Patron de diseño: Observer

Esta herramienta ofrece la posibilidad de definir una dependencia uno a uno entre dos o más objetos para transmitir todos los cambios de un objeto concreto de la forma más sencilla y rápida posible. Para conseguirlo, puede registrarse en un objeto (observado) cualquier otro objeto, que funcionará como observador. El primer objeto, también llamado sujeto, informa a los observadores registrados cada vez que es modificado.

El patrón Observer trabaja con dos tipos de actores: por un lado, el sujeto, es decir, el objeto cuyo estado quiere vigilarse a largo plazo. Por otro lado, están los objetos observadores, que han de ser informados de cualquier cambio en el sujeto.

Sin el patrón Observer, los objetos observadores tendrían que solicitar al sujeto regularmente que les enviase actualizaciones acerca de su estado (status updates). Cada una de estas solicitudes conllevaría tiempo de computación y requeriría, además, ciertos recursos de hardware. El patrón Observer se basa en la idea de centralizar la tarea de informar en manos del sujeto. Para conseguirlo, existe una lista en la que los observadores pueden registrarse. En caso de modificación, el sujeto los informa uno tras otro, sin necesidad de que los observadores lo pidan activamente. Si, más adelante, un observador ya no necesita las actualizaciones automáticas, puede simplemente retirarse de la lista.

Ejemplo de la vida real

Imaginemos que tenemos dos tipos de objetos: un objeto Cliente y un objeto Tienda. El cliente está muy interesado en una marca particular de producto (digamos, un nuevo modelo de: celular, computador, reloj inteligente, etc…) que estará disponible en la tienda muy pronto.

El cliente puede visitar la tienda cada día para comprobar la disponibilidad del producto. Pero, mientras el producto está en camino, la mayoría de estos viajes serán en vano.

Por otro lado, la tienda podría enviar cientos de correos (lo cual se podría considerar spam) a todos los clientes cada vez que hay un nuevo producto disponible. Esto ahorraría a los clientes los interminables viajes a la tienda, pero, al mismo tiempo, molestaría a otros clientes que no están interesados en los nuevos productos.

Código

```
// La clase notificadora base incluye código de gestión de
// suscripciones y métodos de notificación.
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        listeners.remove(eventType, listener)

    method notify(eventType, data) is
        foreach (listener in listeners.of(eventType)) do
            listener.update(data)

// El notificador concreto contiene lógica de negocio real, de
// interés para algunos suscriptores. Podemos derivar esta clase
// de la notificadora base, pero esto no siempre es posible en
// el mundo real porque puede que la notificadora concreta sea
// ya una subclase. En este caso, puedes modificar la lógica de
// la suscripción con composición, como hicimos aquí.
class Editor is
    public field events: EventManager
    private field file: File

    constructor Editor() is
        events = new EventManager()

    // Los métodos de la lógica de negocio pueden notificar los
    // cambios a los suscriptores.
    method openFile(path) is
        this.file = new File(path)
        events.notify("open", file.name)

    method saveFile() is
        file.write()
        events.notify("save", file.name)

    // ...
```

```
interface EventListener is
    method update(filename)

// Los suscriptores concretos reaccionan a las actualizaciones
// emitidas por el notificador al que están unidos.
class LoggingListener implements EventListener is
    private field log: File
    private field message

    constructor LoggingListener(log_filename, message) is
        this.log = new File(log_filename)
        this.message = message

    method update(filename) is
        log.write(replace('%s',filename,message))

class EmailAlertsListener implements EventListener is
    private field email: string

    constructor EmailAlertsListener(email, message) is
        this.email = email
        this.message = message

    method update(filename) is
        system.email(email, replace('%s',filename,message))


// Una  aplicación puede configurar notificadores y suscriptores
// durante el tiempo de ejecución.
class Application is
    method config() is
        editor = new Editor()

        logger = new LoggingListener(
            "/path/to/log.txt",
            "Someone has opened the file: %s")
        editor.events.subscribe("open", logger)

        emailAlerts = new EmailAlertsListener(
            "admin@example.com",
            "Someone has changed the file: %s")
        editor.events.subscribe("save", emailAlerts)
```

Patron de diseño: Visitor

Es un patrón de comportamiento, que permite definir una operación sobre objetos de una jerarquía de clases sin modificar las clases sobre las que opera. Representa una operación que se realiza sobre los elementos que conforman la estructura de un objeto.

Puesto que la estructura de los objetos está compuesta de muchas clases inconexas y requiere cada vez más operaciones, para los desarrolladores es muy inconveniente implementar una nueva subclase para cada nueva operación. El resultado es un sistema plagado con varias clases de nodos diferentes que no solo es difícil de entender, sino también de mantener y modificar. La instancia esencial del visitor pattern es el Visitor, que permite añadir nuevas funciones virtuales a una familia de clases sin modificarlas.

El visitor pattern establece que el objeto Visitor se define por separado y tiene el fin de implementar una operación que se realiza en uno o más elementos de la estructura del objeto. Los clientes que acceden a la estructura del objeto llaman entonces a la operación de envió accept(visitor) en el elemento en cuestión, lo que delega la petición en el objeto visitante aceptado. Así, el objeto Visitor puede realizar la operación necesaria.

Ejemplo de la vida real

Imaginemos un experimentado agente de seguros que está deseoso de conseguir nuevos clientes. Puede visitar todos los edificios de un barrio, intentando vender seguros a todo aquel que se va encontrando. Dependiendo del tipo de organización que ocupe el edificio, puede ofrecer pólizas de seguro especializadas:

Si es un edificio residencial, vende seguros médicos.

Si es un banco, vende seguros contra robos.

Si es una cafetería, vende seguros contra incendios e inundaciones.

Utilizando el patrón, el agente de seguros puede saber previamente a qué entidad se dirigirá y saber a qué se dedica. De acuerdo a lo anterior sabrá cómo abordar a las personas para que la venta tenga más posibilidades.

Código

```
// La interfaz elemento declara un método `accept` (aceptar) que
// toma la interfaz visitante base como argumento.
interface Shape is
    method move(x, y)
    method draw()
    method accept(v: Visitor)

// Cada clase de elemento concreto debe implementar el método
// `accept` de tal manera que invoque el método del visitante
// que corresponde a la clase del elemento.
class Dot implements Shape is
    // ...

    // Observa que invocamos `visitDot`, que coincide con el
    // nombre de la clase actual. De esta forma, hacemos saber
    // al visitante la clase del elemento con el que trabaja.
    method accept(v: Visitor) is
        v.visitDot(this)

class Circle implements Shape is
    // ...
    method accept(v: Visitor) is
        v.visitCircle(this)

class Rectangle implements Shape is
    // ...
    method accept(v: Visitor) is
        v.visitRectangle(this)

class CompoundShape implements Shape is
    // ...
    method accept(v: Visitor) is
        v.visitCompoundShape(this)
```

```
// La interfaz Visitor declara un grupo de métodos de visita que
// se corresponden con clases de elemento. La firma de un método
// de visita permite al visitante identificar la clase exacta
// del elemento con el que trata.
interface Visitor is
    method visitDot(d: Dot)
    method visitCircle(c: Circle)
    method visitRectangle(r: Rectangle)
    method visitCompoundShape(cs: CompoundShape)

// Los visitantes concretos implementan varias versiones del
// mismo algoritmo, que puede funcionar con todas las clases de
// elementos concretos.
//
// Puedes disfrutar de la mayor ventaja del patrón Visitor si lo
// utilizas con una estructura compleja de objetos, como un
// árbol Composite. En este caso, puede ser de ayuda almacenar
// algún estado intermedio del algoritmo mientras ejecutas los
// métodos del visitante sobre varios objetos de la estructura.
class XMLExportVisitor implements Visitor is
    method visitDot(d: Dot) is
        // Exporta la ID del punto (dot) y centra las
        // coordenadas.

    method visitCircle(c: Circle) is
        // Exporta la ID del círculo y centra las coordenadas y
        // el radio.

    method visitRectangle(r: Rectangle) is
        // Exporta la ID del rectángulo, las coordenadas de
        // arriba a la izquierda, la anchura y la altura.

    method visitCompoundShape(cs: CompoundShape) is
        // Exporta la ID de la forma, así como la lista de las
        // ID de sus hijos.
```

```
// El código cliente puede ejecutar operaciones del visitante
// sobre cualquier grupo de elementos sin conocer sus clases
// concretas. La operación `accept` dirige una llamada a la
// operación adecuada del objeto visitante.
class Application is
    field allShapes: array of Shapes

    method export() is
        exportVisitor = new XMLExportVisitor()

        foreach (shape in allShapes) do
            shape.accept(exportVisitor)
```