Diseño de Sistemas 2014



K3001

Jueves TM

**Futbol 5 – 2da 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 – Solución 1

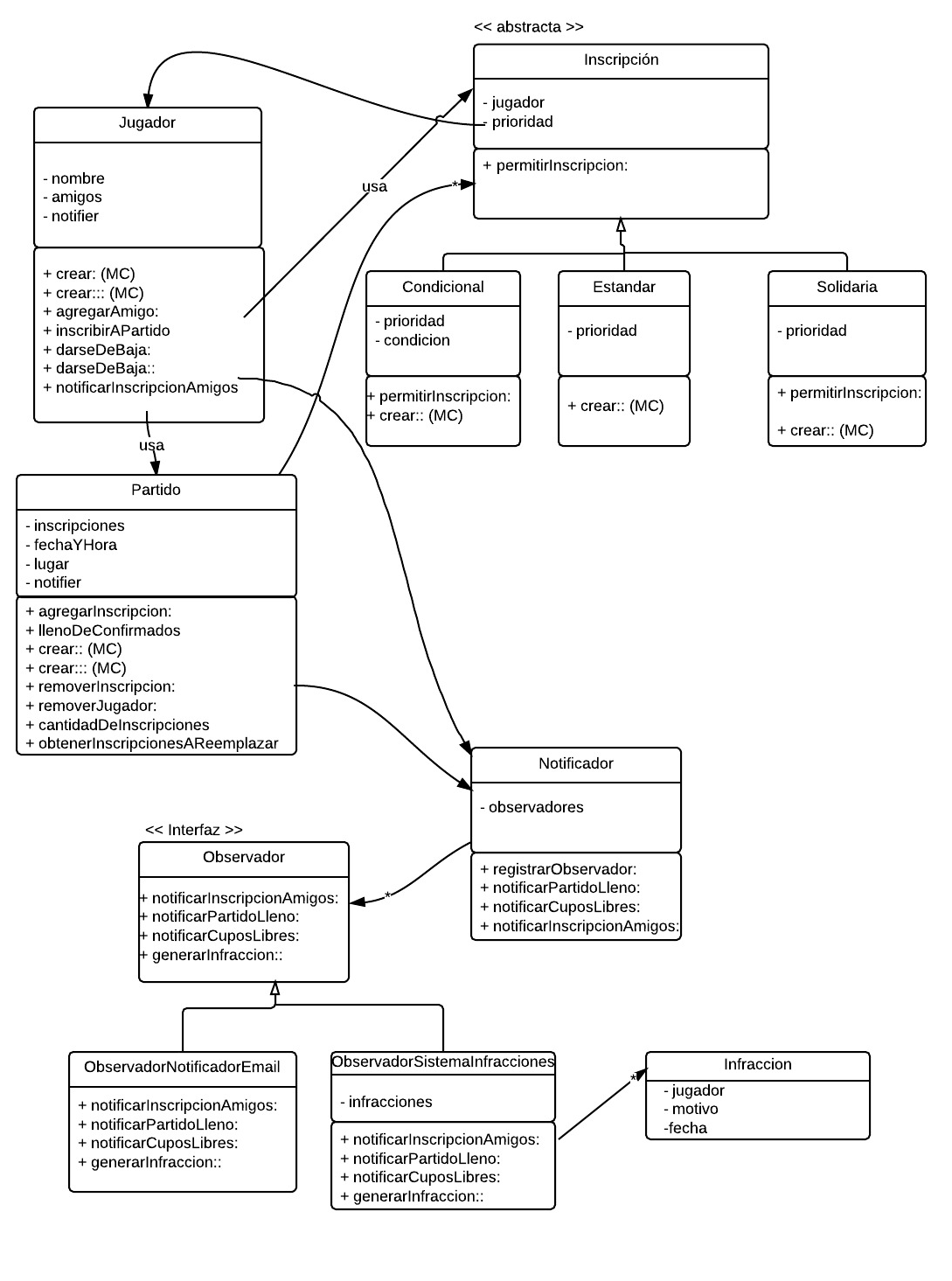
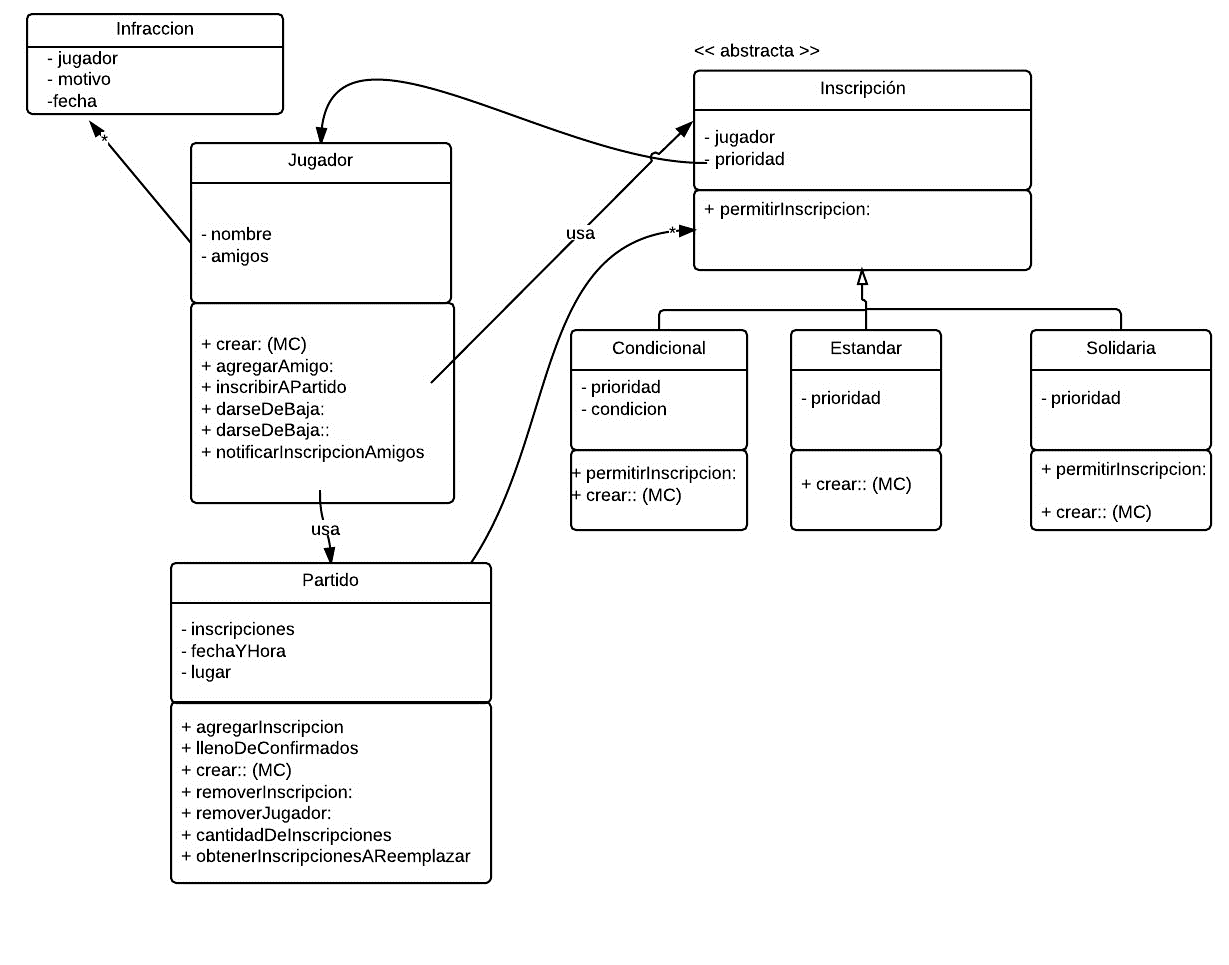


Diagrama de clases – Solución 2



Analisis comparativo de las soluciones

en la solución 1, tuvimos que adaptar algunas clases a los cambios propuestos para esta entrega. En la clase Jugador, hicimos una sobrecarga del método Crear, para que además de crear al jugador con sus características específicas, se setee el notificador. Además, obviamente tuvimos que agregar este atributo (notifier).

También, se agregó la lógica para darseDeBaja, la cual tiene dos sobrecargas. Una se utiliza cuando el jugador se da de baja por algun motivo y no presenta un reemplazo, por lo que se le genera una infracción. Y la otra se usa cuando el jugador en cuestión presenta un reemplazante para inscribir al partido.

Se agregó también una lista de amigos al Jugador, dado que en esta entrega contamos con esa posibilidad.

Por último, apareció otro método nuevo: notificarInscripcionAmigos. Este método hace que todos los observadores que estaban interesados en la inscripción de un jugador, reciban este aviso. particularmente, en este caso hay un observador que lo que hace es enviarle un mail a todos los amigos del jugador inscripto.

Hablando de la clase Partido, similar a lo que pasa en la inscripción de un jugador, si el partido está lleno se le avisa (por mail) al administrador. Al igual que en la clase Jugador, se hizo una sobrecarga del método Crear, para poder setear el notificador. Y al momento de remover una inscripción, se sumó la funcionalidad de notificar al administrador que quedaron cupos libres (en caso de que esto sucediera).

Por otro lado, agregamos la interfaz Observador, la cual es implementada por las clases ObservadorNotificadorMail y ObservadorSistemaInfracciones. Y además, sumamos la clase notificador, que es la que se comunica directamente con los observadores.

Los observadores son manejados por la clase notificador, la cual los va agregando o eliminando de una colección de interesados. Por ejemplo:

**private** ArrayList<Observador> \_observadores = **new** ArrayList<Observador>()

**public** **void** registrarObservador(Observador observador){

\_observadores.add(observador);

}

}

Entrando en las ventajas de utilizar esto, el observer es un patrón que permite definir un objeto observable (el notificador) y varios interesados u observadores (el notificador de mails y el registro de infracciones). El objeto observado no conoce exactamente las clases concretas de cada observer, sino que delega en una colección que provee una interfaz común para todos los objetos observadores.

De esta manera, podemos ver dos características de diseño importantes: composición (por sobre la herencia) y polimorfismo.

Otra de las ventajas es que observador y observado tienen un bajo acoplamiento ya que el comportamiento del observador está casi completamente aislado. lo único que se le provee es una interfaz de salida a la cual se debe adaptar.

El grado de dependencia es bajo, cosa que mejora la mantenibilidad de la solución, aumenta su reutilización y evita la propagación de errores.

Por otro lado, cada observador concreto maneja su implementación en forma independiente de las demás, y si la interfaz del observador no cambia, la solución permite agregar nuevos observadores sin que el objeto observado sufra cambios. (favorece la extensibilidad)

En la solución 2, tuvimos que realizar cambios similares en las clases Jugador y Partido, ya que la lógica de los cambios correspondientes nos lo exigían. Es decir, agregamos el comportamiento para que estas dos clases notifiquen (tanto por mail como creando las infracciones), ya que toda la lógica de las notificaciones se hace de la misma manera: Se notifica a los amigos de un jugador cuando este se inscribe a un partido, se notifica al administrador ante algún cupo libre en un partido o cuando alguno de estos está lleno, y se genera una infracción al jugador que se dio de baja de un partido y no consiguió un reemplazante.

Lo que cambia en esta solución con respecto a la solución 1, es que no existe la interfaz Observador, ni las clases que heredan de ésta. Así que las clases Partido y Jugador llevan adentro toda la lógica que en la otra solución solamente la conocen los observadores.

La desventaja que produce pensar la solución de esta manera, es que para agregar nuevas funcionalidades, debemos modificar la clase que corresponda. Esto hace que la flexibilidad de la solución sea menor, como también su mantenibilidad, ya que hay que esforzarse más al momento de adaptar alguna funcionalidad.

Cómo ayuda cada solución a aumentar la cohesión de los componentes

En la solución 1, podemos ver que, como la lista sólo conoce a la interfaz observador, aumenta su cohesión ya que no tiene que manejar la lógica de envío de mails ni de generar infracciones (que se encuentran dentro de ObservadorNotificadorMail y ObservadorSistemaInfracciones, respectivamente). Es decir, toda la lógica se encuentra dentro de los observadores.

Los objetos observados, simplemente disparan el evento para que los observadores interesados lo tomen y ejecuten lo que corresponde. Se ve claramente en el siguiente ejemplo:

>>clase jugador

**public** Boolean darseDeBaja(Partido unPartido, String motivo){

**if**(unPartido.removerJugador(**this**)){

getNotifier().**generarInfraccion**(**this**, motivo);

**return** **true**;

}

**return** **false**;

}

>>clase notificador  
**public** **void** generarInfraccion(Jugador jugador, String motivo) {

**for**(Observador observador : \_observadores){

observador.generarInfraccion(jugador, motivo);

}

}

>>Clase ObservadorSistemaInfracciones

**public** **void** generarInfraccion(Jugador unJugador, String motivo){

Infraccion nuevaInfraccion = **new** Infraccion();

nuevaInfraccion.setJugador(unJugador);

nuevaInfraccion.setFecha(**new** Date());

nuevaInfraccion.setMotivo(motivo);

\_infracciones.add(nuevaInfraccion);

}

En cambio, en la solución 2, toda la lógica queda en el jugador, que realiza más cosas y tiene por lo tanto menos cohesión:

**public** Boolean darseDeBaja(Partido unPartido, String motivo){

**if**(unPartido.removerJugador(**this**)){

Infraccion nuevaInfraccion = **new** Infraccion();

nuevaInfraccion.setJugador(unJugador);

nuevaInfraccion.setFecha(**new** Date());

nuevaInfraccion.setMotivo(motivo);

\_infracciones.add(nuevaInfraccion);

**return** **true**;

}

**return** **false**;

}

Objeto “impostor” para los tests

La idea del ‘mock’ es cambiar, tanto los casos de prueba como las unidades que estamos probando, sin que uno afecte al otro. Esto lo utilizamos para simular el comportamiento de objetos bastante complejos, de los cuales es complicado reflejar en el test todas las funcionalidades que tienen, o todo lo que necesitan para llevarlas a cabo. Entonces, modelando un objeto impostor, podemos imitar el comportamiento de un observador, por ejemplo, de manera controlada, y orientando los resultados a lo que necesitamos para implementar la prueba.

@Test

**public** **void** testDarseDeBajaConInfraccionOk(){

Mockito.doReturn(**true**).when(unPartido).removerJugador(unJugador);

Assert.assertTrue(unJugador.darseDeBaja(unPartido, "motivo") == **true**);

verify(notifier).generarInfraccion(unJugador, "motivo");

//esto verifica que haya llamado UNA vez al sistema de infracciones

}

Conclusión

Tanto por lo enunciado en el apartado de ‘*Analisis comparativo de las soluciones*’ como en el análisis que hicimos sobre la cohesión, creemos la solución 2 puede ser más simple de entender y también de programar, pero sin embargo la solución 1 es la más correcta de ambas, ya que nos permite que en un futuro, se agreguen nuevos observadores fácilmente (crear la clase y agregar los eventos que necesite en la clase Observador) y esto es algo probable que pase.