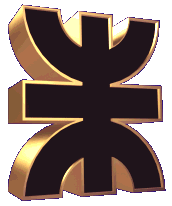
|  |
| --- |
| **Nombre y Apellido** |
| Cappelini Dario |
| Carella Germán |
| Centurion Damian |
| Fage Victor |
| Gonzalez Santiago |

|  |  |
| --- | --- |
| **Fecha De Presentación:** | 21/05/2014 |
| **Fecha de Devolución:** |  |
| **Calificación** |  |
| **Firma Profesor** |  |



**GRUPO N° 7**

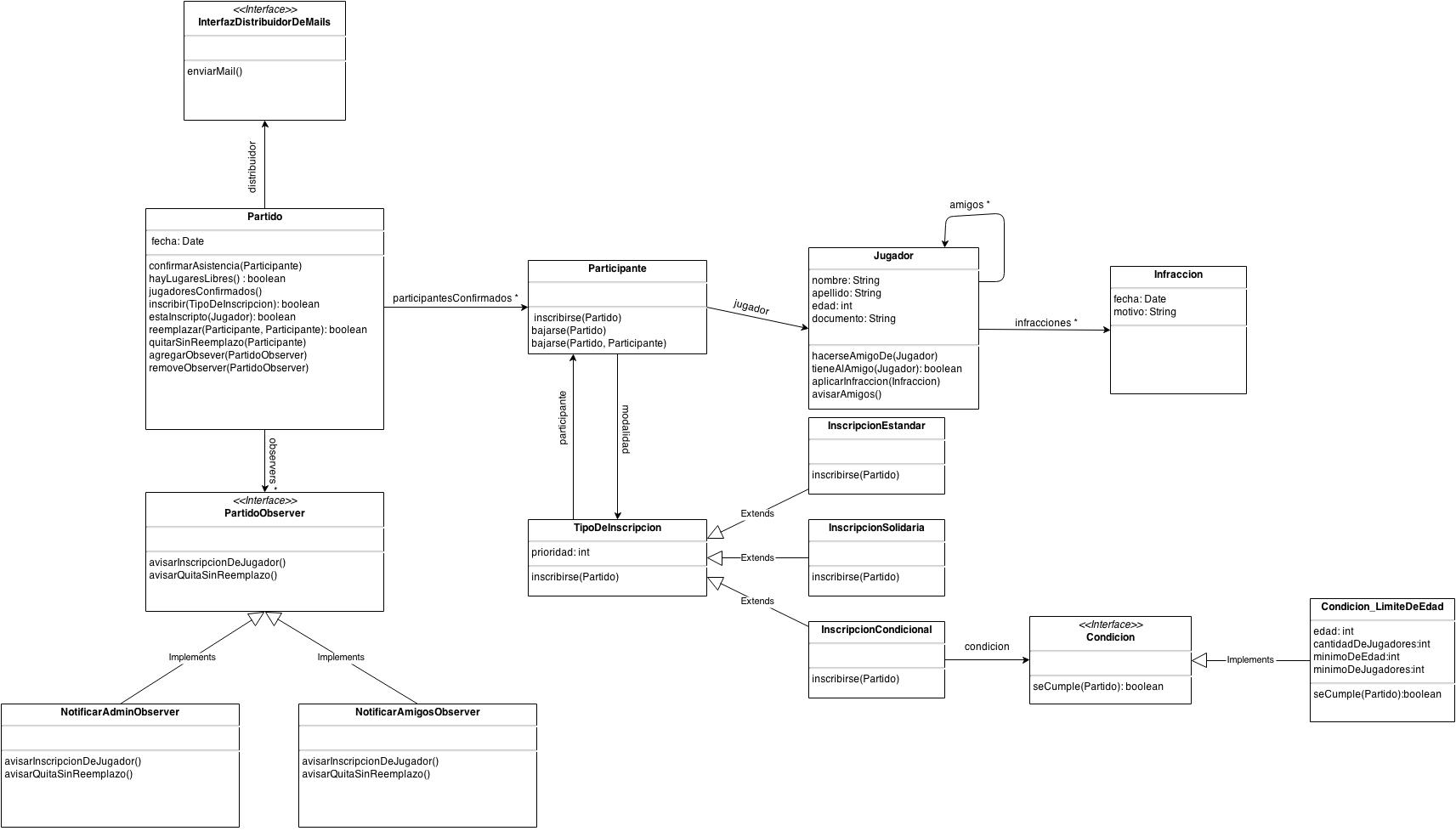
TP Anual: 2da Entrega

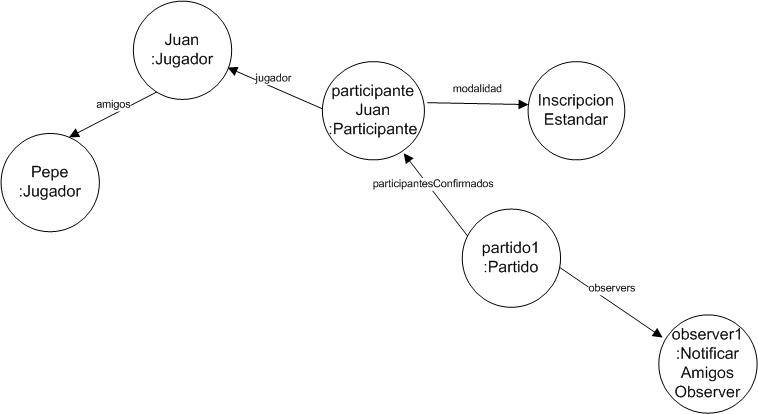
***Diseño de Sistemas 2014***

***Miercoles mañana***

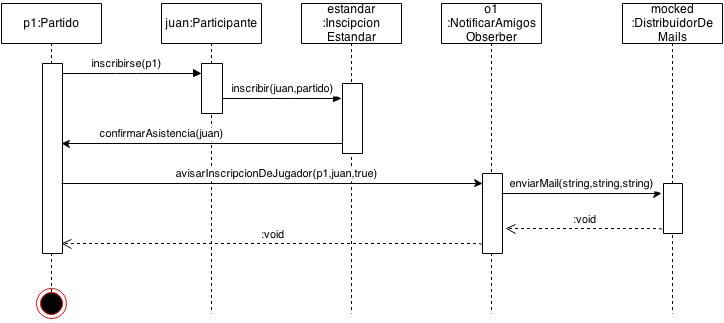
1. Proponemos dos soluciones, una que utiliza el patrón observer y otra que emplea el patrón decorator. Con observer el partido “avisa” a sus observadores cuando ocurren ciertos eventos. Con decorator existen decoradores que interceptan ciertos mensajes que les interesan como forma de enterarse de estos eventos. Como la funcionalidad pedida es la misma, tanto los observer como los decorator realizan las mismas acciones, la diferencia está en cómo son llamados, cómo se enteran de que ocurren los eventos, y a qué objetos conocen. Incluimos algunos diagramas de objetos, de secuencia y de clases para cada solución.

**Utilizando el Patrón “Observer”**

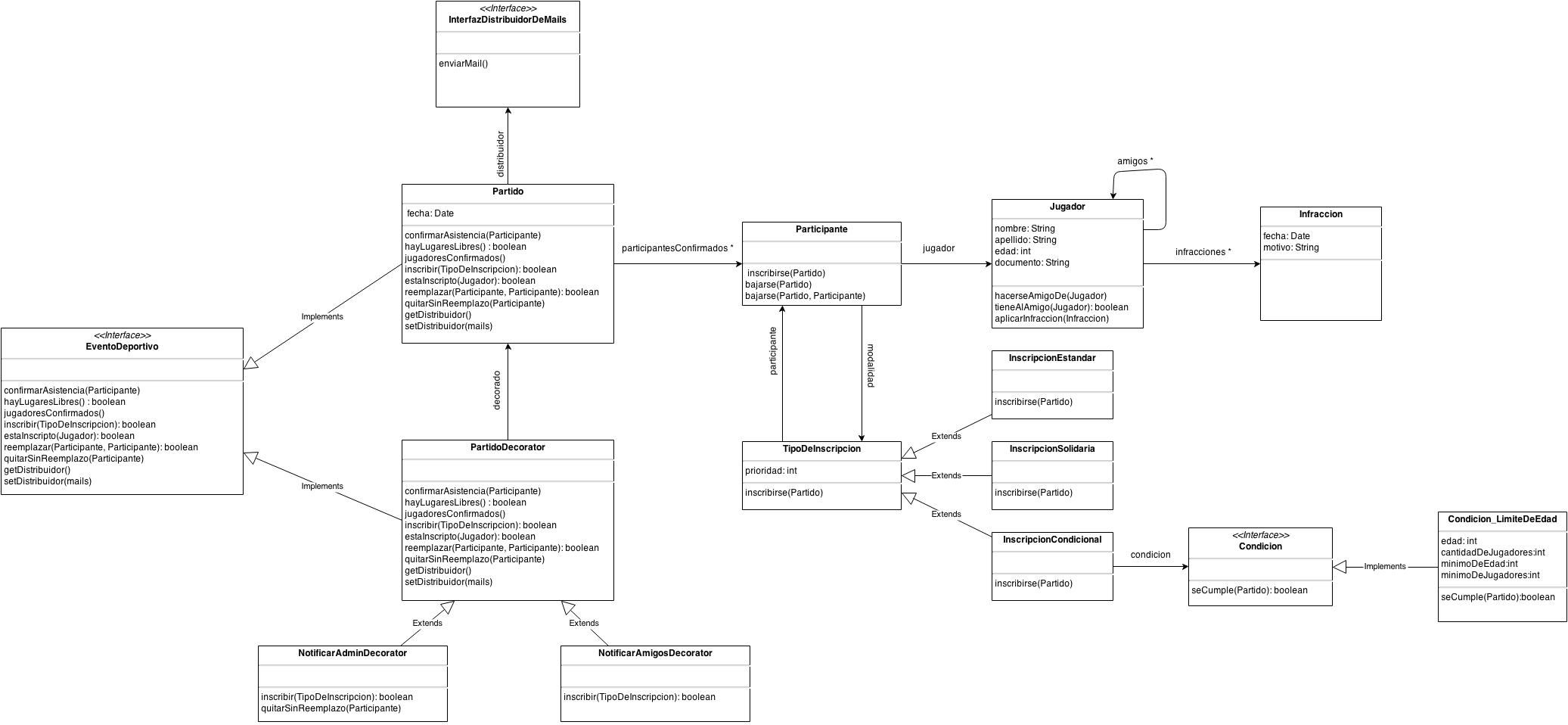
* **Diagrama de Clases**
* **Diagrama de Objetos**

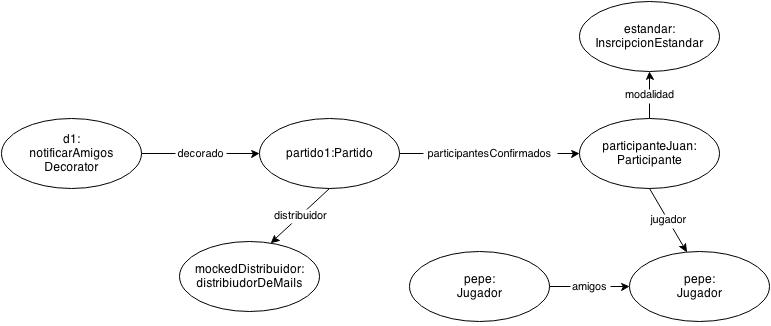


* **Diagrama de Secuencia**

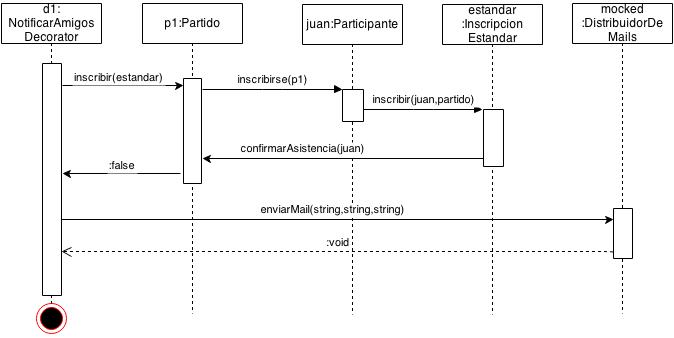


**Utilizando el Patrón “Decorator”**

* **Diagrama de Clases**
* **Diagrama de Objetos**



* **Diagrama de Secuencia**



2. Clasificamos las pruebas automatizadas según los distintos requerimientos: tenemos las pruebas para la inscripción de jugadores que verifican todo lo implementado en la entrega anterior, un conjunto de pruebas para la funcionalidad de la baja del jugador, y dos conjunto de pruebas para el envío de notificaciones. Tenemos dos para esto último porque la construcción y configuración de los objetos necesarios es distinta en la solución observer que en la decorator, así que necesitamos pruebas específicas para cada una. Prueban lo mismo y el comportamiento desde el punto de vista del usuario es el mismo, pero la configuración es distinta.

3. Para armar nuestro objeto impostor usamos el framework Mockito. En los diagramas de clases se observa la interfaz “InterfazDistribuidorDeMail” que no está implementando nadie. Esta interfaz está para que nosotros podamos enviar mensajes pidiendo que se mande un mail, y recibir excepciones si no se pudo. En un sistema real, las implementaciones de esta interfaz serían adapters que trabajen con la biblioteca real de envío de mails, que es externa a nuestro sistema. Como el enunciado no especifica de qué forma se envían las notificaciones, y considerando que no podemos mandar mails reales como parte de nuestros tests, no tenemos ninguna implementación. Lo que funciona como “implementación” son las herramientas que nos da el framework Mockito, con las que verificamos si se “mandó cierto mail” simplemente preguntando si alguien llamó a tal o cual método de la interfaz con los argumentos esperados. Así, el atributo “distribuidor” del partido apunta a un mockedDistribuidor que es un objeto que adopta la interfaz del DistribuidorDeMails y nos permite hacer esas preguntas “estadísticas”. No nos interesa la clase de ese objeto ya que es algo que maneja Mockito, sólo nos interesa la seguridad de que implementa nuestra interfaz.

4. Consideramos que las nuevas funcionalidades se dividen en dos tipos bien diferenciados:  
a. Implementar la baja del jugador incluyendo un sistema de infracciones.  
b. Implementar un sistema de notificaciones que envíe mensajes para ciertos eventos.

Para (a) no presentamos dos alternativas distintas sino que simplemente agregamos comportamiento al jugador, al partido, y agregamos una clase “infracciones” para que el jugador coleccione. No incluimos diagramas para esta parte porque es muy simple: hay un método bajarse() en el participante que usa sobrecargas. Esto permite a un jugador bajarse nombrando a un reemplazante o no. La variable sin reemplazante envía un mensaje especial al partido que, además de quitar al participante de la lista de confirmados, le genera una infracción al jugador correspondiente (enviándole un mensaje que lleva como parámetro una new Infracción()).

Para (b) sí tenemos dos soluciones distintas, que ya fueron explicadas. Para llevarlas a cabo se agregaron los objetos descriptos en el punto (1), y los atributos necesarios para relacionar esos objetos y los que ya había en el sistema.

Más allá de eso, se producen las siguientes consecuencias interesantes en cada solución:  
-En la solución observer, el partido termina teniendo líneas de código dedicadas a avisar él a sus observadores de cosas que en realidad a él no le interesan. Esto reduce la cohesión del partido, ya que por ejemplo el método inscribirJugador() pasa de ser “inscribir jugador” a “inscribir jugador y avisar a los observadores”. Ahora tenemos pocos observadores pero a futuro podemos llegar a tener este tipo de llamadas dispersas por todo el partido dificultando la separación entre el código “útil para el partido” y el código que está sólo para avisar a los observer. Es de destacar, de todas formas, que la pérdida de cohesión sería aún mayor si apuntáramos a una solución parecida pero sin aplicar el patrón, por ejemplo si cuando se inscribe un jugador el mismo partido se fija si ahora está confirmado y en ese caso manda el mail correspondiente.  
-En la solución decorator, la identidad del partido deja de ser confiable. Notar que esto no es un problema ahora porque en nuestra solución actual nadie conoce al partido, pero si de repente algún requerimiento futuro resultara en que alguien lo conociera, deberíamos tener cuidado con este punto. Usando decorator sería mala idea agregar y quitar decoradores una vez creado el objeto, ya que esto implica actualizar todas las referencias en el sistema (que, repetimos, en este caso no es ninguna pero esto podría cambiar). Además, agregar o quitar un decorador dado de una cadena es por demás difícil dada la manera en que se conocen. Considerando esto, queda muy a la vista de que este es un problema que la solución observer no tiene ya que se pueden suscribir y desuscribir observers a voluntad en cualquier momento del ciclo de vida del partido sin mayor dificultad.

En conclusión, la solución decorator nos obliga a que, cuando un partido se creó con cierto comportamiento de notificaciones, permanezca así hasta que desaparezca ya que cambiarle el comportamiento significa cambiarle la identidad. La solución observer, al contrario, nos permite cambiar esto a voluntad de manera sencilla. Como contrapartida, el observer nos obliga a agregar líneas de código en el partido que no son comportamiento que le interesa al partido directamente, afectando la legibilidad del código y la cohesión de los métodos. En este aspecto se hace más fuerte la solución decorator, ya que el partido no debe modificarse para nada, y no sabe que está siendo decorado. Todo el comportamiento de las notificaciones queda en otros objetos que el partido ni siquiera conoce.