GAÑANADA Nº 003

La función que hace absolutamente todo

. . .

CATEGORÍA	ARQUITECTURA CRIMINAL
GRAVEDAD	■■■□□ III — GRAVE
FECHA DEL HALLAZGO	30 DE SEPTIEMBRE DE 2026
ETIQUETAS	#responsabilidad-unica #complejidad #refactorizacion #legacy

§ 1

Descripción del hallazgo

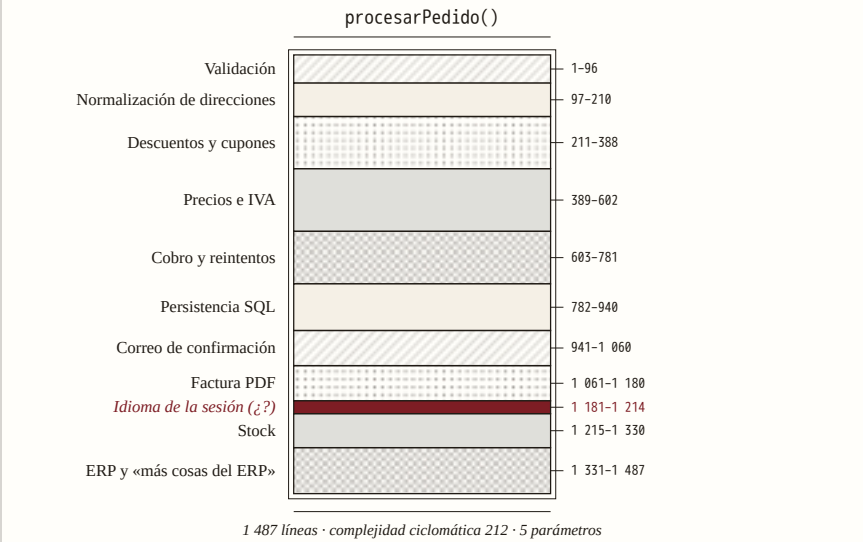
El espécimen responde al nombre de `procesarPedido` y tiene la siguiente firma:

```
export async function procesarPedido(
  pedido: any,
  opciones: any,
  modo: string,
  usuario?: any,
  flags?: Record<string, boolean>,
): Promise<any>;
```

Cinco parámetros, cuatro de ellos de tipo `any` y uno opcional que resulta ser obligatorio en once de los catorce caminos. El cuerpo ocupa 1 487 líneas. La herramienta de análisis estático le asigna una complejidad ciclomática de 212 y, a continuación, sugiere amablemente «considerar dividir la función».

Sus autores, conscientes del tamaño, le añadieron un índice en forma de comentarios:

```
// ===== SECCIÓN 1: VALIDACIÓN =====
// ===== SECCIÓN 4: PRECIOS (NO TOCAR IVA CANARIAS) =====
// ===== SECCIÓN 7: EMAIL =====
// ===== SECCIÓN 9: COSAS DEL ERP =====
// ===== SECCIÓN 9B: MÁS COSAS DEL ERP =====
```



Lám. 3.1 — Corte estratigráfico de `procesarPedido`, con la extensión de cada sección en líneas. La capa en rojo no figura en ningún requisito.

§ 2

Contexto del descubrimiento

En 2017, `procesarPedido` tenía cuarenta líneas y hacía exactamente lo que su nombre prometía. Después llegaron los requisitos. Cada uno, por separado, era razonable; cada uno, por separado, era «solo un `if` más».

La función tenía además una propiedad gravitatoria: como ya disponía de todo el contexto —el pedido, el usuario, la conexión a base de datos, los flags—, era el lugar más cómodo para añadir cualquier cosa. Crear una función nueva exigía decidir qué datos necesitaba. Añadir una sección a `procesarPedido`, no.

El hallazgo formal se produjo al cambiar el tipo de IVA de un producto. El cambio, de una línea, rompió el envío de correos de confirmación. La causa: una variable llamada `total` se reutilizaba en la sección 7 con otro significado.

§ 3

Consecuencias observadas

- **Imposible de probar de forma aislada.** Ejecutarla exige base de datos, pasarela de pago, servidor SMTP, sistema de archivos y una cola de mensajes. Su suite de tests consiste en un único test de extremo a extremo marcado como `@flaky`.
- **Conflictos de fusión permanentes.** Toda funcionalidad nueva toca la misma función, así que cada rama compite con todas las demás.
- **Revisiones de código ceremoniales.** Un *diff* de tres líneas en mitad de 1 487 se aprueba por fe, no por comprensión.
- **Acoplamiento accidental.** Variables reutilizadas entre secciones crean dependencias que no figuran en ningún diseño. Un cambio en precios rompe correos; un cambio en correos, nadie sabe todavía qué rompe.

§ 4

Recomendación de la academia

Una función debe tener una sola razón para cambiar. Cuando una función tiene secciones numeradas, ya no es una función: es un módulo sin el valor de serlo.

No conviene reescribirla desde cero: contiene años de reglas de negocio que nadie ha documentado. El camino seguro es incremental:

1. **Fija el comportamiento actual** con tests de caracterización: entradas reales, salidas observadas, aunque las salidas parezcan incorrectas.
2. **Extrae primero el cálculo puro** —validación, precios, impuestos—, que se prueba sin infraestructura.
3. **Aísla los efectos** —cobro, persistencia, correo, ERP— detrás de interfaces pequeñas.
4. **Deja una orquestación corta**, que se lea como la descripción del proceso:

```
export async function procesarPedido(pedido: Pedido): Promise<PedidoConfirmado> {
  const valido = validarPedido(pedido);
  const importe = calcularImporte(valido, tarifas);
  const cobro = await pasarela.cobrar(importe);
  await pedidos.guardar(valido, cobro);
  eventos.publicar({ tipo: 'pedido-confirmado', id: valido.id });
  return { id: valido.id, importe };
}
```

Los correos, las facturas y el ERP reaccionan al evento; ya no viven dentro del pedido. Para que la especie no vuelva a colonizar el código, establece límites de complejidad y de longitud en la integración continua (`complexity` y `max-lines-per-function` en ESLint; herramientas como lizard en C y C++) y trátalos como una conversación obligatoria, no como un número que se sube cuando estorba.



No trabajes así, no seas un gañán.

REFERENCIAS

1 Michael Feathers, «Working Effectively with Legacy Code» — *Tests de caracterización y búsqueda de costuras («seams») para refactorizar código sin red.*

2 ESLint — reglas «complexity» y «max-lines-per-function»

Separata del volumen I de la Bitácora del Gañán, publicada por AXXA.
https://axxa.embeddedsource.es/bitacora-del-ganan/003-la-funcion-que-hace-absolutamente-todo