**ESCUELA POLITÉCNICA NACIONAL**

**FACULTAD DE INGENIERÍA DE SISTEMAS**

**CONSTRUCCIÓN Y EVOLUCIÓN DE SOFTWARE**

**Nombre:** Belén Cholango.
**Curso:** GR2SW.

**Taller. Clase 06.**

# Código Limpio

## Objetivos

- Aplicar los principios de código limpio en el desarrollo de software con el fin de mejorar la calidad, legibilidad y mantenibilidad del código en proyectos reales.
- Comprender la importancia del código limpio y su impacto en escalabilidad y trabajo colaborativo dentro de un proyecto.
- Identificar áreas de mejora en fragmentos de código mediante la detección de malas prácticas, olores de código y posibles problemas de diseño y proponer refactorizaciones que permitan optimizar la estructura, claridad y organización del código sin alterar su funcionalidad.

## Introducción

El código limpio se refiere a un estilo de programación orientado a producir software claro y legible. Su objetivo es convertir el código funcional en software confiable, fácil de mantener y un placer para los equipos [1]. Este concepto enfatiza la simplicidad, la coherencia y la responsabilidad bien definida de cada componente del sistema. Adoptar código limpio reduce la complejidad innecesaria y permite construir soluciones más estables y sostenibles.

## Impacto en la mantenibilidad, escalabilidad y trabajo en equipo

El impacto es totalmente positivo ya que la aplicación de código limpio mejora significativamente la mantenibilidad al facilitar la detección de errores y la incorporación de nuevas funciones mediante funciones claras, nombres descriptivos y responsabilidades bien definidas. También favorece la escalabilidad, un sistema modular y desacoplado permite añadir funcionalidades sin modificar grandes partes del código ni afectar su rendimiento. Además, en entornos de trabajo colaborativo, el código limpio agiliza la comprensión del proyecto, reduce malentendidos y acelera la integración de nuevos desarrolladores gracias a convenciones consistentes y estructuras organizadas.

## Ejemplos de “código sucio” vs. “código limpio”

### Código sucio

```
function a(b,c){let d=0;for(let
i=0;i<b.length;i++){d+=b[i].p*c;}if(d>50){d=d*0.9;}d+=5;return d;}
```

Este fragmento presenta problemas como nombres poco representativos, uso de una sola función para múltiples tareas y escasa claridad en la lógica.

```
function calcularTotal(items, cantidades) {
  const subtotal = calcularSubtotal(items, cantidades);
```

```
  const descuento = aplicarDescuento(subtotal);
  const envio = calcularEnvio(subtotal);
  return subtotal - descuento + envio;
}
```

En este caso, la función utiliza nombres claros, delega tareas a funciones específicas y mejora la legibilidad general.

**Principios clave**

- **SOLID**: conjunto de reglas orientadas a objetos que promueven la responsabilidad única, la extensibilidad y el bajo acoplamiento.
- **DRY (Don't Repeat Yourself)**: evita repetir código o lógica innecesaria.
- **KISS (Keep It Simple)**: privilegia soluciones simples sobre implementaciones innecesariamente complejas.
- **Separación de responsabilidades**: cada módulo debe enfocarse en una tarea específica.

**Reglas generales de diseño**

- **Alta cohesión y bajo acoplamiento**: los módulos deben relacionarse lo menos posible entre sí, pero internamente conservar coherencia.
- **Encapsulamiento de detalles**: se deben ocultar implementaciones internas y exponer solo lo necesario.
- **Interfaces claras**: las funciones y clases deben contar con entradas y salidas bien definidas.
- **Evitar números mágicos**: usar constantes con nombre para evitar confusión.

**Comprensibilidad y nombres**

Los nombres de funciones, variables y clases deben reflejar claramente su propósito. Entre las buenas prácticas se encuentran usar verbos en funciones como calcularTotal, obtenerUsuario; emplear plurales en colecciones como productos, clientes; evitar abreviaciones y siglas sin significado claro y utilizar nombres que representen la intención como isValido, tienePermiso.

**Funciones y comentarios**

Las funciones limpias deben ser cortas y centradas en una sola responsabilidad, fáciles de leer gracias a parámetros claros y pocos argumentos y consistentes en su nivel de abstracción.
El código bien escrito debería explicar "qué" hace por sí mismo; los comentarios deben aclarar "por qué" se tomó cierta decisión.

**Organización de código, objetos y estructuras**

Se recomienda dividir por capas o por características del sistema (controllers, services, repositories o módulos por funcionalidad), mantener archivos pequeños y con responsabilidades únicas, utilizar clases u objetos que agrupen datos y comportamiento relacionado y evitar objetos anémicos, donde los datos están separados de su lógica natural.

Patrones como DAO, Repository, Factory o Service Layer apoyan una arquitectura más clara y escalable.

**Pruebas y olores de código**

Las pruebas son fundamentales para garantizar que el sistema se mantenga estable a pesar de cambios o actualizaciones.

- **Pruebas unitarias**: validan funciones individuales.
- **Pruebas de integración**: verifican interacción entre módulos.
- **Pruebas end-to-end**: evalúan el flujo completo del sistema.

**Olores de código (code smells)** comunes:

- Funciones demasiado largas.
- Código duplicado.
- Clases con demasiadas responsabilidades.
- Condicionales profundas.
- Variables sin un propósito claro.
- Nombres ambiguos.
- Dependencias globales o no controladas.

Identificar estos olores permite refactorizar a tiempo y evitar problemas futuros.

**Desarrollo**

El repositorio elejido fue: freeCodeCamo - https://github.com/freeCodeCamp/freeCodeCamp

**Archivos elegidos:**
client/src/components/settings/certification.tsx
client/src/components/settings/email.tsx

**Preguntas de análisis:**

**¿Los nombres son claros?**

El archivo “certification.tsx” tiene buenos nombres como “createCertifiedMap”, “handleSolutionModalHide”, “getProjectSolution”, “isCertifiedMap”, pero existen nombre ambiguos como “e” para eventos cuando podría ser “event”. Tambien a “t” para la función de traducción y no es descriptivo.

Con el archivo “email.tsx” hay buenos nombre como “createHandleEmailFormChange”, “getValidationForNewEmail”, “isEmailVerified” pero también tiene nombres poco claros como “t” para traducción “e” para eventos y “maybeEmailRE” donde “RE” no es obvio que significa "Regular Expression".

Por lo tanto, en su mayoría son nombres claros, pero hay nombres de variables temporales y funciones auxiliares que podrían mejorar en cuanto a claridad de nombre.

**¿Las funciones son cortas y hacen una sola cosa?**

En “certification.tsx” hay una función muy larga llamada “getProjectSolution” maneja múltiples responsabilidades como buscar proyectos, preparar datos y crear callbacks. Hay un componente muy largo “CertificationSettings” que maneja estado, renderizado y lógica de negocio

En “email.tsx” hay funciones cortas razonables, pero existe una función media llamada “EmailSettings” (principal, 110 líneas) que podría dividirse.

**¿Hay comentarios útiles o innecesarios?**

En “certification.tsx” solo hay 2 comentarios en 358 líneas dónde el primero indica confusión sobre tipos (code smell) y el segundo es útil.

En “email.tsx” no hay comentarios en 194 líneas y las funciones de validación complejas no tienen explicación. La lógica del formulario está sin documentación.

**¿Hay olores de código (duplicación, funciones largas, variables ambiguas)?**

En “certification.tsx” se encontró lo siguiente:

| Code Smell | Líneas | Descripción |
|---|---|---|
| **Función larga** | 193-237 | “getProjectSolution” con múltiples responsabilidades |
| **Duplicación** | 157-173 | Lógica de “handleClaim” repetida en “LegacyFullStack” y “ProjectsFor” |
| **Parámetros booleanos** | 66-88 | Mapa con 23 propiedades booleanas, por lo que es difícil de mantener. |
| **Demasiados estados** | 242-249 | 6 estados useState cuando debería usar useReducer |
| **Magic strings** | 285, 291 | “projectPreview”, “examResults” deberían ser constantes |
| **Componente anidado** | 250-271 | Componente Certification definido dentro de función |

**En email.tsx” se encontró:**

| Code Smell | Líneas | Descripción |
|---|---|---|
| **Validación duplicada** | 51-62, 64-78 | Lógica similar en dos funciones de validación |

| | | |
|---|---|---|
| **Lógica compleja en componente** | Todo | Validación mezclada con presentación |
| **Conditional rendering anidado** | 137-145 | Condicionales dentro de FormGroup pueden simplificarse |

**¿Cómo está organizada la estructura?**

En “certification.tsx” existe un import de 31 líneas que contiene constantes, redux mapping, helper function (createCertifiedMap) aunque parece estar mal ubicada, constante de mensaje, types, “LegacyFullStack Component”, “CertificationSettings Component”, Estado local con muchos “useState”, Funciones auxiliares, “getProjectSolution” y que es muy larga, “Certification” y “ProjectsFor” que son componentes anidados. Además, no hay separación clara entre lógica y presentación.

Sobre “email.tsx” hay importe de redux mapping, types, “EmailSettings Component”, uso de un solo Estado, Handlers, Validaciones. Entonces, las validaciones deberían estar en un archivo separado

**Mejoras y justificación:**

1. **Extraer componentes anidados a archivos separados en “certification.tsx”.**

Actualmente el archivo contiene componentes definidos dentro de la función principal CertificationSettings como "Certification" que muestra cada tarjeta de certificación y otro llamado "ProjectsFor" que renderiza la lista de proyectos y el botón de reclamar certificado. Esta práctica de definir componentes dentro de otros componentes es un anti-patrón.

La mejora consiste en extraer estos dos componentes anidados a sus propios archivos independientes, creando CertificationCard.tsx y ProjectList.tsx en una carpeta de componentes. Cada archivo tendría su propia interfaz de TypeScript con las props claramente definidas, su lógica aislada y su exportación nombrada. De esta forma, el componente principal CertificationSettings solo se encargaría de orquestar estos componentes hijos y manejar el estado general.

Al extraerlos a archivos separados, seguimos el Principio de Responsabilidad Única donde cada archivo tiene un propósito claro y específico.

2. **Refactorizar múltiples useState a useReducer en “certification.tsx”.**

El componente CertificationSettings actualmente maneja seis estados independientes usando useState: projectTitle, challengeFiles, challengeData, solution, examResults e isOpen. Todos estos estados están relacionados entre sí porque representan la información que se muestra en los diferentes modales del componente.

La mejora consiste en reemplazar estos seis useState por un único useReducer que maneje un objeto de estado cohesivo. Se definiría una interfaz TypeScript llamada ModalState que agrupe todos estos campos relacionados, y un conjunto de acciones con nombres descriptivos como OPEN_PROJECT_MODAL, OPEN_PREVIEW_MODAL, OPEN_EXAM_MODAL y CLOSE_MODAL. El reducer tendría la lógica centralizada para actualizar el estado basándose en estas acciones. Además, se eliminaría la función initialiseState que actualmente llama a cinco setters diferentes.

Con useReducer, garantizamos que todas las actualizaciones del estado del modal ocurren de forma atómica, es decir, o se actualizan todas las propiedades relacionadas o ninguna, evitando estados intermedios inválidos.

3. **Separar lógica de validación en utilidades y custom hooks en “email.tsx”.**

El componente EmailSettings contiene dos funciones grandes llamadas getValidationForNewEmail y getValidationForConfirmEmail que tienen toda la lógica de validación mezclada con el código del componente. Estas funciones verifican expresiones regulares, comparan emails, retornan estados de validación y mensajes de error, todo dentro del mismo archivo del componente.

La mejora consiste en extraer esta lógica de validación a dos capas separadas creando un archivo de utilidades (emailValidation.ts) que contenga funciones puras de validación que solo reciban strings y retornen objetos con isValid y errorKey, sin depender de React ni del estado del componente. Adicionalmente, se reemplazarían los strings mágicos como 'success' y 'error' por un enum llamado ValidationState, y la expresión regular maybeEmailRE tendría un nombre más descriptivo como EMAIL_REGEX con un comentario explicando su propósito.

Al extraer las validaciones a funciones puras, podemos testearlas sin necesidad de renderizar ningún componente React, lo que hace que los tests sean mucho más rápidos y simples de escribir.

4. **Eliminar código duplicado creando un custom hook compartido en “certification.tsx”.**

El archivo contiene código duplicado significativo como la misma lógica para manejar el reclamo de certificados aparece dos veces, una vez en el componente LegacyFullStack y otra vez en el componente ProjectsFor. Ambas implementaciones crean una función handleClaim que previene el comportamiento por defecto del evento, verifica si el usuario es honesto (isHonest), y luego o bien llama a verifyCert o muestra un mensaje flash. La única diferencia es que en LegacyFullStack es una función de orden superior que recibe el certSlug como parámetro.

La mejora consiste en encapsular esa replicación de lógica compartida en algo nuevo llamado useCertificationClaim. Además, el mensaje de honestidad (honestyInfoMessage) debería extraerse a un archivo de constantes en lugar de estar definido localmente en el archivo.

Tener el mismo código en dos lugares no solo duplica líneas innecesariamente, sino que crea un problema a futuro cuando se necesite mejorar el código y no se haga la mejora en ambos lugares, solo en uno.

5. **Reorganizar en arquitectura modular por capas de responsabilidad en ambos archivos.**

Ambos archivos son individuales y todo está mezclado en un solo archivo como sus imports, tipos TypeScript, funciones helper, lógica de negocio, componentes de presentación, y hasta funciones auxiliares que no tienen relación directa con la UI.

La mejora consiste en transformar cada funcionalidad en una arquitectura modular organizada por carpetas donde cada archivo tiene una responsabilidad única y clara. Para certification, se crearía una carpeta principal que contenga: una subcarpeta "components" con todos los componentes de presentación puros (CertificationCard, ProjectList, etc.), una subcarpeta "hooks" con toda la lógica de negocio encapsulada (useModalState, useCertificationClaim, useProjectSolution), una subcarpeta "utils" con funciones puras y helpers (createCertifiedMap, certificationHelpers), una subcarpeta "types" con todas las interfaces TypeScript, y una subcarpeta "constants" con valores inmutables como mensajes y configuraciones. El mismo patrón se aplicaría a email. El archivo principal de cada funcionalidad se reduciría a ser solo el que importa y compone estos módulos.

**Conclusiones**

El análisis de estos dos archivos de producción del repositorio freeCodeCamp ha mostrado que incluso en proyectos open-source exitosos y ampliamente utilizados, existen formas de mejora en términos de código limpio. Los principales problemas identificados son patrones comunes que afectan la mantenibilidad y escalabilidad del software. Este ejercicio demuestra que aplicar los principios de Clean Code no es solo teoría académica, sino una necesidad práctica que impacta directamente en la velocidad de desarrollo, la facilidad para corregir bugs, la capacidad de agregar nuevas funcionalidades y la colaboración efectiva en equipos.

**Bibliografía:**

[1] “A Guide to Clean Coding Principles,” Clean CodeGuy blog, 2025. https://cleancodeguy.com/blog/clean-coding-principles (último acceso: 26 nov 2025).

**[2]** OpenAI, “*ChatGPT*,” *OpenAI*, 2025. [Online]. Available: https://chat.openai.com. [Accessed: 24-Oct-2025].