Escenario de uso: Patrón *Observer*

Este patrón se adapta cómodamente al manejo de logs para movimientos de caja en el sistema, ya que no es necesario hacer modificaciones en la base de datos para su implementación. Podemos usar a los logs como observadores que cada vez que se realiza un movimiento de caja definiendo los movimientos como los eventos.

En cuanto a su implementación, cada vez que se realiza un movimiento de caja, el evento notifica a los observadores registrados que, en nuestro caso, es el sistema de logs. Este último, al recibir la notificación, registra la información correspondiente al movimiento en la tabla de logs de nuestra base de datos, permitiendo un registro de lo que pasa en la aplicación en caso de ser necesario.

De esta manera, obtenemos una serie de ventajas significativas a la hora de desarrollar, añadir o corregir el código de la aplicación. Algunas de estas ventajas son:

- Centraliza la lógica de auditoría: Evita tener logs en cada momento del código donde se intenta realizar un movimiento.

- Escala fácilmente: Para nuestro modelo de datos, pueden introducirse nuevos eventos (cierres, alertas, etc.) que notifiquen al observador y registre sus logs según sea el caso.

- Consistencia: Es sencillo identificar errores relacionados, pues todo pasa por el mismo flujo.

Codigo:

En nuestro proyecto usamos prisma como sistema de ORM, el siguiente código muestra una posible implementación en nuestra aplicación usando la función "*conexionDB"* (Nuestro *singleton* par manipular la base de datos):

```typescript
// Evento del patron
export class MovementEvent {
  private observers: any[] = [];

  attach(observer: any) {
    this.observers.push(observer);
  }

  detach(observer: any) {
    this.observers = this.observers.filter(obs => obs !== observer);
  }

  async notify(eventType: string, data: any) {
    for (const observer of this.observers) {
      if (typeof observer.update === "function") {
        await observer.update(eventType, data);
      }
    }
  }
}

// Modelo para registrar movimiento en BD
export class MovementModel {
  private subject: MovementEvent;

  constructor(subject: MovementEvent) {
    this.subject = subject;
  }

  async createMovement(data: any) {
    const movement = await conexionDB.log.create(data,);
    await this.subject.notify("MOVEMENT_CREATED", movement);
    return movement;
  }
}

// Modelo para insertar logs en BD
export class LogModel {
  static async create(eventType: string, details: any) {
    await conexionDB.log.create(
```

```
      details,
    );
  }
}

// Observador del patron
export class LogObserver {
  async update(eventType: string, data: any) {
    if (eventType === "MOVEMENT_CREATED"){
      await LogModel.create(eventType, data);
    }
  }
}
```