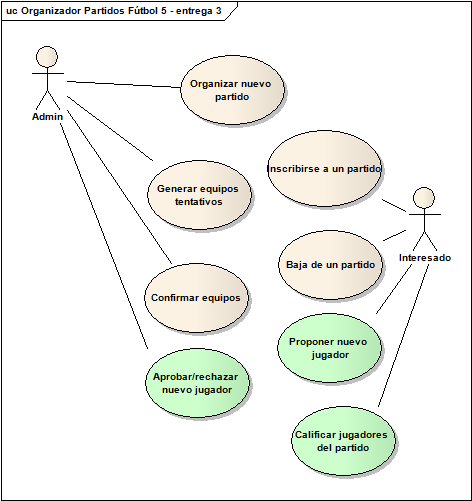
**CU Nuevo Jugador**

(Proponer jugador + Aceptar + Rechazar)

Este diagrama de casos de uso, del enunciado, nos muestra dos cosas: en un momento T1, el interesado es quien propone nuevos jugadores. Y luego en otro momento T2, el admin toma la propuesta, y la aprueba; o la rechaza.



El paso de mensajes para proponer (y aceptar) podría ser así:

En el momento T1, la persona inicia la propuesta indicando el amigo, y la modalidad en que quiere inscribirlo. Despues le pasa la propuesta al admin, quien la almacena en una coleccion de propuestas pendientes. (CU Proponer nuevo jugador)

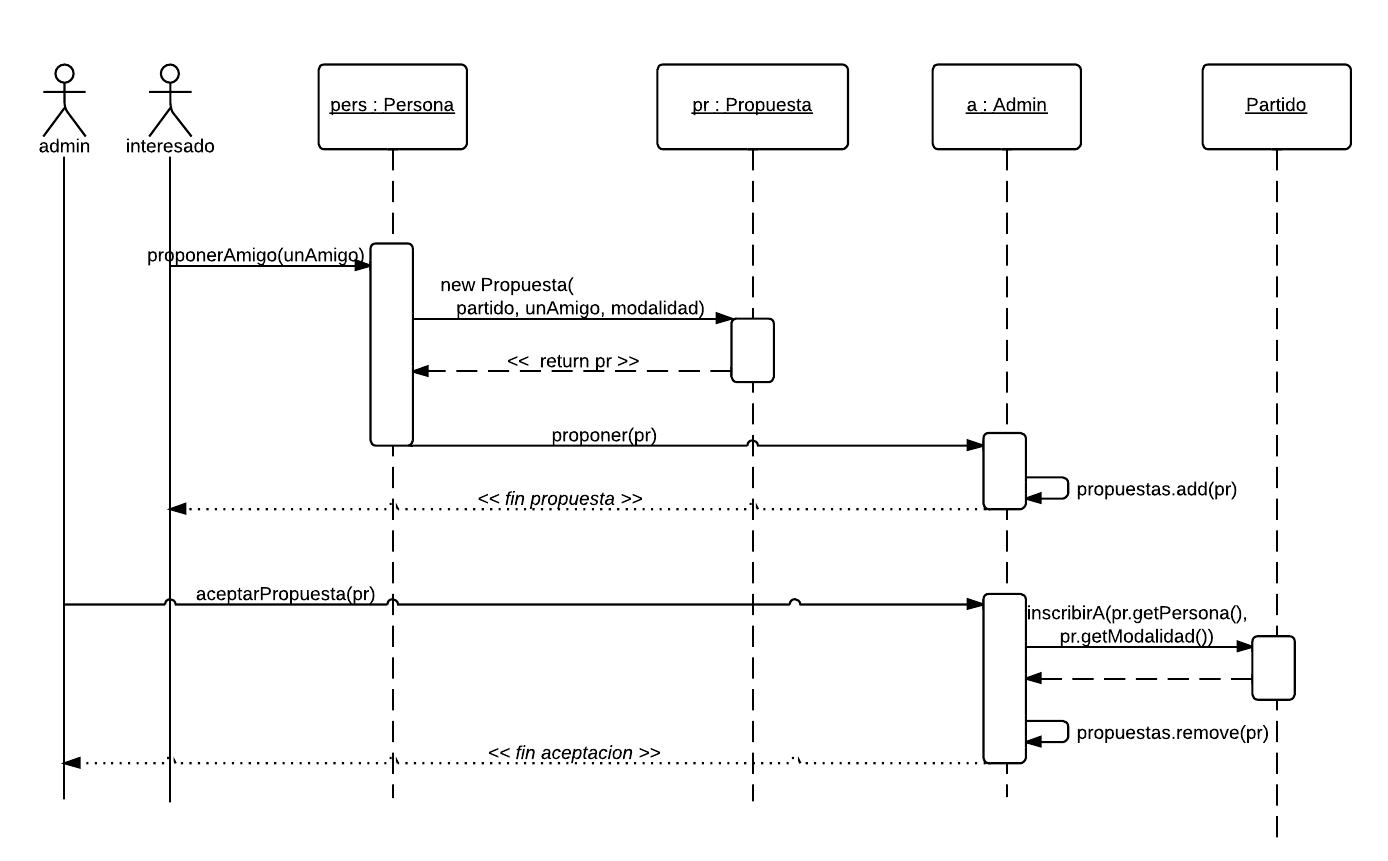
Luego, en un momento T2, pueden ocurrir dos variantes:  
a) El admin selecciona una propuesta sacandola de la lista de las propuestas pendientes, y la acepta. Finalmente, el hecho de aceptarla implica inscribirlo al partido en el que fue propuesto. (CU Aceptar Propuesta)

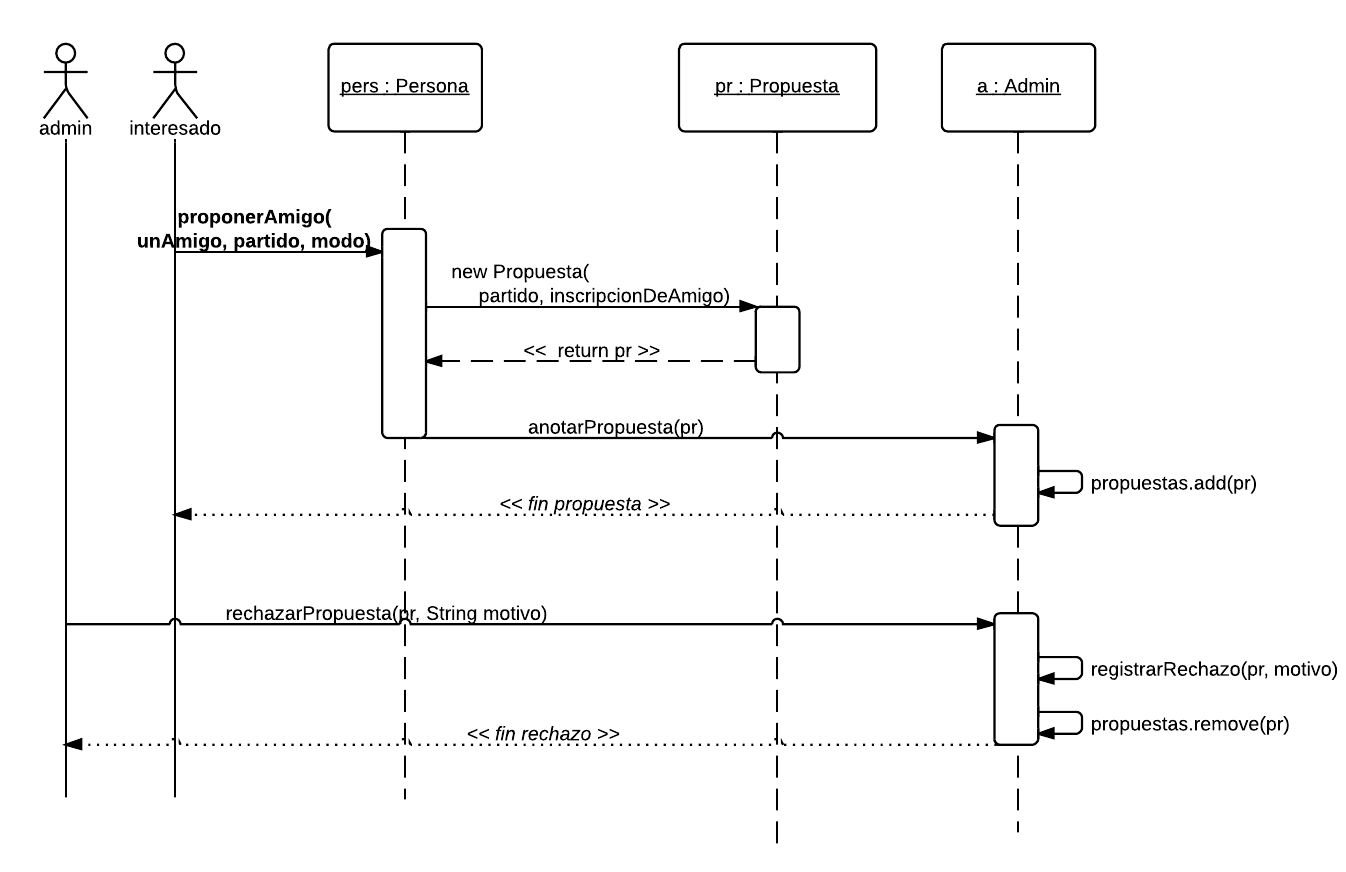
b) Idem a, excepto que la rechaza: en ese caso igualmente se saca de la lista de propuestas, pero se también se registra el motivo de rechazo (y no se intenta inscribir al partido).

Dos ejemplos del paso de mensajes pueden verse en los diagramas de secuencia abajo.

(Nota: iremos mostrando la evolución que fuimos teniendo en el desarrollo, para mostrar el motivo que nos llevó a varias cuestiones que quedaron en la solución final)

Secuencia de propuesta y **aceptacion** (v1.0):



Secuencia de propuesta y **rechazo**:

**Actualización 2:**

Como dijimos, el admin agarra de la propuesta el partido, la persona, y el modo. Y despues lo inscribe, o sea, en *pseudocódigo*:

**Admin** >> aceptarPropuesta(Propuesta unaProp)

{

Partido partido = unaProp.getPartido();

partido.inscribirA(unaProp.getPersona(), unaProp.getModalidad());

/\* hacer otras cosas, como borrar unaProp de lista de propuestas \*/

}

Sin embargo, aca tenemos un code smell, **feature envy**, ya que le estoy pidiendo demasiadas cosas a la propuesta. Aparte, la propuesta en este momento no tiene comportamiento, sólo almacena datos (otro smell, *data class?*). Entonces, por qué no delegarselo al objeto propuesta, y que éste se encargue?

Quedaria:

**Admin** >> aceptarPropuesta(Propuesta unaProp)

{

unaProp.aceptar();

/\* hacer otras cosas, como borrar unaProp de lista de propuestas \*/

}

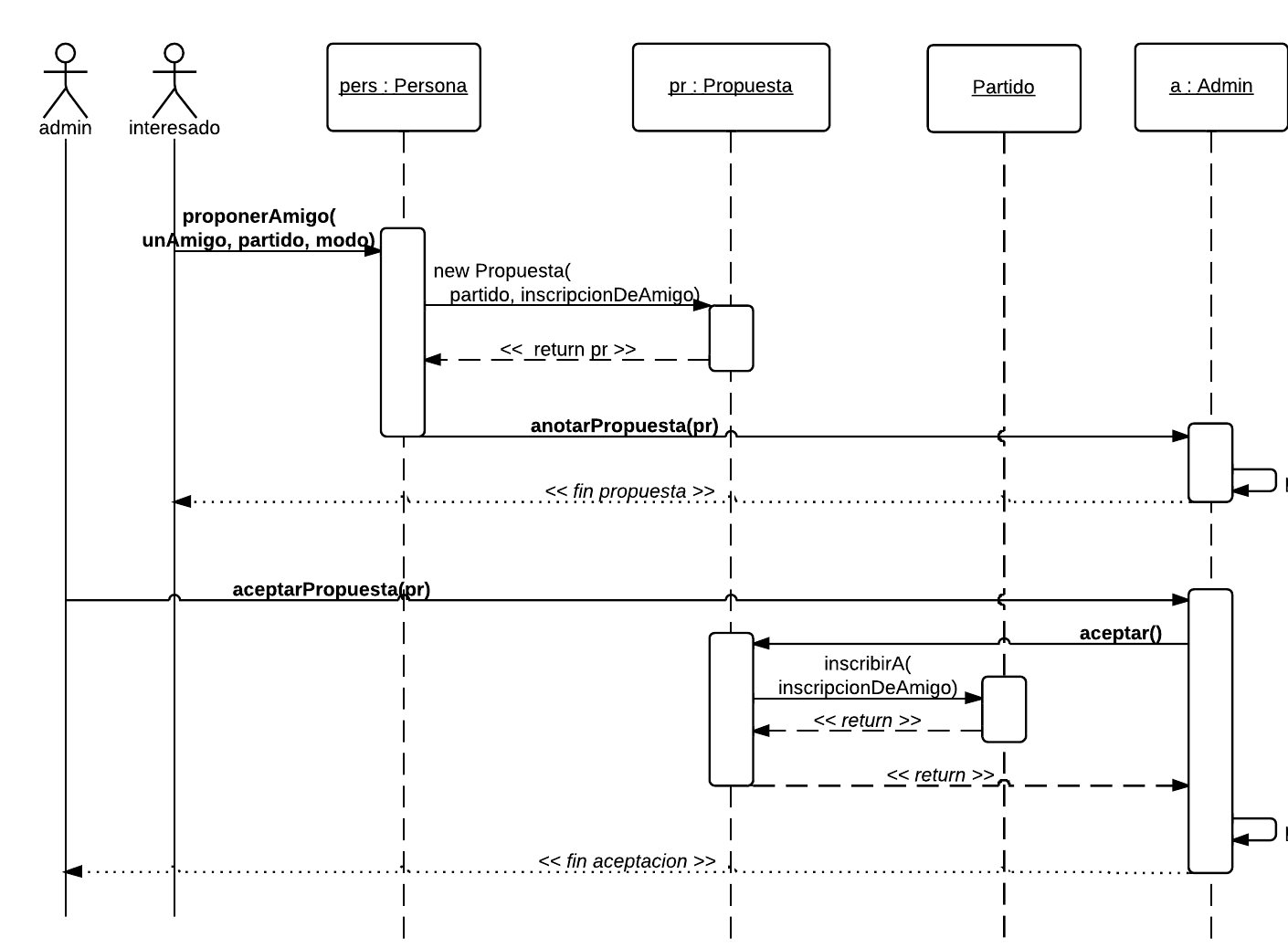
**Propuesta** >> aceptar()

{

partido.inscribirA(persona, modalidad); //partido, persona y modalidad son atributos de Propuesta

}

**Y el diagrama de secuencia v2.0 queda:***(ver abajo)*



Notar cómo el admin ya no interactúa con el partido, ni se encarga de hacer la inscripción, pues todo esto fue delegado a la propuesta. El admin sólo sabe que acepta la propuesta, y la propuesta es quien sabe manejar esa aceptación. En otras palabras, estamos manteniendo **desacoplados** el admin del partido.

**ACTUALIZACION 3:**

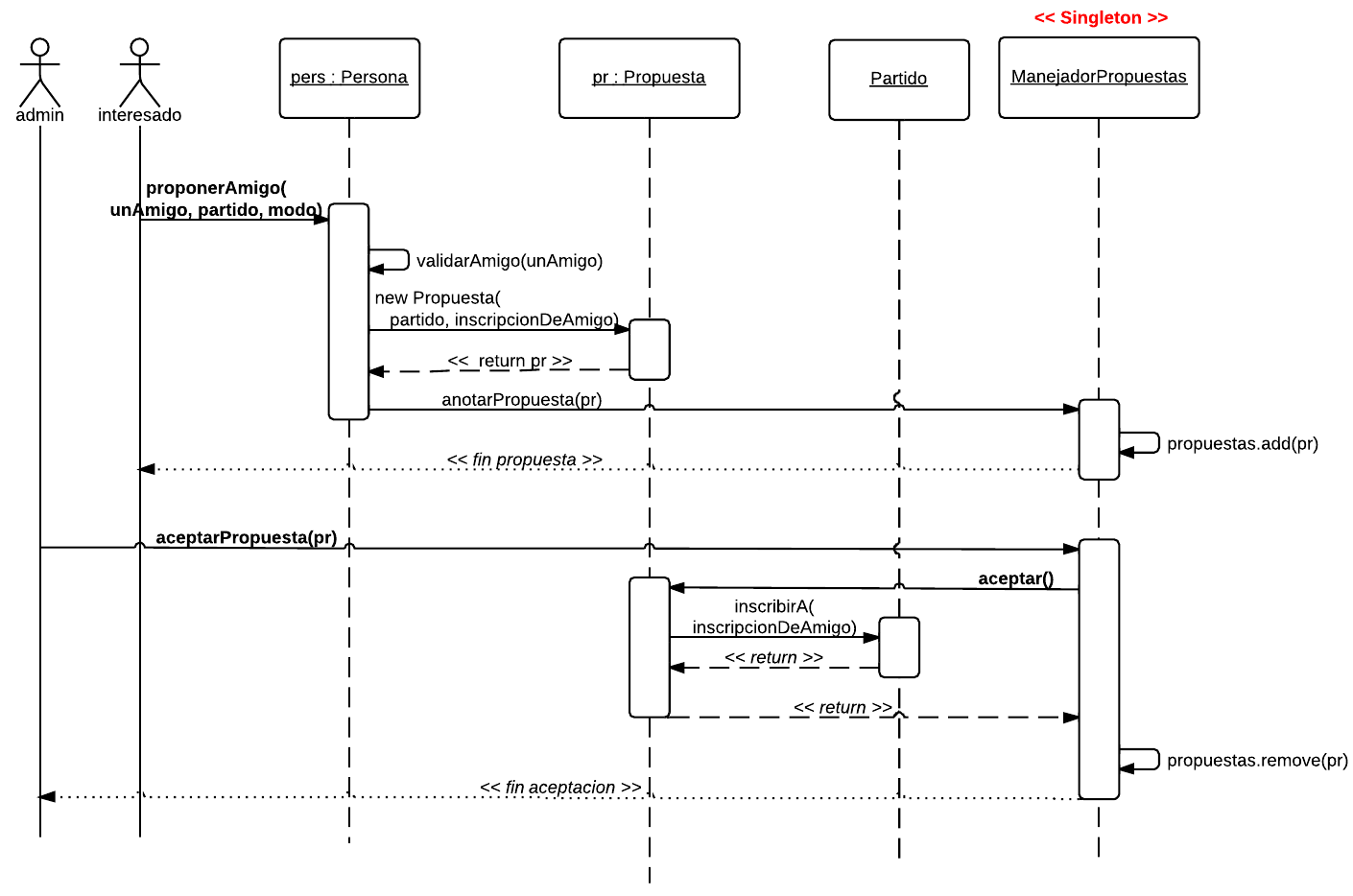
*(parte de la respuesta al item 3 acá)*

Nos dimos cuenta que al hacerlo de la forma planteada en 2.0, la persona tiene que conocer al objeto admin (para poder mandarle el mensaje proponer(pr), que luego llamamos anotarPropuesta(pr)). Esto no es correcto por dos motivos: la persona NO conoce al admin, no tiene por qué hacerlo; y a parte si quisieramos forzar esto el admin no tendría por qué ser unico, es decir cualquier admin debería poder resolver las propuestas.

Entonces, dijimos de pasar la responsabilidad de manejar las propuestas, al partido. Pero nos dimos cuenta que entonces el partido pasaría a ser un God Object (es decir, tendría demasiadas responsabilidades, algunas que no le corresponden necesariamente).

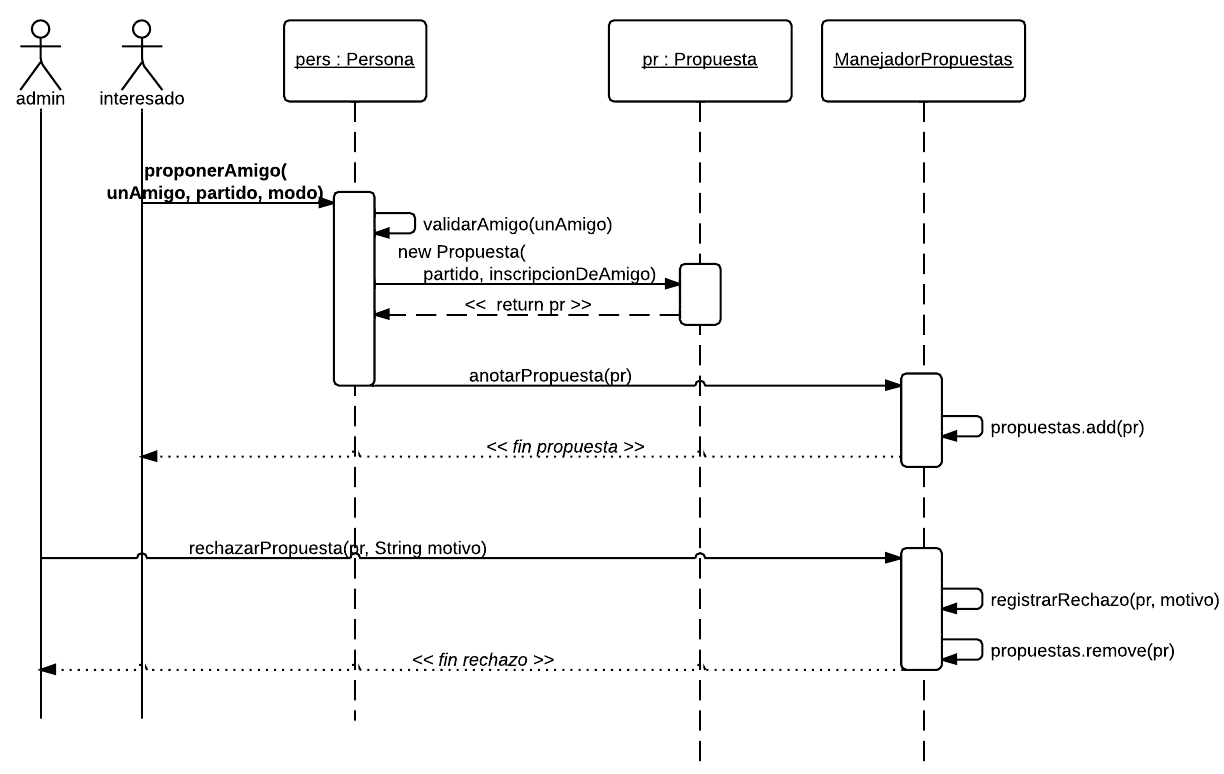
Finalmente, para **desacoplar** este comportamiento, tomamos la decision de crear un Singleton, un objeto ManejadorPropuestas que se encarge de estas tareas. Por ser Singleton, resulta ser global y puede ser accedido desde cualquier punto del sistema. Así, no ensuciamos el código del partido, y no obligamos al objeto admin a ser único ni global.

La secuencia queda para **aceptar (v3.0)**:



**Y para Rechazar (v3.0):**

(ver abajo)



***Notas o consideraciones adicionales:***

*Notamos algunas decisiones que tomamos, respecto a puntos del enunciado que estaban ambiguos o no eran suficientemente claros al respecto.*

- En el caso de que el partido quede confirmado y se juegue, pensamos todas las propuestas que quedaron pendientes para ese partido deberian borrarse. Sin embargo no es algo que pida el enunciado, asique por el momento no lo implementamos.

- Segun el enunciado, una vez que se hace la propuesta, el admin antes de aceptarla, no debería tener los datos del jugador propuesto (nombre, modalidad etc), sino que debería preguntarselas al jugador que hizo la propuesta. Y luego en un tiempo T3, el jugador responde con esos datos; a partir de entonces es que podría ocurrir el tiempo T2 que mencionamos anteriormente (aceptar o rechazar), que en este caso sería T4. Pero para simplificar (ya que agregaría bastante complejidad), simplemente hicimos que la persona que propone ya sepa los datos del amigo y se los pase todos juntos con su propuesta, al admin.

- Decidimos que PropuestaRechazada sea una especie de version decorada de Propuesta:

. Son ambos polimorficos,

. la propuestaRechazada contiene internamente el objeto Propuesta concreto,

. En propuestaRechazada agregamos el “comportamiento” adicional (en este momento es solo el motivo como dato, y la forma en que hacemos un toString(), pero en el futuro podrían haber otras diferencias o comportamiento distinguido).

Sin embargo se diferencia del Decorator, en que: nos interesa componer sólo una PropuestaRechazada con una Propuesta, y sólo ese caso (no rechazadas con rechazadas que tengan una propuesta dentro, por ejemplo).

La ventaja que esto nos aporta es poder tratarlas indistintamente, y sin embargo que cada objeto sepa responder según el tipo que sea, por ejemplo si intentamos aceptar una propuestaRechazada da error, o al pedirle a una Propuesta concreta un motivo este estaría vacío. Otra ventaja: al rechazar una propuesta, el objeto propuesta original no se destruye ni altera, sólo se lo decora, y al final la propuesta concreta no se entera. Es *como si* lo “cambiara” de clase.