Carrera: Licenciatura en Sistemas Materia: Orientación a Objetos II Año: 2018

Equipo docente:
Titular: Prof. María Alejandra Vranić alejandravranic@gmail.com
Ayudantes: Prof. Leandro Ríos leandro.rios.unla@gmail.com
Prof. Gustavo Siciliano gussiciliano@gmail.com
Prof. Romina Mansilla romina.e.mansilla@gmail.com

Patrones de diseño

Patrón Observer (Observador):

Este patrón define una dependencia de uno a muchos entre objetos de manera que cuando uno cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.

La idea es definir un objeto que es el "guardián" del modelo de datos y/o lógica de negocio (el sujeto) y delegar la funcionalidad a los objetos de observadores. Los observadores se registran a sí mismos con el sujeto. Siempre que el sujeto cambia, se difunde a todos los observadores registrados que ha cambiado, y cada observador consulta el Asunto de ese subconjunto de estados.

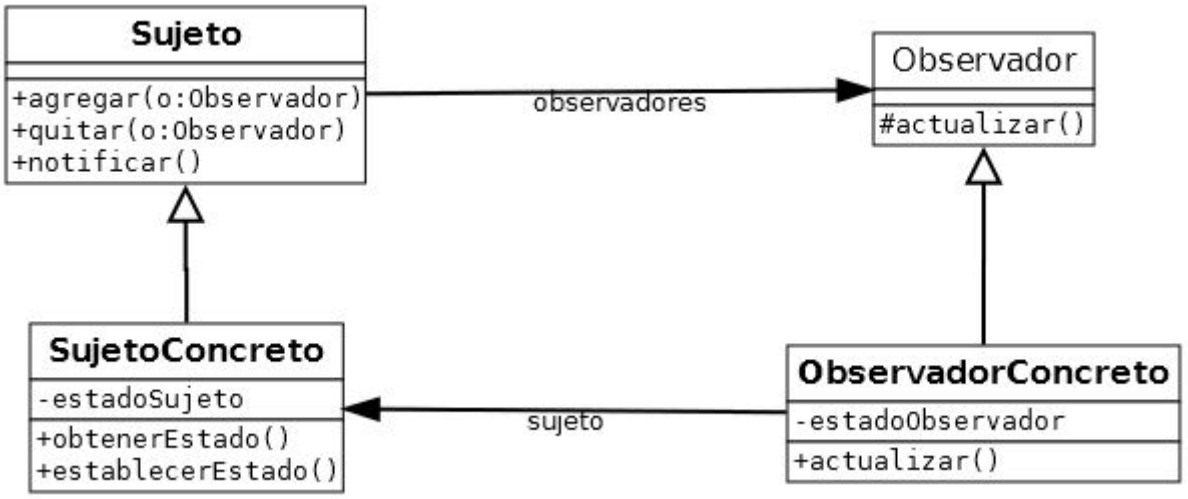
Cuándo utilizar este patrón:

Si una abstracción tiene dos aspectos y uno depende del otro. El encapsulamiento de estos aspectos en objetos separados permite la reutilización en forma independiente.

Si un cambio en un objeto implica cambiar otros y desconocemos cuántos objetos son necesarios cambiar.

Cuando queremos reducir el acoplamiento entre estos objetos.

Estructura:



Este patrón tiene los siguientes componentes:

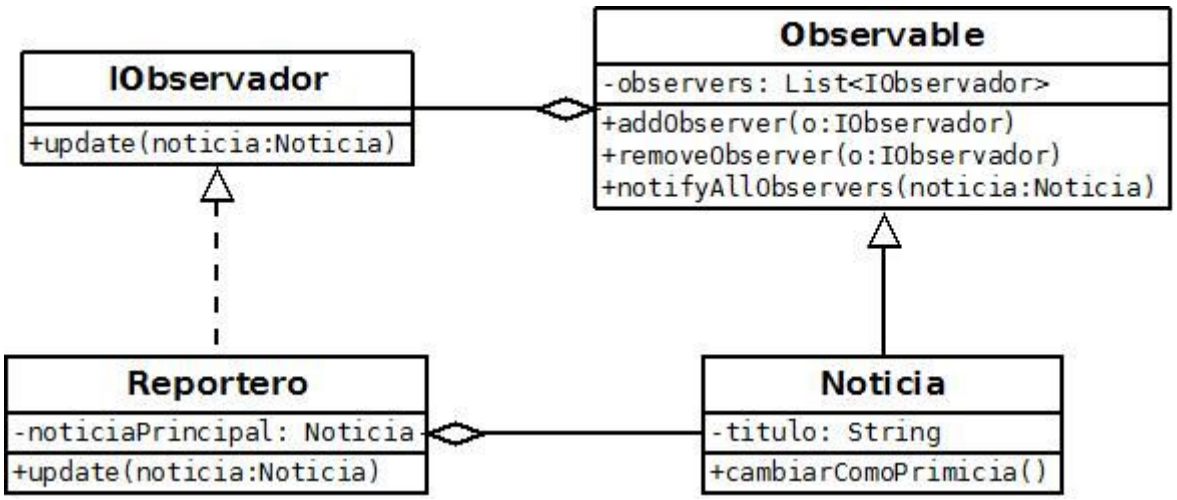
1. Observable (Sujeto): Esta clase se conoce también como Sujeto y conoce todos sus observadores. Clase abstracta que define las operaciones para agregar y sacar observadores al cliente.
2. Observador: Interfaz o clase abstracta que define las operaciones que se usan para notificar cambios en su estado.
3. Sujeto Concreto: Almacena el estado que es de interés para los objetos Observador concreto. Se encarga de enviar una notificación a sus observadores si cambia su estado.
4. Observador concreto: Mantiene una referencia al sujeto concreto e implementa la interfaz de actualización, es decir, guardan la referencia del objeto que observan, así en caso de ser notificados de algún cambio, pueden preguntar sobre este cambio.

Conclusiones:

- Diseñar el patrón Observador permite actualizar en forma independiente los sujetos y observadores.
- Permite agregar observadores sin modificar el sujeto u otros observadores existentes.

Ejemplo:

Supongamos que tenemos que desarrollar un sistema en cual tengamos reporteros y noticias, cada reportero se debe inicializar con una noticia, la cual va a ser su primicia.



Código

```
package observer;

import modelo.Noticia;

public interface IObservador {

	public void update(Noticia noticia);
}
```

```
package observer;

import java.util.List;

import modelo.Noticia;

public class Observable {

	List<IObservador> observers;

	public List<IObservador> getObservers() {
		return observers;
	}

	public void setObservers(List<IObservador> observers) {
		this.observers = observers;
	}

	public void addObserver(IObservador observer){
		this.getObservers().add(observer);
	}

	public void removeObserver(IObservador observer){
		this.getObservers().remove(observer);
	}

	public void notifyAllObservers(Noticia noticia){
		for(IObservador each : this.getObservers())
		{
			each.update(noticia);
		}
	}
}
```

```java
package modelo;

import java.util.ArrayList;
import observer.IObservador;
import observer.Observable;

public class Noticia extends Observable{

	private String titulo;

	public Noticia(String titulo) {
		this.titulo = titulo;
		this.setObservers(new ArrayList<IObservador>());
	}

	public void cambiarComoPrimicia(){
		this.notifyAllObservers(this);
	}

	public String getTitulo() {
		return titulo;
	}
	public void setTitulo(String titulo) {
		this.titulo = titulo;
	}
}
```

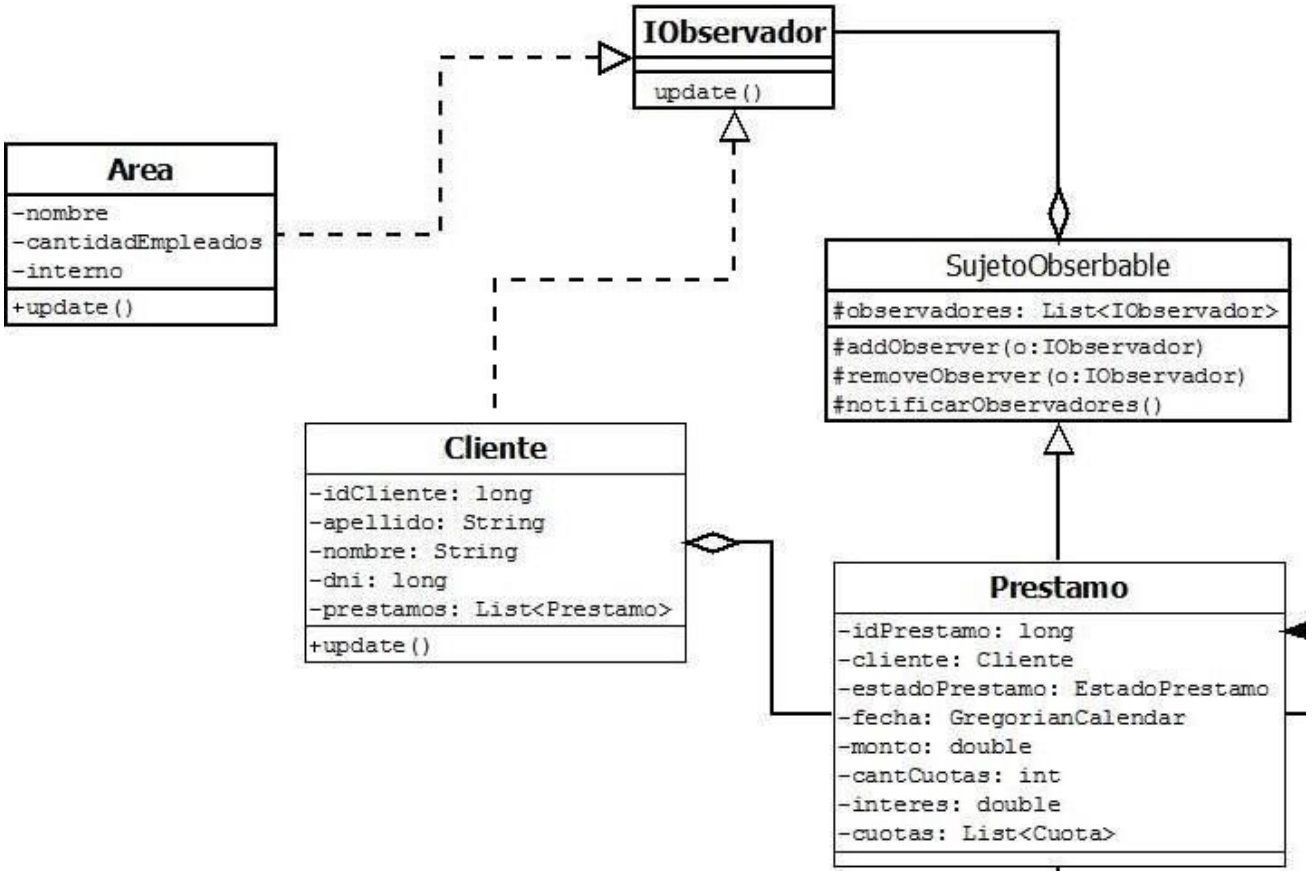
```java
package modelo;

import observer.IObservador;

public class Reportero implements IObservador{

	private Noticia noticiaPrincipal;

	public Reportero(Noticia noticiaPrincipal) {
		this.noticiaPrincipal = noticiaPrincipal;
	}

	@Override
	public void update(Noticia noticia) {
		this.setNoticiaPrincipal(noticia);
	}

	public Noticia getNoticiaPrincipal() {
		return noticiaPrincipal;
	}
	public void setNoticiaPrincipal(Noticia noticiaPrincipal) {
		this.noticiaPrincipal = noticiaPrincipal;
	}
}
```

```java
package test;

import modelo.Noticia;
import modelo.Reportero;

public class Main {

	public static void main(String[] args) {

		Noticia noticia1 = new Noticia("Estrenos de la semana en el cine");
		Noticia noticia2 = new Noticia("Entrevista");
		Noticia noticia3 = new Noticia("Pronóstico extendido");
		Reportero reportero1 = new Reportero(noticia1);
		Reportero reportero2 = new Reportero(noticia2);
		Reportero reportero3 = new Reportero(noticia3);

		System.out.println(reportero1.getNoticiaPrincipal().getTitulo());
		System.out.println(reportero2.getNoticiaPrincipal().getTitulo());
		System.out.println(reportero3.getNoticiaPrincipal().getTitulo());

		Noticia noticia4 = new Noticia("Están explicando Observer Pattern en
la clase de OO2");
		noticia4.addObserver(reportero2);
		noticia4.addObserver(reportero3);

		System.out.println("");
		noticia4.cambiarComoPrimicia(); //Arrancó la clase!

		System.out.println(reportero1.getNoticiaPrincipal().getTitulo());
		System.out.println(reportero2.getNoticiaPrincipal().getTitulo());
		System.out.println(reportero3.getNoticiaPrincipal().getTitulo());
	}

}
```

Ejemplo con el sistema francés:

Se nos solicita que cada vez que un préstamo cambia de estado se solicite al cliente y a la/s área/s correspondiente/s. El caso más visto sucede cuando un préstamo cae en estado de deuda se le debe enviar un mail al cliente para notificarle la situación y además al área de deudores del banco. También se puede avisar al área de créditos en caso de solicitar un préstamo, o al área de ejecutivos y gerencial al momento de rechazarlo. En todo caso podemos ver que hay varios objetos que están atentos al estado de préstamo y que accionan de manera diferente cuando este cambia de estado. Como podemos ver se puede aplicar el patrón Observer.

Volviendo al caso de la notificación cuando un préstamo cae como deudor podemos modificar el diagrama para aplicar y patrón y queda de la siguiente manera:



Preguntas:

1- ¿Porque la lista observadores de SujetoObservable puede contener instancia de Area y Cliente?
2- ¿El patrón observer permite añadir observadores sin modificar al sujeto u otros observadores?

Referencias utilizadas:

Título: Patrones de Diseño

Autores: Erich Gamma - Richard Helm - Ralph Johnson -Johon Vlissides

Editorial: Pearson Addison Wesley

SourceMaking https://sourcemaking.com/design_patterns