Lista de Correo
Eventos ante un post

por
Fernando Dodino

Versión 1.1
Abril 2017

Indice

1 Precondiciones

Asumimos que ya leíste el enunciado original y la propuesta de solución posible. Vamos a trabajar el agregado de funcionalidades cuando ocurre la recepción de un mensaje.

2 El agregado

Recordamos el punto 2 del enunciado:

Nos interesa que haya formas alternativas de recibir mensajes, además del MailSender

- *Por celular, para lo cual hay un singleton PhoneTextSender que entiende un mensaje sendMessage (String telefono, String texto).*
- *Por mensaje grabado, para lo cual hay un singleton PhoneVoiceSender que entiende un mensaje sendMessage (String telefono, String texto).*

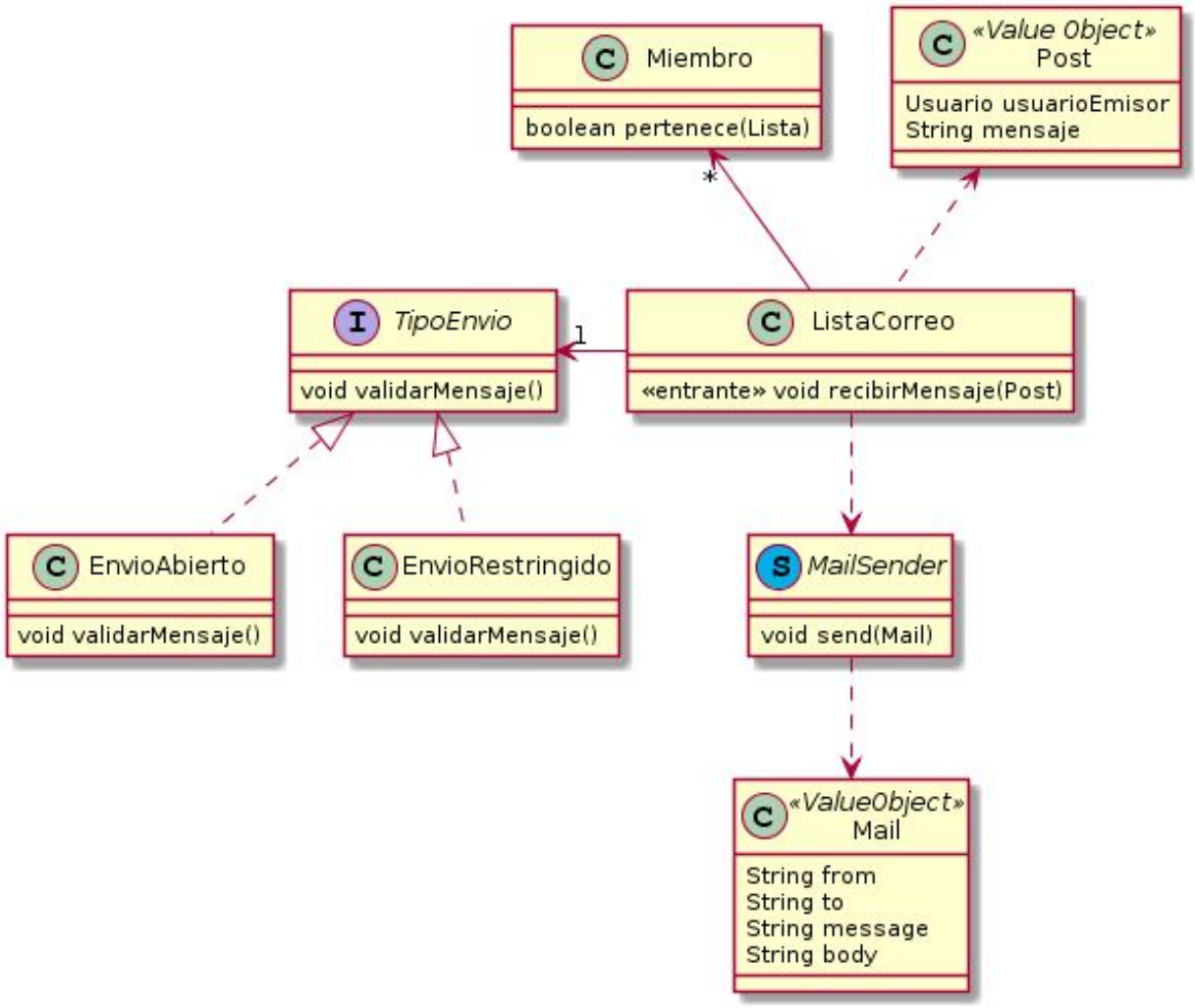
pero además

- *que cada vez que se envíe un mensaje se detecte si en el mensaje hay malas palabras (“hambre”, “guerra”, etc.). En caso afirmativo, hay que enviar un mensaje a los administradores*
- *que cuando un usuario envíe más de 5 mensajes a la lista quede bloqueado hasta tanto el administrador lo vuelva a liberar*

Debe ser fácil incorporar o quitar estas funcionalidades (“a veces quiero auditar malas palabras o usuarios verborrágicos y a veces no”, nos dice el usuario).

3 Estado actual de las cosas

En este ejercicio no diseñamos desde cero sino que comenzamos con un modelo previo del caso de uso *Enviar mensaje*:



Recordamos algunas decisiones:

- Post y Mail modelan dos abstracciones diferentes. Mientras que el Post es un mensaje que un miembro envía a la lista, el Mail es un formato posible de distribución de ese Post. El agregado de mensajes de voz y otras opciones ayuda a separar ambos conceptos.
- El envío de mail lo resuelve la lista de correo. Para ello toma el Post y lo convierte a un objeto Mail, que es lo que entiende el MessageSender / MailSender. En ese sentido funciona como **Adapter**.
- El MailSender no es un objeto, sino que se trabaja con un contrato, una interfaz que debe ser implementada por un objeto. En el diagrama de clases elegimos no graficar las implementaciones posibles, porque implica tomar decisiones por anticipado.

El código asociado es:

```
>>ListaCorreo
def void recibirMensaje(Post post) {
      tipoEnvio.validarEnvio(post, this)
      getMailsDestino(post).forEach [ mailDestino |
          val mail = new Mail => [
             from = post.emisor.mail
             to = mailDestino
             titulo = "[" + encabezado + "] nuevo post"
             message = post.mensaje
          ]
          mailSender.send(mail)
      ]
}
```

4 Cosas en común de los nuevos requerimientos

En los agregados, además de enviar un mail nos piden:

- enviar mensajes de voz
- enviar mensajes a celulares
- informar malas palabras
- detectar usuarios que envían demasiados mensajes.

¿Qué tienen en común todos estos requerimientos? Todos ocurren en un mismo momento: cuando un miembro envía un mensaje.

- Todos los agregados funcionan después del envío del mensaje, no necesitamos validar ni interceptar dicho envío.
- Si el mensaje no es válido (por ejemplo, porque un miembro intenta enviar un mensaje a una lista cerrada sin formar parte de ella) nada debería ocurrir.

5 Soluciones posibles

- Modelar mediante **condicionales** (if) cada funcionalidad. Se incorporaría entonces a la lista una serie de atributos booleanos enviaMensajeDeVoz, enviaMensajeACelular, informaMalasPalabras, etc. + la pregunta que dispara el comportamiento que corresponde

```
>>ListaCorreo
def void enviar(Post post) {
```

```
        tipoEnvio.validarEnvio(post, this)
        if (informaMalasPalabras) {
           ... código asociado a las malas palabras ...
        }
        if (enviaMensajeDeVoz) {
           ... código asociado al envío de mensaje de voz ...
        }
        ...etc...
    }
```

- Encontrar una interfaz común para todos estos requerimientos y trabajarlo mediante **composición**: la lista conocería a un conjunto de objetos polimórficos.
- Como es posible combinar cada uno de los cinco requerimientos, esto descarta de plano utilizar la **herencia** como herramienta (la combinatoria de alternativas lleva inevitablemente a duplicar ideas, y además no podemos activar o inactivar funcionalidades en forma dinámica).

5.1 El evento recibir post y lo que pasa después

Nuestra intención es poder desacoplar:

- lo que hace la lista cuando recibe un post
- y las acciones posteriores a ese evento.

Sobre la lista, que es un objeto de interés, hay una determinada cantidad de objetos interesados. Cuando se recibe el post, la lista avisa a todos sus interesados. Cada interesado se encargará de resolver una funcionalidad específica:

- un objeto puede ser el encargado de enviar un mail, transformando el post en un objeto mail
- otro objeto podrá enviar un mensaje de voz, transformando el post en un objeto con el archivo *.mp3* o uno equivalente para luego transmitirlo
- otro objeto verificará si el mensaje tiene malas palabras para avisar a los administradores, entonces necesita analizar el mensaje del post.

etc.

¿Qué interfaz definimos para todos los objetos “interesados”? En principio

```
def void postRecibido(Post)
```

Vamos a asumir que el Post conoce:

- el emisor del post

- la lista de destino (esto nos va a ayudar por ejemplo para enviar un mail a los administradores cuando haya una mala palabra, o para conocer los destinatarios de un mail)
- el mensaje.

Bajando a detalle, la Lista dispara un evento de notificación:

```
>>ListaCorreo
def void enviar(Post post) {
	tipoEnvio.validarEnvio(post, this)
	postObservers.forEach
		[ postObserver | postObserver.postRecibido(post) ]
}
```

¿Qué hacemos en el envío de mails?

Generamos un objeto que adapta el Post construyendo un Mail.
Entonces:

from => se obtiene del emisor del post

to => puede ser un envío individual de mails o a toda la lista

subject => lo construimos manualmente a partir del encabezado de la lista, en versiones posteriores podríamos incorporarlo al objeto Post, no nos interesa mucho profundizar en este aspecto

message => el mensaje del Post.

Esto ya lo hacíamos antes, pero era algo intrínseco a la lista de correo, ahora está desacoplado y se implementa en uno de los interesados en saber cuando la lista recibe un post, lo llamamos MailObserver.

```
>>MailObserver
override postRecibido(Post post) {
    val lista = post.destino
    val mail = new Mail => [
        from = post.emisor.mail
        // Es fácil cambiar para que envíe los mails de a uno
        to = lista.getMailsDestino(post)
        //
```

```
                titulo = "[" + lista.encabezado + "] nuevo post"
                message = post.mensaje
            ]
            mailSender.send(mail)
        }
```

Estamos separando dos abstracciones que representan cosas diferentes:

1. *MailSender* es una interfaz que permite implementar diferentes formas de enviar mails (utilizando diferentes frameworks como JavaMail, EasyMail, etc. o utilizando distintos protocolos de bajo nivel como SMTP, IMAP, POP3, etc.)
2. *MailObserver* es una interfaz que permite implementar diferentes formas de adaptar el Post a un objeto Mail. Esto incluiría
    a. distintas formas de disparar los envíos a la lista: un único mail con copia a todos los miembros de la lista, un mail individual por cada miembro de la lista, etc.
    b. cambiar el encabezado del asunto del mail

    etc.

En la clase MalasPalabrasObserver conocemos la lista de malas palabras, vemos la implementación para el send:

```
>>MalasPalabrasObserver
override postRecibido(Post post) {
        if (post.tieneMalasPalabras) {
                postConMalasPalabras.add(post)
        }
}
```

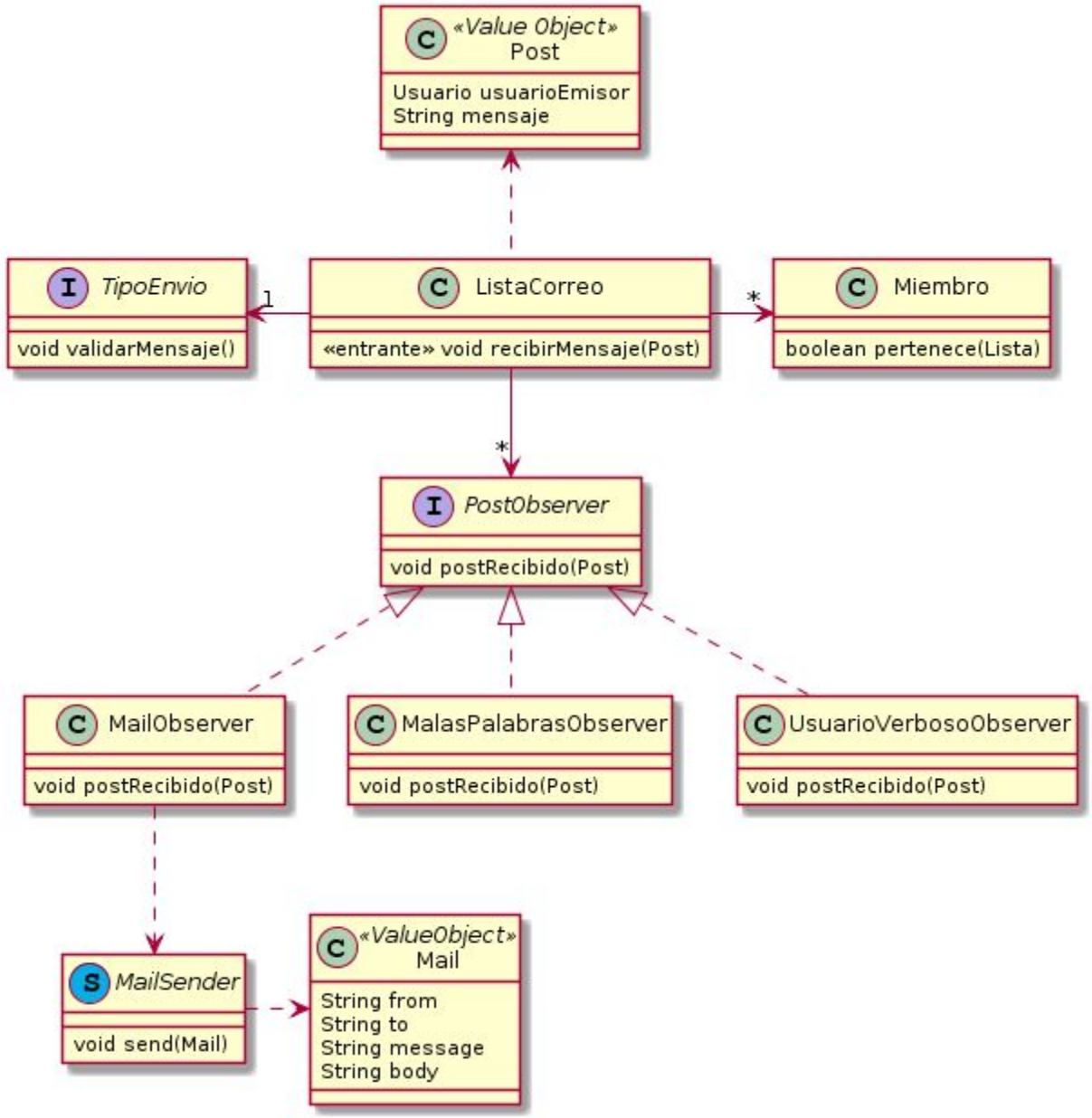
Aquí estamos usando el método tieneMalasPalabras como un extension method, `post.tieneMalasPalabras` es equivalente a hacer `this.tieneMalasPalabras(post)`. Podríamos eventualmente llevar esa responsabilidad al post, pero creemos que es mejor encapsular el conocimiento de las malas palabras en este observer específico. Seguimos:

```
def boolean tieneMalasPalabras(Post post) {
        malasPalabras.exists [ malaPalabra | post.tiene(malaPalabra) ]
}
```

El post sí es el responsable de indicar si está incluyendo una palabra (puede buscar

en el asunto y en el contenido, solo en el contenido, en base a lo que defina el negocio).

Vemos el diagrama de clases de esta solución:



(omitimos las validaciones de envío y de suscripción adrede para visualizar mejor los observadores del post)

5.2 Cómo incorporar dinámicamente cada funcionalidad...

.. a través del mecanismo de observers: Agregando o eliminando cada objeto observer en una colección de interesados: si queremos agregar el control de malas

palabras, debemos hacer lo siguiente:

```
val postQueDigaPodridoObserver = new MalasPalabrasObserver
malasPalabrasObserver.agregarMalaPalabra("podrido")
...
listaAlumnos = Lista.listaAbierta() => [
      agregarPostObserver(new MailObserver(stubMailSender))
      agregarPostObserver(postQueDigaPodridoObserver)
]
```

Para dejar de auditar las malas palabras que escriben los miembros, basta con eliminar al observer de la colección:

```
      listaAlumnos.eliminarPostObserver(postQueDigaPodridoObserver)
```

Recordemos cómo funciona el evento send en la clase Lista:

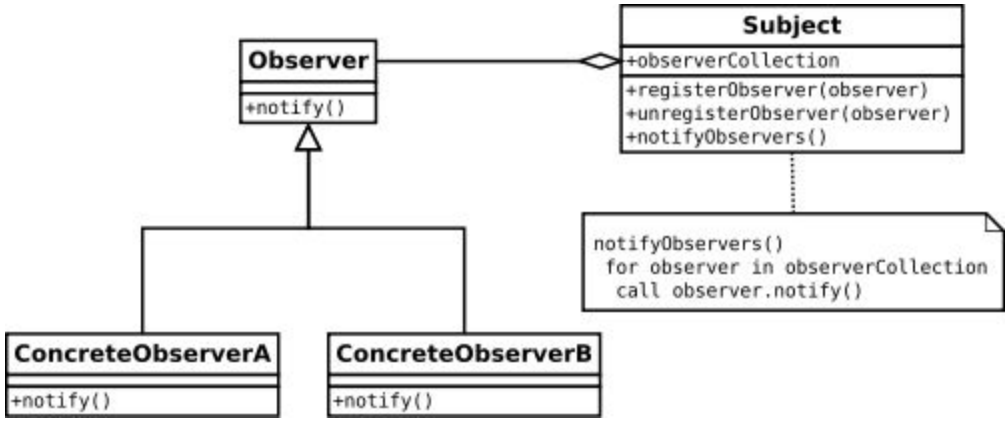
```
      def void enviar(Post post) {
            tipoEnvio.validarEnvio(post, this)
            postObservers.forEach
                  [ postObserver | postObserver.postRecibido(post) ]
      }
```

5.3 El Observer “de libro”

En el libro Design Patterns de Gamma *et al.* [1] se presenta al *Observer* como un patrón que permite definir un objeto observado y *n* interesados u observers, que son notificados ante determinados eventos.

El objeto observado no conoce exactamente las clases concretas de cada interesado, sino que delega en una colección que provee una interfaz común para todos los objetos observadores. Una vez más las herramientas favoritas de los patrones de diseño saltan a la luz: composición (por sobre la herencia) y polimorfismo.

[1] Para más información véase Design Patterns, Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, pág.326



Subject ==> Objeto cuyos determinados eventos son observados (la lista de correo en nuestro contexto)
Observers ==> Objeto/s interesado/s (envío de mails, chequeo de malas palabras, etc.)

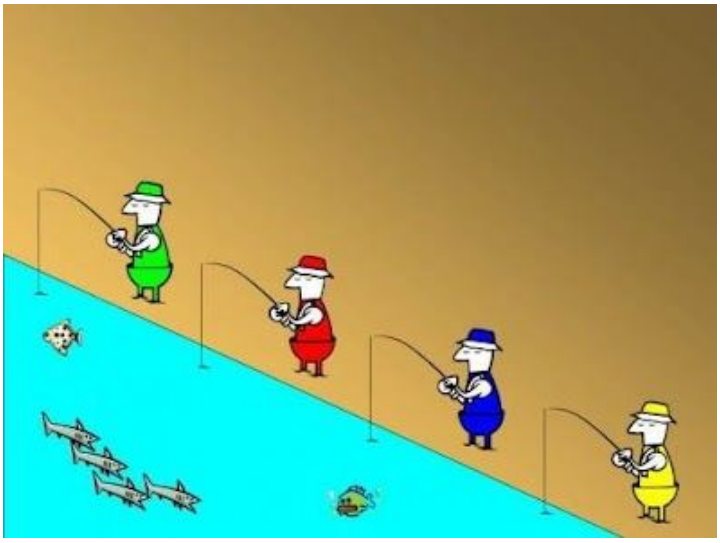
notify() ==> es el postRecibido(Post) en nuestro contexto. Puede haber más de un evento.

- Observador y observado tienen un bajo acoplamiento (determinado por la interfaz del Observer y opcionalmente también del Subject/objeto observado)
- Cada observer concreto maneja su implementación en forma independiente de las demás: esto puede ser tanto una ventaja como una desventaja si necesitamos tener control sobre el algoritmo.
- Si la interfaz del Observer no cambia la solución permite agregar nuevos observadores sin que el objeto observado sufra cambios
- Cada evento puede disparar una notificación distinta a los interesados. Si tenemos una gran cantidad de eventos esto puede representar un trabajo arduo.

5.4 Metáforas asociadas

5.4.1 Pescar

La caña se mueve, la boya se hunde o tengo una campana que suena cuando el pez pica (la campana me notifica el evento “pez mordió el anzuelo”)



5.4.2 Twitter

Cuando elijo seguir a @dodainOk, me suscribo a sus publicaciones[2] (addObserver / registerObserver), cuando él publica un tweet se notifica a todos sus seguidores (notifyObservers):

Cuando dejo de seguir a @dodainOk, se produce un removeObserver/unregisterObserver.

6 Análisis comparativo de soluciones

Solución con condicionales (if)	**Solución con composición (observers)**
Es posible modificar dinámicamente qué funcionalidades activar ante la recepción de un post (seteando true/false cada flag)	También es posible (agregando o eliminando observers).

[2] Otro de los nombres del observer es Publisher-Subscriber

La solución permite trabajar cada funcionalidad en un determinado orden en forma programática. Cambiar ese orden requiere codificar y recompilar. *Ej:* primero chequear malas palabras, luego enviar mensajes de voz, etc.	Como los interesados se guardan en una colección, es posible definir que esa colección sea una lista que respete un cierto orden. Entonces para cambiar el orden de ejecución simplemente se debe modificar el orden de los interesados.
Toda la lógica queda en el objeto Lista, que realiza más cosas y tiene por lo tanto menos cohesión	Como la lista sólo conoce a la interfaz de los objetos interesados, aumenta su cohesión ya que no tiene que manejar la lógica de envío de mails ni lo que hace cada interesado.
Para agregar nuevas funcionalidades debemos modificar la clase Lista.	Para agregar nuevas funcionalidades hay que generar nuevas clases que implementen la interfaz PostObserver. El riesgo está en que necesitemos información que no esté en el parámetro Post que recibimos en el método send, lo que obligaría a modificar la interfaz PostObserver y todas las clases que la implementan, además del objeto Lista.

7 Y para el final

Dejamos un paper revolucionario de Ingo Maier, Tiark Rompf y Martin Odersky, *Deprecating the Observer Pattern*