Patrón Observer,
Fork Join + Recursive Action

TPO1 - Grupo 3

- Trobbiani Perales Donato
- Gallucci Gianfranco Nicolás

Observer.

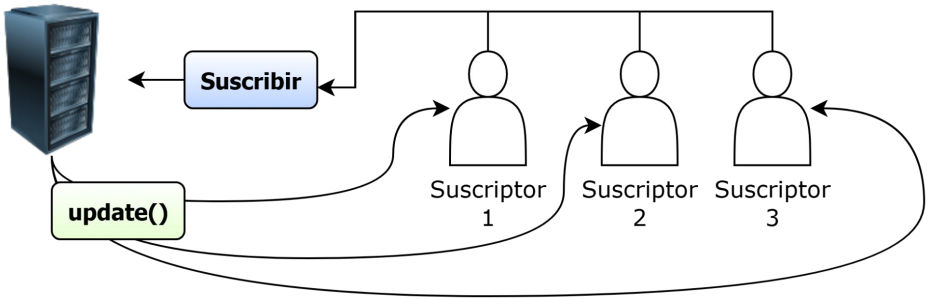
Patrón de diseño de comportamiento que permite definir una relación uno-a-muchos entre objetos.

Incorpora un **mecanismo de suscripción** para objetos interesados (observadores); estos podrán suscribirse a un objeto (sujeto) de manera que serán notificados cuando este cambie su estado.

Componentes.

Subject (Notificador/Publisher/Observable):

- Mantiene una **colección de observadores.**
- Mantiene su propio **estado** interno.
- Ofrece métodos para **añadir y remover observadores.**
- Cuando **cambia de estado**, **notifica** a todos sus **observadores** (notifySuscribers()).



Observer (Observador/Suscriptor):

- Define **interfaz** suscriptora con método **update()**.
- **Cada clase** interesada debe **implementar** (Concrete Subscriber) esta **interfaz** con su propia reacción específica a **update()**.
- Es posible incorporar parámetros en update() para añadir contexto junto a la actualización.

Ventajas y desventajas.

Ventajas

- ✔ Desacoplamiento.
- ✔ Principio abierto-cerrado.
- ✔ Escalabilidad.
- ✔ Establecer relaciones en ejecución.
- ✔ Consistencia.
- ✔ Reutilización.

Desventajas

- ✘ Sobrecarga con muchos observadores.
- ✘ Complejidad en sistemas grandes.
- ✘ Aleatoriedad en actualizaciones.
- ✘ Posible inconsistencia temporal.
- ✘ Dependencias ocultas.

Observer en código.

```java
public class BaseCentral {
    // BaseCentral (Publisher) -> IntegranteBase (Subscriber)
    private final List<IntegranteBase> colonias = new ArrayList<>();
    private Evento eventoActual;

    public void addSuscriber(IntegranteBase nuevaColonia) {
        colonias.add(nuevaColonia);
    }

    public void removeSuscriber(IntegranteBase colonia) {
        colonias.remove(colonia);
    }

    public void recibirEvento(Evento evento) {
        eventoActual = evento;
        notifySuscribers();
    }

    public void notifySuscribers() {
        System.out.println("¡Nuevo evento detectado! " + eventoActual);
        System.out.println("Notificando a " + colonias.size() + " colonias...");
        for (IntegranteBase colonia : colonias) {
            colonia.update(eventoActual);
        }
    }
}
```

```java
public interface IntegranteBase {
    void update(Evento nuevoEvento);
}
```

```java
public class ColoniaCientifica implements IntegranteBase {
    // ColoniaCientifica (suiscriber) - reacciona solo al evento DESCUBRIMIENTO
    @Override
    public void update(Evento nuevoEvento) {
        if (nuevoEvento == Evento.DESCUBRIMIENTO) {
            System.out.println(x:"¡La colonia científica festeja un nuevo descubrimiento!");
        }
    }
}
```

Fork Join + Recursive Action.

Fork Join: Framework y modelo de programación para **ejecutar tareas en paralelo** mediante el enfoque *divide y vencerás,* aprovechando todos los núcleos disponibles de la CPU.

Se basa en:

1. **Dividir** un problema en partes más pequeñas.
2. **Resolver** los problemas en paralelo.
3. **Combinar** todos los resultados uniendolos al final.

RecursiveAction: Clase abstracta del Fork/Join en Java que **representa tareas recursivas sin valor de retorno**, útiles cuando se busca realizar acciones sobre los datos.

Implementación.

Clases

ForkJoinPool: Pool de hilos especializados en ejecutar tareas Fork/Join. Cada hilo mantiene su propia cola de tareas y pueden robar tareas de otras colas (work-stealing) si termina la suya para lograr un buen balance de carga.

RecursiveAction: Subclase abstracta de ForkJoinTask que se usa cuando la tarea no devuelve resultado. Implementa el método **compute()** que divide el trabajo en subtareas y las resuelve.

Métodos principales

Fork(): Envía la tarea al pool para resolverla en paralelo, sin esperar su finalización.

Join(): Espera que la tarea asociada termine, en el caso de RecursiveAction, no devuelve un resultado.

Compute(): Define la lógica de la resolución de la **tarea** en RecursiveAction.

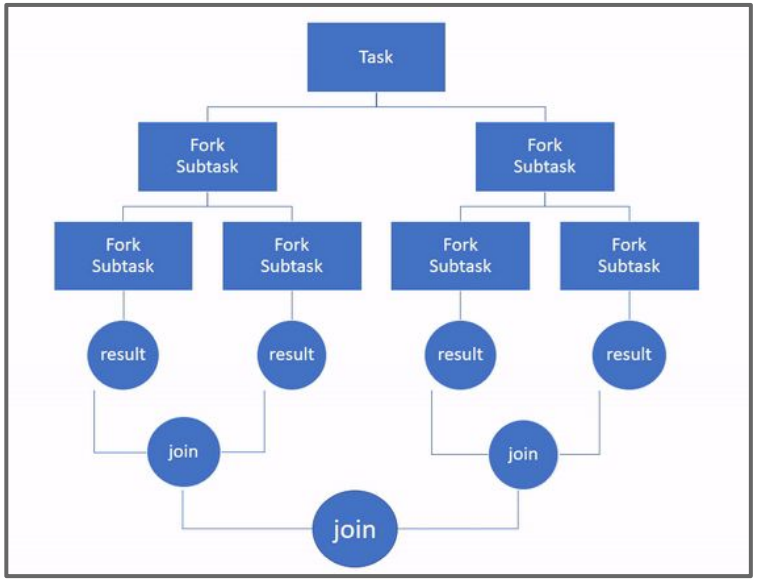
- **Si el problema es pequeño,** lo resuelve.
- **Si el problema es grande,** lo divide (llamando a fork() en algunas subtareas y compute() o join() en otras).

Compute().

Proceso de compute()

Al ejecutar compute(), el proceso que la tarea sigue es el siguiente:

1. **Si es lo suficientemente pequeña**, se ejecuta la lógica diréctamente.
2. **Si es más grande**, se divide en **subtareas más pequeñas,** creando nuevas instancias de la clase de la tarea para cada una.
3. **Fork**: **Una de las subtareas se envía al ForkJoinPool** para que sea ejecutada en paralelo por los hilos del pool.
4. **Join**: El **hilo actual espera** a que la subtarea “forkeada” haya terminado, sincronizando ambas tareas.
5. Luego de que todas las subtareas han completado su procesamiento, **la tarea original se considera terminada.**



Ventajas y desventajas.

Ventajas

- ✔ Paralelismo eficiente.
- ✔ Work-stealing.
- ✔ Divide y vencerás.

Desventajas

- × El costo de dividir puede superar al beneficio.
- × Complejidad en el diseño.
- × Bloqueos de hilos si se abusa de join().

ForkJoin + RecursiveAction en código.

```java
import java.util.concurrent.RecursiveAction;

public class Planeta extends RecursiveAction {

    private String nombre;
    private float[] puntajeSectores;
    final static int UMBRAL = 5; // Umbral para dividir la t
    private int inicio, fin;

    Planeta(String nombre, float[] puntajeSectores, int inic
        this.nombre = nombre;
        this.puntajeSectores = puntajeSectores;
        this.inicio = inicio;
        this.fin = fin;
    }
```

```java
@Override
protected void compute() {
    int longitud = fin - inicio;
    if ((longitud) < UMBRAL) {
        for (int i = inicio; i < fin; i++) {
            puntajeSectores[i] /= 100;
        }
    } else {
        int mid = (inicio + fin) / 2;
        Planeta p1 = new Planeta(nombre + " 1", puntajeSectores, inicio, mid);
        Planeta p2 = new Planeta(nombre + " 2", puntajeSectores, mid, fin);

        p1.fork();
        p2.compute();
        p1.join();
    }
}
```