Curso K3031 | Diseño de Sistemas | 1C 2014

Organizador de partidos

de fútbol 5 (OPF5)

GRUPO 6

Buzzetti, Jonathan P.

Dardis, Marco

Scarazzato, Omar

Lott, Sebastián

# ENTREGA 1

## Enunciado

1. El objetivo de esta entrega es diseñar en detalle la inscripción de un jugador al partido, partiendo por los casos de prueba. Para esto se debe presentar:

Implemente los casos de prueba, donde debe indicar

Escenario (pre-condición)

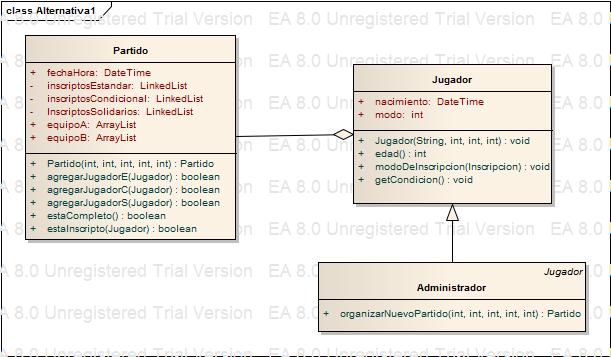
Serie de pasos a ejecutar

Resultado esperado (post-condición)

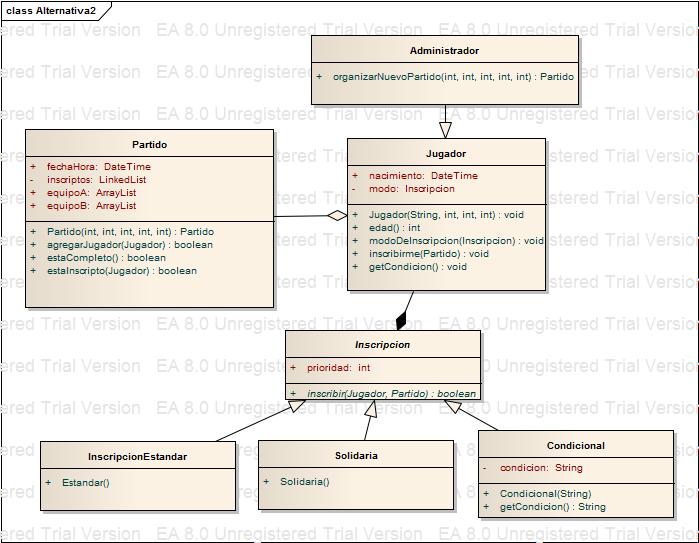
O bien los tests cases que implementen dichos casos de prueba (incluyendo un fixture o juego de datos).

1. Modele dos posibles soluciones alternativas para ese caso de uso. Luego, realice un análisis comparativo de ambas soluciones y elija una de las opciones, justificando dicha decisión.
2. La implementación del código de negocio necesario para resolver lo pedido en los casos de prueba.
3. Una explicación detallada sobre la vinculación entre las decisiones de diseño y la correspondiente implementación/especificación, que ayude a la trazabilidad de la solución. Utilice las herramientas que crea convenientes para explicar las decisiones de diseño que tomó en cada caso (diagramas de clase, secuencia, colaboración, objetos, tests, código o pseudocódigo, prosa castellana, etc.). No olvide indicar qué componentes aparecieron, cuáles son sus responsabilidades y las colaboraciones con otros componentes.
4. Analizar cada una de las soluciones teniendo en cuenta los atributos de diseño simplicidad y mantenibilidad.

## Solución 1



## Solución 2



## Análisis Comparativo

En ambos diseños la clase Jugador tiene 2 atributos: el nacimiento, seteado en el momento de la instanciación, y el modo de la inscripción. El *Administrador* es una subclase de *Jugador* que por el momento, además de inscribirse, puede crear u organizar un *Partido* estableciendo su fecha y hora.

Para el formato de la hora se utilizó la librería Joda Time, que optimiza y facilita el manejo de la fecha y la hora, tanto de partidos como de jugadores con la clase *DateTime*.

En la **alternativa Nº1** se optó por utilizar una lista para cada tipo de inscripción de los jugadores. Para individualizar el agregado a cada una de las listas, se tuvo que implementar un método para cada una, o bien se podrían haber hecho públicas las listas, pero se hubiera roto la encapsulación de *Partido*. De esta manera, el partido tiene la lógica de agregar a los jugadores que se inscriben, y es más simple tomar decisiones e identificar a los inscriptos según su modo debido a las tres listas. Por ejemplo, un partido está completo cuando la lista *inscriptosEstandar* tiene 10 jugadores.

En la **alternativa Nº2** la clase *Partido* tiene una sola lista de inscriptos. Se delegó la lógica de la inscripción en Jugador, respetando la historia del cliente.

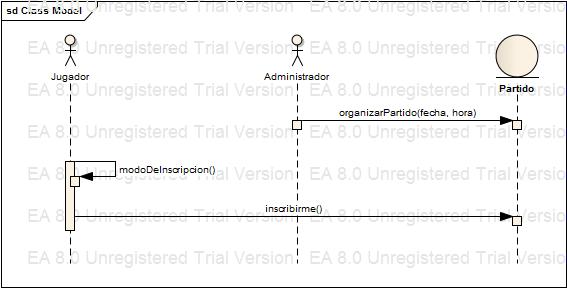
Se utilizó el patrón State para que el jugador, según el atributo “modo” decida que algoritmo utilizar para agregarse a la lista de inscriptos de Partido. De esta manera se hace uso del 1er. principio de diseño al utilizar la composición del modo de inscripción, en el jugador. El jugador tiene por defecto el modo Estándar al ser instanciado.

Seleccionamos la alternativa Nº2 porque es más flexible en cuanto a la implementación futura de otros modos de inscripción, ya que se desacoplan tanto del Jugador como del *Partido*. Así, solamente extendiendo la clase *Inscripcion*, el sistema sigue funcionando perfectamente.

### Conclusión

La solución elegida tiene la cualidad de una mayor mantenibilidad, por lo anteriormente mencionado, y respeta la lógica señalada por el cliente de modo que cada jugador decide de qué manera inscribirse en el partido. Esto hace que los futuros cambios solicitados sean más simples de incorporar al diseño existente.

## Implementación



### Caso de Uso “Organizar nuevo partido”

*mario* = **new** Administrador("Mario", 1985, 5, 20);

*partidoPrueba* = *mario*.organizarNuevoPartido(2014, 6, 18, 15, 30);

### Caso de Uso “Inscribirse a un partido”

*luis* = **new** Jugador("Luis", 1990, 5, 20);

*luis*.modoDeInscripcion(**new** Condicional("Juego si esta nublado"));

*mario*.inscribirme(*partidoPrueba*);

*luis*.inscribirme(*partidoPrueba*);

*jose*.modoDeInscripcion(**new** Solidaria());

*jose*.inscribirme(*partidoPrueba*);

# ENTREGA 2

## Enunciado

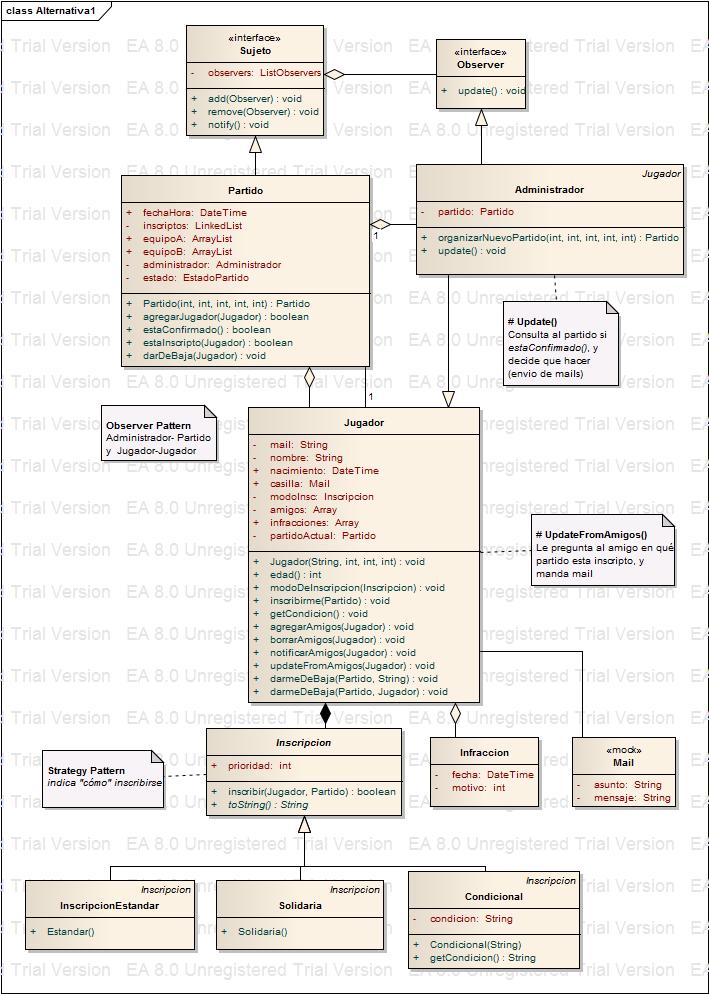
Tras ver los primeros avances, el cliente nos pidió que incorporemos algunas mejoras en el ciclo de vida del partido:

* Cada vez que el partido esté confirmado (esto es, que los 10 jugadores confirmaron su participación según la modalidad en la que se anotaron, *no implica que el administrador confirma los que efectivamente jugaron, cosa que ocurre en otro momento*), se debe notificar al administrador del sistema.
* Incorporar como caso de uso la baja de un jugador a un partido. Cuando esto ocurre
  + el jugador debe indicar qué jugador lo reemplazará, en caso contrario se le generará una infracción. Como el sistema de infracciones es por tiempo limitado, nos interesa poder discriminar infracciones de diferentes momentos y por distintos motivos.
  + si el partido deja de tener los 10 jugadores confirmados, se debe notificar al administrador del sistema.
* Cada vez que un jugador se inscriba al partido se debe notificar a sus amigos.

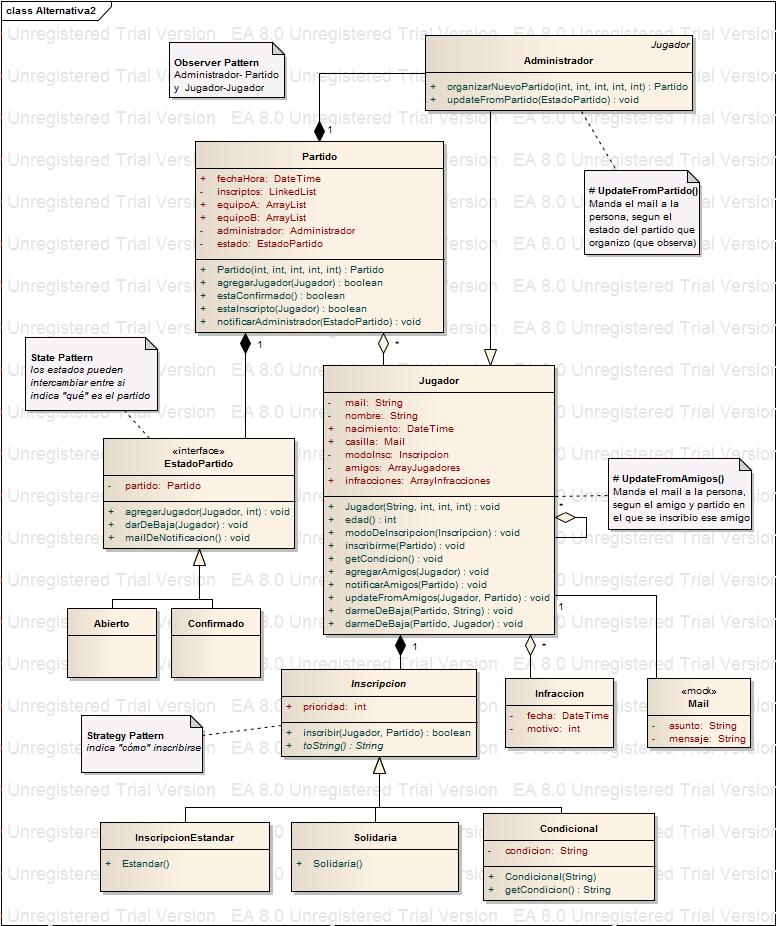
Se pide:

1. Diseñe la solución aplicando dos soluciones posibles. Recuerde utilizar las herramientas que crea convenientes para documentar las decisiones de la misma manera que en la entrega 1.
2. Implemente los casos de prueba automatizados que permitan garantizar la consistencia de las soluciones a. y b, para lo cual
3. Modelar un objeto impostor (mock / stub) que simule el envío de mails. Indique cómo este objeto facilita la realización de pruebas.
4. Haga un análisis comparativo de ambas soluciones en base a:
   * cómo se incorpora cada funcionalidad pedida a lo desarrollado anteriormente
   * qué consecuencias trae en cada solución el manejo de la identidad de los objetos existentes
   * cómo ayuda cada solución a aumentar la cohesión de los componentes.

## Solucion 1



## Solucion 2



## Objeto Impostor

En nuestro caso creamos la clase MailSender, la cual se encarga de gestionar el envío de mails a los jugadores. Por el momento, los receptores son los objetos para poder realizar los tests. La instancia de MailSender recibe los datos necesarios para redactar un mail a través del método *compose*. Luego, se le indica un objeto destinatario para enviarlo. Éste objeto va a guardar el Mail en el atributo *casilla* del Jugador, de manera que pueda ser consultado más tarde.

**public** **void** compose(String from, String to, String subject, String message)

La utilización de un *stub object* facilitó el diseño y las pruebas porque su propósito es la abstracción del proceso de envíos de mails. Podemos considerar que este proceso no se encuentra dentro del dominio del sistema, sí en cambio la utilización del módulo como entidad externa.

Además, se encuentran distribuidas variadas implementaciones de éstos módulos en el mercado.

## Análisis Comparativo

### Funcionalidades

La notificación al administrador del sistema, tenía dos posibilidades: que el partido se “vuelva confirmado” o que “deje de estar confirmado”.

Para implementarlo, en la alternativa N°1 se optó por usar el patrón Observer, donde Partido es el Sujeto y Administrador el Observador. El administrador también debe conocer al partido para, al momento del update, preguntarle cuál es su **estado** mediante el método *estaConfirmado()*. En base a esto, es el mismo administrador quien decide qué debe hacer y cómo con un *if*.

En la alternativa N°2, se tomó la idea del mismo patrón pero personalizándolo de acuerdo a lo necesario. Como el único que necesita saber el estado del partido es el administrador, en lugar de tener una lista dinámica de observadores, pasó a tener un atributo (administrador) generado al momento de la instanciación del partido. Los **estados** del partido se encapsularon en clases derivadas de la interfaz **EstadoPartido**. Allí, además de especificar el contenido de la notificación o mail al administrador, se realiza la lógica de alta y baja de los jugadores.

Para la baja de jugadores, en la alternativa 1 es la clase Partido la que tiene el algoritmo y toma las decisiones de cómo hacerlo en el método *darDeBaja()*. En la alternativa 2, esto se delega a EstadoPartido, teniendo el comportamiento determinado según si ya tiene 10 confirmados o no.

La infracción en ambos casos se genera desde la clase Jugador en el momento que decide darse de baja sin dar un reemplazo. Se utilizó la sobrecarga de métodos de Java.

Para la notificación de amigos no se pudo implementar el patrón Observer directamente ya que es la misma clase la que cumpliría con los dos roles al mismo tiempo. En ambos casos se agregó una lista de amigos a la clase Jugador. En la alternativa 1, el Jugador guarda en un atributo el partido en el que se encuentra inscripto. En la Nº 2 directamente se envía, al momento de la inscripción, el mensaje *notificarAmigos()* con el jugador mismo y el partido al que se inscribió como parámetros. Luego el jugador observador (el que debe actualizarse) realiza el envío del supuesto mail.

**private** **void** notificarAmigos(Partido partido) {

**for**(Jugador amigo : **this**.amigos){

amigo.update(**this**, partido);

}

}

### Consecuencias

En ambos casos los objetos mantuvieron su identidad previa, siendo esto comprobado al correr satisfactoriamente los tests de la primera entrega sin realizarle modificaciones.

Como el envío de mails y el cambio de estado del partido son cosas que las clases manejan internamente, no hubo necesidad de que los objetos modificaran el comportamiento previo.

### Cohesión

La alternativa N°2 tiene una mayor cohesión en cuanto al comportamiento del partido. Ya que gracias al patrón State se agrupan los distintos procesos que debe realizar el partido dependiendo de su estado, permitiendo además extender esta característica en el futuro mucho más fácilmente.

En la N°1 la clase Administrador decide que es lo que tiene que notificarse, disminuyendo así la cohesión.

### Conclusión

Elegimos la segunda alternativa de diseño ya que tiene una estructura más simple de acuerdo a las necesidades y a la vez fomenta la encapsulación de los elementos. Además posibilita que un administrador organice más de un partido porque no es un atributo sino un objeto desacoplado.