TESTING-US

VALIDÁ TU APRENDIZAJE

DRY
KISS
SOLID

CUADERNILLO DE ESTUDIO · NIVEL INTERMEDIO

Código y patrones

escribir código que se pueda cambiar

Código legible, principios DRY, KISS, YAGNI y SOLID, refactorización y señales de mal diseño, Git y revisión de código, y los patrones de diseño más usados, con ejemplos en JavaScript.

1 Código legible

2 SOLID

3 Refactorización

4 Git y revisión

5 Patrones

\+ Ejercicios con respuestas

Testing-US · Validá tu aprendizaje
testing-us.com

Edición octubre 2026

Antes de empezar

El código se escribe una vez y se lee muchas: lo leen tus compañeros, lo lee quien lo revisa y lo vas a leer vos dentro de seis meses. Este cuadernillo reúne las prácticas que hacen que un programa sea fácil de entender y de cambiar, y los patrones de diseño que resuelven problemas que aparecen una y otra vez.

CÓMO APROVECHARLO

- Tené a mano un proyecto propio (aunque sea chico) y buscá en él ejemplos de cada tema.
- Para cada principio o patrón, preguntate qué problema resuelve: eso es lo que se evalúa.
- Hacé los ejercicios del final de cada tema antes de mirar las respuestas.
- Cuando termines, rendí la evaluación **Buenas prácticas de código y patrones de diseño** en testing-us.com.

TIP Las buenas prácticas son criterios, no leyes. Si una regla complica más de lo que ayuda en tu caso, probablemente no es el momento de aplicarla.

Contenido

1. **Código legible**
   Nombres, funciones chicas, números mágicos, comentarios, DRY, KISS y YAGNI.
2. **Principios de diseño: SOLID**
   Cohesión y acoplamiento, los cinco principios, composición y herencia.
3. **Refactorización y señales de mal diseño**
   Refactorizar con pruebas, refactorizaciones frecuentes, code smells, deuda técnica.
4. **Control de versiones con Git y revisión de código**
   Commits, ramas, fusión, conflictos, pull requests e integración continua.
5. **Patrones de diseño**
   Singleton, Factory, Adapter, Decorator, Strategy, Observer y MVC.

TEMA 1

1 Código legible

LO QUE VAS A APRENDER

- Elegir nombres que expliquen.
- Escribir funciones chicas con una sola tarea.
- Reemplazar números mágicos y usar bien los comentarios.
- Aplicar DRY, KISS y YAGNI.

Nombres que explican

- Un buen nombre dice **qué es** o **qué hace**: `totalConIva` es mejor que `t2`, y `calcularEdad()` es mejor que `proc()`.
- Las **funciones** suelen nombrarse con un **verbo** (`enviarFactura`) y las variables con un **sustantivo** (`facturasPendientes`). Los valores verdadero/falso se leen bien como pregunta: `estaVencida`, `tienePermiso`.
- Conviene evitar abreviaturas raras y nombres que mienten: si una función se llama `obtenerUsuario` no debería además borrar datos.
- En un proyecto se respeta la **convención** del equipo y del lenguaje (en JavaScript, `camelCase` para variables y funciones, `PascalCase` para clases).

Funciones chicas y con una sola tarea

- Una función debería hacer **una sola cosa** y hacerla bien. Si para describirla hay que decir "y además...", probablemente son dos funciones.
- Pocos **parámetros**: con más de tres o cuatro cuesta recordar el orden; se puede pasar un objeto con nombres (`crearPedido({ cliente, items, envio })`).
- Las **condiciones complejas** se leen mejor con un nombre: `const puedeComprar = esMayor && tieneSaldo;`.
- Las **cláusulas de guarda** (salir temprano con `return` cuando algo no se cumple) evitan muchos niveles de `if` anidados.

Antes

```
function f(p, e) {
  if (e > 17) {
    return p * 1.21;
  }
  return 0;
}
```

✓ Nombres que explican

✓ Condición con nombre

Después

```
const IVA = 0.21;
function precioFinal(precio, edad) {
  const esMayor = edad >= 18;
  if (!esMayor) return 0;
  return precio * (1 + IVA);
}
```

✓ Constante en lugar de número mágico

✓ Cláusula de guarda: sale temprano

Figura 1.1 El mismo código, antes y después.

Números mágicos y comentarios

- Un **número mágico** es un valor suelto sin explicación (`precio * 1.21`). Se reemplaza por una **constante con nombre** (`const IVA = 0.21;` y `precio * (1 + IVA)`).
- Los **comentarios** explican el **por qué** (una decisión, una limitación, un caso raro), no repiten **qué** hace el código. Si hace falta un comentario para entender qué hace, suele convenir renombrar o dividir.
- Un comentario desactualizado es peor que ninguno: engaña.

TIP Antes de escribir un comentario que explique qué hace un bloque, probá darle a ese bloque un nombre: extraelo a una función.

DRY, KISS y YAGNI

- **DRY** (*Don't Repeat Yourself*, no te repitas): cada conocimiento debería estar en **un solo lugar**. Si la misma regla está copiada en cinco sitios, un cambio obliga a acordarse de los cinco.
- **KISS** (*Keep It Simple*, mantenelo simple): la solución más simple que resuelve el problema suele ser la mejor; la complejidad tiene que justificarse.
- **YAGNI** (*You Aren't Gonna Need It*, no lo vas a necesitar): no programar hoy funciones "por las dudas" para un futuro que quizás no llegue.
- Formato consistente (indentación, espacios, largo de línea): conviene que lo aplique una herramienta automática de formato, así nadie discute por estilo.

Para practicar

1. Proponé un nombre mejor para la función `proc(l)` que suma los importes de una lista de facturas.
2. Reescribí `if (u.r === 3)` sabiendo que 3 significa "administrador".
3. ¿Qué principio no se cumple si la regla del descuento está copiada en tres pantallas?
4. ¿Qué principio recomienda no agregar hoy un exportador a Excel que nadie pidió?

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

TEMA 2

2 Principios de diseño: SOLID

LO QUE VAS A APRENDER

- Distinguir cohesión y acoplamiento.
- Reconocer los cinco principios SOLID.
- Usar inyección de dependencias.
- Elegir entre composición y herencia.

Cohesión y acoplamiento

- **Cohesión alta:** lo que está dentro de un módulo o clase trabaja para un mismo objetivo.
- **Acoplamiento bajo:** los módulos dependen lo menos posible unos de otros, y mejor de **interfaces** (qué se puede pedir) que de detalles internos.
- Con cohesión alta y acoplamiento bajo, un cambio queda **localizado** y no rompe partes lejanas del sistema.
- Señales de acoplamiento alto: **variables globales** que cualquier parte del programa puede cambiar y **dependencias circulares** (A necesita a B y B necesita a A).

Los cinco principios SOLID

- **S — Responsabilidad única (SRP):** una clase o módulo debería tener **una sola razón para cambiar**. Una clase que calcula sueldos, arma el PDF del recibo y manda el mail tiene tres razones para cambiar.
- **O — Abierto/cerrado (OCP):** abierto a la **extensión**, cerrado a la **modificación**: agregar un nuevo medio de pago debería hacerse sumando código nuevo, no editando un `switch` gigante cada vez.
- **L — Sustitución de Liskov (LSP):** donde se usa una clase, tiene que poder usarse cualquiera de sus **subclases** sin sorpresas. Si una subclase rompe lo que el código esperaba de la clase base, la herencia está mal planteada.
- **I — Segregación de interfaces (ISP):** es mejor tener **interfaces chicas y específicas** que una enorme que obliga a implementar métodos que no se usan.
- **D — Inversión de dependencias (DIP):** el código de alto nivel (las reglas del negocio) no debería depender de detalles concretos (una base de datos en particular); ambos dependen de **abstracciones**. En la práctica, la dependencia se **recibe** desde afuera (**inyección de dependencias**) en lugar de crearse adentro. Eso facilita las pruebas: se puede pasar un objeto falso (un **doble de prueba**) en lugar de la base de datos real.

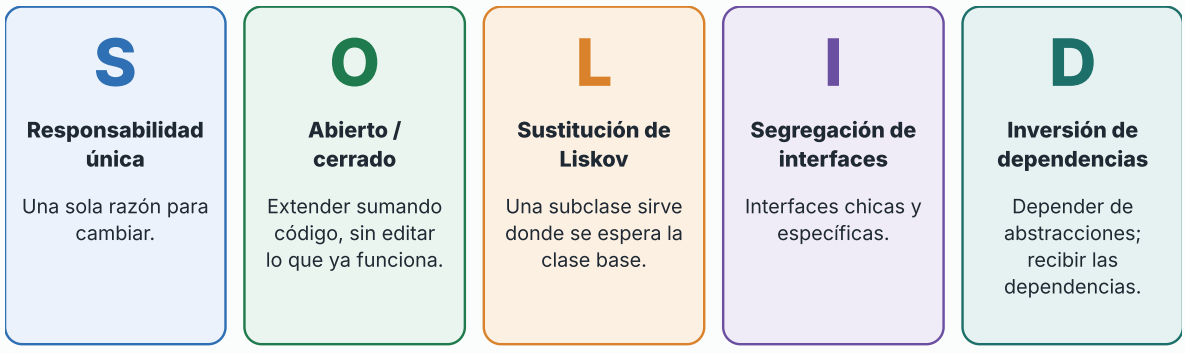


Figura 2.1 SOLID en una línea por principio.

Composición antes que herencia

- La **herencia** ("es un") acopla mucho: un cambio en la clase base afecta a todas las hijas.
- La **composición** ("tiene un"): un objeto usa a otros y delega en ellos. Suele dar diseños más flexibles: se pueden combinar comportamientos sin crear jerarquías profundas.
- Regla práctica: heredar solo cuando la relación "es un" es verdadera en todos los casos.

Por qué importan

- Los principios no son un fin: sirven para que el código sea **fácil de probar** y de **cambiar**. Aplicarlos a la fuerza en un programa chico agrega complejidad sin beneficio (ver KISS).

Para practicar

1. Una clase `Reporte` consulta la base, calcula totales y arma el PDF. ¿Qué principio no cumple y cómo la dividirías?
2. ¿Qué principio se aplica si un medio de pago nuevo se agrega creando una clase, sin tocar las existentes?
3. Un `Cuadrado` hereda de `Rectangulo`, pero al cambiarle el ancho también cambia el alto. ¿Qué principio puede romper?
4. ¿Cómo harías para probar una clase que crea por dentro la conexión a la base de datos?

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

TEMA 3

3 Refactorización y señales de mal diseño

LO QUE VAS A APRENDER

- Refactorizar sin cambiar el comportamiento.
- Usar las pruebas como red de seguridad.
- Reconocer code smells frecuentes.
- Manejar la deuda técnica.

Qué es refactorizar

- **Refactorizar** es cambiar la **estructura interna** del código **sin cambiar su comportamiento** visible, para que sea más fácil de entender y modificar.
- Se hace en **pasos chicos** y, entre paso y paso, se corren las **pruebas automáticas**: son la red de seguridad que avisa si algo se rompió.
- Sin pruebas, refactorizar es arriesgado: primero conviene escribir pruebas que fijen el comportamiento actual.
- Refactorizar no es agregar funciones nuevas: conviene no mezclar las dos cosas en el mismo cambio.
- Muchos editores traen **refactorizaciones automáticas** (renombrar en todo el proyecto, extraer función) que evitan errores de hacerlo a mano.

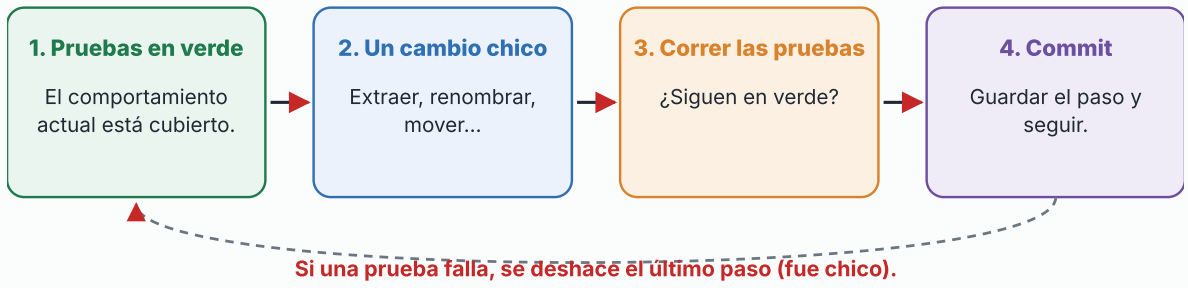


Figura 3.1 Pasos chicos y pruebas entre paso y paso.

Refactorizaciones frecuentes

- **Extraer función:** un bloque con un propósito claro pasa a ser una función con nombre.
- **Renombrar:** cambiar un nombre confuso por uno que explique.
- **Reemplazar número mágico por constante.**
- **Introducir objeto parámetro:** agrupar parámetros que siempre viajan juntos.
- **Reemplazar condicional por polimorfismo:** un `switch` sobre el tipo que se repite en varios lugares se reemplaza por clases (o funciones) que saben comportarse cada una a su manera.

Señales de mal diseño ("code smells")

- **Código duplicado:** la misma lógica copiada en varios lugares.
- **Función larga** y **clase grande:** hacen demasiadas cosas.
- **Lista larga de parámetros.**
- **Cirugía de escopeta:** un cambio chico obliga a tocar muchos archivos distintos.

- **Envidia de funcionalidad:** una función usa más los datos de otro objeto que los propios; probablemente debería vivir en ese otro objeto.
- **Comentarios que tapan código confuso.**
- **Código muerto:** funciones o variables que nadie usa. Se borran; si alguna vez hacen falta, se recuperan de la historia de Git (no hace falta dejarlas comentadas).
- Un "smell" no es un error: es una **pista** de que conviene mirar mejor.

Deuda técnica

- La **deuda técnica** es el costo futuro de elegir hoy una solución rápida pero desprolija. Como una deuda, genera "intereses": cada cambio posterior cuesta más.
- A veces endeudarse tiene sentido (llegar a una fecha), pero conviene **anotarla** y **pagarla** de a poco.
- La **regla del boy scout**: dejá el código un poco más limpio de lo que lo encontraste.

OJO Refactorizar sin pruebas es como hacer una instalación eléctrica sin cortar la luz: puede salir bien, pero el riesgo es innecesario.

Para practicar

1. Nombrá tres code smells y una refactorización para cada uno.
2. ¿Por qué conviene separar en dos commits una refactorización y una funcionalidad nueva?
3. Encontrás una función que nadie usa. ¿Qué hacés con ella?
4. ¿Qué harías primero antes de refactorizar un módulo heredado sin pruebas?

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

TEMA 4

4 Control de versiones con Git y revisión de código

LO QUE VAS A APRENDER

- Usar los comandos básicos de Git.
- Escribir buenos commits.
- Trabajar con ramas y resolver conflictos.
- Revisar código de forma útil.

Para qué sirve Git

- **Git** es un sistema de **control de versiones distribuido**: guarda la historia de los cambios del proyecto, permite volver atrás y que varias personas trabajen en paralelo.
- Un **commit** es una foto del estado del proyecto en un momento, con un **mensaje** que explica qué se hizo y por qué.
- Un **repositorio remoto** (en un servidor) permite compartir el trabajo: `git push` sube los commits y `git pull` trae los de los demás.

Comandos básicos

- `git status`: qué archivos cambiaron y cuáles están preparados para el próximo commit.
- `git add archivo`: prepara cambios para el próximo commit (área de preparación o *staging*).
- `git commit -m "mensaje"`: guarda lo preparado.
- `git log`: muestra la historia.
- `git diff`: muestra los cambios que todavía no se prepararon con `git add` (con `git diff --staged` se ven los ya preparados).
- El archivo `.gitignore` lista lo que **no** se versiona (dependencias descargadas, archivos generados, archivos con contraseñas o claves). Solo afecta a archivos que todavía no están versionados: si uno ya se subió, agregarlo después no lo saca de la historia.

Buenos commits

- **Chicos y con un solo propósito:** más fáciles de revisar y, si algo falla, de deshacer.
- Mensaje claro, breve y con contexto: "Corrige el cálculo del IVA en notas de crédito" dice más que "cambios".
- **Nunca** subir contraseñas ni claves al repositorio: aunque después se borren, quedan en la historia. Si pasa, hay que **cambiar la clave** de inmediato.

OJO Si una clave o contraseña llegó al repositorio, borrarla no alcanza: queda en la historia. Hay que cambiarla.

Ramas y fusión

- Una **rama** (*branch*) es una línea de trabajo independiente: se crea una rama para cada funcionalidad o corrección y la principal queda estable.
- Al terminar, la rama se **fusiona** (*merge*) con la principal. Si dos ramas cambiaron las mismas líneas (o una edita un archivo que la otra borró), aparece un **conflicto** que hay que resolver a mano, eligiendo o

combinando los cambios.

- `git revert` crea un commit nuevo que deshace otro anterior, sin reescribir la historia: es la forma segura de deshacer algo que ya se compartió.
- `git rebase` reescribe la historia de una rama para apoyarla sobre otra; no conviene reescribir historia que otras personas ya descargaron, porque su copia deja de coincidir y les genera conflictos.

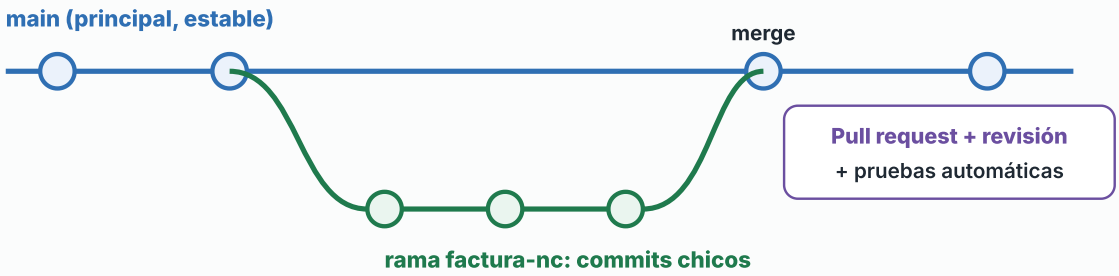


Figura 4.1 Una rama de funcionalidad que vuelve a la principal.

Pull requests y revisión de código

- Un **pull request** (o *merge request*) es el pedido de incorporar una rama a la principal; ahí el equipo **revisa** el código antes de fusionarlo.
- La **revisión de código** busca errores, mejora la legibilidad y comparte conocimiento. Se comenta el **código, no a la persona**, con sugerencias concretas.
- Muchos equipos practican **integración continua**: integran cambios chicos a la principal con frecuencia y cada cambio (por ejemplo, cada pull request) dispara una compilación y las **pruebas automáticas**; si fallan, no se fusiona.

Para practicar

1. ¿Qué comando usás para ver qué archivos cambiaste?
2. Mejorá este mensaje de commit: "fix".
3. ¿Qué hay que hacer cuando aparece un conflicto al fusionar?
4. Reescribí este comentario de revisión para que sea útil: "esto está mal".

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

TEMA 5

5 Patrones de diseño

LO QUE VAS A APRENDER

- Saber qué es un patrón de diseño.
- Reconocer patrones creacionales y estructurales.
- Reconocer Strategy, Observer y MVC.
- Saber cuándo no usar un patrón.

Qué es un patrón de diseño

- Un **patrón de diseño** es una **solución probada** a un problema de diseño que aparece seguido, descripta con un nombre, cuándo usarla y sus consecuencias. No es código para copiar: es una **idea** que se adapta y sirve en distintos lenguajes.
- El catálogo clásico ("Design Patterns", 1994, de Gamma, Helm, Johnson y Vlissides, la "banda de los cuatro") describe **23 patrones** en tres familias: **creacionales** (cómo se crean objetos), **estructurales** (cómo se combinan) y **de comportamiento** (cómo se reparten responsabilidades y se comunican).
- Tener un **vocabulario común** es una gran ventaja: decir "acá usamos un Observer" explica mucho en pocas palabras.

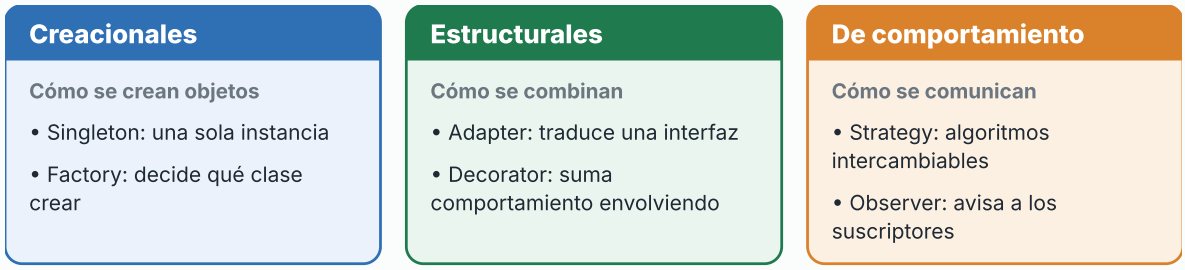


Figura 5.1 Las tres familias del catálogo clásico, con ejemplos.

Patrones creacionales

- **Singleton:** asegura que exista **una sola instancia** de una clase y da un punto de acceso global (por ejemplo, la configuración). Usarlo con cuidado: es un estado global y complica las pruebas.
- **Factory (fábrica):** un método o función se encarga de **crear objetos** y decidir de qué clase concreta, así el resto del código no depende de las clases concretas. Ejemplo: `crearNotificacion("sms")` devuelve un objeto que sabe enviar SMS. En el catálogo de 1994 aparece como **Factory Method** y **Abstract Factory**; la función que elige la clase según un parámetro es su versión más simple.

Patrones estructurales

- **Adapter (adaptador):** convierte la **interfaz** de una clase en otra que el código espera; como un adaptador de enchufe. Ejemplo: envolver una biblioteca de pagos externa para que responda a `cobrar(monto)`.
- **Decorator (decorador): agrega comportamiento** a un objeto envolviéndolo, sin modificar su clase. Ejemplo: un objeto que registra en un log cada llamada y después delega en el original.

Patrones de comportamiento

- **Strategy (estrategia):** define una familia de **algoritmos intercambiables** y permite elegir uno en tiempo de ejecución. Ejemplo: distintas formas de calcular el envío (`envioMoto`, `envioCorreo`) que se pasan como parámetro.
- **Observer (observador):** un objeto (sujeto) **avisa automáticamente** a todos los que se suscribieron cuando cambia su estado. Es la base de los eventos: `boton.addEventListener("click", manejador)`.

MVC

- **Modelo-Vista-Controlador** es un patrón de **arquitectura**: el **modelo** tiene los datos y las reglas, la **vista** los muestra y el **controlador** recibe las acciones del usuario y coordina. Separar responsabilidades permite cambiar la pantalla sin tocar las reglas del negocio.

Cuándo no usarlos

- Un patrón agrega piezas. Si el problema no existe todavía, usarlo "por las dudas" va contra **KISS** y **YAGNI**. Primero el problema, después el patrón.

TIP Primero el problema, después el patrón: si no podés decir qué problema resuelve, todavía no hace falta.

Para practicar

1. Una app manda avisos por mail, SMS o WhatsApp según lo que elija el cliente, y una función crea el objeto adecuado. ¿Qué patrón es?
2. Varias pantallas tienen que actualizarse cuando cambia el carrito de compras. ¿Qué patrón usarías?
3. Tenés que usar una biblioteca con una interfaz distinta de la que espera tu código. ¿Qué patrón aplica?
4. En MVC, ¿dónde va la regla "no se puede vender sin stock"?

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

PARA CONTROLAR

Respuestas de los ejercicios

Tema 1

1. Por ejemplo, `sumarImportes(facturas)` o `totalFacturado(facturas)`.
2. `const ROL_ADMIN = 3; if (usuario.rol === ROL_ADMIN)`, o mejor una función `esAdministrador(usuario)`.
3. DRY: la regla debería estar en un solo lugar.
4. YAGNI: no programar lo que todavía no se necesita.

Tema 2

1. Responsabilidad única: separar el acceso a datos, el cálculo y la generación del PDF en piezas distintas.
2. Abierto/cerrado: se extiende sumando código nuevo.
3. Sustitución de Liskov: el código que espera un rectángulo con ancho y alto independientes se rompe.
4. Recibiendo la conexión desde afuera (inyección de dependencias), para pasarle una falsa en la prueba.

Tema 3

1. Por ejemplo: código duplicado → extraer función; lista larga de parámetros → objeto parámetro; número mágico → constante con nombre.
2. Porque si algo falla es más fácil saber qué lo causó, revisar cada cambio y deshacerlo.
3. La borro: si alguna vez hace falta, se recupera de la historia de Git.
4. Escribir pruebas que fijen su comportamiento actual.

Tema 4

1. `git status` (y `git diff` para ver el detalle).
2. Por ejemplo: "Corrige el redondeo del total en el resumen de cuenta".
3. Abrir los archivos marcados, decidir qué cambios quedan (o combinarlos), probar y confirmar la fusión.
4. Por ejemplo: "Si el carrito está vacío, esta función devuelve undefined; ¿la hacemos devolver 0?".

Tema 5

1. Factory.
2. Observer: las pantallas se suscriben y el carrito les avisa cuando cambia.
3. Adapter.
4. En el modelo, que tiene los datos y las reglas del negocio.

Validá lo que aprendiste

Rendí la evaluación **Buenas prácticas de código y patrones de diseño** en **testing-us.com**. Es gratis: en cada intento te tocan 10 preguntas al azar, dos de cada tema de este cuadernillo. Tenés 20 minutos y aprobás con 70%.

Si aprobás, recibís un certificado con código de verificación que podés sumar a tu currículum y compartir con quien te contrate.

MIS NOTAS

© 2026 Testing-US. Material de estudio propio. Se permite imprimirlo y compartirlo sin modificaciones y sin fines comerciales.

Referencias: Gamma, Helm, Johnson y Vlissides, Design Patterns (1994); Martin Fowler, Refactoring y el catálogo de refactoring.com; Robert C. Martin, principios SOLID; Scott Chacon y Ben Straub, Pro Git (git-scm.com). Ejemplos de JavaScript verificados con Node.js en octubre de 2026.

El certificado de Testing-US acredita la aprobación de una evaluación de conocimientos. No es un título oficial ni habilita para el ejercicio profesional.