Patrones de Comportamiento

Los patrones de comportamiento se centran en las interacciones entre objetos y la forma en que se comunican. Estos patrones definen cómo los objetos colaboran entre sí para cumplir con tareas complejas, asegurando que la responsabilidad esté bien distribuida y que el flujo de control esté bien organizado.

Características de los Patrones de Comportamiento:

1. **Delegación de Responsabilidades:** Promueven la delegación de responsabilidades entre objetos, asegurando que las clases no se vuelvan demasiado grandes o complejas.
2. **Control del Flujo:** Estos patrones ayudan a manejar el flujo de control en aplicaciones complejas, mejorando la coordinación entre diferentes partes del sistema.
3. **Flexibilidad y Extensibilidad:** Facilitan la modificación del comportamiento del programa de manera dinámica, permitiendo a los desarrolladores cambiar la forma en que los objetos interactúan entre sí sin alterar las clases subyacentes.

Ejemplos Comunes de Patrones de Comportamiento:

1. **Estrategia (Strategy):** Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite al algoritmo variar independientemente de los clientes que lo usan.
2. **Observador (Observer):** Define una dependencia uno-a-muchos entre objetos, de modo que cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente.
3. **Comando (Command):** Encapsula una petición como un objeto, permitiendo parametrizar a los clientes con diferentes solicitudes, encolar o registrar solicitudes, y soportar operaciones que se puedan deshacer.
4. **Estado (State):** Permite a un objeto alterar su comportamiento cuando su estado interno cambia. El objeto parecerá cambiar su clase.

```
Run | New Tab | Active Connection
import java.util.ArrayList;
Run | New Tab
import java.util.List;

Run | New Tab
// Interface Observador
interface Observador {
    void actualizar(String mensaje);
Run | New Tab
}

Run | New Tab
// Clase Sujeto que notifica a los observadores
class Sujeto {
    private List<Observador> observadores = new ArrayList<>();
    Run | New Tab
    private String estado;

    Run | New Tab
    public void agregarObservador(Observador observador) {
        observadores.add(observador);
    Run | New Tab
    }

    Run | New Tab
    public void eliminarObservador(Observador observador) {
        observadores.remove(observador);
    Run | New Tab
    }

    Run | New Tab
    public void setEstado(String estado) {
        this.estado = estado;
        Run | New Tab
        notificarObservadores();
    Run | New Tab
    }
```

```
    Run | New Tab
    private void notificarObservadores() {
        for (Observador observador : observadores) {
            observador.actualizar(estado);
        Run | New Tab
        }
    }
}

Run | New Tab
// Implementaciones concretas de Observador
class ObservadorConcreto implements Observador {
    private String nombre;

    Run | New Tab
    public ObservadorConcreto(String nombre) {
        this.nombre = nombre;
    Run | New Tab
    }

    Run | New Tab
    @Override
    public void actualizar(String mensaje) {
        System.out.println(nombre + " recibió actualización: " + mensaje);
    Run | New Tab
    }
}

Run | New Tab
// Uso del patrón Observador
public class Main {
    public static void main(String[] args) {
        Sujeto sujeto = new Sujeto();

        Run | New Tab
        Observador observador1 = new ObservadorConcreto("Observador 1");
        Run | New Tab
        Observador observador2 = new ObservadorConcreto("Observador 2");
```

```
        Run | New Tab
        sujeto.agregarObservador(observador1);
        Run | New Tab
        sujeto.agregarObservador(observador2);

        Run | New Tab
        sujeto.setEstado("Nuevo Estado");
    Run | New Tab
    }
}
```

Explicación:

En este ejemplo, Sujeto mantiene una lista de Observadores. Cuando el estado de Sujeto cambia, todos los Observadores son notificados automáticamente y pueden reaccionar al cambio. Este patrón es útil cuando el cambio en un objeto requiere que otros objetos dependientes se actualicen.