PATRONES APLICADOS EN EL PROYECTO DEL TALLER MECÁNICO

# 1. Patrón de Comportamiento: Template Method

**Ubicación:**  
Clase BaseService<T, ID> y sus subclases (VehiculoService, HistorialArregloService, etc.)

**Concepto teórico:**  
El patrón **Template Method** define la estructura general de un algoritmo en una clase base, permitiendo que las subclases redefinan partes específicas del comportamiento sin cambiar la estructura global.  
En otras palabras, la clase padre define “el esqueleto” del proceso y deja que las hijas completen o personalicen ciertos pasos.

**Aplicación en el proyecto:**  
El método alta(T entidad) en BaseService actúa como **plantilla**:

public T alta(T entidad) throws ErrorServiceException {

validar(entidad); // Paso 1: validación (hook)

preAlta(entidad); // Paso 2: antes de guardar (hook)

entidad.setEliminado(false);

T guardado = repository.save(entidad); // Paso principal (común)

postAlta(guardado); // Paso 3: después de guardar (hook)

return guardado;

}

Las subclases (HistorialArregloService, VehiculoService, etc.) pueden redefinir los métodos *hook* (validar, preAlta, postAlta, etc.) para agregar reglas específicas sin duplicar lógica.

**Ventajas obtenidas:**

* Centralizamos la lógica común de CRUD.
* Evitamos código repetido.
* Permitimos extender comportamientos sin alterar la plantilla base.

**Resumen:**

Este patrón fue elegido porque permite estructurar de forma homogénea las operaciones comunes de todas las entidades, asegurando consistencia y fácil mantenimiento del código.

# 2. Patrón Creacional: Factory Method

**Ubicación:**  
En la clase **BaseService** → método abstracto **protected abstract T createEmpty();**  
y en **HistorialArregloService**, **VehiculoService**, etc., donde se implementa el método concreto.

**Concepto teórico:**  
El patrón **Factory Method** delega la creación de objetos a las subclases, de manera que el código que necesita instancias no se acople a clases concretas.  
Esto favorece la **extensibilidad** y **encapsula la lógica de creación** de objetos.

**Aplicación en el proyecto:**  
El **TallerFacade** utiliza el método de fábrica definido en el servicio:

HistorialArreglo h = historialService.createEmpty();

De esta forma, el *Facade* no necesita saber cómo construir un HistorialArreglo; delega la responsabilidad al *service* correspondiente, que puede aplicar reglas específicas de inicialización (por ejemplo, setEliminado(false) o inicializar colecciones).

**Ventajas obtenidas:**

* Aisla la creación de instancias de HistorialArreglo, Vehiculo, etc.
* Facilita el mantenimiento y las pruebas unitarias.
* Permite cambiar las clases concretas sin modificar el resto del sistema.

**Resumen:**

Este patrón se eligió para desacoplar la creación de entidades y centralizar su inicialización en cada servicio, evitando dependencias directas entre capas.

# 3. Patrón Estructural: Facade

**Ubicación:**  
Clase TallerFacade

**Concepto teórico:**  
El patrón **Facade** proporciona una interfaz unificada y simplificada para un conjunto complejo de subsistemas.  
Oculta la complejidad interna de las interacciones entre múltiples servicios, exponiendo un único punto de acceso coherente.

**Aplicación en el proyecto:**  
La clase TallerFacade coordina múltiples servicios (VehiculoService, MecanicoService, HistorialArregloService, ClienteService) para realizar una operación de negocio compleja:

public void registrarReparacion(String idVehiculo, String idMecanico, String detalle, String fechaArregloStr) {

Vehiculo v = vehiculoService.obtener(idVehiculo).orElseThrow(...);

Mecanico m = mecanicoService.obtener(idMecanico).orElseThrow(...);

HistorialArreglo h = historialService.createEmpty();

h.setVehiculo(v);

h.setMecanico(m);

h.setDetalleArreglo(detalle);

h.setFechaArreglo(...);

historialService.alta(h);

}

El *controller* (HistorialArregloController) ya no necesita conocer la lógica de negocio ni manipular múltiples servicios, solo llama:

tallerFacade.registrarReparacion(...);

**Ventajas obtenidas:**

* Simplifica la interfaz del subsistema del taller.
* Reduce el acoplamiento entre controladores y servicios.
* Mejora la legibilidad y cohesión del código.

**Resumen:**

Se implementó este patrón para ofrecer una interfaz unificada que encapsule la lógica de negocio de registrar una reparación, facilitando la comunicación entre capas y reduciendo la complejidad del controlador.

**🧠 Conclusión general**

| **Categoría** | **Patrón** | **Objetivo** | **Resultado en el proyecto** |
| --- | --- | --- | --- |
| **Comportamiento** | Template Method | Estandarizar la estructura de los servicios | Unificación de lógica CRUD y mayor reutilización |
| **Creacional** | Factory Method | Desacoplar la creación de entidades | Flexibilidad y consistencia en la inicialización |
| **Estructural** | Facade | Simplificar la interacción entre subsistemas | Controladores más limpios y baja complejidad |