



Integrantes: Gabriel Maldonado, Daniel Moncayo y Jhair Zambrano

Curso: GR2SW

Fecha: 8 de noviembre de 2025

Aplicando principios de código limpio en proyectos reales

Introducción

1. ¿Qué es el código limpio?

Es una filosofía de desarrollo de software que sostiene que el código debe escribirse pensando en el lector, que pueden ser otros programadores o para el mismo desarrollador del futuro.

El código limpio es aquel que es fácil de entender, mantener y extender. Si el código funciona, pero es ininteligible, es “código sucio” o spaghetti code, muchos autores reconocidos como:

- Bjarne Stroustrup (creador de C++) .
- Grady Booch (autor de Object Oriented Analysis and Design with Applications).
- Dave Thomas (fundador de OTI, el padrino de la estrategia Eclipse).
- Michael Feathers (autor de Working Effectively with Legacy code).
- Ron Jeffries (autor de Extreme Programming Installed y Extreme Programming Adventures in C#).
- Ward Cunningham (inventor de Wiki, fit y uno de los inventores de extreme programming).

Incentivaron a cambiar el paradigma sobre el cómo se debe presentar el código, y los principios que más destacaron fueron la simplicidad, la expresividad y la ausencia de duplicidad. Todos ellos convergen en la idea de que la calidad del código se mide por la rapidez con la que se puede leer y comprender. A raíz de esto, se popularizaron reglas de diseño esenciales que guiarán este taller, tales como KISS (Manténlo simple), DRY (No te repitas) y la importancia crítica de usar Nombres Significativos y Funciones de Responsabilidad Única.

2. ¿Por qué es importante?

Escribir código limpio no es solo estética; tiene un impacto directo en el negocio y la salud del proyecto:

- **Mantenibilidad:** El software pasa el 80% de su vida en modo "mantenimiento". El código limpio permite encontrar errores y corregirlos rápidamente sin romper otras partes del sistema.
- **Escalabilidad:** Un código ordenado permite agregar nuevas funcionalidades de manera ágil. En código sucio, cada nueva función aumenta el riesgo de introducir bugs.



- **Trabajo en Equipo:** El código se lee muchas más veces de las que se escribe. Escribir limpio es un acto de respeto hacia tus compañeros. Facilita la incorporación de nuevos desarrolladores (onboarding).

3. Ejemplo: Código sucio vs. Código limpio

a. Código sucio

```
1 // ¿Qué es d? ¿Qué es 0.15? ¿Qué significa tipo === 1?
2 function calc(l, tipo) {
3     let t = 0;
4     for (let i = 0; i < l.length; i++) {
5         t += l[i].p;
6     }
7     if (tipo === 1 && t > 100) {
8         return t * 0.85;
9     }
10    return t;
11 }
```

b. Código limpio

```
1 const PORCENTAJE_DESCUENTO_VIP = 0.15;
2 const UMBRAL_COMPRA_MINIMA = 100;
3
4 /**
5  * Calcula el total de La compra aplicando descuentos según el tipo de cliente.
6  * @param {*} items lista de artículos con su 'precio'.
7  * @param {*} tipoCliente Tipo de cliente
8  * @returns Total de la compra con el descuento aplicado si corresponde.
9 */
10 function calcularTotalConDescuento(items, tipoCliente) {
11     let totalCompra = items.reduce((total, item) => total + item.precio, 0);
12     if (esClienteVIP(tipoCliente) && cumpleMontoMinimo(totalCompra)) {
13         return aplicarDescuento(totalCompra);
14     }
15     return totalCompra;
16 }
17 function esClienteVIP(tipo) {
18     return tipo === "VIP";
19 }
20 function cumpleMontoMinimo(total) {
21     return total > UMBRAL_COMPRA_MINIMA;
22 }
23 function aplicarDescuento(total) {
24     return total * (1 - PORCENTAJE_DESCUENTO_VIP);
25 }
```

4. Principios Clave



a. Reglas Generales de diseño

Para guiar la arquitectura general del código, utiliza estos acrónimos:

- **KISS (Keep It Simple, Stupid):** No busques soluciones complejas si una simple funciona. La complejidad innecesaria es enemiga de la calidad.
- **DRY (Don't Repeat Yourself):** Evita la duplicidad. Si tienes que cambiar una lógica, deberías hacerlo en un solo lugar.
- **YAGNI (You Ain't Gonna Need It):** No agregues funcionalidades "por si acaso". Implementa solo lo que necesitas hoy.

b. Comprensibilidad y Nombres

El nombre de una variable, función o clase debe responder: ¿Por qué existe? ¿Qué hace? ¿Cómo se usa?

- **Intención clara:**
`int d;` (**mal**)
`int diasTranscurridos;` (**bien**).
- **Evita desinformación:** No uses `listaCuentas` si la estructura real es un mapa o un arreglo. Usa `grupoCuentas`.
- **Nombres pronunciables:** Si no puedes leer el nombre en voz alta en una llamada, es un mal nombre (`genymdhms` vs `generationTimestamp`).
- **Contexto:** Evita prefijos innecesarios. Si estás en una clase `Usuario`, no llames a la variable `nombreUsuario`, solo `nombre` es suficiente.

c. Funciones y comentarios

• Funciones:

- **Responsabilidad Única (SRP):** Una función debe hacer una sola cosa y hacerla bien.
- **Tamaño:** Deben ser pequeñas. Si tu función tiene 100 líneas, probablemente hace demasiadas cosas.
- **Argumentos:** Lo ideal es 0, 1 o 2 argumentos. Si necesitas más de 3, considera pasar un objeto como parámetro.

• Comentarios:

- El mejor comentario es el código bien nombrado.
- Usa comentarios para explicar el "Por qué" (la razón de negocio o una decisión compleja), no el "Qué" (lo que hace el código, eso debe ser obvio al leerlo).
- No comentes código muerto (código comentado).
¡Bórralo! Para eso existe Git.

d. Organización de código, objetos y estructuras



- **La metáfora del periódico:** El código debe leerse como un artículo. Arriba los titulares (conceptos generales y de alto nivel) y abajo los detalles técnicos (funciones auxiliares).
- **Ley de Demeter:** Un objeto no debe conocer los detalles internos de los objetos que manipula ("No hables con extraños"). Evita encadenamientos largos como `obj.getA().getB().getC().doSomething()`.
- **Estructuras de datos vs Objetos:**
 - Las estructuras exponen datos y no tienen lógica significativa.
 - Los objetos esconden datos (encapsulamiento) y exponen comportamiento.

e. Pruebas y olores de código

- **Pruebas (Tests):**
 - El código limpio debe tener pruebas unitarias. Sin pruebas, tienes miedo de refactorizar; si tienes miedo, el código se pudre.
 - **Regla F.I.R.S.T.:** Las pruebas deben ser Rápidas (Fast), Independientes, Repetibles, Self-Validating (pasan o fallan, sin interpretación manual) y Oportunas (escritas junto o antes del código).
- **Olores de código:**
 - **Rigidez:** El software es difícil de cambiar; un cambio pequeño causa una cascada de cambios.
 - **Fragilidad:** El software se rompe en muchos lugares tras un solo cambio.
 - **Complejidad innecesaria:** Uso de patrones de diseño donde no hacen falta.
 - **Clases Dios:** Clases que hacen todo y controlan todo el sistema.

5. ACTIVIDAD PRINCIPAL

Repositorio seleccionado:

- <https://github.com/mastodon/mastodon>

Mastodon es un proyecto masivo con +49k estrellas y +7.3k forks. Su éxito y supervivencia se basa en estos tres pilares de Código Limpio:

Importancia	Explicación en el Contexto de Mastodon
Mantenibilidad	Los bugs en el sistema de federación o en la API deben arreglarse rápido. Un código limpio (bien organizado en sus



	Services y Models) permite a un desarrollador localizar el error sin tener que leer el 100% del proyecto.
Escalabilidad	Mastodon necesita manejar millones de posts y notificaciones. Su arquitectura está dividida (Ruby on Rails para la lógica principal y Node.js para el <i>streaming</i> en tiempo real). El código limpio en el backend (RoR) permite agregar funciones sin afectar al rendimiento del <i>streaming</i> .
Trabajo en Equipo	Con cientos de colaboradores, el código limpio es la norma de comunicación. Un <i>Pull Request</i> (PR) se revisa más rápido si el código es legible y sigue los estándares. Es un acto de cortesía con el compañero.

Código Sucio vs Código Limpio (La Lógica del "Toot")

Imaginemos que estamos en el modelo Status (un post de Mastodon). Queremos aplicar una lógica de visibilidad.

Código Sucio (Múltiples responsabilidades)	Código Limpio (Responsabilidad Única)
Una función gigante en el controlador que verifica el tipo de cuenta, el estado de privacidad del post y la visibilidad de archivos adjuntos.	Un Service Object llamado ProcessStatusVisibilityService que delega la verificación de medios a una función separada como AttachmentPolicy.check(media).
Problema: Si algo falla en la visibilidad del archivo, tienes que depurar toda la función de publicación.	Ventaja: Cada parte hace una cosa. El controlador solo interactúa con el Service, y el Service se encarga de llamar a funciones pequeñas y enfocadas.

Principios Clave en la Práctica con Mastodon

- DRY (Don't Repeat Yourself):** Usan Ruby on Rails que, por naturaleza, promueve la reutilización. Por ejemplo, la lógica de interacción con la base de datos se centraliza en los Modelos (app/models), no se repite en cada controlador.
- KISS (Keep It Simple, Stupid):** A pesar de la complejidad de la federación, el *core* de la aplicación sigue el patrón MVC clásico. No se complican con una arquitectura de microservicios si no es estrictamente necesario, manteniendo la arquitectura central relativamente simple de entender.



B. Comprensibilidad y Nombres

- En lugar de crear un controlador gigante (PostsController), Mastodon aísla la lógica en Service Objects.
 - **Ejemplo:** Mira la carpeta app/services. Encuentras clases con nombres como PostStatusService o AccountDeleteService.
 - **Resultado:** El nombre de la clase ya te dice exactamente qué hace y dónde buscar si algo falla en ese proceso. (Principio de Nombres Explícitos).

C. Funciones y Comentarios

- **Funciones Pequeñas (SRP):** Aquí se aplica el principio de Responsabilidad Única a nivel de función:
 1. El Controller recibe la petición (hace una cosa).
 2. Llama al Service (PostStatusService).
 3. El Service (que coordina la lógica de negocio) llama a muchas funciones privadas muy pequeñas para hacer cosas como: notify_followers(), federate_status(), schedule_attachments().
 - Esto garantiza que cada método en el código solo haga una operación específica y sea fácil de probar.
- **Comentarios:** En el código bien escrito de Mastodon, los comentarios son escasos y se centran en el "Por qué" de una decisión de diseño compleja (por ejemplo, por qué se usa una implementación específica del protocolo ActivityPub), no en el "Qué" hace una línea de código.

D. Organización de Código, Objetos y Estructuras

Mastodon está organizado en capas:

Capa	Ubicación en el Repo	Rol y Principio
Persistencia	app/models	Encapsula la lógica de la base de datos (PostgreSQL).
Frontend	app/javascript/mastodon	Es una aplicación React separada. Se comunica vía API, logrando un alto desacoplamiento.
Tareas Asíncronas	app/workers (Sidekiq)	Maneja procesos pesados como el envío de correos o la



	federación. Evita la rigidez y la lentitud del sistema principal.
--	---

- **Pruebas:** Mastodon tiene una carpeta spec masiva, donde se encuentran sus pruebas unitarias y de integración (usando RSpec).
 - Estas pruebas son el seguro de vida del proyecto. Permiten al equipo de Mastodon refactorizar (limpiar) el código antiguo sin introducir regresiones ni miedo a romper el sistema.
- **Olores de Código (*Code Smells*):** El equipo usa herramientas de análisis estático como RuboCop (el linter para Ruby).
 - **Función:** RuboCop "huele" los problemas: clases demasiado largas, funciones con demasiados parámetros, o nombres de variables confusos.
 - **Resultado:** Si el código enviado por un colaborador tiene olor, el *workflow* de GitHub (GitHub Actions) automáticamente rechaza el PR. Esto obliga a los desarrolladores a limpiar el código antes de que se fusionen, manteniendo la base de código sana.