Sobre los efectos en el observer

o cuánto control tiene un observador sobre el sujeto y los demás observadores

Franco Bulgarelli, Gisela Decuzzi, Javier Fernándes
Versión 1.0
Mayo 2013

1 Introducción

Una discusión interesante que Gamma no da al hablar sobre el patrón Observer es sobre los efectos que los observadores producen ante la recepción de eventos.

Para abrir el juego, Hamlet toma un observer, y mirándolo fijo, nos interpela:

"Modificar o no modificar, esa es la cuestión"

¿En que está pensando? Podemos hacer algunos análisis:

- ¿Podemos vetar los eventos?
- ¿Podemos modificar al sujeto?
- ¿Podemos cambiar la forma de reaccionar dinámicamente?

2 Vetar eventos

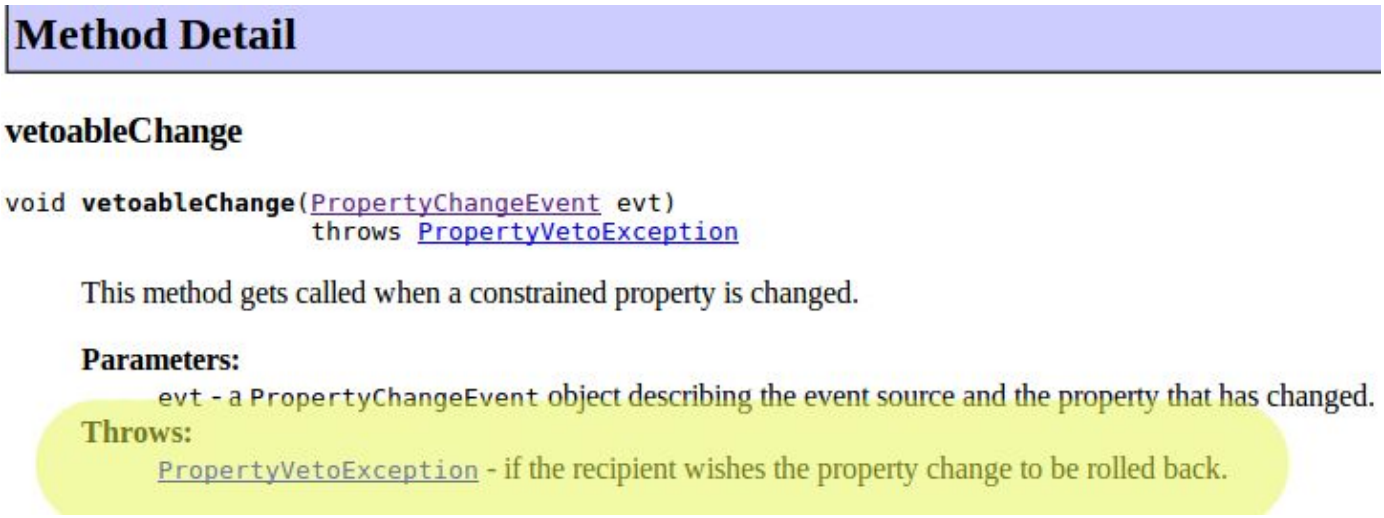
El observador, ante un evento ¿puede modificar la fuente del evento para que éste resulte anulado y no se propague a los otros observadores? Dicho en otras palabras: ¿puede el observador vetar al evento?

En el observer de libro, esto no ocurre: el observer es un componente "pasivo" que no tiene control al respecto. Sin embargo, existen variantes que soportan esta

noción.

Por ejemplo, veamos el API Java de Vetoable: VetoableChangeListener

Nótese que esta interfaz se ve como la de un observador común y corriente. Sin embargo, si miramos con cuidado veremos que hay algo diferente en la firma de vetoableChange (que es nuestro “recibirNotificacion”):



Method Detail

vetoableChange

```
void vetoableChange(PropertyChangeEvent evt)
                    throws PropertyVetoException
```

This method gets called when a constrained property is changed.

Parameters:

`evt` - a `PropertyChangeEvent` object describing the event source and the property that has changed.

Throws:

`PropertyVetoException` - if the recipient wishes the property change to be rolled back.

Es decir, para señalar que el evento debe ser vetado, lo que estos observers deben hacer es lanzar una excepción. Esta excepción no indicará un fallo, sino que será utilizada como forma de comunicación entre el observador y el observado. Y el observado deberá manejar esta excepción para que no se propague, y deje de notificar a sus observadores del evento en cuestión.

Entonces, ahora vemos que el orden de registración de los observadores resulta importante, dado que los primeros ahora tienen el control de veto de eventos.

Y como consecuencia, surge un cierto **acoplamiento implícito** entre los observadores, dado que los observadores registrados antes son conscientes de la existencia potencial de otros registrados después.

Un cambio en uno de los primeros que haga que la excepción no sea lanzada, provocará que otro observador se ejecute, con lo que ya no son totalmente independientes el uno del otro. Y para peor, este acoplamiento no es tan fácil de detectar a partir de la interfaz de los componentes.

El problema del acoplamiento implícito será aún más evidente en la próxima sección.

3 Modificar al sujeto

El observador ¿puede afectar al (sujeto) observado? Acá hay un gris. Gamma tampoco habla de esto, y deja abierta la posibilidad a que esto ocurra. Y ciertamente, es posible y válido.

Sin embargo, acá hay que tratar este tipo de observadores con cuidado. No por nada filosófico, sino por cuestiones de consistencia: dado que estamos en lenguajes sincrónicos, probablemente mientras el observador esté modificando al sujeto, el mismo estará aún dentro de un bloque de estilo:

```
observadores.forEach [
   observador | observador.notificar(evento)
]
```

con lo que, aun ejecutando en un solo hilo, puede introducir inconsistencias y notificar cosas incorrectas (lo cual es peor que notificar cosas desactualizadas).

Por ejemplo, supongamos el siguiente problema: tenemos guerreros, que pueden recibir daño. Si el guerrero, como consecuencia del daño recibido, queda moribundo, notificará a sus observadores.

```
class Guerrero
  int energia
  List<GuerreroObserver> observers

  def void recibirDanio() {
    ...
    if (this.estaMoribundo()) {
       notificarMoribundo()
    }
    ...
  }

  def estaMoribundo() {
```

```
    energia <= 10
  }

  def void sanar() {
    ...subirle la energia mucho mucho ...
  }

  def void notificarMoribundo() {
    observers.forEach [ observer | observer.notificarMoribundo(this) ]
  }
}
```

Uno de sus observadores es una enfermera, que intentará curarlo sacándolo del estado moribundo

```
class Enfermera implements GuerreroObserver {
  def void notificarMoribundo(Guerrero guerrero) {
    guerrero.sanar()
  }
}
```

Y otro observador será su aseguradora de vida, que quiere empezar a hacer papeles en caso de que el guerrero esté próximo a su defunción.

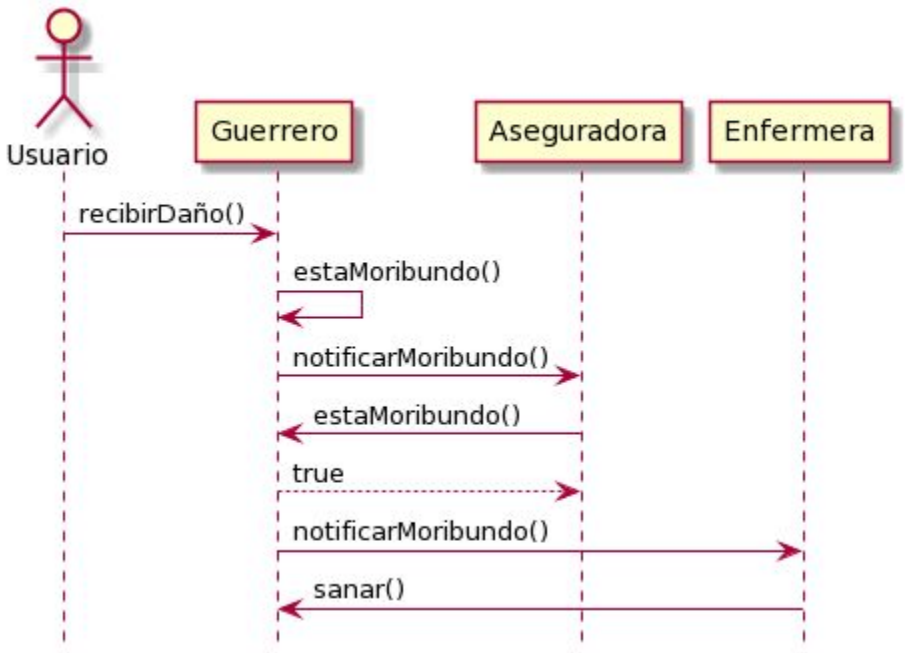
```
class AseguradoraDeVida implements GuerreroObserver {
  def void notificarMoribundo(Guerrero guerrero) {
    empezarElPapelerio(guerrero)
  }
}
```

Dado que la aseguradora no quiere gastar dinero de más, lanzará una excepción si se intenta hacer los papeles para un guerrero que no está moribundo

```
class AseguradoraDeVida implements GuerreroObserver {
  def void empezarElPapelerio(Guerrero guerrero) {
    if (!guerrero.estaMoribundo()) {
      throw new BusinessException("no se puede empezar el papelerio si
esta sano!!")
    }
```

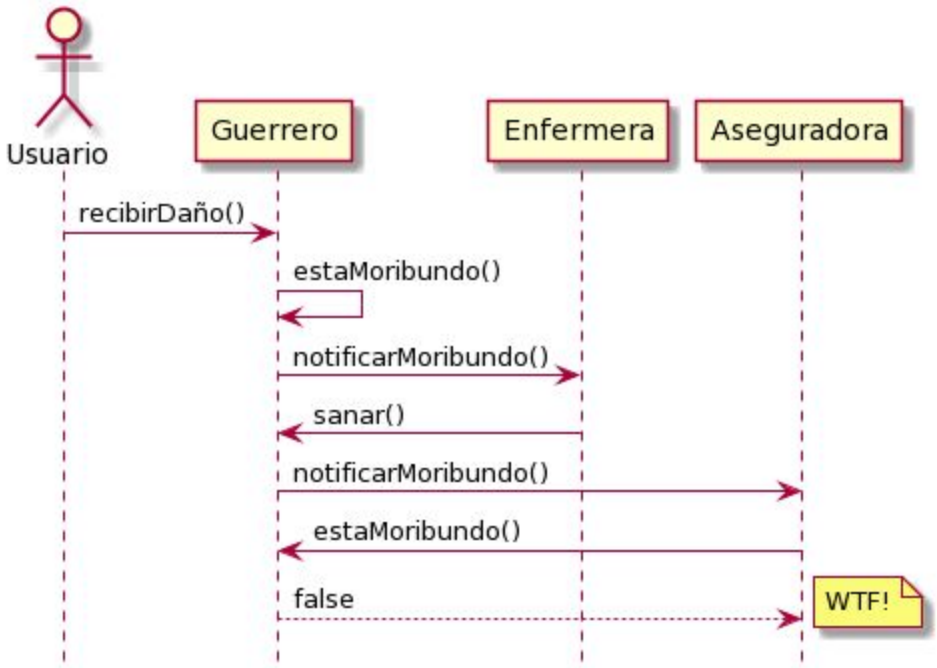
```
    ...
  }
}
```

Lo probamos con un ejemplo: tenemos un guerrero que tiene una enfermera y aseguradora de vida observando su convalecencia.



Esto funciona. El guerrero primero avisa a la aseguradora, que empieza el papelerío. Luego, le avisa a la enfermera, quien lo cura, haciendo que el guerrero deje de estar moribundo.

Ahora supongamos que registramos a los observadores en orden inverso, y la enfermera resulta notificada primero, sanando al guerrero.



Entonces, se dará un extraño "WTF?[1]": la aseguradora de vida recibe una notificación de que está moribundo, pero como la enfermera fue notificada primero, y sanó al guerrero, la aseguradora va a fallar.

Éste no es un problema fácil de arreglar. Sin embargo es, de hecho, bastante común en sistemas reales, como por ejemplo al utilizar la API de Android.

Obvio, no siempre ocurre eso, y si lo que modifico no está asociado con la generación de eventos, o todos los objetos evento son independientes del contexto y son variables locales, ahí seguramente no rompa nada.

El libro de Gamma sólo tira el warning *"make sure subject state is self consistent before notifications"*, con lo que se deja abierta la puerta a la duda.

4 Alterando dinámicamente la forma de reaccionar

A veces sucede que un Observer al recibir el evento, por su "dominio" o lógica tiene que desregistrarse, porque ya no le interesa más observar. Esto podría explotar, porque el observable recorre la lista de observers y va llamando uno por uno.

[1] Expresión del inglés que denota asombro

Si uno de esos le llama al método "remove" consigo mismo de la lista, por restricciones de las colecciones de muchos lenguajes, esto puede fallar. Por ejemplo, en Java, cuando el iterador quiera pedir un elemento más, va a lanzar ConcurrentModificationException. Es decir, si alguien modifica la lista mientras la estamos recorriendo, va a explotar.

La fácil para solucionar esto, es que siempre el observado al notificar un evento tiene que hacerlo "sobre una copia de la lista de observers y no sobre la original".

Quedaría:

```
def void notificarMoribundo() {
  observadores.clone().forEach
       [ observer | notificarMoribundo(this) ]
}
```

El mensaje clone() puede llamarse copy() o un nombre similar.

5 Algunas conclusiones

Como se ve, el patrón observer es poderoso, pero en ciertos casos, cuando la forma de reaccionar es más compleja e involucra efectos, vetos o reacciones dinámicas, etc, nos puede quedar corto.

Es por eso que existen detractores de este patrón y proponen otro superador: Reactor Pattern (que obviamente, escapa al alcance de la materia)