

Integrantes: Sebastián León

Curso: GR2SW

Fecha: 09 de noviembre de 2025

Clase 006 – Taller Aplicando Principios de Código Limpio

Objetivo

Comprender los principios del código limpio y aplicarlos en un contexto real, analizando código de repositorios públicos para identificar buenas prácticas, detectar olores de código y proponer mejoras que incrementen la mantenibilidad, claridad y calidad del software.

Introducción

¿Qué es Código Limpio y por qué es importante?

Código Limpio es aquel que combina elegancia, claridad y eficacia. Según Bjarne Stroustrup, el código limpio hace bien una sola cosa, con lógica directa que evita errores ocultos, mínimas dependencias para facilitar mantenimiento, y un manejo de errores completo y consistente. Es un código que se disfruta leer, como un objeto bien diseñado, y que tiende a reducir la probabilidad de introducir fallos al modificarlo. [1]

Es importante porque permite reducir errores, mejorar la velocidad de desarrollo y facilita que nuevos programadores puedan integrarse sin esfuerzo al proyecto. Un código limpio agrega valor a largo plazo. [1]

Impacto en mantenibilidad, escalabilidad y trabajo en equipo

Mantenibilidad: El desarrollo de software rara vez es un trabajo de una sola vez. Un código limpio permite que los cambios se realicen de manera más sencilla sin necesidad de grandes refactorizaciones, lo que facilita corregir errores más rápido y reduce la probabilidad de introducir nuevos fallos en el sistema. [2]

Escalabilidad: A medida que la aplicación crece, también lo hace la base de código. Mantener un código limpio ayuda a controlar la complejidad mediante una estructura modular y fácil de seguir, lo que hace más sencillo agregar nuevas funcionalidades y disminuye el riesgo de acumular deuda técnica. [2]

Trabajo en equipo: La calidad del código es fundamental en entornos colaborativos. Un código limpio permite que los miembros del equipo comprendan con claridad lo que hace cada parte, construyan sobre el trabajo de otros sin confusión y facilita la incorporación de nuevos desarrolladores, quienes pueden integrarse rápidamente al proyecto gracias a la claridad y consistencia del código. [2]

Ejemplo de “código sucio” vs “código limpio”.

Un ejemplo claro de “código sucio” es cuando encontramos variables con nombres genéricos como `x` o `data1`, funciones demasiado largas que hacen muchas tareas a la vez, comentarios explicando cosas que deberían ser obvias, lógica duplicada o formatos de código inconsistentes. En contraste, el código limpio se caracteriza por utilizar nombres significativos y expresivos, funciones pequeñas con una sola responsabilidad, menos comentarios innecesarios, reutilización de lógica evitando duplicación, y un estilo uniforme que facilita leer y entender el código rápidamente. [1]

Principios clave

- **Reglas generales de diseño:** priorizar claridad, simplicidad y reducir complejidad innecesaria.
- **Comprensibilidad y nombres:** elegir nombres de variables y funciones que describan claramente su propósito.
- **Funciones y comentarios:** las funciones deben ser cortas y realizar una sola tarea; los comentarios deben ser solo cuando aporten valor, y no para explicar código mal escrito.
- **Organización del código, objetos y estructuras:** mantener una estructura lógica, modular y coherente para facilitar la navegación del proyecto.
- **Pruebas y olores de código:** identificar indicadores de mala calidad como duplicación, métodos enormes, variables sin contexto, e implementar pruebas que respalden el comportamiento esperado.

[1]

Actividad Principal

Elige un repositorio público en GitHub

<https://github.com/spring-projects/spring-petclinic>

Archivos seleccionados del proyecto

Archivos elegidos:

1. `OwnerController.java`
2. `PetController.java`

<https://github.com/spring-projects/spring-petclinic/blob/main/src/main/java/org/springframework/samples/petclinic/owner/PetController.java>

OwnerController.java

Análisis aplicando los principios de Código Limpio

¿Los nombres son claros?

Sí. Los nombres de métodos describen bien su intención: `initCreationForm`, `processFindForm`, `findPaginatedForOwnersLastName`, `addPaginationModel`.

¿Las funciones son cortas y hacen una sola cosa?

La mayoría sí. PERO `processFindForm` (Figura 1) tiene demasiada lógica (validación, branching, consulta, UI). Hace 3 cosas mínimo.

¿Hay comentarios útiles o innecesarios?

Hay muy pocos comentarios. No hay comentarios innecesarios. Por ejemplo, los comentarios en `processFindForm` (Figura 1) son explicativos y útiles. Pero sí faltan comentarios explicativos en partes donde se toman decisiones de lógica.

¿Hay olores de código?

Sí, se observa una ligera duplicación en la validación de formularios, y también existen métodos donde se mezclan responsabilidades del controlador con pequeñas piezas de lógica y preparación del modelo, lo que indica que parte de esa responsabilidad debería extraerse hacia servicios u otros componentes para mantener el principio de responsabilidad única.

¿Cómo está organizada la estructura?

La estructura es buena, sigue arquitectura MVC. Sin embargo este controlador podría ser más pequeño si delega más tareas. El controlador tiene demasiadas responsabilidades: *buscar* + *paginar* + *validar* + *UI redirects*.

```
@GetMapping("/owners")
public String processFindForm(@RequestParam(defaultValue = "1") int page, Owner owner, BindingResult result,
    Model model) {
    // allow parameterless GET request for /owners to return all records
    String lastName = owner.getLastName();
    if (lastName == null) {
        lastName = ""; // empty string signifies broadest possible search
    }

    // find owners by last name
    Page<Owner> ownersResults = findPaginatedForOwnersLastName(page, lastName);
    if (ownersResults.isEmpty()) {
        // no owners found
        result.rejectValue("lastName", "notFound", "not found");
        return "owners/findOwners";
    }

    if (ownersResults.getTotalElements() == 1) {
        // 1 owner found
        owner = ownersResults.iterator().next();
        return "redirect:/owners/" + owner.getId();
    }

    // multiple owners found
    return addPaginationModel(page, model, ownersResults);
}
```

Figura 1. Método `processFindForm` en el `OwnerController.java`

Mejoras propuestas

1. Extraer paginación a un servicio (ej: OwnerService): el controlador no debería decidir pageSize ni la forma de buscar propietarios, esto ayuda a mantener SRP.
2. Simplificar processFindForm() en métodos más pequeños: ese método hace flujo condicional + consulta + redirect + validación; dividirlo reduciría complejidad y mejora testeo.
3. Usar constantes para las viewStrings repetidas: "owners/findOwners", "owners/ownersList" se repiten y puede causar errores si el nombre de vista cambia solo en una parte del código.
4. Validar id en processUpdateOwnerForm mediante un Validator dedicado: esa validación manual en el controller debería estar en un validador para mantener responsabilidades claras.
5. Retornar ResponseEntity o al menos un DTO en métodos futuros tipo AJAX/REST: el controller está muy acoplado al modelo basado en plantillas, eso hace más costosa una migración futura a JSON o API REST.

PetController.java

Análisis aplicando los principios de Código Limpio

¿Los nombres son claros?

Sí. Los métodos tienen nombres descriptivos: initCreationForm, processCreationForm, initUpdateForm, processUpdateForm, findPet, updatePetDetails. Los nombres comunican bien la intención.

¿Las funciones son cortas y hacen una sola cosa?

La mayoría sí, pero processCreationForm y processUpdateForm tienen demasiada lógica agrupada: validación manual + verificación de duplicado + validación de fecha + persistencia + redirects. Eso indica múltiples responsabilidades dentro del controlador.

¿Hay comentarios útiles o innecesarios?

Los comentarios son pocos y los que existen son útiles. Por ejemplo, el comentario sobre “checking if the pet name already exists” está bien (Figura 2). No hay comentarios redundantes.

¿Hay olores de código?

Sí, se observa lógica **duplicada** entre processCreationForm y processUpdateForm (validar nombre duplicado + validar fecha futura). También el controlador está asumiendo decisiones de validación que deberían estar en un validador o servicio. Esa mezcla rompe SRP en varios puntos.

¿Cómo está organizada la estructura?

La estructura sigue MVC. Sin embargo el controlador está ocupándose de lógica que le corresponde a un servicio (especialmente validación de reglas de negocio y persistencia del Owner completo para registrar un pet). Esto hace que el controller sea más grande de lo necesario.

```
@PostMapping("/pets/{petId}/edit")
public String processUpdateForm(Owner owner, @Valid Pet pet, BindingResult result,
    RedirectAttributes redirectAttributes) {

    String petName = pet.getName();

    // checking if the pet name already exists for the owner
    if (StringUtils.hasText(petName)) {
        Pet existingPet = owner.getPet(petName, false);
        if (existingPet != null && !Objects.equals(existingPet.getId(), pet.getId())) {
            result.rejectValue("name", "duplicate", "already exists");
        }
    }

    LocalDate currentDate = LocalDate.now();
    if (pet.getBirthDate() != null && pet.getBirthDate().isAfter(currentDate)) {
        result.rejectValue("birthDate", "typeMismatch.birthDate");
    }

    if (result.hasErrors()) {
        return VIEWS_PETS_CREATE_OR_UPDATE_FORM;
    }

    updatePetDetails(owner, pet);
    redirectAttributes.addFlashAttribute("message", "Pet details has been edited");
    return "redirect:/owners/{ownerId}";
}
```

Figura 2. Método processUpdateForm en el OwnerController.java

Mejoras propuestas

1. Extraer la validación de nombre duplicado hacia un servicio o validador: hoy está repetida entre processCreationForm y processUpdateForm, rompe SRP y DRY.
2. Extraer la validación de fecha de nacimiento futura hacia un Validator dedicado: esa regla está en ambos flujos, debería centralizarse para consistencia.
3. Mover la lógica de persistencia a un PetService: el controller no debería estar guardando un Owner para actualizar un Pet; la persistencia es responsabilidad del service.
4. Separar processUpdateForm en métodos más pequeños (validate, persist, redirect): ese método ejecuta múltiples responsabilidades; dividirlo facilita testeo unitario y reduce complejidad.
5. Cambiar updatePetDetails para que retorne el Pet actualizado en lugar de void: hoy oculta el resultado y obliga a leer el Owner completo del repositorio para ver cambios; devolver el Pet actualizado hace el flujo más explícito y facilita unit testing.

Conclusión

Spring-petclinic es un muy buen ejemplo de arquitectura limpia en proyecto real, pero aún así tiene oportunidades de refactorización en controladores demasiado ocupados, duplicación de validaciones y servicios un poco genéricos. Esto demuestra que incluso proyectos “bien escritos” siguen siendo mejorables continuamente desde el enfoque de Clean Code.

Referencias

- [1] R. C. Martin, *Código limpio: Manual de estilo para el desarrollo ágil de software*, 2ª ed., Madrid: Anaya Multimedia, 2010.
- [2] P. Natekar, *The Importance of Clean Code: Writing Maintainable Code for the Future*, Medium, 1 ene. 2024. [Online]. Available: <https://pawannatekar220.medium.com/the-importance-of-clean-code-27515760c2cc>