CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

Patrones de Diseño

20. *Observer*

Luis Fernández Muñoz

https://www.linkedin.com/in/luisfernandezmunyoz

setillofm@gmail.com

INDICE

1. Problema

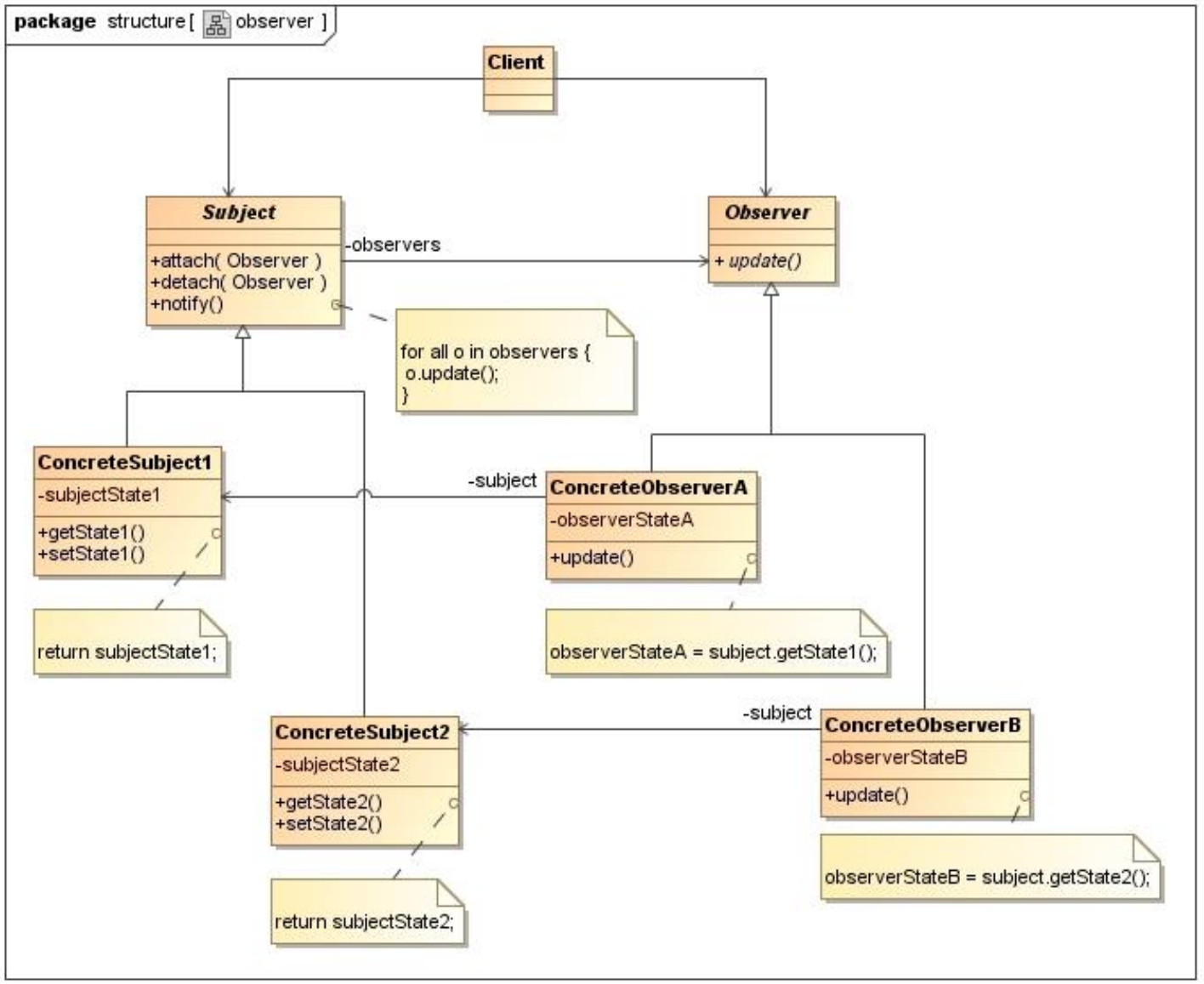
- *Cuando en una sucursal bancaria se produce un atraco hay que avisar de inmediato a los cuerpos de seguridad públicos y privados particulares del banco para reforzar a los primeros.*
- *Una alternativa es dotar a los empleados de los mecanismos particulares para avisar a cada cuerpo de seguridad (p.e. números de teléfono o páginas web o …). Lo cual obliga a gestionar esos mecanismos cada vez que haya un cambio (p.e. nuevos cuerpos, …)*
- *Una alternativa más eficiente sería pulsar un botón que dispara automáticamente el aviso a todos los cuerpos registrados en el automatismo. Esto independiza a los empleados de intervenir en la gestión de esos mecanismos, solo pulsa y ya se avisará quien esté registrado*

- *Define una dependencia uno-a-muchos entre objetos de tal forma que cuando un objeto cambia su estado, todos los dependientes son notificados y modificados automáticamente*

1. Problema

- El suscriptor sería el observador y el publicador sería el observado (o sujeto).
- Un efecto lateral habitual al partir un sistema en una colección de clases cooperantes es la necesidad de mantener la consistencia entre los objetos relacionados. Pero no se quiere mantener la consistencia haciendo las clases altamente acopladas porque reduce su reusabilidad
- Aplíquese en las siguientes condiciones:
  - Cuando una abstracción tiene dos aspectos, donde una depende de otra. Encapsula estos aspectos con objetos separados que permite variar y reusarlos independientemente.
  - Cuando un cambio sobre un objeto requiere cambiar otros y no conoces cuántos necesitan ser cambiados.
  - Cuando un objeto debería ser capaz de notificar a otros objetos sin hacer asunciones sobre quiénes son estos objetos. En otras palabras, no se quiere estos objetos altamente acoplados

2. Solución


package structure [observer]
Client
Subject
+attach(Observer)
+detach(Observer)
+notify()
-observers
Observer
+ update()
for all o in observers {
o.update();
}
ConcreteSubject1
-subjectState1
+getState1()
+setState1()
return subjectState1;
-subject
ConcreteObserverA
-observerStateA
+update()
observerStateA = subject.getState1();
ConcreteSubject2
-subjectState2
+getState2()
+setState2()
return subjectState2;
-subject
ConcreteObserverB
-observerStateB
+update()
observerStateB = subject.getState2();

2. Solución

- El observado puede tener cualquier número de observadores dependientes. Todos los observadores son notificados si el observado sufre un cambio de estado. En respuesta, cada observador requerirá al observado para sincronizar su estado con el estado del observado.
- Se envían notificaciones sin tener conocimiento de quienes son su observadores. Cualquier número de observadores puede suscribirse
- La forma más simple par que un observado realice el seguimiento de los observadores a los que debería notificar sería almacenar explícitamente las referencias a éstos. Sin embargo esto puede ser muy caro cuando hay muchos observados y pocos observadores. Una solución es un compromiso espacio/tiempo usando un array asociativo que mantiene la correspondencia entre observados a observadores. Por otro lado, este enfoque incrementa el coste de acceso

2. Solución

- Puede tener sentido en alguna situación que un observador dependa de más de un observado. Entonces, es necesario extender el interfaz de *Update* en tal caso para permitir al observador conocer qué observado está enviando la notificación. El observado puede simplemente enviarse a sí mismo como parametro de la operación *Update* permitiendo al observador conocer a qué observado examinar

2. Solución

- ¿Qué objeto llama a *Notify* para disparar las modificaciones?:
  - Tener operaciones de establecimiento del estado sobre el observado que llaman a *Notify* después de que cambie el estado del observado. La ventaja de este enfoque es que los clientes no tienen que recordar llamar *Notify* sobre el observado. La desventaja es que varias operaciones consecutivas pueden causar varios cambios consecutivos, que pueden ser ineficientes.
  - Hacer a los clientes responsables de llamar a *Notify* en el momento adecuado. La ventaja aquí es que el cliente puede esperar para activar la actualización hasta que se haya realizado una serie de cambios de estado, evitando de este modo actualizaciones intermedias innecesarias. La desventaja es que los clientes tienen una responsabilidad añadida para activar la actualización. Esto hace que los errores sean más probables, ya que los clientes pueden olvidar llamar *Notify* .

2. Solución

- La eliminación de un observado no debería producir referencias "colagadas" de sus observadores. Una forma de evitar referencias "colagadas" es hacer que el observado notifique a sus observadores, cuando se elimina de modo que puedan reinicializar la referencia a él. En general, borrar los observadores no es una opción, ya que otros objetos pueden hacer referencia a ellos, o se pueden observar otros observados también.
- Es importante asegurar que el estado del observado esta auto-consistente antes de llamar a *Notify* porque los observadores preguntarán al observado por su estado actual en el curso de modificar su propio estado. Esta regla de autoconsistencia es fácil de violar no intencionadamente cuando subclases del observado llaman a operaciones heredadas. Se puede evitar este fallo enviando las notificaciones desde métodos plantilla

2. Solución

- A menudo el observado transmite información adicional acerca del cambio. El observado pasa esta información como un argumento para la actualización. La cantidad de información puede variar ampliamente.
  - En un extremo, lo que llamamos el modelo de inserción (*push*), el observado envía información detallada a los observadores sobre el cambio, lo quieran o no. En el otro extremo, se encuentra el modelo de extracción (*pull*), el observado envía nada más que la notificación mínima, y los observadores preguntan por los detalles de forma explícita a partir de ahí.
  - El modelo de extracción hace hincapié en la ignorancia del observado de sus observadores, mientras que el modelo de inserción asume que los observados saben algo acerca de las necesidades de sus observadores.

2. Solución

- Cuando la relación de dependencia entre los observados y los observadores es particularmente compleja, un objeto que mantiene estas relaciones podría ser necesario, llamado *ChangeManager*. Su objetivo es reducir al mínimo el trabajo necesario para hacer que los observadores reflejen un cambio en sus observados. Por ejemplo, si una operación involucra cambios en varios observados interdependientes, puede que tenga que asegurarse de que sus observadores son notificados después de que todos los observados se han modificado para evitar la notificación a los observadores más de una vez. *ChangeManager* tiene tres responsabilidades:
  - Relaciona un observado a sus observadores y proporciona una interfaz para mantener esta relación. Esto elimina la necesidad de que los observados mantengan las referencias a sus observadores y viceversa.
  - Define una estrategia de actualización en particular

2. Solución

- Se puede mejorar la eficiencia en la actualización mediante la ampliación de interfaz de registro del observado permitiendo que los observadores se registran sólo para eventos específicos de interés. Cuando tal evento se produce, el observado informa sólo aquellos observadores que se han registrado interés en ese evento. Una forma de apoyar esto es utilizar la noción de los aspectos de los objetos observado.
  - *void Subject::Attach(Observer*, Aspect& interest);*
  - *void Observer::Update(Subject*, Aspect& interest);*

3. Consecuencias

- Permite variar independientemente observados y observadores. Se pueden reusar observados sin reusar sus observadores y viceversa. Permite añadir observadores sin modificar los observados u otros observadores
- Todo observado conoce que tiene una lista de observadores, conforme a la sencilla interfaz de la clase abstracta *Observer*. El observado no conoce la clase concreta de cualquier observador. El acoplamiento entre observados y observadores es abstracto y mínimo. Pueden pertenecer a capas diferentes de abstracción del sistema
  - Si se agrupan, el objeto resultante debe abarcar dos capas (y violar la arquitectura de capas) o se debe forzar a convivir en una capa u otra (lo que podría comprometer las abstracciones de las capas)

3. Consecuencias

- Debido a que los observadores no tienen conocimiento de la presencia de otros, pueden no ver el coste final de cambiar el observado. Una operación aparentemente inocua sobre el observado puede provocar una cascada de cambios en los observadores y sus objetos dependientes. Por otra parte, los criterios de dependencia que no están bien definidos o mantenidos por lo general conducen a cambios espurios, que puede ser difícil de localizar.
  - Este problema se ve agravado por el hecho de que un protocolo simple de actualización no proporciona ningún detalle sobre lo que ha cambiado en el observado. Sin un protocolo adicional para ayudar a los observadores a descubrir lo que cambió, pueden verse obligados a trabajar duro para deducir los cambios.

3. Consecuencias

- La notificación que envía el observado no necesita especificar sus receptores. La notificación es multidifundida (*broadcast*) automáticamente a todos los objetos que estén suscritos. El observado no se preocupa de cuántos objetos interesados existen; su responsabilidad solo es notificar a sus observadores. Esto da la libertad de añadir y borrar observadores en cualquier momento. El observador ya verá si maneja o ignora la notificación

4. Comparativa

- ***Observer*** vs ***Mediator***
  - Ver *Mediator* vs *Observer*