



# Dominando el Patrón Template Method en Node.js

De la Duplicación de Código a la Abstracción Elegante

*Una guía práctica para desarrolladores intermedios y expertos.*

# El Problema: Un Déjà Vu en el Código

GeneradorReportePDF.js

```
class GeneradorReportePDF {  
    generar() {  
        this.autenticarUsuario();  
        const datos = this.obtenerDatos();  
        const pdf = this.formatearAPDF(datos);  
        this.guardarPDF(pdf);  
        this.enviarNotificacion();  
    }  
    // ... otros métodos  
}
```

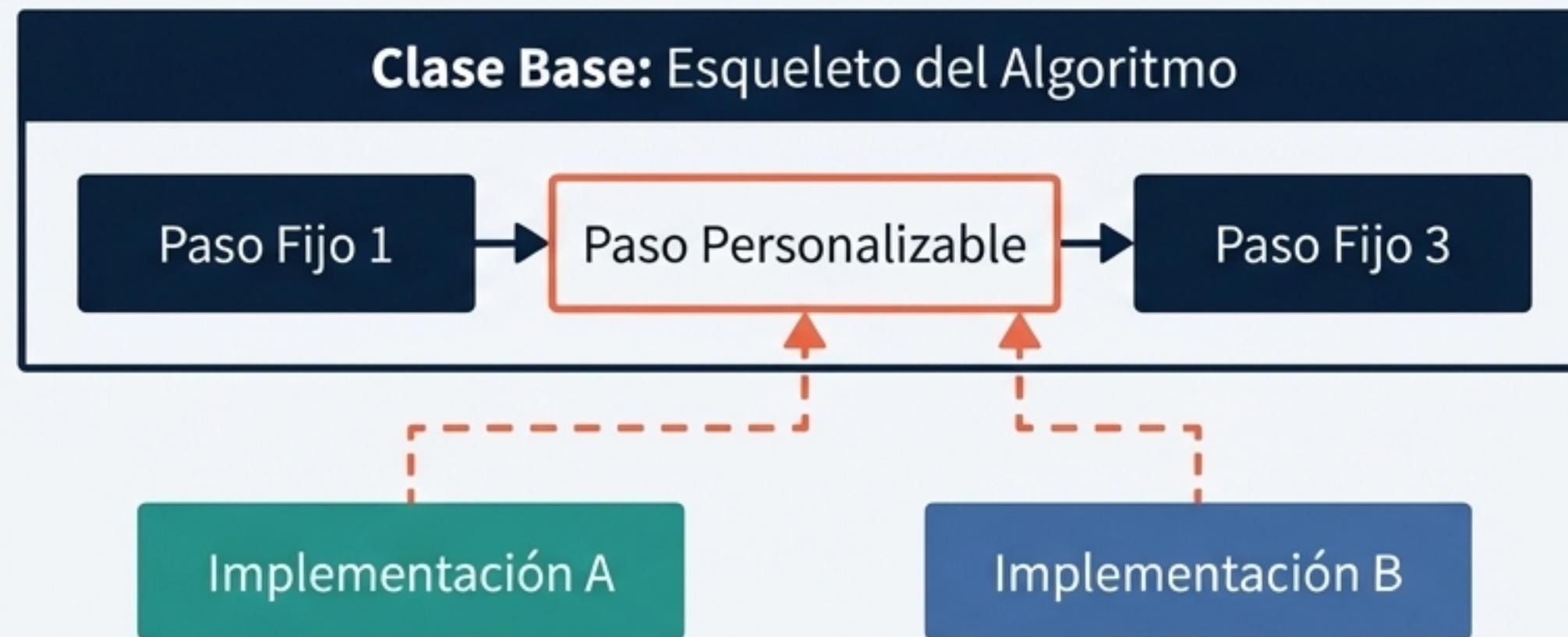
GeneradorReporteCSV.js

```
class GeneradorReporteCSV {  
    generar() {  
        this.autenticarUsuario();  
        const datos = this.obtenerDatos();  
        const csv = this.formatearACSV(datos);  
        this.guardarCSV(csv);  
        this.enviarNotificacion();  
    }  
    // ... otros métodos  
}
```

LÓGICA  
DUPLICADA  
LÓGICA  
ESPECÍFICA

¿Cómo eliminamos la duplicación sin sacrificar la flexibilidad para diferentes formatos?

# La Solución: El Patrón Template Method

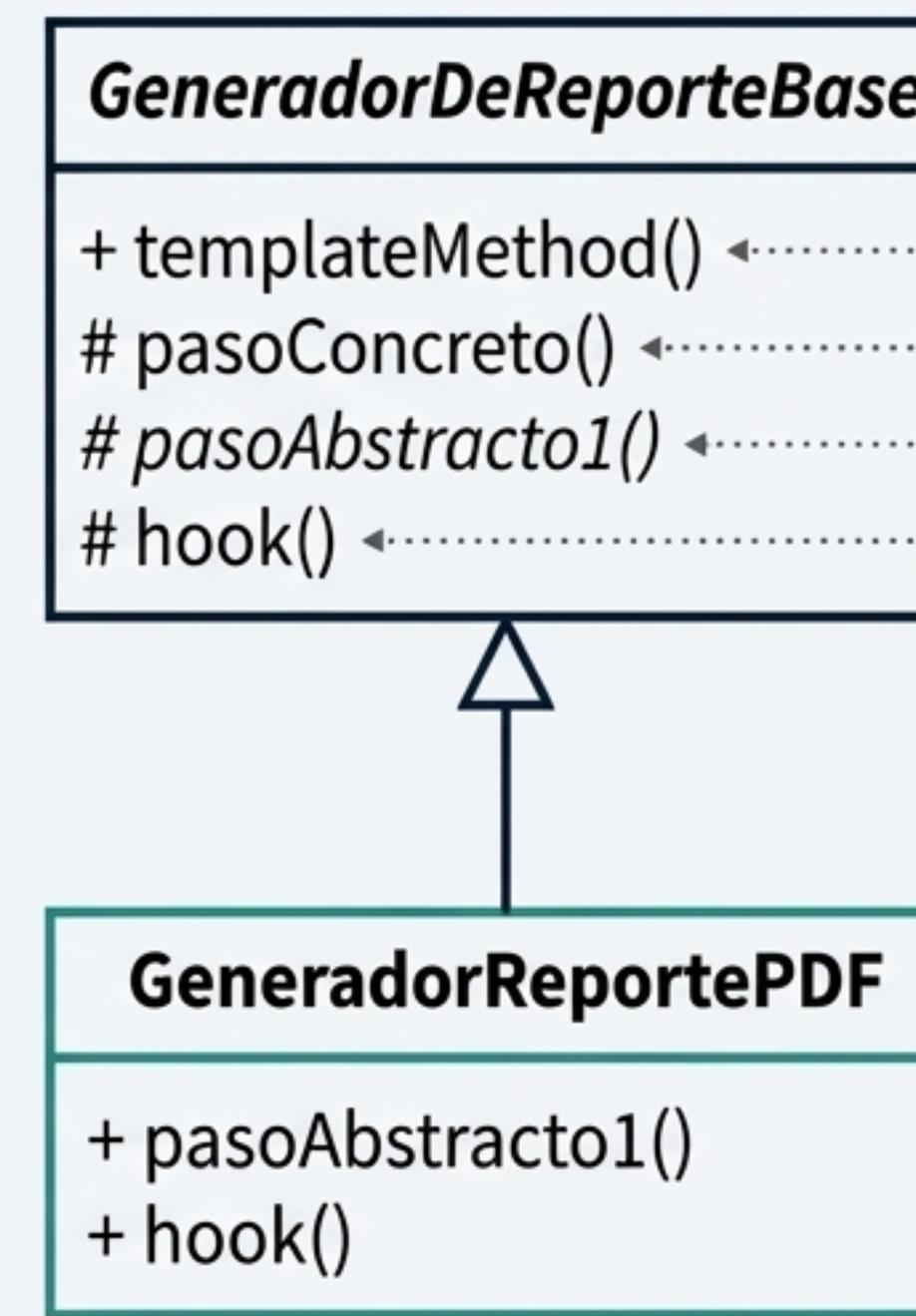


> El Patrón Template Method es un patrón de comportamiento que define el esqueleto de un algoritmo en una superclase, pero delega la implementación de pasos específicos a las subclases. La estructura general del algoritmo no cambia.

## Concepto Clave: Inversión de Control

También conocido como el **Principio de Hollywood**:  
**“No nos llames, nosotros te llamaremos.”**  
La clase base (el framework) controla el flujo del algoritmo y llama a las implementaciones de las subclases cuando es necesario, no al revés.

# Anatomía del Patrón: Estructura y Participantes



- ..... **EL ESQUELETO.** Define el orden de los pasos. Es **'final'** en algunos lenguajes para evitar su sobreescritura.
- ..... **PASO CONCRETO.** Implementación compartida por todas las subclases.
- ..... **PASO ABSTRACTO.** **Debe** ser implementado por cada subclase. Define un punto de variación obligatorio.
- ..... **HOOK (GANCHO).** Un paso opcional con una implementación por defecto (incluso vacía). Las subclases **pueden** sobreescribirlo para añadir comportamiento extra.

# Refactorizando, Paso 1: Crear la Clase Base Abstracta

Identificamos el algoritmo común y lo movemos a una clase base. Los pasos que varían se declaran como métodos abstractos que las subclases deberán implementar.

```
class GeneradorDeReportePDF {
  generar() {
    this.autenticarUsuario();
    const datos = this.obtenerDatos();
    const contenidoFormatado = this.formatearDatos(datos);
  }
  // Métodos concretos
  autenticarUsuario() { /* ... lógica común ... */ }
  obtenerDatos() { /* ... lógica común ... */ }
  enviarNotificacion() { /* ... lógica común ... */ }
}
```

```
class GeneradorDeReporteExcel {
  generar() {
    this.autenticarUsuario();
    const datos = this.obtenerDatos();
    const contenidoFormatado = this.formatearDatos(datos);
  }
  // Métodos concretos
  autenticarUsuario() { /* ... lógica común ... */ }
  obtenerDatos() { /* ... lógica común ... */ }
  enviarNotificacion() { /* ... lógica común ... */ }
}
```

```
class GeneradorDeReporteBase {
  // El Template Method
  generar() {
    this.autenticarUsuario();
    const datos = this.obtenerDatos();
    const contenidoFormatado = this.formatearDatos(datos);
    this.guardarArchivo(contenidoFormatado);
    this.enviarNotificacion();
  }
  // Métodos concretos (lógica duplicada)
  autenticarUsuario() { /* ... lógica común ... */ }
  obtenerDatos() { /* ... lógica común ... */ }
  enviarNotificacion() { /* ... lógica común ... */ }
  // Métodos abstractos (a ser implementados por subclases)
  formatearDatos(datos) {
    throw new Error('Debe ser implementado por una subclase');
  }
  guardarArchivo(contenido) {
    throw new Error('Debe ser implementado por una subclase');
  }
}
```

# Refactorizando, Paso 2: Implementar las Clases Concretas

Las clases hijas ahora solo se preocupan de sus responsabilidades únicas.  
El código es más limpio, más corto y más fácil de mantener.

Antes

GeneradorReportePDF.js

```
class GeneradorDeReportePDF {
    generar() {
        this.autenticarUsuario();
        const datos = this.obtenerDatos();
        const contenidoFormateado = this.formatearDatos(datos);
        this.guardarArchivo(contenidoFormateado);
        this.enviarNotificacion();
    }
    autenticarUsuario() { /* ... lógica común ... */ }
    obtenerDatos() { /* ... lógica común ... */ }
    formatearDatos(datos) {
        // Lógica para convertir datos a PDF...
        return pdfContent;
    }
    guardarArchivo(pdfContent) {
        // Lógica para guardar el archivo .pdf...
    }
    enviarNotificacion() { /* ... lógica común ... */ }
}
```

Lógica de orquestación



Sobrecargado

Después

GeneradorReportePDF.js

```
class GeneradorReportePDF extends GeneradorDeReporteBase {
    formatearDatos(datos) {
        // Lógica para convertir datos a PDF...
        return pdfContent;
    }
    formatearDatos(datos) {
        // Lógica para convertir datos a PDF...
        return pdfContent;
    }
    guardarArchivo(pdfContent) {
        // Lógica para guardar el archivo .pdf...
    }
}
```



De 25 líneas a solo 8. Toda la lógica de orquestación ha sido abstraída.

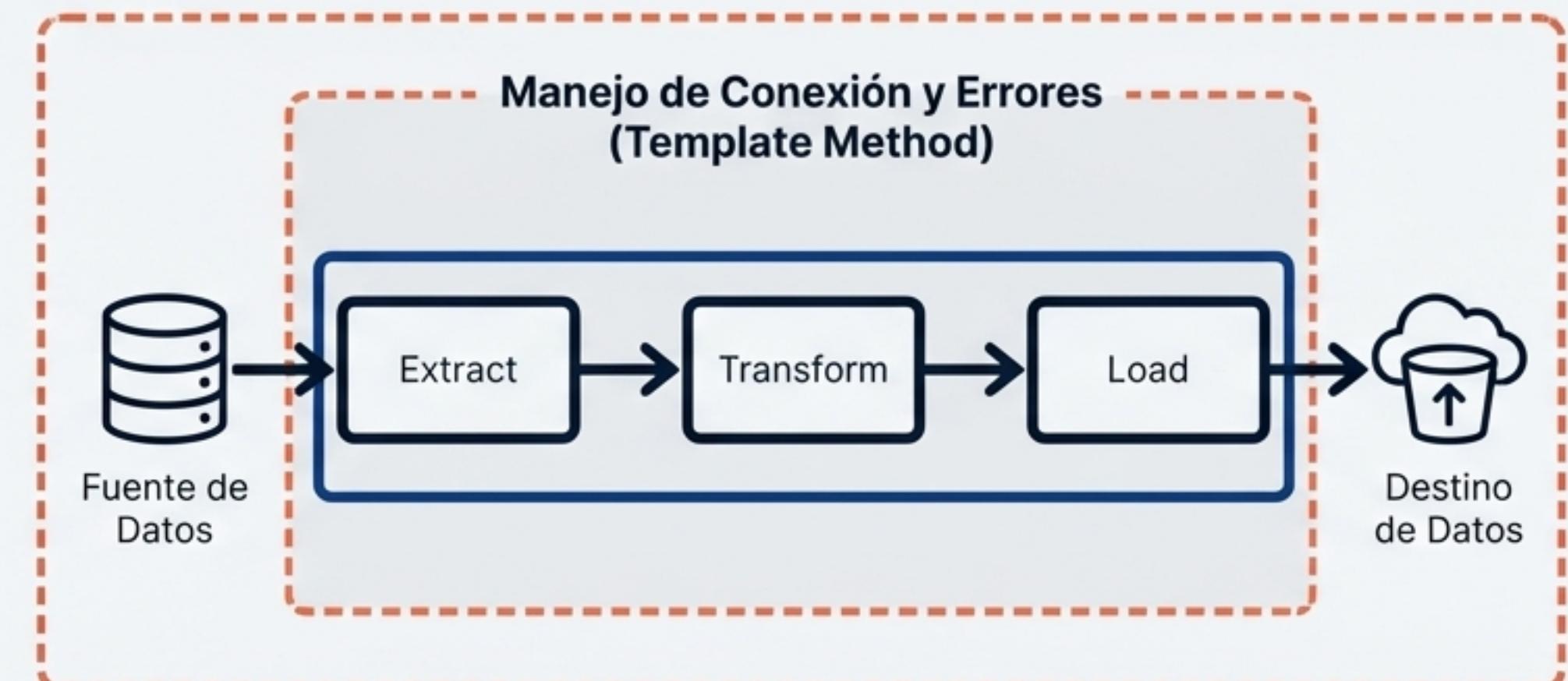
# Nivel Experto: Un Pipeline de ETL Asíncrono en Node.js

## Escenario

- Necesitamos un sistema para procesos de Extract, Transform, Load (ETL).
- El objetivo es mover datos desde múltiples fuentes (PostgreSQL, MongoDB API) a múltiples destinos (Amazon S3, Google BigQuery).

## El Reto Técnico

- La orquestación del proceso (conectar, extraer, transformar, cargar, desconectar, manejar errores) es siempre la misma.
- Los detalles de implementación para cada fuente y destino son completamente diferentes y asíncronos.



# La Abstracción: `AbstractETLProcess.js`

```
class AbstractETLProcess {  
    // El Template Method asíncrono  
    async run() {  
        try {  
            await this.connect();  
            const extractedData = await this.extract();  
            const transformedData = await this.transform(extractedData);  
            await this.load(transformedData);  
            await this.onSuccessHook(); // Hook opcional  
        } catch (error) {  
            console.error('Fallo el proceso de ETL:', error);  
            await this.onErrorHook(error); // Hook opcional  
        } finally {  
            await this.disconnect(); // Se ejecuta siempre  
        }  
    }  
  
    // Pasos abstractos asíncronos  
    async connect() { throw new Error('...'); }  
    async extract() { throw new Error('...'); }  
    async transform(data) { return data; } // Hook con implementación base  
    async load(data) { throw new Error('...'); }  
    async disconnect() { throw new Error('...'); }  
  
    // Hooks opcionales  
    async onSuccessHook() {}  
    async onErrorHook(error) {}  
}
```

Garantiza la limpieza de recursos, una ventaja clave del patrón en operaciones con I/O.

# La Implementación: `PostgresToS3.js`

Una clase concreta que hereda de `AbstractETLProcess` e implementa los pasos utilizando las librerías `pg` para PostgreSQL y `aws-sdk` para S3.

```
import { Client } from 'pg';
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';

class PostgresToS3ETL extends AbstractETLProcess {
  async connect() {
    this.pgClient = new Client({ /* ...config... */ });
    await this.pgClient.connect();
    this.s3Client = new S3Client({ /* ...config... */ });
  }

  async extract() {
    const result = await this.pgClient.query('SELECT * FROM sales_data');
    return result.rows;
  }

  async load(data) {
    const command = new PutObjectCommand({
      Bucket: 'my-data-bucket',
      Key: `sales-${new Date().toISOString()}.json`,
      Body: JSON.stringify(data),
    });
    await this.s3Client.send(command);
  }

  async disconnect() {
    await this.pgClient.end();
  }
}
```

# Análisis: Ventajas y Desventajas

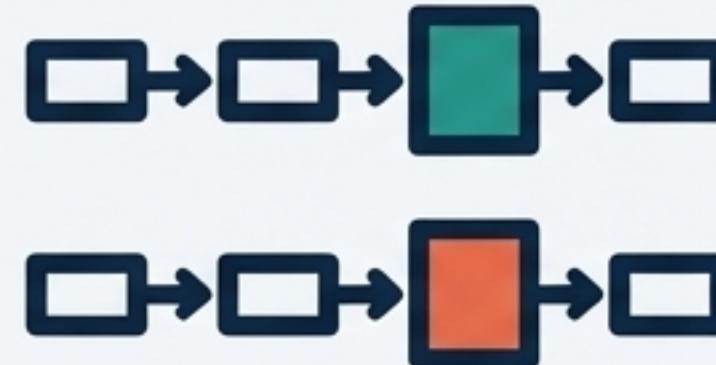
## ✓ Ventajas

- **Reutilización de Código:** Centraliza el esqueleto del algoritmo, cumpliendo con el principio DRY (Don't Repeat Yourself).
- **Estructura Forzada:** Garantiza que las subclases adhieran a la estructura del algoritmo definido, reduciendo errores.
- **Extensibilidad:** Añadir nuevas variaciones es simple: solo se necesita crear una nueva subclase.

## ⚠ Desventajas

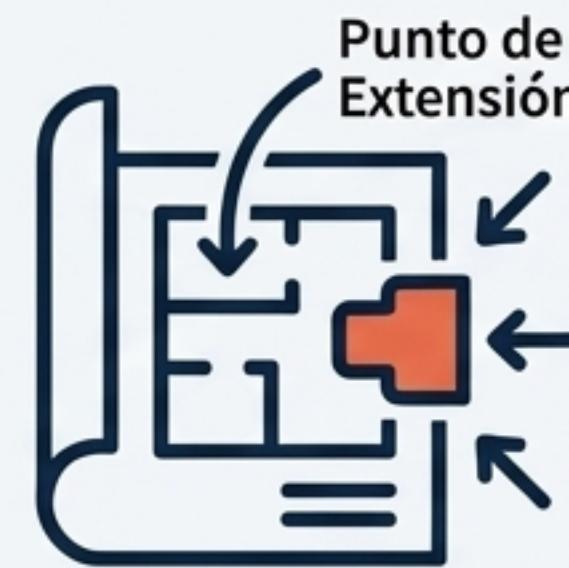
- **Rigidez:** La estructura del esqueleto está fija en la clase base y es difícil de alterar para una subclase específica.
- **Complejidad de la Jerarquía:** Puede llevar a un árbol de herencia profundo y difícil de navegar si hay muchas variaciones.
- **Possible Violación del Principio de Sustitución de Liskov:** Las subclases pueden introducir un comportamiento que rompa las suposiciones de la clase base.

# ¿Cuándo Usar el Patrón Template Method?



## Para Algoritmos Similares

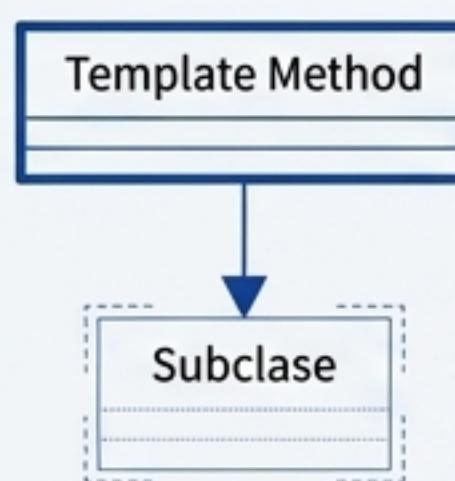
Cuando tienes múltiples clases que implementan algoritmos con pasos idénticos pero detalles diferentes. Es una señal clara de que el refactoring 'Form Template Method' es aplicable.



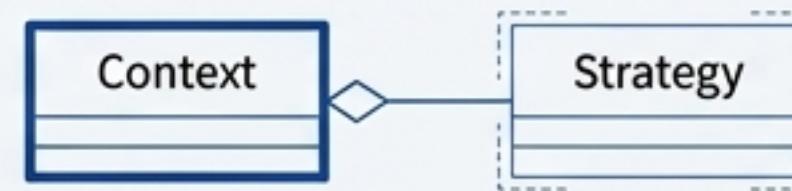
# En el Ecosistema de Patrones de Diseño

## Template Method vs. Strategy

La Diferencia Clave: Herencia vs. Composición



HERENCIA



COMPOSICIÓN

**Template Method:** Usa **herencia**. Varía partes de un algoritmo fijo. La relación se establece en **tiempo de compilación**.

**Strategy:** Usa **composición y delegación**. Permite cambiar el algoritmo completo en **tiempo de ejecución**. Es **más flexible**.

## Sinergia con Factory Method

Una Combinación Común



Es muy común que un **Template Method** utilice un **Factory Method** en uno de sus pasos.

Ejemplo: El esqueleto del algoritmo podría llamar a un método **crearConexion()** que actúa como una fábrica para devolver diferentes tipos de conexiones a bases de datos.

# Puntos Clave para Recordar

-  **Define el esqueleto, delega los detalles:** Esta es la esencia del patrón.
-  **Fomenta la reutilización de código** a través de la herencia y la centralización de la lógica común.
-  **Usa 'hooks'** para ofrecer puntos de extensión opcionales y flexibles sin forzar su implementación.
-  Es una **poderosa herramienta de refactoring** para eliminar la duplicación en flujos de trabajo compartidos.
-  **Ideal para lógica asíncrona en Node.js**, donde la orquestación y el manejo de recursos son críticos.

# Recursos y Próximos Pasos



## Código Fuente de los Ejemplos

Explora y ejecuta el código de esta presentación.

[github.com/your-repo/template-method-nodejs-example](https://github.com/your-repo/template-method-nodejs-example)



## Lectura Adicional

Profundiza en la teoría y otros patrones de comportamiento.

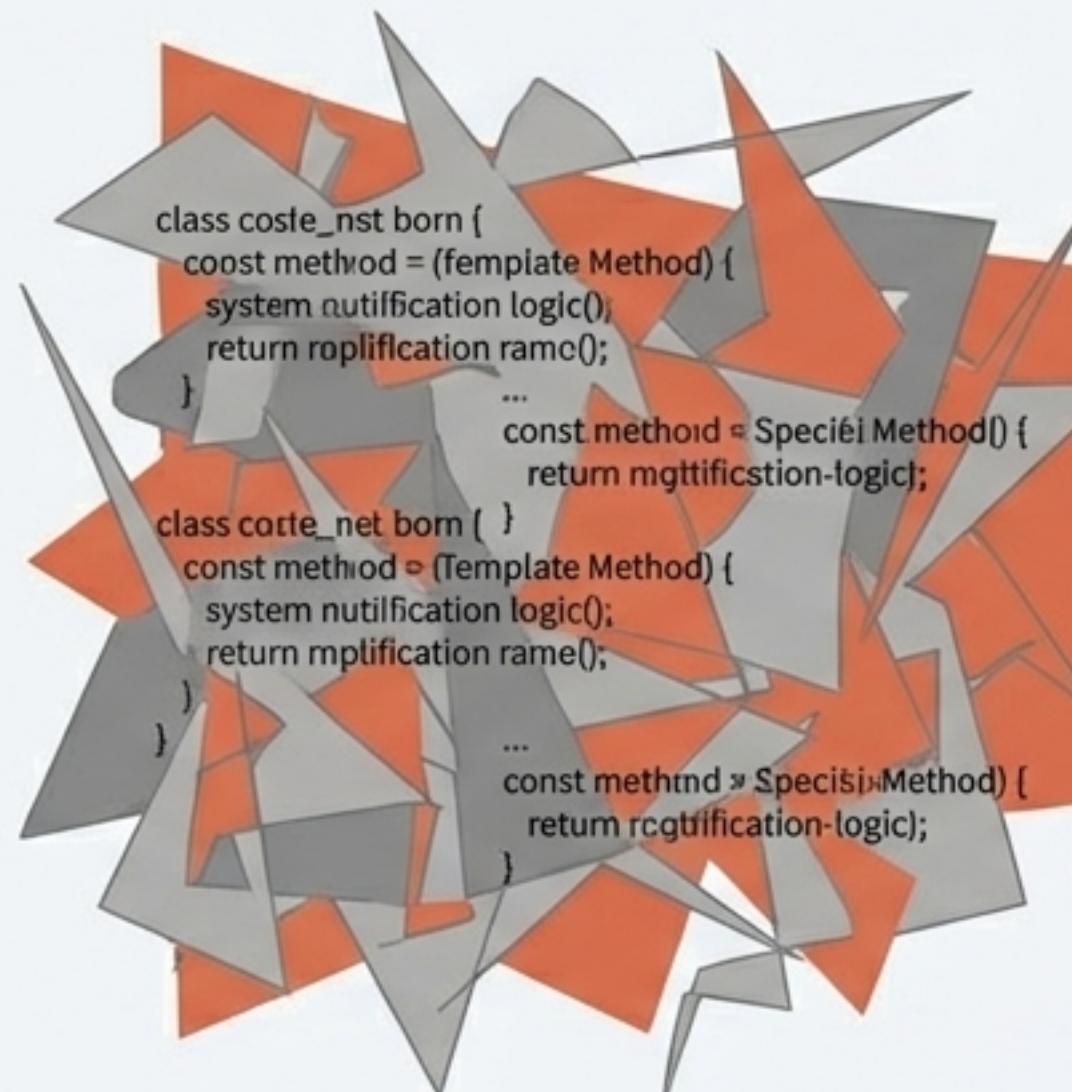
[refactoring.guru/design-patterns/template-method](https://refactoring.guru/design-patterns/template-method)



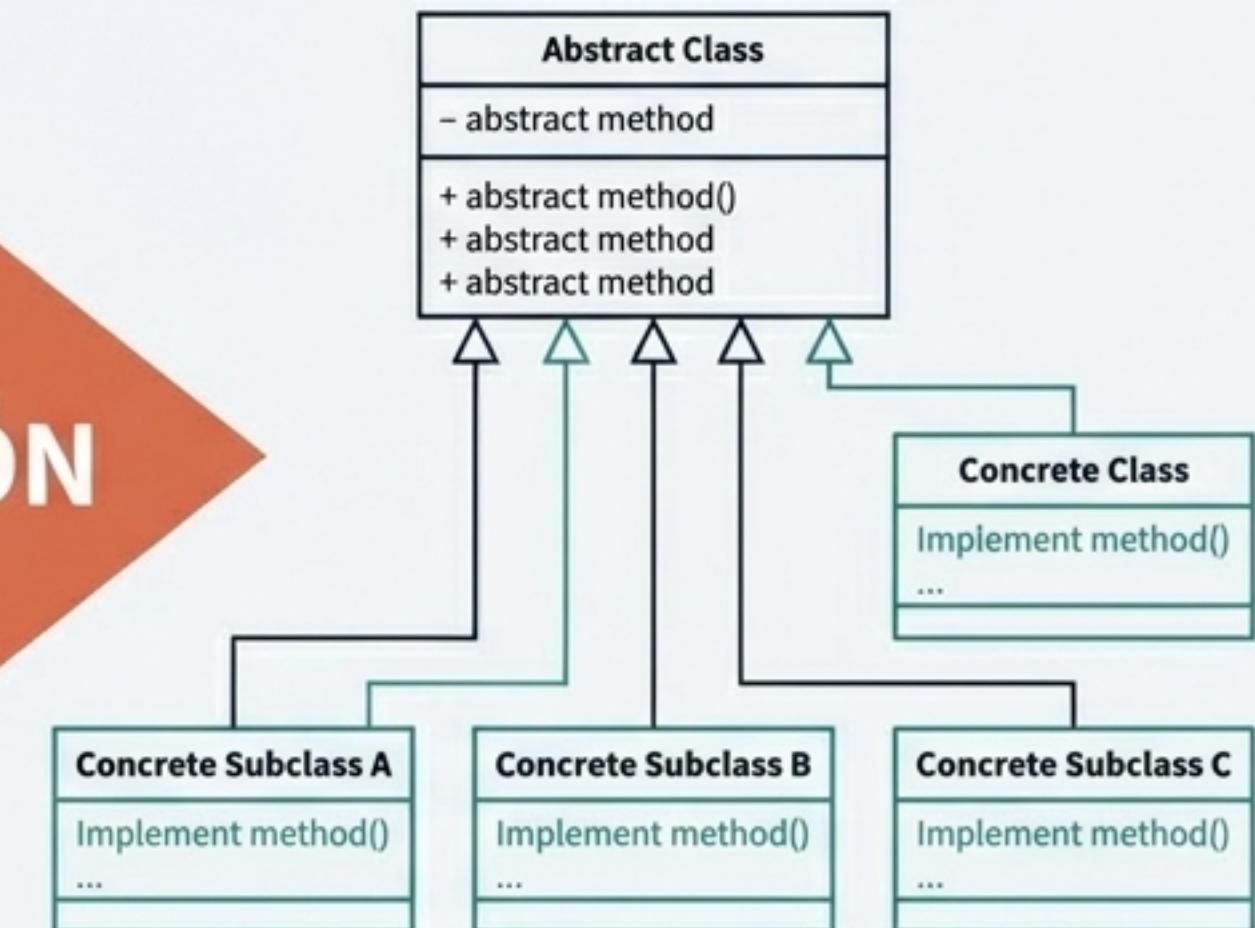
## Tu Turno

Desafío: Identifica en tu propio proyecto un lugar donde la refactorización "**Template Method**" pueda limpiar código duplicado.  
**¡Empieza por buscar dos métodos con pasos similares!**

# Transformación: Del Caos a la Claridad



REFACTORIZACIÓN



*Abstacta lo común, implementa lo específico.*