Investigación Invocación Implícita

Observer

Este patrón de diseño permite reaccionar a ciertas clases llamadas observadores sobre un evento determinado.

Es usado en programación para monitorear el estado de un objeto en un programa. Está relacionado con el principio de invocación implícita. La motivación principal de este patrón es su utilización como un sistema de detección de eventos en tiempo de ejecución. Es una característica muy interesante en términos del desarrollo de aplicaciones en tiempo real.

Debe ser utilizado cuando:

Un objeto necesita notificar a otros objetos cuando cambia su estado. La idea es encapsular estos aspectos en objetos diferentes permite variarlos y reutilizarlos independientemente.

Cuando existe una relación de dependencia de uno a muchos que puede requerir que un objeto notifique a múltiples objetos que dependen de él cuando cambia su estado.

Este patrón tiene un uso muy concreto: varios objetos necesitan ser notificados de un evento y cada uno de ellos deciden como reaccionar cuando esta evento se produzca.

Un caso típico es la Bolsa de Comercio, donde se trabaja con las acciones de las empresas. Imaginemos que muchas empresas estan monitoreando las acciones una empresa X. Posiblemente si estas acciones bajan, algunas personas esten interesadas en vender acciones, otras en comprar, otras quizas no hagan nada y la empresa X quizas tome alguna decisión por su cuenta. Todos reaccionan distinto ante el mismo evento. Esta es la idea de este patrón y son estos casos donde debe ser utilizado.

En Java este patrón podemos implementarlo usando una clase, la clase Observable, y una interfaz, la interfaz Observer proporcionadas en el propio JDK. La clase que queremos que reciba los eventos deberá implementar la interfaz Observer y el objeto que queremos que produzca los eventos debe extender o contener una propiedad de tipo Observable. La interfaz Observer contiene un único método de nombre update, que recibe dos parámetros que son la instancia del objeto observable sobre la que se ha producido el evento y un Object a modo de argumento que el objeto observable envía. La clase Observable contiene métodos para añadir y eliminar observadores que queremos que sean notificados, obtener un contador con el número de observadores y unos métodos para conocer y establecer si un objeto ha cambiado con el método hasChanged.

El patrón Observer proporciona algunas propiedades de los sistemas que se comunican mediante mensajes:

No hay que estar monitorizando un objeto en búsqueda de cambios, se hace la notificación cuando un objeto cambia.

Permite agregar nuevos observadores para proporcionar otro tipo de funcionalidad sin cambiar el objeto observador.

Bajo acoplamiento entre observable y observador, un cambio en el observable u observador no afecta al otro.

Esta funcionalidad es la ofrecida en el propio JDK, los sistemas que permiten comunicar las aplicaciones usando mensajes como RabbitMQ o JMS proporcionan estas características deseables además de otras propiedades como funcionamiento asíncrono.

Implementar el patrón no es muy complicado basta con extender de una clase (Observable) e implementar una interfaz (Observer), sin embargo, tener que extender de la clase Observable puede ser un problema si la clase que queremos observar ya extiende de otra clase, esto es así porque en Java no hay herencia múltiple de varias clases, si es posible implementar varias interfaces. ¿Que podemos en este caso? La solución es usar composición en vez de herencia aunque con composición no tendremos acceso a los métodos clearChanged ni setChanged si nos son necesarios.

Supongamos que tenemos una clase Producto con un precio y queremos emitir un mensaje en la terminal cuando un producto cambie de precio. Para implementar este patrón Producto debería extender de Observable y otra clase hacer que implemente la interfaz Observer, sin embargo, si no queremos o no podemos hacer que nuestra clase extienda de Observable para no limitarnos en nuestra jerarquía de clases o porque ya extiende de otra podemos usar composición, por otro lado, si no queremos registrar el observador en cada instancia de Producto sino observar cualquier instancia que se cree podemos implementar el Observer de forma estática en la clase Producto. El observable ProductoObservable amplia la visibilidad del método setChanged para poder hacer uso de él usando composición, deberemos invocarlo para que los observadores sean notificados.