Universidad Nacional de Colombia - sede Bogotá
Facultad de Ingeniería
Departamento de Sistemas e Industrial
Curso: Ingeniería de Software 1 (2016701)

Patrón de diseño: Command

Command es un patrón de diseño de comportamiento que convierte una petición en un objeto autónomo que contiene toda la información requerida para ejecutar dicha petición.
El objetivo principal es desacoplar el objeto que envía una solicitud del objeto que la recibe, lo que permite un procesamiento de comandos flexible en un sistema. Para implementar el patrón de diseño en el proyecto se plantea refactorizar la lógica del backend, ya que es el entorno que puede acoplar de una mejor manera el principio de separación de responsabilidades.

Para el correcto funcionamiento del patrón de diseño, se deben implementar los siguientes elementos:

1. **Command:** se crea una clase base *Command,* que contiene un método execute() que las clases concretas deberán implementar.
2. **Receiver:** en este caso, será la lógica de base de datos que está dispersa en los endpoints, ya que actuarán como receptores de los comandos.
3. **Concrete Commands:** son comandos específicos para operaciones (en este caso, para la base de datos). En este caso, uno de ellos será para la creación de una venta.
4. **Invoker:** se crea una clase *Invoker,* en la que tomará un objeto de comando para ejecutarlo. Con esto, se cumple con el desacople del cliente que realiza la solicitud con la lógica del servidor.

De acuerdo a lo anterior, los endpoints se encargarán de crear una instancia del comando pertinente, en lugar de tener toda la lógica. Luego, pasarán el comando al Invoker, que ejecutará el comando y devolverá un resultado que se enviará al cliente.

En este caso, se utilizará un ejemplo en específico para comprender la implementación en su totalidad. El archivo *servidor.cjs* contiene los endpoints que generan la comunicación con la base de datos. Un procedimiento en específico, *POST /api/venta,* es complejo ya que realiza operaciones en varias tablas de la base de datos: crea la venta, inserta los detalles de los productos, y actualiza el inventario. Por lo tanto, es ideal refactorizar este procedimiento, utilizando el patrón *Command*.

Flujo de ejecución de ejemplo

1. Cliente (servidor.cjs): el endpoint *POST /api/venta* recibe una solicitud con los datos de la venta. En lugar de procesar la venta directamente, su única responsabilidad es:
   - Crear una instancia de *CrearVentaCommand*, pasándole los datos de la venta.

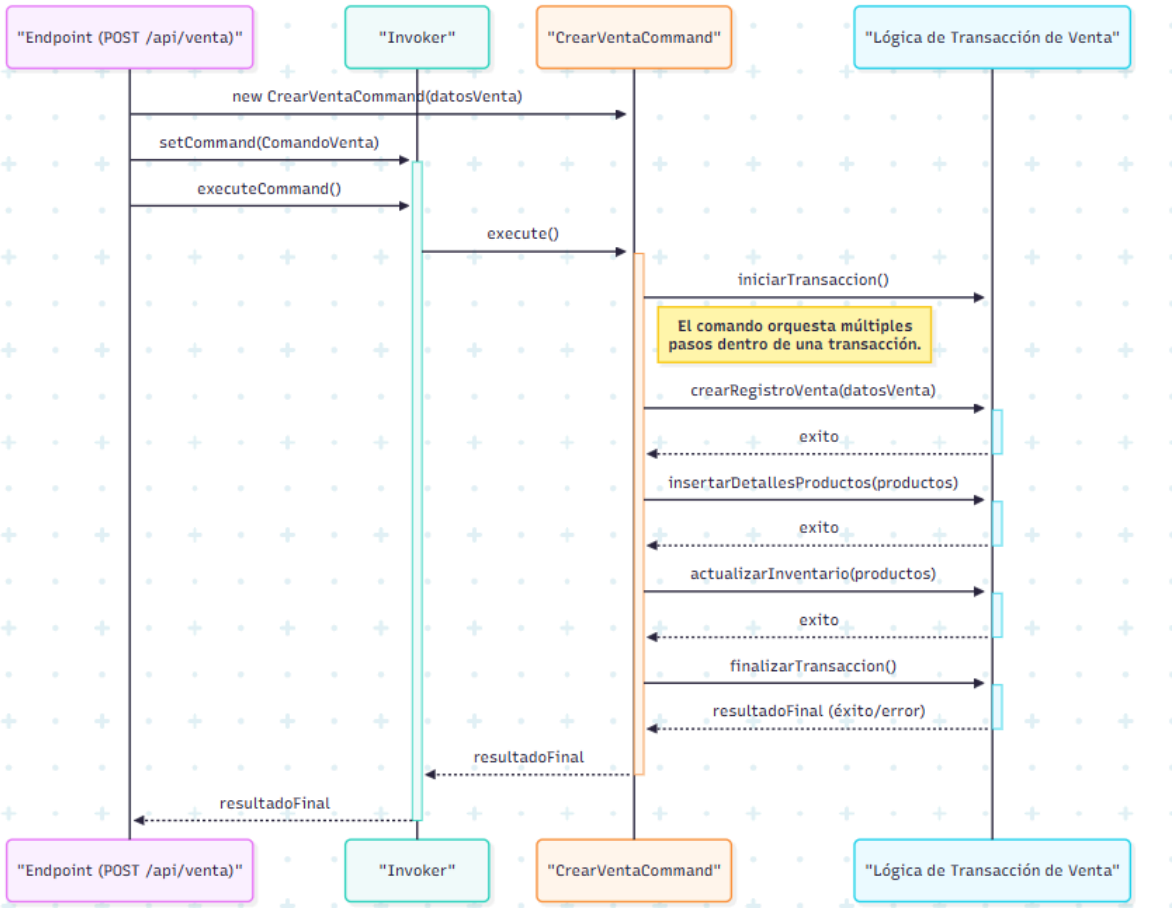
- Crear una instancia del Invoker.
- Asignar el comando al Invoker y llamarlo para que lo ejecute.

2. Invocador (Invoker.js): es una clase simple que tiene un método *executeCommand()*. Este método simplemente llama al método *execute*() del comando que se le ha asignado.

3. Comando Concreto (*CrearVentaCommand.js*):
- Hereda de la clase base *Command*.
- En su método execute(), llama al Receptor (*ServicioVenta*) para que realice la lógica de negocio.
- Orquesta la llamada a los métodos del servicio, encapsulando toda la complejidad de la operación de venta.

4. Receptor (*ServicioVentas.js*):
- Contiene la lógica de negocio real para crear una venta.
- Implementa métodos como *crearVentas*, que internamente maneja la transacción de la base de datos: insertar en la tabla *VENTA*, luego en *DETALLE_PRODUCTO_VENDIDO* y finalmente actualizar el inventario en *MOVIMIENTO_INVENTARIO*.
- Es el único componente que interactúa directamente con la base de datos para esta operación.

El siguiente diagrama muestra el flujo general al momento de registrar una venta.



Comparativa

Antes (Lógica monolítica en *servidor.cjs*): el código realizado antes de la implementación del patrón de diseño muestra la lógica directamente en el endpoint, con una alta cohesión y bajo desacoplamiento.

JavaScript

```
app.post('/api/venta', async (req, res) => {
  try {
    const v = req.body;
    // ... ~50 líneas de código para:
    // 1. Generar código de venta.
    // 2. Buscar cliente.
    // 3. Calcular totales.
    // 4. Iniciar transacción.
    // 5. Insertar en VENTA.
    // 6. Insertar en DETALLE_PRODUCTO_VENDIDO.
    // 7. Insertar en MOVIMIENTO_INVENTARIO.
    // 8. Finalizar transacción.
    // 9. Manejar errores de cada paso.
    res.json({ ok: true, codigo: codigoVenta });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});
```

Después (Uso del Patrón Command): el endpoint ahora es limpio y declarativo.

JavaScript

```
app.post('/api/venta', async (req, res) => {
  try {
    const command = new CreateSaleCommand(req.body);
    const invoker = new Invoker();
    invoker.setCommand(command);
    const result = await invoker.executeCommand();

    res.status(201).json({ ok: true, codigo: result.saleId });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
```

```
});
```

Justificación del uso

1. Separación de Responsabilidades (SoC): el servidor.cjs se vuelve más limpio y simple. Su única responsabilidad es gestionar las rutas y delegar el trabajo, sin contener lógica de negocio. Esto cumple con el Principio de Responsabilidad Única (SRP).

2. Encapsulación de Operaciones Complejas: la creación de una venta no es una única inserción en la base de datos; es una transacción que afecta a múltiples tablas. El patrón Command permite encapsular esta transacción completa en un solo objeto (CrearVentaCommand), tratándola como una unidad atómica.

3. Mejora de la Mantenibilidad y Escalabilidad: añadir nuevas operaciones se reduce a la creación de una nueva clase de comando, sin necesidad de modificar el *servidor.cjs* o el *Invoker*. La lógica de negocio está centralizada en los `services` (Receptores), lo que facilita su mantenimiento y prueba.

4. Flexibilidad para el Futuro: aunque no se implemente de inmediato, esta arquitectura sienta las bases para funcionalidades avanzadas como:

- Cola de Tareas: Podríamos poner los objetos de comando en una cola para ser procesados en segundo plano (por ejemplo, para generar reportes pesados sin bloquear la respuesta al usuario).
- Historial y Auditoría: Se podría mantener un log de todos los comandos ejecutados, facilitando la auditoría del sistema.
- Operaciones Deshacer/Rehacer: El patrón es la base para implementar la funcionalidad de deshacer, guardando el estado necesario en el objeto de comando.