararía desde el método inscribirJugador que comprobaría si con el jugador agregado se completó el partido. Esta idea fundamentalmente aporta mucha simplicidad ya que solamente habría que crear una clase administrador, agregar un atributo y un método a la clase partido. Sin embargo le quita cohesividad a la clase partido ya que, además de manejar el estado de los jugadores, pasaría a enviar mensajes al administrador. Si vamos a agregar cosas, no caguemos lo otro. La descartamos.

Quedamos varados y a la deriva. Estábamos convencidos de que agregarle responsabilidad no cohesiva a partido estaba mal. Sabiamos que algo de administrador tiene que haber, tal vez no una clase porque no tiene comportamiento, pero algo hay. Cada vez más nos gustaba la idea de un sistema de notificaciones.

Dejamos de lado por un rato este requerimiento y seguimos con el otro.

Notificar cuando se “descompleta” un partido

Este requerimiento tiene capitales semejanzas con el anterior. La idea de algo del administrador, de un sistema de mensajes, de que la clase partido no se puede encargar de enviar mensajes. La dejamos en stand-by a la espera de una idea.

Notificar cuando se inscribe un amigo

La idea de esta funcionalidad es que cuando un jugador se inscriba el sistema le avise a sus amigos, simplemente eso, o no tan simple.

Aparecen varias cositas nuevas acá y algunas que ya se mencionaron.

En principio, aparece la idea de amigo. ¿Qué es un amigo? Bueno, nosotros pensamos que un amigo era también un jugador de fútbol (sino para que le iba a avisar). Entonces descartamos la idea de una abstracción nueva llamada amigo.

Si un jugador tiene amigos jugadores, se nos ocurrió que tenga un atributo colección de amigos. Es decir, que un jugador de fútbol sabe quiénes son sus amigos (tiene sentido). A partir de ahora, comienza a tener más sentido la clase jugador que, a propósito, codificamos en la primera iteración.

Pero, ¿qué hacemos con el envio de mensajes? ¿Quién tiene la responsabilidad de avisarle a los amigos del jugador: el sistema o el jugador? Dificil decisión.

Y salió la luz: Observers

Leyendo y releyendo los apuntes, nos convencimos de que la solución estaba en los Observers. Según tenemos entendido, los observadores observan a los objetos, más precisamente a los cambios de estados de éste. En base a eso y ante la señal del objeto, estos actúan.

Los eventos que acontecían eran tres: el partido se completa, el partido se descompleta, un jugador se inscribe.

Sabíamos que esta era la respuesta pero había dos posibilidades de implementarla:

1. Crear una interfaz Observador que observe a la clase partido y tenga varios métodos de acuerdo a que evento sucedía.

interfaz ObservadorPartido
    metodo notificarCompleto
    metodo notificarDescompleto
    metodo notificarAmigos(amigos)

2. Crear varias interfaces Observador que observen cada una a un evento determinado y tengan un método que notifique que el evento sucedió.

interfaz ObservadorCompleto

metodo notificar
interfaz ObservadorDescompleto
metodo notificar
interfaz ObservadorInscripcion
metodo notificar(amigos)

Las analizamos:

- Ambas soluciones desacoplan. ¿Cómo lo hacen? Bueno, liberando a la clase partido de tener que incorporar o referenciar a un sistema de mensajes para tener que avisar según corresponda.
- Ambas soluciones aportan considerable flexibilidad. Es muy fácil agregar nuevos observadores que actúen ante los eventos del sistema. Estos tienen que implementar nada más la interfaz Observador correspondiente y redefinir los métodos. Se complica si necesitan cosas que no son pasadas por los parámetros del método.
- Ambas soluciones son complejas. Es un concepto no tan común ni trivial para los programadores novatos, al principio puede costar pero es una complejidad necesaria.
- La gran diferencia entre las dos opciones es que los observadores que implementen el observador partido es probable que no redefinan todos los métodos que le pasa la interfaz ya que no tiene asignado comportamiento ante ese evento. En cambio, en la otra solución, los observadores solamente van a redefinir un método y si o si van a redefinir ése método porque es lo que le importa. En el primer caso implementás métodos que no usas, en el segundo no.