



Patrones de Diseño III

UNJu - Programación Orientada a Objetos

Ing. José Zapana



Interceptor Pattern

- Propósito

Centralizar el manejo de errores en una aplicación Spring Boot para:

- Evitar duplicación de lógica en controladores.
- Proveer respuestas consistentes y estructuradas.
- Mejorar la experiencia del cliente y la documentación (Swagger).

- Motivación

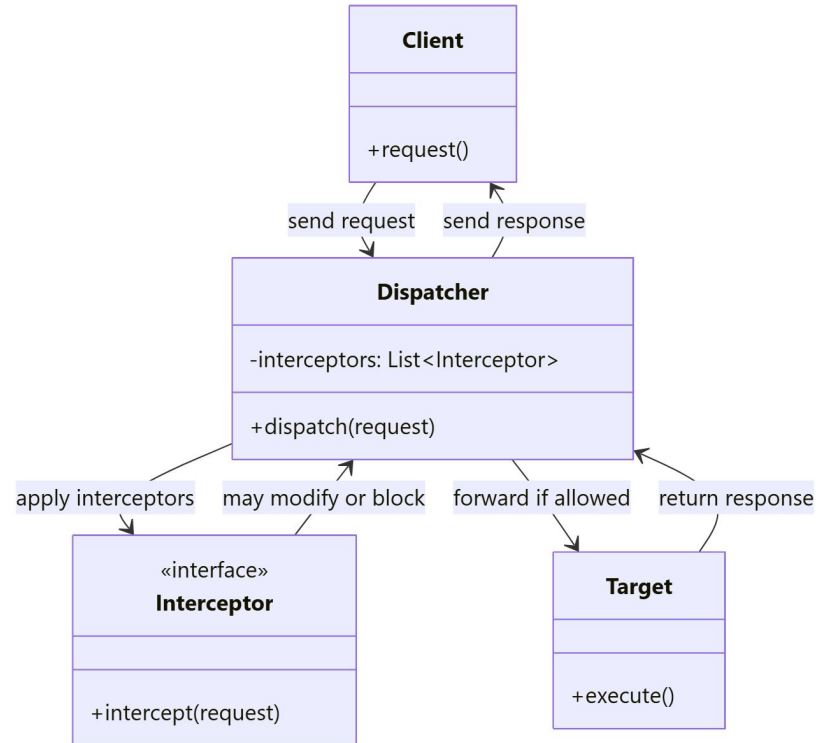
- Los errores lanzados en servicios o controladores pueden generar respuestas poco claras (ej. 500 sin mensaje).
- Capturar y transformar excepciones en respuestas HTTP con cuerpo explicativo mejora la comunicación con el frontend.
- Facilita la trazabilidad, el logging y la documentación de errores.

- Aplicabilidad

- Se desea interceptar excepciones como `IllegalArgumentException`, `MethodArgumentNotValidException`, etc.
- Se quiere devolver objetos como **MensajeError** con mensajes personalizados.
- Se trabaja con APIs RESTful que deben comunicar errores de forma clara y estructurada.
- Se usa Swagger/OpenAPI para documentar respuestas esperadas.



Diagrama de clases





Implementación

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<MensajeError>
manejarIllegalArgumentException(IllegalArgumentException ex) {
        MensajeError error = new MensajeError(ex.getMessage());
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(error);
    }
}
```

```
public class MensajeError {
    private String mensaje;

    public MensajeError(String mensaje) {
        this.mensaje = mensaje;
    }

    public String getMensaje() {
        return mensaje;
    }

    public void setMensaje(String mensaje) {
        this.mensaje = mensaje;
    }
}
```

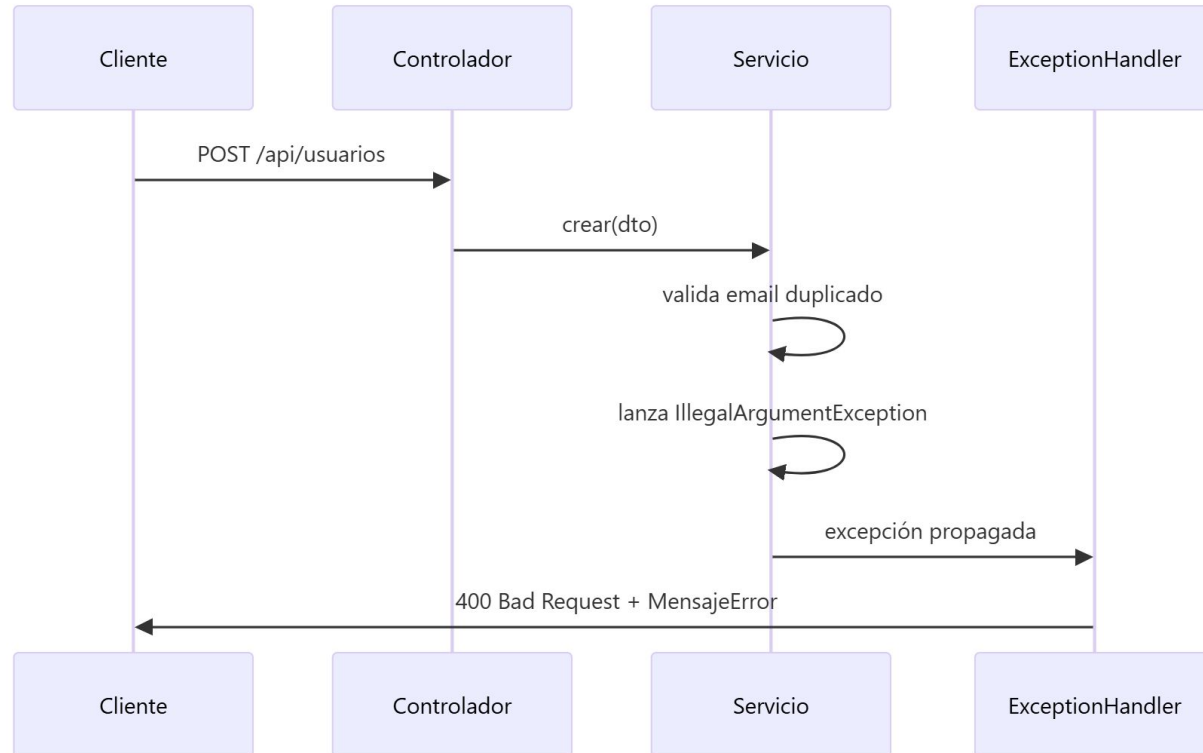


Implementación

```
@Operation(  
    summary = "Crear un nuevo usuario",  
    description = "Crea un usuario validando email y username únicos",  
    responses = {  
        @ApiResponse(  
            responseCode = "201",  
            description = "Usuario creado",  
            content = @Content(mediaType = "application/json",  
schema = @Schema(implementation = UsuarioDTO.class))  
        ),  
        @ApiResponse(  
            responseCode = "400",  
            description = "Error de validación o negocio",  
            content = @Content(mediaType = "application/json",  
schema = @Schema(implementation = MensajeError.class))  
        )  
    }  
)  
@PostMapping  
public ResponseEntity<UsuarioDTO> crear(@Valid @RequestBody UsuarioDTO  
dto) {  
    log.info("Creando un nuevo usuario");  
    UsuarioDTO creado = usuarioService.crear(dto);  
    return ResponseEntity.created(URI.create("/api/usuarios/" +  
creado.getUsername())).body(creado);  
}
```



Ejemplo API crear usuario





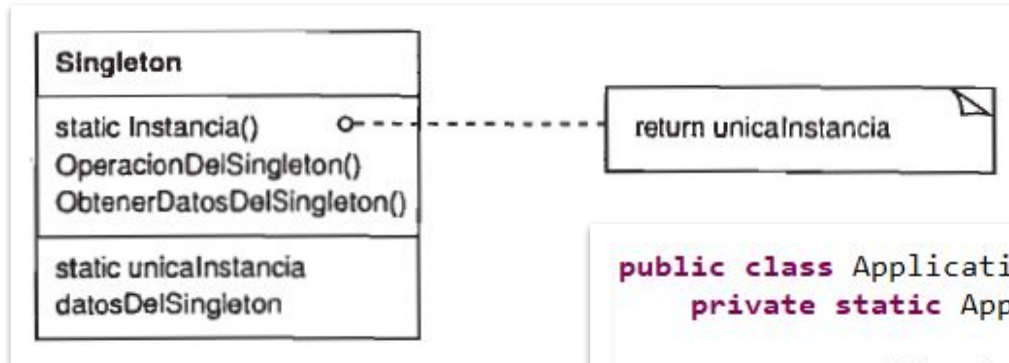
Patrón Singleton (único)

- Propósito
 - Garantiza que una clase tenga solo una instancia, y proporciona un punto de acceso global a ella.
- Motivación
 - Cola de impresión, sistema de ficheros, gestor de ventanas
 - Una solución es hacer que sea **la propia clase la responsable de su única instancia**, y puede proporcionar un modo de acceder a la instancia.
- Aplicabilidad
 - Cuando deba haber exactamente una instancia de una clase, y ésta deba ser accesible a los clientes desde un punto de acceso conocido.
 - La única instancia debería ser extensible mediante herencia, y los clientes deberían ser capaces de usar una instancia extendida sin modificar su código.



Singleton - Estructura

- Estructura y ejemplo



```
public class ApplicationProperties {  
    private static ApplicationProperties instance;  
  
    private ApplicationProperties(){  
  
    }  
  
    public static ApplicationProperties getInstance() {  
        if (instance == null)  
            instance = new ApplicationProperties();  
        return instance;  
    }  
}
```



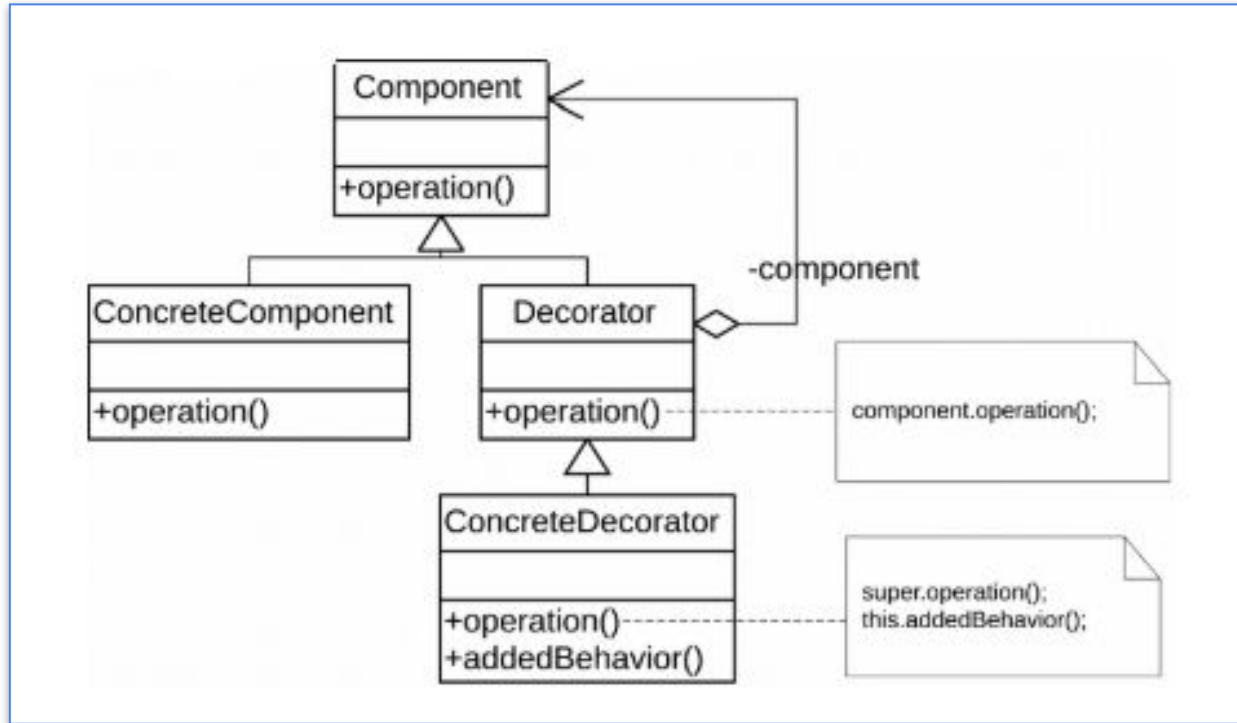
Patrón Decorator

- **Objetivo**
 - Agregar comportamiento a un objeto dinámicamente y en forma transparente.
- **Problema**
 - Cuando queremos agregar comportamiento extra a algunos objetos de una clase puede usarse herencia. El problema es cuando necesitamos que el comportamiento se agregue o quite dinámicamente, porque en ese caso los objetos deberían “mutar de clase”. El problema que tiene la herencia es que se decide estáticamente.
- **Solución**
 - Definir un decorador (o “wrapper”) que agregue el comportamiento cuando sea necesario.
 - Prestar atención al polimorfismo



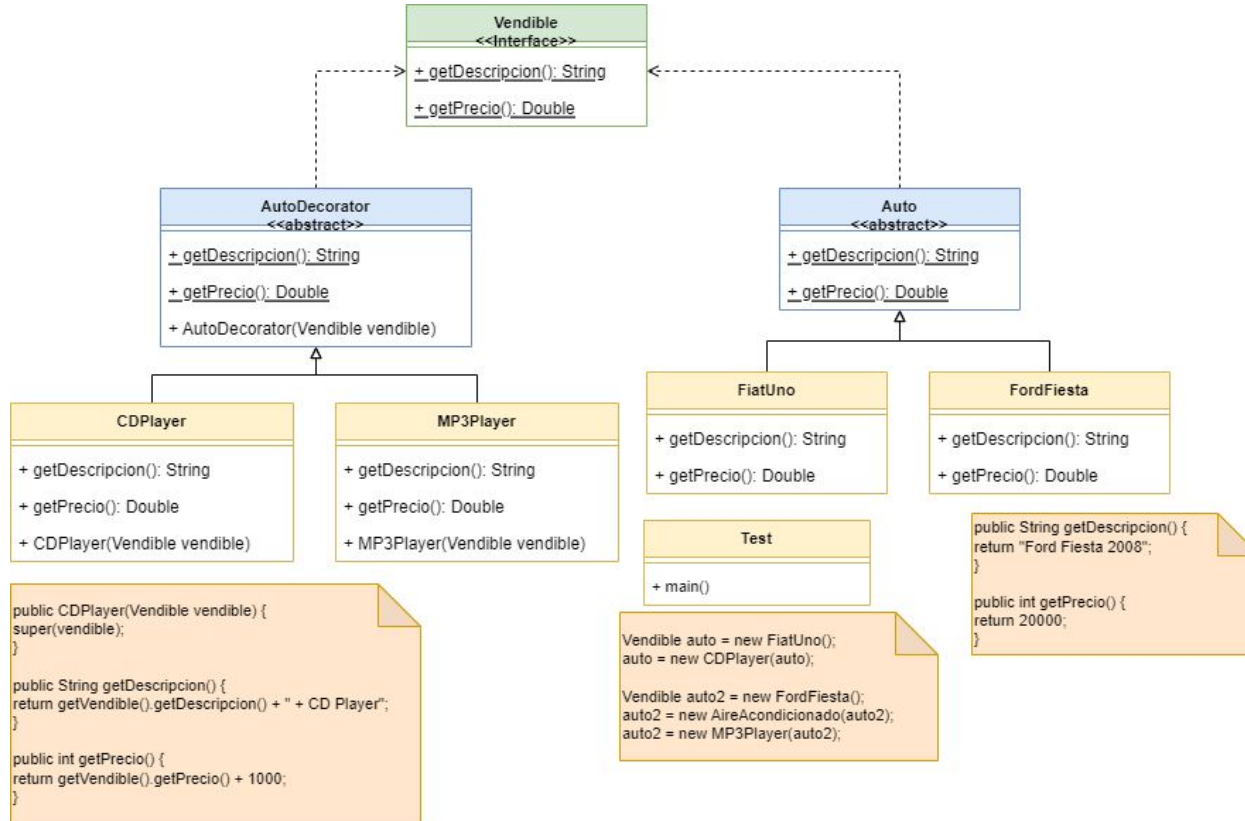
Patrón Decorator - Estructura

- Estructura general





Ejemplo Decorator





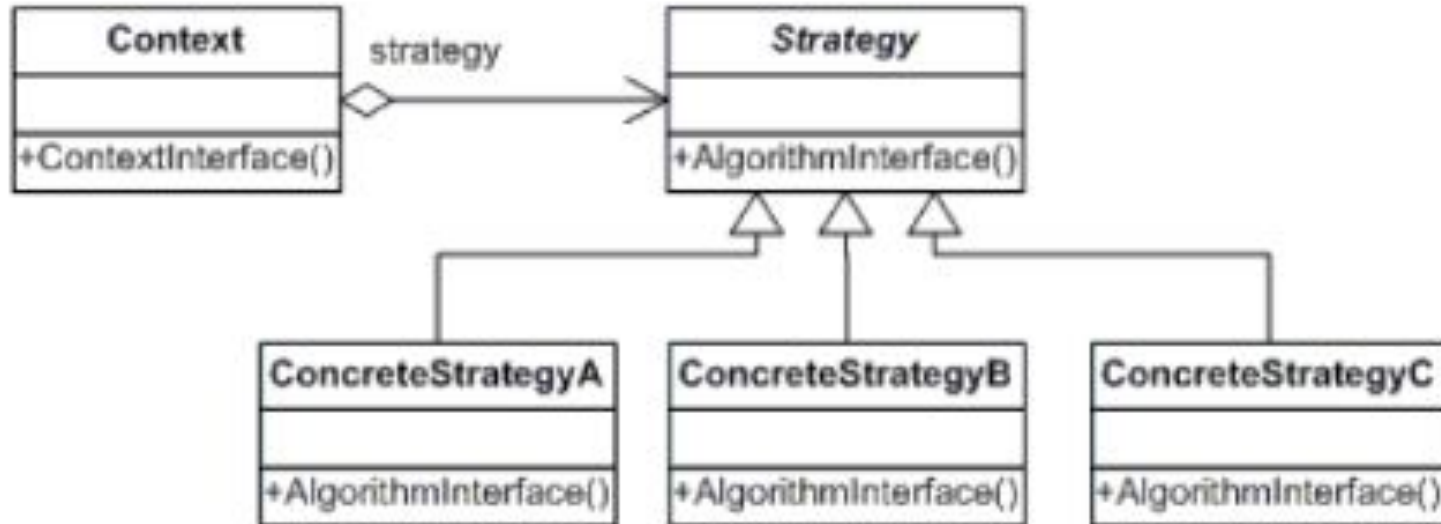
Patrón Strategy

- **Objetivo:**
 - Desacoplar un algoritmo del objeto que lo utiliza.
 - **Permitir cambiar el algoritmo que un objeto utiliza en forma dinámica.**
 - Brindar flexibilidad para agregar nuevos algoritmos que lleven a cabo una función determinada.
- **Problema:**
 - Existen muchos algoritmos para llevar a cabo una tarea.
 - No es deseable codificarlos todos en una clase y seleccionar cuál utilizar por medio de sentencias condicionales.
 - Cada algoritmo utiliza información propia. Colocar esto en los clientes lleva a tener clases complejas y difíciles de mantener.
 - Es necesario cambiar el algoritmo en forma dinámica.
- **Aplicabilidad:**
 - Muchas clases relacionadas difieren solo en su comportamiento. Las estrategias permiten configurar una clase con un determinado comportamiento de entre muchos posibles.
 - Se necesitan distintas variantes de un algoritmo
 - Un algoritmo usa datos que los clientes no deberían conocer. Evita exponer estructuras de datos complejas y dependientes del algoritmo.
 - Una clase define muchos comportamientos, y estos se representan como múltiples sentencias condicionales en sus operaciones.
- **Solución:**
 - Definir una familia de algoritmos, encapsulando a cada uno en un objeto.



Patrón Strategy - Estructura

- Estructura General





Strategy vs State

- Mismo diagrama de clases.
- Misma idea de delegación.
- Pero
 - El estado es privado del objeto, ningún otro objeto sabe de él.
 - Un State define una máquina de estados con sus transiciones.
 - Un strategy suele tener un único mensaje público



Referencias

- [Builder Design Pattern in Java](#)
- [Factory Method Design Pattern in Java](#)
- [Catálogo de ejemplos en Java](#)