 Diseño de Sistemas 2014

K3001

Jueves TM

**Futbol 5 – 1era 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

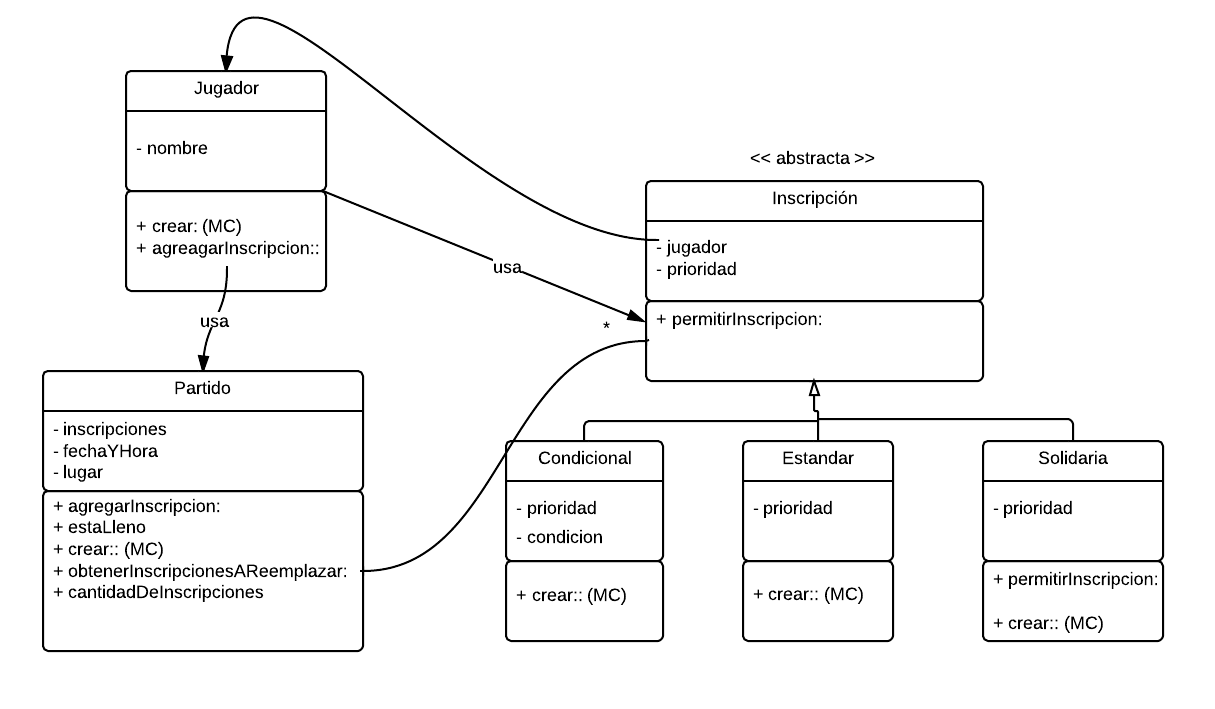
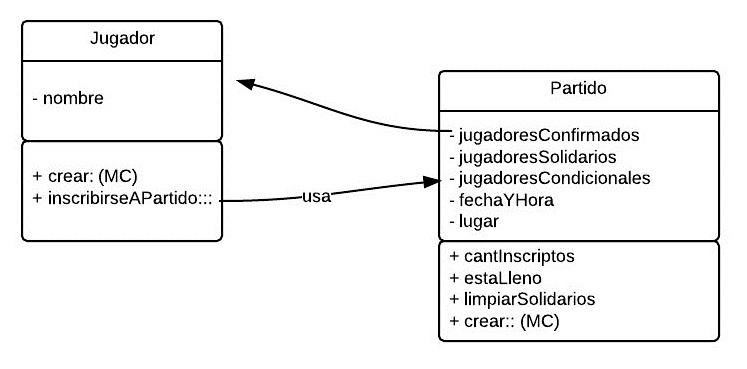


Diagrama de clases – Solución 2



Analisis comparativo de las soluciones

en la solución 1, hay una clase abstracta Inscripcion de la cual hereden las subclases InscripcionCondicional, InscripcionEstandar, e InscripcionSolidaria. De esta manera, no solo nos ahorramos código pudiendo reutilizar funciones, sino que también permitimos redefinir la responsabilidad particular que tengan las subclases como por ejemplo la determinar cuando una inscripción de un jugador puede ser agregada a la lista de inscripciones de un partido. Esto nos permite además, agregar atributos propios de cada tipo de inscripcion, como la subclase InscripcionCondicional, que posee un atributo condicion, que en caso que cumplirse permite agregar la inscripción. Lo que logramos, es que cada subclase posea lo minimo y necesario para que cada una cumpla con los requisitos del mercado, es decir, que las otras clases que implementan la clase Inscripcion, no tienen por qué conocer este atributo Condicion, ya que no la utilizan.

Además, para el proceso que determina la posible inscripción a reemplazar en caso que la lista de inscripciones ya este completa (cuando no estan todos confirmados), fue necesario definirle un atributo prioridad a cada tipo de inscripcion (Estandar=1, Solidaria=2, Condicional=3), quedando así definida la condicion para saber que inscripcion reemplazar: va a ser la primera que se encuentre presente en la lista de inscripciones de un partido cuya prioridad sea numericamente mayor.

A la hora de agregar una inscripción a un partido (cualquiera sea el tipo de inscripción), se debe llamar al método ‘agregarInscripcion’ de la clase Jugador. En primera instancia va a preguntar si el partido ya se encuentra completo de inscripciones Estandares. En caso que no lo esté, va a llamar polimorficamente al método ‘permitirInscripcion’ perteneciente a la subclase de inscripción que se quiere agregar. En caso que se le permita inscribir, se llamará al método agregarInscripcion de la clase Partido.

El tipo de inscripción condicional posee un atributo Condicion que es del tipo Predicate<Partido>. Predicate es una interfaz incluida a partir de Java 1.7 que permite hacer exactamente lo que nosotros deseamos: a partir de determinada condición (representada mediante una lambda) evaluarla para determinado partido y a partir de allí decidir si agregar la inscripcion a la lista de inscripciones de dicho partido.

En la solución 2, esta herencia no se encuentra presente. Sino que dentro de la clase Partido hay tres colecciones de jugadores, donde dependiendo de cada tipo de inscripcion se van a ir agregando a dichas listas. En primera instancia, podemos sacar como conclusión que va a haber presente un Code-Smell a lo largo de esta solución: Type-Test (explicado más abajo). Como podemos ver en el diagrama de clases, ni siquiera podemos observar que exista la clase Inscripcion, sino que directamente depende del tipo de inscripcion

(va a ser una cadena de caracteres, o un tipo de datos primitivo). En este punto, es donde vemos un nuevo Code-Smell**: Primitive Obsession.**

El Type Test se observa en el momento de agregar el jugador al cual se quiere inscribir, ya que va el sistema va a preguntar si la inscripcion es de tipo Estandar, Condicional o Solidaria. Este “problema” no se observa en la solución 1 ya que al poseer una clase abstracta y tres subclases que implementan dicha clase, basandonos en la idea del polimorfismo (poseer distintos tipos de objetos a los cuales puedo enviarles un mensaje y que todos lo puedan entender e implementar su correspondiente metodo), delegamos la responsabilidad de determinar si una inscripcion es valida a cada subclase.

El otro code-smell (Primitive Obsession) se observa en el método que agrega el jugador a la lista de inscripciones correspondientes, cuando el parametro que recibe para determinar que tipo de inscripcion va a ser una cadena de caracteres, o un int determinado, cuando realmente se podría reemplazar por una tipo de de datos complejo, como se observa en la Solución 1.

En el proceso de determinacion de la inscripcion a reemplazar, se puede volver a observar el Code-Smell de Type-test ya que para identificar dicha inscripcion, primero debe ir preguntando de que tipo es la nueva inscripcion, y una vez definido, debo comenzar a buscar la lista de inscripciones a reemplazar.

En la segunda solucion, como no estan presentes las clases correspondientes a los distintos tipos de inscripciones, tampoco vamos a encontrar un atributo correspondiente a la condicion que deba cumplir el partido para luego poder agregarle una inscripcion condicional. Para poder lograr esta funcionalidad, el metodo de agregar inscripcion (presente en la clase Jugador) va a poseer un parametro que sea una funcion Lambda. Un defecto de esto, es que cuando la inscripción no sea del tipo condicional, el método va a seguir esperando un parámetro que sea la condición, cuando realmente no es necesario ya que en el relevamiento nunca dice que las inscripciones estandares/solidarias conozcan a un objeto que represente una condición.

La segunda solucion, A la hora de agregar una inscripción a un partido (cualquiera sea el tipo de inscripción), se debe llamar al método ‘inscribirseAPartido’ de la clase Jugador. En primera instancia va a preguntar si el partido ya se encuentra completo de inscripciones Estandares. En caso que no lo esté, va a determinar el tipo de inscripción (nuevamente vemos el Type-Test). dentro de la misma clase Jugador, dependiendo del tipo identificado, va a depender de lo que va a hacer (según lo obtenido en el “informe de relevamiento” que tenemos como consigna).

A demás La segunda solución presenta otro Code-Smell: duplicated-code. Esto viene aparejado tras el Type-Test, ya que en todo momento, se esta determinando el tipo de inscripcion que se trata de agregar y a su vez las decisiones a tomar dependiendo de dicho tipo.

Analisis comparativo según los atributos de diseño

Mantenibilidad

La primer solucion, es mucho más mantenible que la solución 2. Esto se debe a que el hecho de haber delegado las responsabilidades a las clases que realmente deberían poseerlas, nos permite evitarnos ampliamente la repetición de código presente en la solución 2. Esto facilita que en el caso de presentarse un error o tener que adaptar una funcionalidad existente, tanto saber perfectamente en que clase o parte de código se encuentra presente la situación a corregir como tambien reescribir el código en un único lugar, permitiría adaptar la situación sin tener que volver a escribir los métodos que utilizan dicha funcionalidad a adaptar.

En la solución 2, al tener código repetido, en caso de encontrarse un error, o tener que adaptar una funcionalidad, va a haber que reescribir tanto cada una de las implementaciónes de dicha parte de código duplicado, como así tambien los métodos que utilizan a dicho código.

Este es un concepto importante, ya que los recursos (costo, tiempo, personas, etc.) son muy importantes a la hora del desarrollo del software, por lo que al hacer algo más mantenible, nos permite ahorrarnos dichos recursos y poder utilizarlos posteriormente para realizar mejoras sustanciales al proyecto.

Simplicidad

Como podemos ver a simple vista en el diagrama de clases, una solución (la primera) presenta seis clases mientras que la segunda presenta solamente 2. Más allá de esto (que no nos permite asegurar nada todavía acerca de la simplicidad), podemos observar que para lograr entender el flujo de datos dentro del sistema, previamente quien lo lea tiene que conocer que es una clase abstracta, para que sirve, porque se realizó de dicha forma. Además va a tener que conocer los conceptos de orden superior y las expresiones lambda, tanto para entender el método que determina las posibles inscripciones a reemplazar de determinado partido, como así también el que define la condición a evaluar a la hora de agregar una inscripción condicional.

Sin embargo, la solución 2 es más simple, basta con conocer pocos conceptos tecnicos para lograr entender lo que hace el sistema. Claramente, la incorporación de tres colecciones distintas para manejar los tipos de inscripciones simplifica bastante, pese a que algoritmicamente sea un horror (tener que preguntar con que tipo de inscripción estoy trabajando en todo momento antes de la toma de cualquier decisión).

Conclusión

Tanto por lo enunciado en el apartado de ‘*Analisis comparativo de las soluciones*’ como en

‘*Analisis comparativo según los atributos de diseño*’, creemos que pese a que la solución 2 sea más simple de entender, la solución 1 es la más correcta de ambas, ya que por un lado, las responsabilidades estan perfectamente delegadas, cada clase es responsable de hacer lo que debe. Por otro lado, evitamos el Type-Test presente a lo largo de todo la solución 2 (ya que no estan presente la clase Inscripcion y sus subclases). Además, nos permite que en un futuro, se agreguen nuevos tipos de inscripciones facilmente (crear la clase y hacer que implemente la clase abstracta Inscripcion) y esto es algo probable que pase. A su vez, gracias a la inclusión de las subclases y la herencia, nos evitamos la repetición de código debido a la necesidad tanto de determinar el tipo de inscripcion y las acciones a realizar dependiendo de esto.