ChatGPT

Manual de Buenas Prácticas: Código Limpio, Robusto y Mantenible

Checklist Esencial (Ready-to-Use)

1. **Escribe funciones y clases pequeñas, con una sola responsabilidad.**
   Por qué: Facilita la lectura, las pruebas unitarias y reduce el acoplamiento [1] [2]. Si cada unidad de código tiene un propósito claro (principio SRP), el sistema es más modular.
   Cómo verificarlo:
2. Verifica en la PR que cada función/método tenga *una sola razón para cambiar* (p. ej., no mezclar lógica de negocio con acceso a datos).
3. Asegúrate de que las clases no tengan nombres genéricos como `Helper` o `Utils` con demasiadas responsabilidades.
   Ejemplo:

```
#   Mal: función hace demasiadas cosas
def procesar_pago_y_enviar_email(pago, email_service): ...

#   Bien: responsabilidades separadas
def procesar_pago(pago): ...
def notificar_pago(email_service, pago): ...
```

4. **Usa nombres descriptivos y consistentes (PEP8, estilo Google).**
   Por qué: Los nombres claros hacen el código *autodocumentado*, reduciendo necesidad de comentarios y malentendidos [3]. Un nombre correcto revela intención, facilitando la revisión.
   Cómo verificarlo:
5. Comprueba en la PR que los identificadores (variables, funciones, clases) describan su propósito o contenido (evita nombres crípticos como `a`, `data2` o abreviaturas innecesarias).
6. Confirma que se siguen las convenciones de estilo (ej.: snake_case en Python según PEP8).
   Ejemplo:

```
#   Mal: nombres vagos o incorrectos
val = 1234        # ¿Valor de qué?
def doStuff(x):   # ¿Qué hace exactamente?

#   Bien: nombres expresivos
total_pedido = 1234
def calcular_total(pedido): ...
```

7. **Evita la *sobreingeniería*: implementa solo lo necesario (YAGNI).**
   Por qué: Añadir código “por si acaso” introduce complejidad innecesaria, bugs potenciales y

mantenimiento extra [4] [5]. YAGNI ("*You Aren't Gonna Need It*") previene funciones o abstracciones no requeridas por requisitos actuales.
Cómo verificarlo:

8. Pregunta en la revisión: *"¿Esta abstracción/funcionalidad resuelve un requerimiento actual comprobable?"* [5]. Si no, probablemente se debe eliminar o posponer.
9. Busca *señales de sobreingeniería*, como múltiples capas de abstracción sin beneficio tangible, uso de patrones complejos en problemas simples, o "problemas fantasmas" resueltos que el cliente no pidió [3].
   Ejemplo:

```
#   Mal: Clase genérica innecesaria para un solo caso de uso
class ExportadorFactory:
    def obtener_exportador(tipo):
        if tipo == 'CSV': return ExportadorCSV()
        if tipo == 'JSON': return ExportadorJSON()
exportador = ExportadorFactory.obtener_exportador('CSV')

#   Bien: Directo, sin abstracción prematura (solo CSV requerido)
exportador = ExportadorCSV()
```

10. **Favorece la claridad sobre la brevedad o la "magia" del lenguaje.**
    Por qué: Código explícito y simple minimiza errores. Trucos excesivamente ingeniosos dificultan el onboarding de otros devs y esconden bugs. El principio KISS ("*Keep It Simple, Stupid*") aplica aquí.
    Cómo verificarlo:
11. Comprueba que no haya expresiones extremadamente complejas en una sola línea; un código más largo pero legible es preferible a uno corto pero críptico.
12. En la PR, si algo es difícil de entender a primera vista, pide refactorizarlo de forma más explícita (p. ej., usando variables intermedias con nombres descriptivos en lugar de una sola expresión anidada).
    Ejemplo:

```
#   Mal: comprensión de lista complicada y poco clara
output = [func(x) for x in items if cond(x) and complex_condition(x)]

#   Bien: expansión a bucle for legible con condiciones descriptivas
output = []
for item in items:
    if cond(item) and complex_condition(item):
        output.append(func(item))
```

13. **Gestiona los errores de forma explícita y desacoplada (no abuses de excepciones).**
    Por qué: Un manejo consistente de errores previene flujos ocultos y facilita recuperarse de fallos. En lugar de usar excepciones para control de flujo normal, es preferible el *patrón Result* (retornar objetos Ok/Error) para errores esperables [6]. Esto evita sorpresas y hace el código más explícito.
    Cómo verificarlo:
14. Revisa que las excepciones se utilicen solo para *situaciones verdaderamente excepcionales* (p. ej., un error inesperado de red) y no para lógica esperada (p. ej., retorno de búsqueda sin resultados).

15. Verifica si existe una estructura unificada de error (clases o valores Result) y que los llamados comprueben dichos resultados en lugar de obviarlos.
    Ejemplo:

```
#   Mal: usando excepción para flujo normal (usuario no encontrado)
def obtener_usuario(id):
    usuario = db.buscar(id)
    if not usuario:
        raise UsuarioNoExisteError()
    return usuario

#   Bien: patrón Result indicando éxito o error
def obtener_usuario(id):
    usuario = db.buscar(id)
    if not usuario:
        return Result.Fallo("Usuario no existe")
    return Result.Exito(usuario)
```

16. **Escribe pruebas para comportamiento importante antes o justo después de codificar.**
    Por qué: Las pruebas actúan como *red de seguridad* ante cambios. Ayudan a documentar contratos y a prevenir regresiones. En código nuevo, usar TDD (desarrollo guiado por pruebas) o al menos escribir tests unitarios inmediatamente asegura entender el diseño deseado. En legacy, *tests de caracterización* capturan el comportamiento existente para refactorizar con confianza [7].
    Cómo verificarlo:
17. Comprueba que cada PR incluya pruebas unitarias o de integración relevantes para la funcionalidad cambiada o añadida (especialmente en casos límite y errores esperados).
18. Verifica que las pruebas sean legibles y cubran casos felices y de error. Si es código legacy sin pruebas previas, exige al menos una prueba de caracterización que documente el comportamiento actual antes de cambiar el código.
    Ejemplo:

```
def test_calcular_total_sin_items():
    carrito = Carrito(items=[])
    resultado = carrito.calcular_total()
    assert resultado == 0  # definición del comportamiento esperado
```

19. **Elimina duplicación *relevante*, pero evita abstracciones incorrectas (DRY vs WET).**
    Por qué: DRY ("Don't Repeat Yourself") sugiere no duplicar conocimiento, pero **es peor una abstracción errónea que unas pocas líneas duplicadas** [8]. Duplica temporalmente si la abstracción no está clara; refactoriza solo cuando entiendas el patrón común. Recuerda: "*duplication is far cheaper than the wrong abstraction*" [8].
    Cómo verificarlo:
20. Identifica en la revisión si hay lógica compleja copiada en varios lugares; si sí y los casos son realmente iguales, sugiere extraer una función común.

21. *Pero*, si la duplicación difiere ligeramente en cada caso o añadir abstracción requeriría lógica condicional compleja, considera dejar la duplicidad hasta tener más claridad (evita abstracción prematura).
22. Asegúrate de que cualquier abstracción introducida simplifique el código en los consumidores y no introduzca excepciones/flags para casos especiales (señal de abstracción forzada).
    Ejemplo:

```
#   Mal: Abstracción incorrecta que mezcla casos distintos
def procesar(items, tipo):
    if tipo == 'A': ...  # lógica A
    elif tipo == 'B': ...  # lógica B
    # Este método intenta ser genérico pero crea alto acoplamiento y casos
especiales.

#   Bien: Mantener separado si no hay un patrón común claro
def procesar_tipo_a(items): ...
def procesar_tipo_b(items): ...
```

23. **Aísla dependencias externas y efectos secundarios.**
    Por qué: Para lograr código predecible y testeable, las dependencias (bases de datos, APIs externas, reloj del sistema, etc.) deben estar abstraídas. Así aplicas *inyección de dependencias*: pasas recursos externos como parámetros o usas interfaces, en lugar de acoplar la lógica a llamadas concretas. Esto reduce la fricción en pruebas y cambios futuros [9] [10].
    Cómo verificarlo:
24. Revisa si el código accede directamente a recursos globales (ej.: `datetime.now()`, consultas SQL embebidas, `requests.get` directo). Recomienda encapsular eso en funciones o clases (p. ej., un `Reloj` inyectable, un repositorio de datos).
25. Comprueba la existencia de "costuras" para pruebas: por ejemplo, si un módulo usa `api_cliente = APICliente()` dentro, no es fácil de simular; preferible pasarlo en el constructor o parámetro (costura/inyección).
    Ejemplo:

```
#   Mal: dependencia global directa dificulta pruebas
def registrar_evento(evento):
    requests.post("https://api.externa.com/log", json=evento)

#   Bien: dependencia inyectada (p. ej., un cliente HTTP)
def registrar_evento(evento, http_client):
    http_client.post("/log", json=evento)
```

26. **Utiliza estructuras de datos adecuadas y encapsúlalas en objetos de valor.**
    Por qué: Elegir la estructura correcta (lista, conjunto, diccionario, árbol, etc.) simplifica la lógica y mejora rendimiento. Además, envolver datos crudos en *Value Objects* les da significado y garantiza invariantes (por ejemplo, una clase `Dinero` con moneda y cantidad siempre válida) [11] [12]. Esto mejora la *semántica* del código y previene estados inválidos.
    Cómo verificarlo:

27. En la PR, identifica valores “mágicos” o grupos de datos sin contexto (ej.: pasar `("USD", 100)` a funciones); sugiere encapsularlos (ej.: `Money(moneda="USD", cantidad=100)`).
28. Verifica que los *Value Objects* implementen validaciones en el constructor (p. ej., email válido, no permitir cantidad negativa) y que sean inmutables si es posible [11] [12].
29. Revisa el uso de estructuras: ¿se itera una lista para buscar un elemento cuando pudo ser conjunto/hash? Sugiere mejoras si afecta claridad/rendimiento sin añadir mucha complejidad.
    Ejemplo:

    ```
    #   Mal: usando tupla para representar coordenadas (poco semántico)
    pos = (10, 20)  # ¿10 qué? ¿metros? ¿coordenadas X,Y?

    #   Bien: usando Value Object para claridad y seguridad
    class Coordenada:
        def __init__(self, x: int, y: int):
            self.x = x; self.y = y
    pos = Coordenada(x=10, y=20)
    ```

30. **Implementa logging estructurado y desacoplado de la lógica de negocio.**
    Por qué: Un buen registro de logs ayuda a depurar y monitorear en producción sin ensuciar la lógica principal. Debe ser *encapsulado*: es decir, la decisión de qué, cuándo y dónde loguear se centraliza (por ejemplo, usando un módulo de logging o AOP), evitando llamadas `print` o `logging` dispersas por el código. Además, logs estructurados (JSON u otro formato) facilitan la correlación de eventos y evitan problemas de información sensible.
    Cómo verificarlo:
    - Busca en la PR llamadas directas a logging en medio de funciones de dominio. Si hay muchas, sugiere encapsular en funciones helper o decoradores que manejen la generación de logs.
    - Comprueba que los mensajes de log incluyan contexto útil (identificadores de solicitud, ID de correlación, etc.) para facilitar el *tracing* [13]. Verifica también que no registren PII o secretos en texto plano.
    - Asegúrate de que se respetan niveles de log (info, debug, error) y que los logs de error *no* son absorbidos silenciosamente (p. ej., imprimir el error sin lanzar o manejar).
      Ejemplo:

      ```
      #   Mal: logging intrusivo mezclado con la lógica
      def procesar_pedido(pedido):
          print(f"Procesando pedido {pedido.id}")  # imprime en stdout
          ...  # lógica de procesamiento
          print("Pedido procesado")

      #   Bien: uso de logger desacoplado y niveles apropiados
      logger = logging.getLogger(__name__)
      def procesar_pedido(pedido):
          logger.info(f"Iniciando procesamiento pedido {pedido.id}")
          ...  # lógica
          logger.info(f"Pedido {pedido.id} procesado con éxito")
      ```

Heurísticas para Decidir (Evitar Sobrecodificación y Complejidad)

Matriz de decisión pragmática:

- **¿Añadir abstracción o no?** Si tienes **3+ usos idénticos** de un mismo bloque de código *y* estás seguro de que los casos comparten las mismas reglas → *Extrae una abstracción reutilizable*. Si los usos son sólo *similares pero no iguales*, y anticipas excepciones, *duplica por ahora* en lugar de forzar una abstracción (aplica el consejo de Sandi Metz: "prefiere duplicación a la abstracción incorrecta" [8]). Revisa más adelante si surge un patrón unificado.
- **¿Cuándo crear una generalización (p. ej., clase base o genérica)?** Solo *después* de haber implementado al menos dos casos concretos y verificar que se benefician de la generalización. **Si X y Y tienen lógica duplicada >50% y añadir un parámetro/flag las simplifica sin agregar múltiples condiciones → abstrae Z**. Si no, mantén separado (**YAGNI**: no generalices para un tercer caso hipotético) [5].
- **¿Añadir una dependencia o librería externa?** Si resuelve un problema complejo *central* para tu proyecto (p. ej., un cliente HTTP robusto, un parser XML) y está bien mantenida → *sí, mejor reutilizar*. Si la funcionalidad es simple o podrías implementarla en pocas líneas *conociendo bien los requisitos*, evita depender de terceros (menos peso y acoplamiento).
- **¿Dividir en micro-servicio o módulo separado?** Si el componente en cuestión tiene **altísima tasa de cambio independiente** y fuerte aislamiento de dominio → considera separarlo (módulo o servicio). Si no, *KISS*: mantenlo junto hasta que duela. Separar prematuramente añade sobrecarga (red, despliegues múltiples) sin beneficio claro.

Señales tempranas de sobreingeniería:

- *Solucionas problemas inexistentes:* Si ves código manejando casos ultra-especulativos o configuraciones que ningún cliente pidió (p. ej., soporte multi-base de datos "por si un día…") [3]. **Detección:** Comenta y pregunta si hay un requerimiento real detrás de ese código; si no, es probable sobreingeniería.
- *Capas excesivas:* Por ejemplo, clases *Manager*, *Factory de Factory*, adaptadores anidados sin una ganancia clara. Si para entender una simple operación debes atravesar 5 clases/interfaces, es complejo de más. **Detección:** Cuenta las capas en la traza de una funcionalidad; si se pudo lograr en 2 y hay 5, cuestiona cada una.
- *DRY llevado al extremo:* Abstraer código que *solo se repite dos veces* o juntar funciones que *hacen cosas parecidas pero no iguales*. Esto produce condicionales internos para manejar diferencias, complicando el código [14] [15]. **Detección:** Busca funciones con muchos `if tipo == ...` o parámetros booleanos que alteran comportamiento; a veces es mejor dos funciones separadas que una "genérica" con banderas.
- *Patrón por moda, no por necesidad:* Implementar un **micro-kernel plugin** o un **framework interno** solo porque es interesante, cuando una solución más simple bastaría. **Detección:** Pregunta *"¿Qué problema concreto resuelve esta complejidad adicional?"*. Si la respuesta es difusa (p. ej., "nos da flexibilidad futura…" sin caso de uso actual), es sospechoso.
- *Wrappers innecesarios:* Crear wrappers para absolutamente todo (p. ej., una clase que envuelve una lista estándar sin agregar comportamiento real) [16]. **Detección:** Si la clase A solo delega en B sin lógica adicional, evalúa removerla y usar B directamente.

Criterios YAGNI (You Ain't Gonna Need It):

- Implementa hoy solo lo que se necesita hoy. Si una función "podría ser útil" pero no está confirmada su necesidad, no la incluyas [5].
- Pregunta constantemente: *"¿Qué pasa si no implemento esto ahora?"*. Si la respuesta es "nada en el corto plazo, solo sería para un hipotético futuro", entonces YAGNI: no hacerlo.
- Documenta brevemente las ideas futuras (en un TODO, ticket o comentario) para revisitarlas cuando el requerimiento exista, en lugar de codificarlas prematuramente.
- **Excepción a YAGNI:** cuestiones de arquitectura base difíciles de cambiar más adelante (p. ej., elegir usar un framework que soporte modularidad si **seguro** escalará). Incluso así, mantén la solución lo más simple posible en la implementación inicial.

¿Cuándo refactorizar o abstraer?

Refactoriza **ahora** si:
- Hay *deuda técnica* que complica la incorporación de una feature prioritaria o está generando bugs recurrentes. El riesgo de no refactorizar (seguir parchando) es mayor que el costo de hacerlo.
- Identificas un *code smell* claro (clase Dios de 1000 líneas, métodos duplicados en varias clases) y estás en esa zona del código modificando algo relacionado. Es más eficiente arreglarlo mientras lo tienes cargado en la mente, siempre y cuando tengas tests o puedas escribirlos (refactoring seguro).
- El código viola algún principio crítico (SRP, Demeter) de forma que *dificulta* pruebas o mantenimiento (p. ej., función que realiza 5 cosas en secuencia – acoplamiento temporal – y ya causó errores de orden). En este caso, fragmentarlo reducirá futuros fallos.

Pospone refactor/abstracción si:
- **Funciona y no duele**: El código está feo pero raramente toca y no causa problemas. Documenta la deuda pero no la atiendas hasta que haya que trabajar en esa parte (evita "refactorings de vanidad").
- Estás en medio de una entrega crítica (time-to-market urgente). Solo refactoriza lo mínimo indispensable para agregar la funcionalidad requerida sin añadir bugs. Planifica pagar la deuda después de la entrega.
- No tienes aún pruebas que cubran el comportamiento esencial y la funcionalidad es compleja. Primero escribe *tests de caracterización* o mínimamente tests de humo para asegurarte de no romper nada [7]; luego refactoriza respaldado por esos tests.

Resumiendo, decide siempre en función del *valor vs. costo*: agrega complejidad solo si resuelve un problema real y presente. Mantén un ojo en las señales de sobreingeniería [4]; es más fácil añadir capas después que quitarlas cuando todo está entremezclado sin necesidad. Prioriza soluciones simples y explícitas, y abstrae o optimiza cuando la simplicidad actual empieza a mostrar sus límites.

Estrategias de Pruebas (Nuevo y Legacy)

Test de caracterización (characterization tests): Son pruebas que **capturan el comportamiento actual de un código existente** para poder cambiarlo con seguridad [7]. En código *legacy* (generalmente "código sin tests" según la definición de Feathers [17]), primero escribe tests que describan qué hace actualmente el sistema, incluso si ese comportamiento es erróneo o no deseado. La idea es **congelar** la salida actual como

referencia (*golden master* o *snapshot*) [18]. Así, cuando refactorices o corrijas algo, cualquier cambio inesperado será detectado por estas pruebas.

- *Aplicación:* Úsalo en sistemas existentes donde temes romper algo al hacer cambios. Por ejemplo, llama a una función compleja con entradas conocidas y aserta que la salida coincide con la que produce hoy (aunque sea incorrecta), luego refactoriza. Tras refactor, puedes actualizar las expectativas de tests a los valores correctos esperados una vez entendido el dominio.
- *Verificación:* Un test de caracterización exitoso suele ser uno que *pasa inmediatamente* antes de cambios (confirma la captura fiel del comportamiento) y que *falla* si se altera algo no previsto. En la PR, comenta que estas pruebas son temporales para proteger durante refactor, y actualiza o elimina las que ya no sean relevantes después.

TDD inverso (o *TDD al revés*): Es una forma de aplicar principios TDD en código existente: primero escribes una prueba para un bug reportado o una nueva característica *después* de tener algo de implementación. Es "inverso" porque en TDD clásico se escribe la prueba antes del código; aquí quizás tienes código sin pruebas, y empiezas escribiendo la prueba para documentar el bug o comportamiento deseado. En esencia:
- Cuando encuentres un error, **escribe un test que falle** reproduciendo ese bug (rojo), luego corrige el código hasta que el test pase (verde). Esto asegura que el bug no reaparezca en el futuro.
- Para nuevas features en legacy sin test, podrías primero implementar rápido la funcionalidad, luego escribir tests que cubran casos principales ("TDD al revés"), y finalmente refactorizar. No es lo ideal, pero es pragmático cuando entrar con TDD puro desde cero es complicado.
- En la revisión, busca evidencia de estos tests añadidos cuando se soluciona un bug (un buen PR arreglando bug incluirá `test_<lo_que_sea> regressing bug #123`).

Golden tests (Approval/Golden Master Testing): Consisten en **fijar una salida esperada "dorada"** (por ejemplo, un archivo JSON de salida, un reporte) y comparar futuras ejecuciones contra ella. Se usan cuando el resultado es complejo pero relativamente estable. Por ejemplo, en una librería de generación de reportes PDF, generas un PDF "golden" conocido; los tests futuros comparan que el nuevo PDF coincida bit a bit con el original.

- *Cuándo sirven:* Cuando tienes funcionalidades donde *cualquier pequeño cambio en la salida debe ser intencional*. Especialmente útil en refactors donde no se debe alterar el comportamiento externo. También en test de integración de API: puedes almacenar la respuesta JSON esperada de un endpoint como golden.
- *Cuándo dañan:* Si el output es volátil (incluye timestamps, IDs aleatorios) o si la definición de "correcto" puede variar. En esos casos, un golden test fallará por diferencias irrelevantes. Evítalos si generan muchos falsos positivos o si actualizar el "golden" se vuelve rutina sin entender cambios (indicador de sobre-test).
- *Buenas prácticas:* Mantén los artefactos golden en repositorio (ej., archivos de texto/JSON de referencia). Implementa una forma fácil de actualizar los golden outputs con una revisión manual, para cuando hay cambios legítimos. En la PR, un cambio en golden output debe ser revisado cuidadosamente para asegurarse de que el cambio es deseado.

Cobertura útil vs. cobertura ilusoria: No te dejes engañar por un porcentaje alto de cobertura de código si las pruebas no son efectivas. Una cobertura útil es aquella que cubre caminos de ejecución *relevantes* y valida resultados esperados, no solo ejecuta líneas. Por ejemplo, tests que sólo llaman funciones sin asserts pueden sumar cobertura pero no detectar errores lógicos (cobertura ilusoria). En la revisión de PR, más que

pedir “X% cobertura”, fíjate en la *calidad de los casos*: ¿prueban condiciones normales y excepcionales? ¿Fallan si el código hace algo mal? Prefiere **pocos tests significativos** sobre muchos tests triviales.

Para evaluar calidad, se puede usar **pruebas de mutación**: una herramienta que introduce pequeños cambios (mutantes) en el código y verifica que tus tests los detectan [19] [20]. Si tus tests no fallan ante un mutante obvio (p. ej., cambiaron un `+` por `-` y ningún test se enteró), entonces la suite no es suficientemente rigurosa. No es necesario ejecutar mutación en cada PR, pero mentalmente puedes simular: *“Si esta línea fuera errónea, ¿tenemos un test que lo cace?”*.

Pruebas contractuales (de contrato): En sistemas integrados (microservicios, cliente-servidor), son tests que **verifican que el contrato entre componentes se cumple**. Ejemplo: un *consumer test* define cómo espera recibir los datos el cliente y se valida contra un *provider* simulado, o viceversa (pactos). Aplicado a tu contexto: si consumes APIs de terceros, podrías tener tests que mockean la respuesta esperada (contrato) y validan que tu código la procesa correctamente; o si expones API, tests que verifiquen que no rompiste el formato de respuesta entre versiones. Esto evita regresiones cuando servicios evolucionan. En la práctica, en una PR que cambia un DTO o esquema de datos, debería haber ajuste de pruebas contractuales correspondientes.

Estrategias en Legacy vs Nuevo:

- En **código nuevo**, intenta aplicar TDD clásico o al menos escribe los tests junto con el código. Así diseñas pensando en testabilidad (por ejemplo, evitando acoplar cosas innecesariamente). Usa *test de unidad* donde sea posible (rápidos, aislados) para la lógica pura, y complementa con *test de integración* para verificar que los componentes encajan (ej., llamadas reales a BD o a APIs externas simuladas).
- En **código legacy**, si no puedes escribir unit tests inmediatos por el diseño acoplado, emplea *tests de caracterización* de más alto nivel. Por ejemplo, un test que invoque un módulo grande de inicio a fin con ciertos inputs y compruebe salidas. Una vez que tengas algunos en verde, podrás refactorizar internamente y luego ir reemplazándolos por tests más unitarios conforme creas “costuras” en el diseño [21] [22].

Ejemplos mínimos de estructura de pruebas:

- *Test unitario simple:* comprueba una función pura:

```
def test_formato_nombre():
    nombre = formatear_nombre("juan", "perez")
    assert nombre == "Juan Perez"
```

- *Test de excepción esperada:*

```
import pytest
def test_error_si_balance_insuficiente():
    cuenta = Cuenta(balance=100)
```

```
    with pytest.raises(SaldoInsuficienteError):
        cuenta.retirar(200)
```

- *Test de integración (usando fixtures de Pytest):* digamos, probar un repositorio con base de datos en memoria:

```
@pytest.fixture
def repo_temp():
    return UserRepository(db=DBMemoria())

def test_creacion_y_busqueda_usuario(repo_temp):
    usuario = Usuario(id=1, nombre="Ana")
    repo_temp.guardar(usuario)
    assert repo_temp.buscar(1) == usuario
```

La idea clave: las pruebas deben dar **confianza** para cambiar el código. Usa caracterización para entender legacy, TDD para guiar diseño en lo nuevo, y mantén la suite en verde siempre (integrando tests en CI). Revisa regularmente que los tests realmente están validando condiciones útiles, no solo existiendo para llenar un número.

Errores y Resultados

Patrón Result (Ok/Err): Consiste en retornar un objeto que indica **éxito o fallo** en lugar de lanzar excepciones para el flujo de control esperado. Es común en lenguajes funcionales (como `Result<T, E>` en Rust) pero aplicable en cualquier lenguaje (p. ej., usar tuplas `(resultado, error)` o clases). En lugar de propagar excepciones y obligar a capturarlas, el patrón Result hace explícito en la firma de la función que puede fallar. Por ejemplo, una función `dividir(a,b)` podría devolver `Result(resultado=a/b)` o `Result(error="División por cero")`. El llamador **debe chequear** qué caso fue [6], volviendo el manejo de error obligatorio y claro.

- *Ventajas:* Evita que excepciones checkeadas se olviden de manejar; elimina *throws* como control de flujo (que pueden afectar rendimiento y legibilidad). Permite **encadenar operaciones** facilmente comprobando cada resultado sin try/except anidados. Los errores viajan como datos, se pueden loguear o transformar en valores de dominio.
- *Aplicación:* Úsalo para errores esperables que el programa sepa manejar. Ej: validaciones de negocio (usuario no encontrado, saldo insuficiente) – en vez de lanzar `UserNotFoundException`, devuelves `Result<Fallo, msg>` que la capa superior convierte quizás en 404 HTTP. Para errores no recuperables (p.ej. fallo de memoria), se siguen lanzando excepciones o dejando que revienten.
- *Ejemplo rápido (Python pseudo)*:

```
class Result:
    def __init__(self, ok=None, err=None):
        self.ok = ok; self.err = err
    def es_exito(self): return self.err is None
```

```python
def analizar_datos(datos) -> Result:
    if not datos:
        return Result(err="Datos vacíos")
    # ... procesamiento
    return Result(ok=resultado)

res = analizar_datos(payload)
if res.es_exito():
    procesar(res.ok)
else:
    logger.error(f"Fallo: {res.err}")
```

Aquí la función retorna siempre un `Result`; quien llama no necesita un try/except sino un `if`.

Mapeo de errores (domain, infra, validación): Es buena práctica categorizar las fallas en tu sistema para tratarlas diferenciadamente:

- *Errores de Dominio:* Violaciones a reglas de negocio (ej., "saldo insuficiente", "usuario no puede auto-seguirse" [23]). Suelen representarse con tipos/clases específicas o códigos (`ErrorDominio("saldo_insuficiente")`). Se manejan normalmente cerca de la superficie de la aplicación, mostrando mensajes al usuario o pasos compensatorios.
- *Errores de Infraestructura:* Issues técnicas como pérdida de conexión a BD, timeouts de red, fallo de disco. Deben encapsularse para que la lógica de negocio no dependa directamente de detalles técnicos. Por ej., al capturar un `socket.timeout`, lo envuelves en un `ErrorInfra(timeout)` para notificar a capas superiores que es un problema técnico. Estos errores muchas veces disparan *reintentos* o *circuit breakers* (ver abajo) en lugar de lógica de negocio.
- *Errores de Validación:* Datos de entrada inválidos (formato incorrecto, campos requeridos faltantes). Idealmente se detectan en las capas de entrada (al validar DTOs o requests) y se reportan de forma amigable. Se pueden representar con un tipo `ErroresValidacion` que contiene, por ejemplo, un mapa campo->error.
- *Errores Desconocidos:* Siempre considera la posibilidad de fallos imprevistos. Un patrón es tener un *"catch-all"* (último recurso) que captura cualquier excepción no manejada, la loguea con nivel *error* y devuelve un `Result.Fallo("Error desconocido")` genérico o lanza la excepción hacia un manejador global (p. ej., middleware de servidor que retorna 500). Esto evita que excepciones no capturadas tumben toda la app sin registro.

Al mapear errores en categorías, puedes establecer **políticas de manejo** coherentes por categoría, lo que mejora la robustez.

Política de lanzamiento y resiliencia: Define de antemano cómo tu sistema reacciona ante fallos en operaciones críticas:

- **Retry (Reintento):** Para operaciones transitorias (llamadas de red, transacciones a BD) implementa reintentos con backoff exponencial. Por ejemplo, si un servicio externo no responde, reintenta 3 veces aumentando el intervalo (p. ej., 1s, 2s, 4s) [24]. *Criterio:* reintenta solo si es probable que el error sea pasajero (timeout, error 503); no reintentes errores permanentes (p.ej., 404 Not Found o validación fallida).

- **Circuit Breaker:** Evita golpear un recurso caído repetidamente. Un *circuit breaker* abre el circuito tras N fallos seguidos, de modo que futuras llamadas fallan rápido sin intentar la operación hasta que pase un tiempo de *cooldown* [25]. Útil para prevenir saturación y cascadas de fallos en sistemas distribuidos. *Verificación:* asegúrate de que ante repetidos timeouts a un servicio, tu código deja de intentarlo por un periodo (ej., usa un estado global o compartido para ese circuito).
- **Fallback / Degradación:** Ten planeado comportamientos alternativos cuando algo falle. Ejemplo: si tu servicio de recomendaciones está caído, en lugar de no mostrar nada, mostrar opciones por defecto; si no se puede guardar en cache, seguir con la operación sin cache y loguear. Un fallback mantiene la funcionalidad base aunque sea con menos features.
- **Idempotencia:** Al diseñar reintentos o procesamiento de eventos, haz que las acciones sean *idempotentes* (ejecutar la operación N veces tiene el mismo efecto que 1 vez). Esto es crítico en manejadores de eventos en EDA y en endpoints que el cliente pueda reintentar. Por ejemplo, si un pago se procesa y luego hay un timeout en la respuesta, el reintento no debe duplicar el cargo. Usa identificadores únicos de operación para no repetir side-effects.

Estas políticas a menudo se combinan: por ejemplo, en una operación de pago, puedes tener reintento + idempotencia en el servidor (reintento hasta 3 veces, pero el servidor detecta transacción duplicada y la ignora si ya se procesó).

Logging desacoplado y encapsulado: Como mencionamos en la checklist, el logging debe ser transversal pero sin ensuciar la lógica. Algunas tácticas:

- **Correlación:** Asigna IDs de correlación por petición o flujo (p. ej., un UUID por cada request entrante). Inclúyelo en cada log relacionado [13]. Así en análisis puedes agrupar todos los logs de, digamos, la operación "Orden #1234 desde que entra hasta que sale". Esto ayuda muchísimo en sistemas concurrentes/distribuidos para seguir el hilo.
- **Niveles de log apropiados:** Define qué va en DEBUG (datos detallados para dev), INFO (eventos normales, e.g. "Usuario registrado"), WARNING (situaciones anómalas recuperadas, e.g. "Reintento #2 de envío de email"), ERROR (fallos no recuperados que requieren atención) y CRITICAL (caídas de sistema). En PR, verifica que los niveles usados correspondan a la severidad. Por ejemplo, no hagas `logger.error` de algo que es simplemente un caso de negocio manejado (eso debería ser warning o incluso info).
- **No log duplicativo ni ruidoso:** Evita loggear lo mismo en múltiples capas (p. ej., un error genera 5 trazas de error en distintos lugares). Puede ser mejor centralizar la captura y logging de ciertas excepciones en un solo punto (middleware, decorador).
- **Protección de datos sensibles:** Revisa que no se loguee información confidencial (p. ej., números de tarjeta, contraseñas, PII) en texto plano. Si es necesario para debug, enmascara o anonimiza. Aplica políticas de *sampling* si el volumen de logs es enorme (p. ej., loguear 1 de cada 100 eventos de cierto tipo para no inundar).
- **Estructurado vs libre:** Prefiere logs en formato estructurado (JSON, campos clave=valor) en lugar de strings libres, así las herramientas pueden parsearlos. Ejemplo: en vez de `logger.info("Pedido %s completado en %s seg." % (id, tiempo))`, haz `logger.info({"evento": "pedido_completado", "pedido_id": id, "duracion_seg": tiempo})`. Esto facilita búsquedas y análisis.

En suma, manejar errores de forma robusta implica: comunicar claramente los fallos (Result, excepciones bien categorizadas), reaccionar según la naturaleza del error (reintentar, fallback, abortar con mensaje

claro), y dejar un rastro (logs) útil para diagnósticos sin entorpecer el diseño. Un código que anticipa y maneja errores es más *mantenible* y reduce sorpresas en producción.

🔗 Concurrencia y Condiciones de Carrera

Escribir código concurrente (*threads*, *async*, procesos múltiples) agrega complejidad significativa. Principios para mantenerlo limpio y evitar *race conditions*:

- **Inmutabilidad donde sea posible:** Si múltiples hilos/tareas acceden a un objeto, lo más seguro es que ese objeto no cambie su estado. Los objetos inmutables eliminan de raíz condiciones de carrera porque no hay actualización concurrente [26] [27]. Ejemplo: en vez de tener una lista global a la que varios threads agregan elementos, considera que cada thread produzca una lista y luego las combines (o uses estructuras thread-safe). *Verificación:* busca en código concurrente variables globales o estáticas modificables; si existen, comprueba que estén protegidas con locks o cámbialas a inmutables.
- **Bloqueos (locks) granulares y bien documentados:** Cuando necesites estado mutable compartido, usa mecanismos de sincronización (mutex, semáforos) lo más **acotados** posible. Bloquea solo el mínimo código crítico para evitar condiciones de carrera. Por ejemplo, si actualizas un contador global, bloquea solo esa operación, no toda la función. *Cómo detectarlo:* en code review, localiza secciones `lock.acquire()` / `release()` (o `with lock:` en Python) y verifica que engloban únicamente lo necesario. Añade comentarios que expliquen qué recurso protege el lock. Evita locks anidados si puedes, para prevenir *deadlocks* (bloqueos mutuos).
- **Diseños libres de bloqueo:** Considera *STM (Software Transactional Memory)* o modelo de *actores* donde aplicable. Por ejemplo, en Python puedes usar colas (`queue.Queue`) para que threads pasen mensajes en vez de compartir variables. Un *Event Loop* asíncrono (como `asyncio`) evita muchas races ejecutando tareas cooperativamente en un solo hilo. Si tu carga lo permite, un diseño single-thread con cola de trabajo puede ser más simple y suficientemente concurrente (productor-consumidor). *Verificación:* si se opta por colas, asegúrate de manejar casos de cola llena (backpressure) o de consumidores más lentos que productores (ver backpressure abajo).
- **Idempotencia en operaciones concurrentes:** Similar a EDA, si existe la posibilidad de reejecución (p.ej., un thread repite trabajo después de fallo parcial), diseña las operaciones para que repetir no cause daño. Ej: una función que transfiere dinero debería comprobar "si ya transfirió" para no duplicar en caso de reintento. Esto aplica en concurrencia distribuida (varios servicios). *Verificación:* en PR de componentes concurrentes, pregunta *"¿qué pasa si esta función se ejecuta dos veces a la vez?"*. Si la respuesta es "duplica efectos", entonces no es idempotente; evalúa proteger con lock o lógica de confirmación.
- **Detección y pruebas de *race conditions*:** Las *race conditions* son notorias por ser esporádicas. Incorpora pruebas de estrés multi-hilo: por ejemplo, lanza 100 hilos incrementando el mismo contador miles de veces y verifica resultado; sin sincronización adecuada, a veces fallará. Usa herramientas si existen (p.ej., *thread sanitizers* en C/C++). También considera *fuzz testing* en entornos concurrentes: variar órdenes de ejecución para intentar forzar interleavings extraños. *Verificación:* difícil de hacer manual en PR, pero si el PR modifica código concurrente crítico, sugiere correr tests en loop o con sanitizadores de data-races, o al menos revisar mentalmente todas las rutas posibles de interacción de threads.
- **Condiciones de carrera comunes a prevenir:**

- *Check-then-Act:* Ej: `if not exists(file): create(file)`. Entre el check y create, otro hilo pudo haberlo creado → resultado duplicado. Solución: agrupar esas acciones bajo lock o usar operaciones atómicas del sistema (p.ej., "crear si no existe").
- *Actualizaciones perdidas:* Dos hilos leen valor X, ambos lo incrementan y escriben, uno de los incrementos se pierde. Solución: locks o tipos atómicos (p.ej., `AtomicInteger`).
- *Iterar y modificar:* Un hilo iterando sobre una colección mientras otro la modifica → error. Solución: bloquear la colección o usar variantes concurrentes (ej., CopyOnWriteList, ConcurrentHashMap).

- *Salida prematura:* Un hilo lanza una tarea en segundo plano y termina el programa antes de que la tarea acabe (porque no se hizo join o espera). Solución: sincronizar finalización (join threads, usar AsyncAwait, etc.).

- **Cuándo NO paralelizar:**

- Si la sección crítica del trabajo es no paralelizable (Ley de Amdahl): e.g., 95% del tiempo el programa espera respuesta de una API externa; lanzar 10 hilos no lo hará 10 veces más rápido y solo complicará la lógica.
- Si los datos son *muy difíciles de sincronizar* y el volumen no es tan grande: a veces procesar secuencialmente 1000 registros con lógica sencilla es más rápido (y seguro) que manejar hilos para dividir 1000 registros con overhead de coordinar y luego mergear resultados.
- En sistemas I/O-bound sencillos, usar asincronía (un hilo pero no bloqueante) puede bastar en vez de múltiples threads. Ej: en Python por el GIL, CPU-bound no escala con threads, y para I/O-bound, `asyncio` puede ser más limpio.
- Si no tienes experiencia en concurrencia para ese caso, empieza simple (mono-hilo) y solo paraleliza cuando identifiques claramente el cuello de botella.

Límites y backpressure:
Cuando implementas concurrencia (sobre todo en productor/consumidor), controla la velocidad: *backpressure* es la capacidad de un sistema de **frenar productores cuando los consumidores no dan abasto** [28] [27]. Ejemplo: si tienes un pool de threads procesando tareas de una cola, define un límite de tamaño de cola. Si la cola está llena, los productores deben esperar o desechar mensajes menos prioritarios. Esto previene consumo excesivo de memoria y *thrashing*.

- *Verificación:* Identifica puntos donde se acumulan tareas (colas, buffers). Pregunta: "¿qué pasa si recibimos 100k solicitudes en un minuto?". Si la respuesta es "se acumulan en memoria sin control", es falta de backpressure. Soluciones: límites de cola, rechazo elegante de exceso (por ejemplo, retornar HTTP 429 Too Many Requests), o degradación (descartar tareas no críticas).
- En sistemas *reactivos* o con *streams*, usa sus mecanismos de demanda: e.g., en ReactiveX/Reactor, el subscriber puede solicitar N ítems a la vez. Asegúrate de utilizarlos en vez de emitir infinito sin control.

Herramientas adicionales:
- Considera *transacciones* para concurrencia compleja en estructuras de datos: Algunas plataformas (ej. Clojure STM) permiten que múltiples actualizaciones ocurran de forma atómica sin locks explícitos.
- Modelo de *Actores* (Ej: Akka, Orleans): cada actor tiene su propio mailbox y no hay compartición de estado, solo mensajes. Simplifica razonamiento ya que elimina data races (pero cuidado con orden de mensajes).
- Si usas *async/await* (ej: Python asyncio, JavaScript), recuerda que aunque no hay múltiples threads, puede

haber *race conditions lógicas* si dos corutinas acceden a la misma variable intercaladamente. Async no te salva de sincronizar el acceso a recursos compartidos (podrías necesitar semáforos asíncronos).

En resumen, la concurrencia limpia se logra minimizando la necesidad de sincronización (diseño inmutable o por mensajes) y, cuando es necesaria, haciéndola explícita y pequeña. Siempre pregúntate en PR: *"¿Este código concurrente es comprensible y seguro?"*; si hay dudas, sugiere agregar bloqueos o refactorizar hacia un modelo más sencillo.

Diseño y Acoplamiento

Ley de Demeter (Principio del Menor Conocimiento): Indica que una unidad de software (por ejemplo, un objeto o función) **solo debería interactuar con sus "amigos" cercanos, no con extraños** [1] [2]. En la práctica: un método *M* de un objeto *O* solo debe llamar a métodos de: *O* mismo, sus atributos directos, sus parámetros, o objetos que *M* cree. *No* debería encadenar llamadas a sub-objetos lejanos. Por ejemplo, si tienes `pedido.cliente.direccion.calle`, estás atravesando 3 objetos; violación clásica de Demeter. Lo correcto sería que `pedido` ofrezca un método para obtener la calle del cliente, o que se pase directamente la `direccion` donde se necesite. En términos coloquiales: *"Don't talk to strangers"* (no hables con extraños) [29] [30].

- *Ejemplo:*

```
#   Violación de Demeter:
total = pedido.cliente.obtener_cuenta().calcular_total()
# Aquí, pedido conoce cliente, y cliente conoce cuenta. Pedido llama
métodos de cuenta a través de cliente.

#   Cumpliendo Demeter:
total = pedido.calcular_total_cuenta_cliente()
# El objeto pedido se encarga internamente de navegar a cliente->cuenta.
```

  Esto reduce el acoplamiento, porque si luego cambia la estructura interna (por ejemplo, el cliente ya no tiene cuenta), solo `pedido.calcular_total_cuenta_cliente` debe cambiar, no todos los lugares que hacían la cadena.

- *Verificación:* Busca en el código `.` encadenados ("train wrecks"). Dos puntos seguidos (`obj.a.b` generalmente está bien, indica relación directa; pero tres o cuatro empiezan a oler). Si encuentras `algo.algo2().algo3()`, considera sugerir que *algo* provea un método para lo que se necesita. También fíjate en métodos que "se meten" en estructuras internas de otros objetos para modificarlas; mejor delegar esa tarea al propio objeto.

- *Beneficio colateral:* Cumplir Demeter tiende a aumentar la *cohesión*: la lógica referente a un objeto se mantiene en sus métodos en lugar de dispersarse por otras clases.

SRP (Single Responsibility Principle): Cada módulo (clase, función) debería tener **una sola razón de cambio**. Es decir, agrupa una única funcionalidad o preocupación. Por ejemplo, una clase que maneja la

lógica de facturación no debería también enviar emails de confirmación (eso debería ser otra clase). SRP suele significar cohesión alta: todo en la clase está relacionado con esa responsabilidad única.

- En revisión, identifica clases "multitasking": p. ej., `UsuarioService` que valida datos, construye SQL y envía correo de bienvenida – al menos 3 razones de cambio (reglas de validación, esquema de BD, plantilla de email). Recomienda dividir: quizás `ValidadorUsuario`, `RepositorioUsuario`, `EmailNotificador`.
- También se aplica a funciones: si un método se alarga mucho y en su interior ves secciones comentadas "// calcular X", "// enviar Y", es señal de que hace más de una cosa. Divídelo en sub-funciones.
- **Ojo:** SRP no implica tener cientos de mini clases sin sentido; agrupa lógicamente. A veces la "responsabilidad" puede ser un poco amplia pero cohesiva. El criterio es: si describir la clase requiriera usar "y también", probablemente rompe SRP.

DRY vs WET (y la anti-DRY): DRY ("No te repitas") ya lo cubrimos en la checklist y heurísticas. Vale reiterar la frase: *"Duplication is better than wrong abstraction"* [8]. La contra-cara de DRY es WET ("Write Everything Twice" o a veces "We Enjoy Typing") que humorísticamente alienta la duplicación. En realidad, el equilibrio es: Duplica si no estás seguro de abstraer, pero apunta a eliminar *duplicación de conocimiento* (reglas de negocio, fórmulas, etc.) para que la lógica exista en un solo lugar.

- Ejemplo de DRY bien aplicado: la fórmula de cálculo de impuestos se usa en varias partes -> extrae una función `calcular_impuesto()` en un módulo común. Ahora la fórmula está centralizada; si cambia la tasa de impuesto, modificas un punto.
- Ejemplo de mala abstracción por sobre-aplicar DRY: dos procesos diferentes tenían un bucle similar, pero con diferencias sutiles. Para no repetir el bucle, alguien hizo una función genérica con muchos ifs para manejar ambos casos. Resultado: aumentó la complejidad y cuando se añadió un tercer caso, la función genérica explotó en condicionales. Hubiese sido mejor mantener dos funciones separadas y luego ver si realmente se podía abstraer de forma limpia.
- *Consejo en revisión:* Permite duplicación pequeña si mejora la claridad o aislamiento de contextos. Por ejemplo, duplicar una constante en dos microservicios separados es aceptable (no hace falta un módulo común para una constante). En cambio, duplicar 50 líneas de lógica de cálculo en la misma codebase probablemente no.

"Las costuras" (seams) e inyección de dependencias (DIP): Michael Feathers habla de *costuras* como puntos donde puedes variar el comportamiento sin cambiar el código [31]. En diseño, estas costuras aparecen naturalmente si aplicas *Dependency Inversion Principle (DIP)*: las partes altas del sistema definen interfaces que las partes bajas implementan. Por ejemplo, en lugar de que tu servicio de negocio llame directamente a una clase concreta de base de datos, defines una interfaz `RepositorioClientes` y tu servicio conoce solo eso (depende de la abstracción, no de la concreción). Luego pasas/inyectas una implementación concreta (PostgresRepositorioClientes) al iniciar. Esto permite poner un stub o mock en tests, o reemplazar la implementación sin tocar la lógica de negocio [9] [32].

- *Identificación de costuras:* En revisión, nota si el código es muy difícil de testear porque *no hay costuras*: p. ej., funciones que dentro instancian objetos concretos (`self.db = SQLClient()` en el constructor). Para crear costura, sugerimos pasar ese SQLClient de afuera (constructor o método). Cada *dependency injection* es efectivamente una costura creada.
- **Ports & Adapters (Hexagonal):** Este estilo arquitectónico fomenta explícitamente costuras: los *ports* son interfaces (lado del dominio) y los *adapters* son implementaciones (lado de infraestructura). Un

ejemplo: tu dominio define un port `NotificadorPedidos` (puede enviar notificaciones de pedido), y tienes adapters como `NotificadorEmail`, `NotificadorSMS` implementando ese port. El dominio usa solo `NotificadorPedidos` sin saber qué hay detrás. Esto mantiene el núcleo desacoplado y fácilmente testeable (puedes meter un adapter fake en tests).

- *Verificación:* Si ves dependencias fuertes, sugiere: "¿Podemos extraer una interfaz aquí?". En Python, no hay interface formal, pero puedes lograrlo con duck typing o clases base abstractas (`ABC`). En Java/C#, usar interfaces. Asegúrate de que la creación de la implementación ocurra en la capa de composición (por ejemplo, en el *main* o en un *contenedor IoC*), no dentro de la lógica.

Value Objects (Objetos de Valor): Ya los mencionamos en checklist: objetos sin identidad, inmutables, que representan un concepto del dominio con sus reglas. Desde el punto de vista de acoplamiento, los Value Objects **reducen errores porque encapsulan lógica relacionada**. Por ejemplo, si tienes que pasar varios parámetros que siempre van juntos (latitud, longitud), podrías usar un VO `Coordenada`. Esto no solo mejora legibilidad sino que evita que se mezclen parámetros (error clásico: pasar lat por long al revés). También proveen métodos útiles relacionados (p.ej., `distancia_ate(otra_coordenada)` en la clase Coordenada, en vez de tener funciones utilitarias sueltas).

- *Invariantes:* Un VO garantiza en su constructor que sólo puede existir en estado válido. Si no se puede crear, lanza excepción o retorna un Result de error. Ej: un VO `Porcentaje` que acepta 0 a 100; no podrás instanciar `Porcentaje(150)`. Así evitas propagar valores inválidos y tener que checarlos en mil sitios.
- *Comparación por valor:* A diferencia de entidades, dos VO con mismos valores se consideran iguales (deberían ser *comparables* fácilmente). Esto ayuda a reducir acoplamiento a la identidad arbitraria.
- *Ejemplo:* `EmailAddress` VO que siempre guarda un email válido. Cualquier función que reciba EmailAddress puede confiar en su formato.
- *Verificación:* Mira dónde varios métodos siempre toman los mismos 2-3 parámetros juntos, o un dict con ciertos campos, etc. Considera sugerir un VO para agruparlos. También verifica que los VO tengan métodos de utilidad (si corresponde) para no filtrar demasiada info interna (i.e., Law of Demeter dentro del VO: en vez de pedir `coord.x` y `coord.y` fuera repetidamente, tal vez dotar a `Coordenada` de método `esta_en_cuadrante(superior_derecho)` por ejemplo).

Acoplamiento secuencial (Temporal Coupling): Esto ocurre cuando el orden de ejecución de métodos es crítico. Por ejemplo, debes llamar `obj.inicializar()` antes de `obj.procesar()`, si no, falla. Es un acoplamiento no entre clases distintas, sino entre métodos de la misma clase u objeto a lo largo del tiempo [33] [34]. Es considerado un *code smell* porque obliga a los llamadores a conocer la secuencia exacta – viola encapsulamiento (el objeto expone un protocolo frágil).

- *Detección:* Busca métodos con nombres `init`, `setup`, `begin`, etc., o flags internos como `estado == CONFIGURADO` que son chequeados por otros métodos [35]. También revisa documentación/comentarios que digan "llamar X *después* de Y".
- *Mitigación:* Una solución es usar el patrón *Template Method* o un constructor más completo. Por ejemplo, en vez de:

```
conexion = Conexion()
conexion.config(url, credenciales)
```

```
conexion.conectar()
conexion.enviar_datos(data)
```

Podrías hacer:

```
conexion = Conexion(url, cred)  # config en constructor
conexion.enviar_datos(data)     # internamente conecta si no estaba
conectado
```

O exponer un método de alto nivel que haga la secuencia correcta: `conexion.enviar(data)` que se encargue de configurar, conectar, etc., en orden.

- *Template Method:* Consiste en tener una función en la clase base que llama en orden específico a pasos (métodos) que pueden ser sobreescritos por subclases [36] [37]. Así garantizas el orden dentro de la implementación abstracta.
- *Verificación:* Si no puedes eliminar la secuencia (quizá es inevitable: p.ej., *transaction.begin(), transaction.commit()*), asegúrate de que esté claramente documentada y quizás controlada: ej., que `commit()` verifique internamente que `begin()` fue llamado, dando error claro en lugar de fallos silenciosos [38]. En la PR, si veo secuencia, pregunto "¿podemos reordenar o consolidar esto para que el orden correcto esté garantizado por diseño?".

En general, el objetivo de esta sección es reducir *acoplamiento*: tanto el acoplamiento *entre componentes* (Demeter, DIP, SRP) como el acoplamiento *temporal* dentro de un flujo. Un diseño desacoplado tiene módulos autónomos con interfaces claras y dependencias bien gestionadas, lo que facilita cambiar una pieza sin afectar otras, y probar piezas en aislamiento.

🏗️ Patrones por Caso de Uso (Cuándo Sí / Cuándo No)

A continuación listamos varios patrones de diseño (creacionales, estructurales, de comportamiento) con guía rápida:

- **Factory (Fábrica):**
- **Úsalo cuando:** Necesitas delegar la lógica de **creación de objetos complejos** o dependientes de contexto. Por ejemplo, según la configuración o tipo de datos, crear instancias de diferentes subclases. También útil para ocultar el proceso de creación al cliente (principio de abstracción).
- **Evítalo cuando:** La creación es trivial (una llamada a `new` con constructor simple). Introducir una fábrica para solo devolver `X()` sin lógica es sobreingeniería. Si solo hay una implementación posible, no hay qué fabricar.
- **Complejidad: Baja.** Añade una función o clase adicional, pero suele ser código simple. Mantenimiento bajo, ya que centraliza cambios de construcción (si el constructor de `X` cambia, solo la fábrica se modifica).

- **Ejemplo mínimo:**

```
class ExporterFactory:
    @staticmethod
```

```
    def create(format):
        if format == "CSV": return CSVExporter()
        if format == "JSON": return JSONExporter()
        raise ValueError("Formato no soportado")
# Uso:
exp = ExporterFactory.create(config.output_format)
exp.export(datos)
```

Aquí el Factory decide qué clase concreta (CSV/JSON) instanciar según formato.

- **Strategy (Estrategia):**

- **Úsalo cuando:** Tienes varios **algoritmos intercambiables** que cumplen la misma tarea con distintas implementaciones. Ejemplo clásico: estrategias de ordenamiento (rápido, burbuja) o de cálculo de precio (estándar vs descuento vs promociones). Permite cambiar de algoritmo en tiempo de ejecución configurando el objeto apropiado.
- **Evítalo cuando:** No hay realmente variantes del algoritmo. Si solo existe un método de hacerlo, meterlo en una interface `Strategy` con una sola implementación es innecesario. Tampoco si el cambio de comportamiento estaría mejor modelado con simples *parametrizaciones* (por ejemplo, pasar un comparador a sort podría ser suficiente en vez de estrategias separadas).
- **Complejidad: Baja.** Requiere definir una interfaz (abstracta) y múltiples clases concretas. Fácil de entender: cada estrategia es aislada. El coste es tener varias clases, pero cada una enfocada.

- **Ejemplo mínimo:**

```
class TaxStrategy:
    def calcular(self, compra): ...
class TaxEstandar(TaxStrategy):
    def calcular(self, compra): return compra.total * 0.21
class TaxReducido(TaxStrategy):
    def calcular(self, compra): return compra.total * 0.10

# Uso:
estrategia = TaxReducido() if compra.tipo == "alimentos" else TaxEstandar()
impuesto = estrategia.calcular(compra)
```

Según el tipo de compra escogemos la estrategia adecuada.

- **Adapter (Adaptador):**

- **Úsalo cuando:** Debes integrar una clase/módulo con **interfaz incompatible** en tu sistema, o cuando migras a nueva API manteniendo la vieja interfaz. El Adapter convierte la interfaz de una clase en otra esperada por el cliente. Ej: tienes una librería de terceros con métodos con nombres distintos; creas un Adapter que ofrece tus nombres esperados y delega a la librería.

- **Evítalo cuando:** Puedes modificar directamente la clase objetivo (si es código propio) para que siga la interfaz deseada. O si la diferencia de interfaz es mínima, a veces es sobrekill hacer una clase entera; en su lugar podrías usar funciones wrapper simples.
- **Complejidad añadida: Baja.** Es básicamente código de mapeo. Mantenimiento bajo salvo que las interfaces cambien; en ese caso actualizas el adaptador. Cuidado de no encadenar adaptadores (Adapter del Adapter es señal de mal diseño).

- **Ejemplo mínimo:**

```
class PayPalAPI:
    def send_payment(self, amount): ...
# Nuestro sistema espera interfaz PagoOnline con método pagar(cantidad)
class PayPalAdapter(PagoOnline):
    def __init__(self, paypal_api):
        self.paypal = paypal_api
    def pagar(self, cantidad):
        return self.paypal.send_payment(cantidad)
```

Aquí `PayPalAdapter` implementa la interfaz esperada (`PagoOnline.pagar`) adaptando la de PayPal (`send_payment`).

- **Facade (Fachada):**

- **Úsalo cuando:** Quieres proporcionar **una interfaz simplificada y unificada** a un subsistema complejo. Ej: una clase Fachada `SistemaPedidos` que internamente llama a Módulo de Clientes, Inventario y Facturación, exponiendo un método `procesarPedido()` en vez de que el código cliente llame a 3 sistemas. Ideal para ocultar complejidad o dependencias múltiples detrás de algo más simple.
- **Evítalo cuando:** La capa de fachada añade overhead sin mucho valor. Si el subsistema es simple o ya tienes una clase que coordina, la fachada extra no aporta. Tampoco confundir con controladores: si ya hay un patrón MVC/MVP, la fachada puede ser redundante.
- **Complejidad: Baja.** Es básicamente una clase que agrega llamadas. Puede considerarse *complejidad estructural media* si no se diseña bien, porque la Fachada puede crecer demasiado si absorbe mucha lógica. Debe permanecer delgada, delegando a componentes reales.

- **Ejemplo mínimo:**

```
class PedidoFacade:
    def __init__(self, mod_clientes, mod_invent, mod_fact):
        self.cli = mod_clientes; self.inv = mod_invent; self.fact =
mod_fact
    def procesar_pedido(self, pedido):
        cliente = self.cli.obtener(pedido.cliente_id)
        self.inv.reservar_stock(pedido.items)
```

```
        factura = self.fact.generar_factura(pedido, cliente)
        return factura
```

El cliente de alto nivel solo usa `PedidoFacade.procesar_pedido`, sin tratar directamente con las subsistemas de clientes, inventario, facturación.

- **Observer/Listener (Observador):**

- **Úsalo cuando:** Necesitas notificar a múltiples partes (desacopladas entre sí) sobre **eventos** que ocurren en un objeto. Patrón típico para implementar Event Emitter, suscripción a eventos (GUI, EDA local). Por ejemplo, un objeto *Tema* mantiene una lista de *Observadores* y les avisa cuando cambia de estado. Útil en arquitectura dirigida por eventos interna (no necesariamente microservicios, sino módulos dentro de una app).
- **Evítalo cuando:** La comunicación puede ser simplemente un llamado directo y la relación es uno a uno. Observer añade complejidad (registro/deregistro, orden indeterminado de notificación). Si no requieres un verdadero *desacople* (p. ej., en lugar de observer, puedes llamar un callback pasado), quizás no lo necesites.
- **Complejidad añadida: Media.** Implica manejar listas de suscriptores, posiblemente threads si es asíncrono, cuidado con referencias (evitar memory leaks manteniendo observadores muertos). Mantenimiento moderado: agregar nuevos observadores es fácil (se suscriben), pero debugging puede ser difícil porque la ejecución es discontinua.

- **Ejemplo mínimo:**

```
class EventoNuevoUsuario:
    pass  # solo una señal

class GestorUsuarios(Observable):
    def __init__(self):
        self.observers = []
    def subscribe(self, obs):
        self.observers.append(obs)
    def alta_usuario(self, usuario):
        ... # lógica de alta
        for obs in self.observers:
            obs.notificar(EventoNuevoUsuario(), usuario)
# Observador ejemplo:
class EnviarEmailBienvenida:
    def notificar(self, evento, usuario):
        if isinstance(evento, EventoNuevoUsuario):
            enviar_email(usuario.email, "Bienvenido!")
```

`GestorUsuarios` notifica a todos sus suscriptores cuando se da de alta un usuario nuevo.

- **Command (Comando):**

- **Úsalo cuando:** Quieres encapsular una **solicitud o acción como objeto**, de modo que puedas parametrizar clientes con colas de peticiones, soporte a undo/redo, o ejecutar operaciones diferidas. Un caso es un sistema de tareas: cada tarea es un objeto Command con un método `execute`. Es útil también para separar quién pide una acción de quién la ejecuta (ej: GUI genera comandos que un motor procesa).
- **Evítalo cuando:** La acción es simple y no necesitas toda esa flexibilidad. Implementar undo/redo o colas si no son requeridas es YAGNI. No conviertas cada llamada a método en un Command por default, úsalo intencionalmente.
- **Complejidad: Media.** Requiere crear una clase por cada tipo de comando, o al menos una jerarquía. Pero aporta gran flexibilidad (logging de comandos, reintentos, etc.). Mantenimiento: si hay muchos comandos, puede ser pesado, pero cada uno es aislado.

- **Ejemplo mínimo:**

```
class Command:
    def execute(self): pass  # interface

class CrearUsuarioCmd(Command):
    def __init__(self, repo, datos): self.repo = repo; self.datos = datos
    def execute(self):
        self.repo.guardar_usuario(self.datos)

# Cola de comandos
cola = []
cola.append(CrearUsuarioCmd(repo, {"nombre": "Ana"}))
# ... más comandos
# Ejecución diferida:
for cmd in cola:
    cmd.execute()
```

Aquí encapsulamos la acción de crear usuario. Podríamos loguear comandos, rehacerlos, etc., independientemente del origen.

- **Template Method (Método Plantilla):**

- **Úsalo cuando:** Tienes un **algoritmo general con pasos variables**, y quieres que las subclases definan esos pasos sin cambiar la estructura global. La clase base implementa el método plantilla que llama a métodos abstractos en cierto orden. Ej: una clase base `ExportadorDatos` con método `exportar()` que hace `abrirArchivo(); escribirCabecera(); escribirDatos(); cerrarArchivo`. Las subclases (`ExportadorCSV`, `ExportadorXML`) implementan los métodos específicos (cabecera, formato de datos) pero la secuencia de abrir/escribir/cerrar está fija.
- **Evítalo cuando:** La variación entre casos es muy grande, al punto que es difícil diseñar una plantilla común. Si terminas con muchos métodos abstractos que algunas subclases ni usan, mejor usar estrategia u otra composición. También evítalo si puedes lograrlo con composición claramente; Template Method obliga a herencia, que puede limitar flexibilidad.

- **Complejidad: Baja-Media.** En cuanto a código, es simple: herencia con métodos abstractos. Pero la desventaja es la rigidez de la estructura y la curva de aprendizaje para entender el flujo (hay que saltar entre clase base y subclase mentalmente). Mantenimiento bueno si los pasos son estables; si el algoritmo cambia mucho, podrías tener que tocar la base y subclases coordinadamente.

- **Ejemplo mínimo:**

```
class ProcesadorArchivo:
    def procesar(self):
        self.leer()       # llamados en orden fijo
        self.transformar()
        self.guardar()
    def leer(self): raise NotImplementedError
    def transformar(self): raise NotImplementedError
    def guardar(self): raise NotImplementedError

class ProcesarMayus(ProcesadorArchivo):
    def leer(self): self.data = obtener_texto()
    def transformar(self): self.data = self.data.upper()
    def guardar(self): guardar_texto(self.data)
```

Llamar `ProcesarMayus().procesar()` ejecuta la plantilla: leer->transformar->guardar con la implementación concreta.

- **Builder (Constructor en etapas):**

- **Úsalo cuando:** Debes construir un objeto complejo **paso a paso**, especialmente si tiene muchos parámetros opcionales o configuraciones. El patrón Builder separa la creación de la representación: en lugar de un gran constructor con 10 params, usas métodos encadenados para ir armando el objeto. Ej: construcción de una consulta SQL, de un objeto JSON grande, de una `Casa` con muchas partes opcionales (piscina, garage, etc.).
- **Evítalo cuando:** El objeto a crear es simple o siempre necesitas todos los parámetros. En tal caso, un constructor normal o factores estáticos con diferentes combinaciones bastan. También si el lenguaje soporta *named optional parameters* cómodamente (como Python), el beneficio del Builder es menor.
- **Complejidad añadida: Baja.** Es básicamente sintáctico: una clase builder con métodos que setean campos y devuelven self. El costo es más líneas de código, pero mejora la legibilidad de la creación cuando hay muchos campos. Mantenimiento bajo; si el objeto agrega un atributo, agregas un método al builder.

- **Ejemplo mínimo:**

```
class UsuarioBuilder:
    def __init__(self): self.attrs = {}
    def con_nombre(self, nombre):
        self.attrs["nombre"] = nombre; return self
    def con_edad(self, edad):
```

```
        self.attrs["edad"] = edad; return self
    def build(self):
        return Usuario(**self.attrs)
# Uso:
usuario = UsuarioBuilder().con_nombre("Ana").con_edad(30).build()
```

Se evitó tener `Usuario(nombre, edad, direccion=None, telefono=None, ...)` con muchos None. Solo se establece lo necesario.

- **Repository (Repositorio):**

- **Úsalo cuando:** Quieres una **capa de abstracción sobre el acceso a datos**, para desacoplar la lógica de negocio de los detalles de persistencia. Un repositorio actúa como una colección en memoria aunque detrás haya BD u otro storage. Ej: `UsuarioRepository` con métodos `obtener_por_id`, `buscar_por_nombre`, etc., en lugar de que la lógica tenga SQL/raw queries. Facilita pruebas (puedes meter un repo en memoria) y centraliza las consultas.
- **Evítalo cuando:** Tienes un ORM o framework que ya provee una abstracción similar (por ej., Active Record, donde los modelos ya manejan persis). Hacer repositorios encima de Active Record puede ser redundante. O si la aplicación es pequeña, a veces consultar directamente no es pecado; añade repos si planeas cambiar de fuente de datos o si quieres aislar consultas complejas.
- **Complejidad añadida: Media.** Añade una capa, pero bien justificada. Mantenimiento moderado: los repos crecen a medida que agregas métodos de búsqueda. Debes evitar que se conviertan en "clases Dios" de acceso a cualquier cosa – mantenlos específicos por agregado o entidad.

- **Ejemplo mínimo:**

```
class UsuarioRepository:
    def __init__(self, db_conn): self.db = db_conn
    def obtener(self, id):
        fila = self.db.query("SELECT * FROM usuarios WHERE id=%s", [id])
        return Usuario.from_row(fila) if fila else None
    def guardar(self, usuario):
        self.db.execute("INSERT ...", [usuario.nombre, usuario.email])
```

La capa de negocio usaría `repo.obtener(id)` en vez de meter SQL. En tests, podrías sustituir `db_conn` por un stub que retorna datos fijos.

- **CQRS (Command Query Responsibility Segregation):**

- **Úsalo cuando:** La forma óptima de **escribir datos es muy diferente de la de leerlos**, o tienes altísima carga de lectura que justifica separarlas. CQRS separa el modelo de comandos (mutaciones) del modelo de consultas (lecturas). Por ejemplo, en una app, las operaciones de actualizar inventario usan un modelo de dominio rico con validaciones, mientras que para mostrar el catálogo quizás sea mejor tener proyecciones ya listas (denormalizadas) para rendimiento. También se usa junto a Event Sourcing y escalabilidad (es más fácil escalar lecturas y escrituras independientemente).

- **Evítalo cuando:** La complejidad no lo amerita. Si tu dominio es sencillo, dividir cada entidad en dos modelos puede duplicar código y confundir. También si las lecturas no son un problema de performance o las escribis/lees consistentemente de la misma estructura, una aproximación tradicional basta.
- **Complejidad: Alta.** Introduce dos modelos y a menudo eventual consistency (lo escrito no refleja instantáneamente en lo leído, hay un delay de propagación). Requiere pensar en sincronicación de proyecciones, duplicación de datos. Mantenimiento complejo: cualquier cambio de esquema quizás deba replicarse en dos lugares (comando y consulta). Por eso se justifica solo en sistemas grandes.

- **Ejemplo (conceptual):**

```
# Comando:
class OrdenCommandHandler:
    def manejar_crear_orden(cmd):
        # valida negocio, guarda en BD normalizada
        publicar_evento(OrdenCreada(...))
# Consulta:
class OrdenesQueryService:
    def listar_ordenes_cliente(cliente_id):
        # lee de una vista/tabla optimizada para consulta (ej. join ya
materializado)
        return DB.query("SELECT * FROM vista_ordenes WHERE cliente_id=?",
cliente_id)
```

Aquí tras crear orden, un evento poblaría la vista_ordenes (o se hace mediante triggers/ proyecciones). El query service no usa las entidades de dominio sino una representación optimizada.

- **Saga/Orchestrator (Saga de compensación):**

- **Úsalo cuando:** Tienes una **transacción distribuida** que involucra múltiples servicios o agregados y necesitas mantener consistencia eventual sin locks globales. Una Saga orquesta pasos de una operación, con la posibilidad de ejecutar pasos de compensación si alguno falla, deshaciendo los previos. Ej: en un pedido en microservicios: Order Service crea orden, Payment Service cobra, Inventory Service reduce stock. Si Payment falla, la Saga invoca compensación (cancelar orden, quizás revertir stock). Úsalo para flujos **longos y distribuídos** donde 2PC no es viable [39] [40] .
- **Evítalo cuando:** Todo se puede hacer en una transacción ACID local (monolito) más simple; o si el flujo no requiere deshacer nada (entonces una simple secuencia con manejo de errores basta). Implementar Saga es complejo, no vale la pena para operaciones triviales o una llamada remota única.
- **Complejidad añadida: Alta.** Debes manejar múltiples estados, orquestación centralizada vs coreografía [41] , y casos de inconsistencia parcial. Testing es complicado (simular fallos en mitad de saga). Mantenimiento complejo: cambiar una saga implica pensar en consecuencias en cadena.
- **Ejemplo mínimo (orquestación):**

```
class SagaCrearPedido:
    def __init__(self): self.estado = "INIT"
```

```
    def iniciar(pedido):
        pedido.estado = "PENDING"; save(pedido)
        evento = EventoOrdenCreada(pedido.id)
        publicar(evento)
        self.estado = "ORDER_CREATED"
    def compensar():
        if self.estado == "ORDER_CREATED":
            cancelar_pedido(pedido.id)
```

Nota: En orquestación real, la saga recibiría respuestas de cada servicio y decidiría siguiente paso [42]. En coreografía, cada servicio escucha eventos y emite sus propios eventos. Este ejemplo es simplificado: tras crear orden, publica un evento; Payment Service escuchará, intentará cobrar. Si falla, enviará evento de pago fallido que Order Service capturará para compensar cancelando la orden [43].

- **Coste de mantenimiento:** Alto, porque debes pensar en *todos los caminos de fallo* y mantener las compensaciones actualizadas con los cambios en servicios. Úsalo con moderación y documenta bien el flujo.

- **Circuit Breaker (Cortacircuitos):**

- **Úsalo cuando:** Llamas a **servicios externos** o recursos poco fiables que pueden fallar o tardar. Un circuit breaker evita que tu aplicación quede colgada intentando algo repetidamente que probablemente fallará. Tras X fallos seguidos, el breaker se "abre" y las siguientes llamadas fallan inmediatamente durante un intervalo, en lugar de tratar y volver a fallar lentamente [25]. Después de un tiempo, el breaker puede pasar a "medio abierto" y probar una llamada para ver si se recuperó. Es vital en arquitecturas de microservicios para **evitar cascadas** (si servicio A llama a B, y B cae, A no se quede esperando indefinidamente o saturando B con peticiones).
- **Evítalo cuando:** Las dependencias son muy confiables o locales, y un retry simple es suficiente. En un monolito con llamadas a base de datos síncronas, un circuit breaker no aplica (la BD es local o falla catastróficamente). O si la frecuencia de llamadas es baja, quizá manejar error directamente sea aceptable.
- **Complejidad añadida: Media.** La implementación añade estados (Closed, Open, Half-Open), contadores de fallos, temporizadores para reset. Hay librerías que lo proveen (e.g. Polly en .NET, resilience4j en Java, etc.). Mantenimiento: hay que tunear umbrales (¿cuántos fallos para abrir? ¿timeout de qué duración?) y monitorear.
- **Ejemplo conceptual:**

```
class CircuitBreaker:
    def __init__(self, max_fallas=5, tiempo_reset=60):
        self.estado = "CLOSED"
        self.fallas = 0
        self.ultimo_fallo = None
        self.max_fallas = max_fallas
        self.tiempo_reset = tiempo_reset
    def llamada(self, funcion, *args):
        if self.estado == "OPEN":
```

```
            if time() - self.ultimo_fallo < self.tiempo_reset:
                raise CircuitOpenError()
            else:
                self.estado = "HALF"
        try:
            resp = funcion(*args)
            self.fallas = 0; self.estado = "CLOSED"
            return resp
        except Exception as e:
            self.fallas += 1; self.ultimo_fallo = time()
            if self.fallas >= self.max_fallas:
                self.estado = "OPEN"
            raise e
```

Uso: `breaker.llamada(servicio.remota, param1)`. Si falla repetidamente, el breaker abre y lanza inmediatamente `CircuitOpenError` sin ejecutar `servicio.remota`.

- **Mantenimiento:** calibrar max_fallas y tiempo_reset según tu caso (demasiado corto: falso positivo abre; demasiado largo: tardas en recuperarte).

- **Outbox (Transactional Outbox):**

- **Úsalo cuando:** Necesitas garantizar **entrega de eventos/mensajes** al mismo tiempo que una operación en BD, sin usar transacción distribuida (2PC). El patrón Outbox [44] dice: cuando algo pasa (p.ej. creaste una orden en tu DB), en lugar de intentar publicar un evento en el mismo momento (riesgo de que la publicación falle después de commit, etc.), *escribe el evento en una tabla outbox en la misma transacción que la orden*. Luego un proceso separado lee periódicamente la tabla outbox y envía los eventos al broker, marcándolos como enviados [45]. Así aseguras atomicidad: o se guarda todo (orden y evento), o nada; y si la publicación falla, puedes reintentarlo leyendo la tabla.
- **Evítalo cuando:** Puedes usar directamente Event Sourcing (donde la fuente de verdad son eventos, lo cual inherentemente resuelve esto) o si tu sistema es monolítico sin necesidad de eventos persistentes. También, si la pérdida ocasional de un mensaje no es catastrófica y la complejidad de outbox no compensa.
- **Complejidad: Media.** Requiere infraestructura: una tabla extra, un demonio/servicio para reenviar mensajes, manejo de duplicados (si se reenvía un evento que ya salió). Pero mejora confiabilidad de integraciones. Mantenimiento: vigilar la tabla outbox (limpieza de eventos viejos), asegurar idempotencia en consumidores porque puede haber duplicados [25].
- **Ejemplo mínimo (concepto):**
  - Al guardar Pedido: en la misma transacción SQL: `INSERT Pedido...; INSERT Outbox(evento="PedidoCreado", pedidoId=..., payload=...)`.
  - Un job cada 5s: `SELECT * FROM Outbox WHERE enviado=false;` por cada fila: intenta publicarla a Kafka/Rabbit; si éxito, marca enviado=true (o borra la fila). Si falla, la deja para el siguiente intento.

- **Trade-off:** Duplica ligeramente datos (eventos en DB), pero elimina inconsistencia entre estado y eventos. Los consumidores deben estar preparados para posibles duplicados (si se publicó y falló justo antes de marcar enviado, se reenviará) [25].

- **Event Sourcing (Fuente de Eventos):**

- **Úsalo cuando:** Quieres almacenar **todos los cambios de estado como una secuencia de eventos** en lugar del estado actual. Es útil para tener un historial completo (audit trail) y reconstruir estado pasado o derivar proyecciones múltiples. Por ejemplo, en finanzas, guardar cada crédito/débito en vez de solo el saldo final. Combinado con CQRS a menudo: eventos -> proyecciones. Útil si necesitas *replay* de eventos para depurar o aplicar lógica nueva retroactivamente.
- **Evítalo cuando:** La complejidad es innecesaria. Si no necesitas el historial detallado o podrías guardarlo en logs, mejor usar modelos tradicionales. Event Sourcing complica las lecturas (hay que reconstruir estado sumando eventos), y el versionado de eventos es un dolor (cuando cambian esquemas de datos en eventos). No usar a menos que explicitamente requieras historización o escalabilidad de writes/reads muy diferenciada.
- **Complejidad: Alta.** Necesitas definir esquemas de eventos, manejar *snapshots* (tomar estado completo cada X eventos para no re-reproducir miles siempre), herramientas para migrar eventos antiguos al cambiar lógica. Testing también cambia de enfoque. Mantenimiento engorroso: un cambio en regla produce un nuevo tipo de evento; debes mantener compatibilidad con eventos viejos o migrarlos.
- **Ejemplo conceptual:**

  ```
  # En lugar de: cuenta.saldo = 100
  # Tienes:
  eventos_cuenta = [CuentaCreada(id, cliente), DepositoRealizado(id, 50),
  RetiroRealizado(id, 20)]
  # Estado actual se calcula aplicando: saldo = 0 +50 -20 = 30.
  ```

  La “fuente de verdad” es la lista de eventos. Para obtener el saldo, los eventos se reproducen (o se aplica un acumulador).
- **Cuándo sí:** Auditoría completa requerida (aplicaciones financieras, historiales de datos), alta frecuencia de actualizaciones pequeñas donde escribir eventos append-only es más rápido que actualizar campos (p. ej. IoT), y cuando se prevén muchas proyecciones diferentes de los mismos datos (puedes generar distintas vistas a partir de los eventos).
- **Cuándo no:** App CRUD típica, o sistemas donde borrar datos es necesario (en event sourcing, borrar es complicado, porque los eventos históricos quedan; hay que aplicar eventos de compensación).

(Nota: Existen más patrones – p. ej. Decorator, State, Visitor, Proxy *– pero nos centramos en los mencionados en la consulta y los más relevantes al contexto dado.)*

Cada patrón conlleva un **coste cognitivo**, así que aplícalos solo cuando su beneficio (flexibilidad, claridad, extensibilidad) supere ese coste. Siempre pregúntate: *“¿Este patrón resuelve un problema real que tenemos, o lo estoy forzando por teoría?”*. Si es lo segundo, mejor no añadirlo.

📣 Arquitectura Dirigida por Eventos (EDA)

En EDA, los componentes no interactúan directamente sino que **se comunican enviando y recibiendo eventos** a través de un bus o cola. Esto logra un acoplamiento laxo y sistemas asíncronos escalables, pero introduce consideraciones especiales:

Event Bus / Listeners (Publicación/Suscripción):

- **Contratos de eventos:** Define claramente la estructura de cada tipo de evento. Un evento es un mensaje con un esquema (campos obligatorios, opcionales). Es buena práctica versionarlos: si necesitas cambiar el esquema, en lugar de romper el existente, crea una nueva versión de evento (`UsuarioCreado.v2`) [46]. Mantén compatibilidad hacia atrás siempre que sea posible, para no obligar a todos los consumidores a actualizar simultáneamente. Usa un *schema registry* o, al menos, documenta los formatos (ej: JSON schema) y asegúrate de actualizar documentación en cada cambio.
- **Idempotencia de consumidores:** *At-least-once delivery* es lo común en la práctica (los brokers suelen entregar duplicados si no confirman) [47]. Diseña cada listener de evento para poder recibir el mismo evento dos o más veces sin causar efectos duplicados. Cómo lograrlo: por ejemplo, incluye un `event_id` único en cada evento [13], y haz que el consumidor lleve un registro de IDs ya procesados (en memoria o base de datos). O si aplicas cambios de estado, verifica si ya se aplicó la operación antes de repetir (p. ej., un evento "PagoRealizado" con ID de pago, ignorar si ese pago ya se marcó como realizado).
- **Orden y concurrencia:** Un bus pub/sub por defecto no garantiza orden global, solo quizás orden por tópico o por clave de partición. No asumas que eventos llegarán en secuencia exacta de emisión si van a distintos tópicos o particiones (p.ej., "UsuarioCreado" podría llegar después de "UsuarioActualizado" en ciertas condiciones). Si el orden importa, usa la misma clave de partición para eventos relacionados o introduce un mecanismo de secuencias/version en los datos. Si no, haz consumidores robustos para procesar eventualmente cuando lleguen todos los eventos necesarios (idempotencia + comprobaciones).
- **Durabilidad vs efímero:** Decide si los eventos deben persistir (event streaming) o solo transmitir y descartar. Para integración entre servicios, por lo general los brokers (Kafka, RabbitMQ) almacenan hasta que son consumidos. Asegúrate de configurar retenciones adecuadas (tiempo, tamaño) según necesidades de replay o reprocessing.

Semántica de entrega:

- *At-least-once:* Significa que un evento será entregado **al menos una vez**, pero potencialmente duplicado. Es el más común. Requiere idempotencia en consumidores [47].
- *At-most-once:* Entrega como mucho una vez (puede perderse). Pasa si no hay reintentos, o si usamos UDP por ejemplo. No es deseable para eventos críticos, pero a veces aceptable para logs/telemetría.
- *Exactly-once:* Es el ideal teórico: entregar cada evento una vez. En la práctica, muy difícil sin coordinar extremos. Se logra con combinaciones de idempotencia y outbox (producir sin duplicar) y consumidores idempotentes. Algunos sistemas de streaming intentan exactly-once (Kafka tiene *transactions* para evitar duplicar en sinks), pero igual asume idempotencia en algún nivel. **Realidad:** Diseña pensando en al menos una vez. "Exactly-once" suele implicar *dos veces de trabajo, y luego deduplicar*.

Trazabilidad (Observabilidad) en EDA:

- **Correlación de trazas distribuidas:** Usa un identificador de correlación end-to-end. Por ejemplo, un request entra al sistema (HTTP request ID), se genera un evento A con ese ID en header; un servicio B procesa evento A, al emitir evento B copia el ID; así todos los logs de eventos relacionados comparten el mismo `correlation_id` [13]. Esto permite usar herramientas de tracing (Jaeger,

Zipkin) para reconstruir el flujo a través de servicios. Emplea formatos estándar si posible (Trace Context W3C, etc.).

- **Logs estructurados en consumidores:** Loguea cada evento recibido con su tipo, ID y resultado del procesamiento (ej. "PedidoEntregado event X procesado ok"). Si algo falla, incluye el ID de evento en el error log para buscar en cola si fue reentregado etc.
- **Métricas:** Instrumenta contadores de eventos procesados por tipo, tasas de procesamiento, tamaños de cola. También métricas de lag (cuánto se retrasó un evento desde que fue enviado hasta procesado, si el broker lo permite). Estas métricas alertan de cuellos de botella (si lag crece, tus consumidores no dan abasto → escalar o revisar).
- **Dead Letter Queues (DLQ):** Configura colas de basura para eventos que no se pudieron procesar tras ciertos reintentos. En EDA robusta, un consumidor no debería bloquearse para siempre con un mensaje tóxico; tras n intentos fallidos, envía ese evento a una DLQ para análisis manual, y sigue con el siguiente. Monitorea la DLQ: es un síntoma de casos no manejados.

Eventual consistency y diseño EDA: Acepta que en EDA, los datos **serán eventualmente consistentes**: cuando ocurre un evento, varios servicios reaccionan y sus estados locales se actualizan con algún retraso. Aplica *diseño por compensación*: si consultas rápidamente en un servicio B algo que cambió en A, puede que aún no reciba el evento, así que maneja la ausencia de información con gracia (p. ej., reintentar consulta después de un intervalo, o diseñar tus UI para reflejar estados pendientes). A veces se introducen eventos de confirmación ("OrderCompleted") que los consumidores esperan para saber que ya está todo listo.

Event Sourcing vs Event-driven: No confundir: EDA puede usar eventos como mecanismos de integración, pero el estado interno de cada servicio puede ser tradicional (CRUD). Event Sourcing es guardar estado como eventos. Puedes hacer EDA sin event sourcing, y viceversa.

Seguridad y gobernanza de eventos: Define claramente quién puede producir qué eventos (evitar productores inesperados), y versiona con compatibilidad. Un esquema registry con compatibilidad forward/backward es muy útil. En PRs que modifican eventos, revisa que no rompa consumidores existentes o planifica actualizaciones coordinadas.

En resumen, EDA aporta gran flexibilidad y escalabilidad, pero exige: idempotencia, trazabilidad, buen diseño de contratos de eventos, y consciencia de la consistencia eventual. En cada PR de un componente EDA, verifica: *¿Se maneja duplicados? ¿Se loguea/correlaciona bien? ¿Se actualizó la documentación del evento si cambió? ¿Qué pasa si el evento llega fuera de orden?*. Con este rigor, evitas muchos dolores de cabeza en producción.

📚 Glosario Breve (Conceptos y Definiciones)

- **Acoplamiento secuencial (Temporal Coupling):** Dependencia en el **orden** de ejecución de operaciones. Ocurre cuando un módulo requiere que sus métodos sean llamados en secuencia específica para funcionar correctamente [33]. Es un antipatrón que puede mitigarse reestructurando el código para garantizar el orden internamente (p. ej., con Template Method).
- **Backpressure:** Mecanismo de **control de flujo** que frena la emisión de datos cuando el receptor está abrumado [27]. Previene overflow de buffers/colas comunicando al productor que baje el ritmo o espere hasta que el consumidor se ponga al día.

- **Circuit Breaker:** Patrón de resiliencia que **corta llamadas a un servicio** tras repetidas fallas, evitando insistir en algo caído [25]. Tiene estados (cerrado, abierto, medio-abierto) y restaura llamadas cuando detecta recuperación. Protege sistemas distribuidos de cascadas de fallos.
- **Clean Code:** Conjunto de prácticas y principios para escribir código **legible, simple y libre de suciedad** (bugs ocultos, duplicaciones, nombres pobres). Popularizado por Robert C. Martin, enfatiza: nombres claros, funciones pequeñas, minimizar dependencias, manejo explícito de errores, etc.
- **Costuras (Seams):** Puntos en el código donde puedes **alterar el comportamiento sin modificarlo directamente** [31]. Ejemplo: interfaces o parámetros que permiten inyectar implementaciones diferentes en tests. Identificar costuras facilita pruebas y extensibilidad.
- **Demeter, Ley de (Principio del menor conocimiento):** Una unidad de software solo debe interactuar con sus **colaboradores directos** y no con los internamente anidados de otros [1]. "No hables con desconocidos." Mejora encapsulamiento reduciendo llamadas en cadena.
- **Dependencia (Inyección de –):** Técnica donde las **dependencias de un objeto** (sus colaboradores necesarios) se proporcionan desde fuera, en lugar de crearlas dentro. Permite decoupling y cumplir DIP [9]. Ej: pasar un `Repositorio` al servicio en el constructor.
- **DRY (Don't Repeat Yourself):** Principio de **no duplicar conocimiento** en el código. Cada pieza de lógica o información debe tener una única fuente autorizada. Evita divergencia de comportamiento y reduce esfuerzo de cambio. (Ver también: Duplication vs Wrong Abstraction).
- **EDA (Event-Driven Architecture):** Arquitectura donde las **comunicaciones se basan en eventos** asíncronos. Los componentes publican eventos y otros los consumen reaccionando a ellos, en lugar de invocaciones directas. Promueve bajo acoplamiento y escalabilidad, a costa de mayor complejidad en consistencia y trazabilidad.
- **Event Sourcing:** Patrón de almacenamiento donde el **estado se deriva de eventos históricos** almacenados. En lugar de guardar el estado actual de una entidad, se almacenan todas las modificaciones (eventos) y se recrea el estado calculándolos. Provee historial completo y flexibilidad de proyección, con sobrecarga de complejidad.
- **Idempotencia:** Propiedad de una operación que, si se ejecuta **múltiples veces**, el resultado es el mismo que ejecutarla una vez. En sistemas distribuidos y reintentos es vital: por ejemplo, procesar dos veces un mismo mensaje produce un único efecto. Se logra controlando que la segunda ejecución reconozca que ya se hizo la acción antes (p.ej., ignorar duplicados mediante IDs) [47].
- **Overengineering (Sobrecodificación):** Diseñar o implementar con **más complejidad de la necesaria** para cumplir los requisitos actuales. Incluye anticipar requerimientos no confirmados, usar patrones/herramientas "por si acaso", abstraer en exceso. Resulta en código difícil de entender y mantener [3]. Se combate con KISS, YAGNI y evaluando ROI de cada decisión técnica.
- **Refactoring:** Proceso de **reorganizar/limpiar el código sin cambiar su funcionalidad** externa. Mejora la estructura interna: nombra mejor, extrae funciones, separa responsabilidades, elimina duplicación. Idealmente respaldado por tests que garanticen que nada se rompió. Es una actividad continua para pagar deudas técnicas.
- **Repository (Repositorio):** Patrón que actúa como **colección simulada para objetos de dominio**, abstrae operaciones de persistencia. Por ej., en lugar de usar SQL en la lógica, se llama `repo.obtener_cliente(id)`. Facilita intercambiar la implementación (BD vs mock) y centraliza consultas.
- **Result (Patrón de Resultado):** Objeto que **representa éxito o error** de una operación en lugar de usar excepciones para control de flujo. Contiene típicamente o bien el valor resultante (Ok) o información de error (Err), obligando al llamador a manejar ambos casos explícitamente [6]. Muy utilizado en lenguajes funcionales para un manejo de errores más explícito.

- **Sequential Coupling (Acoplamiento Secuencial):** (Ver Acoplamiento secuencial) Sinónimo en inglés de dependencia en orden de llamadas.
- **SOLID:** Acrónimo de cinco principios de diseño orientado a objetos: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion. Enfocados en crear software **fácil de mantener y extender** evitando acoplamiento excesivo y diseños frágiles. (SRP y DIP ya se mencionaron; OCP: módulos abiertos a extensión pero cerrados a modificación; LSP: subclases deben poder usarse donde la base sin sorpresa; ISP: no forzar a implementar interfaces gigantes - mejor muchas pequeñas).
- **SRP (Single Responsibility Principle):** (Ver Single Responsibility). Cada clase o módulo con **una sola responsabilidad** o motivo de cambio.
- **Feature Toggle (Flag de Función):** Técnica para **habilitar o deshabilitar funcionalidad en producción sin desplegar código nuevo** [48] . A través de configuraciones (flags booleanas, porcentajes de habilitación, etc.), permite realizar *deploy contínuo* de código inactivo hasta que se "enciende" la feature. Debe usarse con moderación y gestionarse (limpiar toggles viejos), porque agrega lógica condicional en el código (ruta on/off).
- **Race Condition (Condición de carrera):** Situación en concurrencia donde el **resultado del software depende del orden/timing no determinista** de ejecuciones. Dos hilos accediendo a la misma variable pueden intercalar operaciones de manera imprevista, causando inconsistencias. Prevenir con sincronización, inmutabilidad, etc.
- **YAGNI:** "You Ain't Gonna Need It" – **No lo vas a necesitar** [5] . Principio XP que aconseja *no implementar algo hasta que realmente sea necesario*. Evita trabajo desperdiciado y sobreingeniería enfocándose solo en requisitos concretos actuales.

(Los términos están en orden alfabético para fácil referencia; algunos están en inglés pues se usan tal cual en jerga técnica.)

Antipatrón vs. Patrón

Antipatrón	**Síntoma** (qué se ve)	**Impacto negativo**	**Patrón/Solución propuesto**
God Object/Clase Dios	Módulo (clase) hace de todo: miles de líneas, campos y métodos de varias temáticas.	Difícil de mantener, cada cambio arriesga efectos colaterales; testing casi imposible por dependencias internas.	*Single Responsibility*, dividir en clases más pequeñas por rol. Aplicar *Facade* si es interfaz externa.
Spaghetti Code	Código sin estructura clara, flujos enmarañados con dependencias globales y secuencias lógicas ocultas.	Altísima fragilidad ante cambios; nadie entiende completamente el flujo, prolifera la deuda técnica.	*Refactoring* hacia estructura: introducir funciones bien nombradas, eliminar dependencias globales con *inyección*, quizás imponer capas (MVC, etc.).

Antipatrón	**Síntoma** (qué se ve)	**Impacto negativo**	**Patrón/Solución propuesto**
Gold Plating (Sobreingeniería)	Funcionalidad o abstracción que no tiene uso actual (código “de más”).	Complejidad extra a leer/ejecutar, posibilidad de bugs en algo que ni se necesita; tiempo de desarrollo/ performance malgastado.	*YAGNI*: eliminar código innecesario. Re-evaluar requisitos, borrar o deshabilitar secciones no utilizadas. Mantenerlo simple (*KISS*).
Shotgun Surgery (cambios dispersos)	Un pequeño cambio de requisito implica modificar muchas clases/módulos a la vez.	Alto riesgo de error, baja cohesión. Indica mal encapsulamiento (lógica única es duplicada o repartida).	*Refactor* para localización de la responsabilidad: agrupar lo que cambia junto. Aplicar *DRY* donde relevante o introducir *Design Patterns* (p.ej., Strategy en vez de varios ifs dispersos).
Lava Flow	Se arrastra código heredado que nadie usa (muerto) o entiende, pero que sigue en la base.	Dificulta navegación, puede ocultar bugs, confunde a nuevos devs; posible deuda de seguridad.	*Remoción de código muerto.* Escribir *tests de caracterización* si se teme borrarlo, luego eliminar gradualmente componentes no usados.
Memory Leaks lógicos (acoplamiento oculto)	Observadores no removidos, caches estáticas sin limpiar, singletons que acumulan estado.	Degrada performance con el tiempo, comportamiento impredecible en larga ejecución.	*Weak references* para observers, asegurar unsubscribe. Patrón *Observer* bien implementado. Monitoreo de recursos, aplicar *Flyweight/Pooling* donde corresponda para reutilización controlada.
Premature Optimization	Código complejo para optimizar algo no crítico (ej. hacks de bajo nivel, caches prematuras).	Aumenta complejidad, a veces empeora rendimiento general por agregar overhead en todas las ejecuciones cuando el caso optimizado es raro.	*KISS* + *Profile First*: escribe simple, luego mide. Optimiza solo partes hot-spot identificadas. Documenta las decisiones de optimización (y solo hazlas si el beneficio es claro).

Antipatrón	**Síntoma** (qué se ve)	**Impacto negativo**	**Patrón/Solución propuesto**
Sequential Coupling	Múltiples métodos deben ser invocados en secuencia fija; p.ej., init() -> doX() -> finish().	Uso incorrecto rompe funcionalidad; difícil de usar correctamente sin leer documentación interna.	*Template Method* para forzar orden interno, o un método fachada que llame en orden adecuado. Alternativamente, construir objeto en estado válido desde el inicio (constructor con todos los datos).
Excepción para flujo normal	Usar throw/except para casos esperables (p. ej., fin de iteración en vez de condición, o devolver de función un resultado válido vía excepción).	Reduce rendimiento (lanzar excepciones es caro), confunde flujo, puede llevar a capturas vacías que esconden problemas.	*Patrón Result* para retornos esperables (ej. devolver None o Result.Err en vez de excepción). Excepciones solo para errores verdaderamente excepcionales [6] .
Microservicios "en cualquier caso" (granularidad excesiva)	Dividir la aplicación en demasiados servicios muy finos sin necesidad clara.	Complejidad de despliegue, sobrecarga de red, dificultad para mantener consistencia; básicamente un monolito distribuido con más fallas.	*Monolito modular* o servicios más cohesivos. Aplicar *Domain-Driven Design* para definir bounded contexts sensatos. Solo extraer a microservicio cuando hay justificación (equipos independientes, escalas diferentes, aislamiento de fallo requerido).

(Cada antipatrón puede tener varias soluciones; aquí se muestra la principal o la mencionada en este documento. Esta tabla sirve para reconocer "malos olores" típicos y recordar posibles remedios.)

Checklist de Revisión de PR (Resumen)

Al revisar un Pull Request, verifica rápidamente estos puntos para asegurar calidad y mantenibilidad:

- [ ] **Nombres claros y significativos:** Los identificadores (variables, funciones, clases) describen su propósito; no hay nombres confusos o abreviaturas innecesarias. *¿Puedes entender qué hace algo solo por su nombre?*
- [ ] **Tamaño y responsabilidad de módulos apropiados:** Las funciones no son excesivamente largas o haciendo múltiples tareas dispares. Clases enfocadas en un rol. *¿Cumplen SRP?*
- [ ] **Manejo explícito de errores:** No se silencian excepciones (p.ej. catch vacío). Se usan resultados o excepciones específicas. *¿Cada posible fallo se contempla y reporta de forma controlada?*
- [ ] **Evitando sobreingeniería:** No hay patrones/código añadido sin caso de uso actual. *¿Este PR añade complejidad que no resuelve un problema actual?* Si sí, sugerir eliminar o comentar por qué se anticipa necesario.

- [ ] **Eliminación de duplicación indebida:** Si se introdujo lógica similar a otra existente, se justifica (o se refactoriza común). *¿Hay funciones/blocks duplicados que deban unificarse?* O al contrario, ¿se unificó prematuramente algo que sería más claro separado?
- [ ] **Cumplimiento de convenciones (PEP8/estilo):** Formato consistente, indentación, espacios, linting pasado. *¿El código sigue las guías acordadas?* Nombres de constantes en mayúsculas, etc., según estándar.
- [ ] **Pruebas incluidas y significativas:** El PR viene con tests unitarios/integración para lo nuevo o cambiado. Las aserciones prueban la lógica (no solo "llamar y no explotar"). *¿Fallarían los tests si el código tuviera un bug?* [19] .
- [ ] **No rompe contratos existentes:** Cambios en interfaces públicas, esquemas de eventos o APIs tienen manejo de versión o migración. *¿Los consumidores existentes seguirán funcionando?* Si se deprecó algo, está anotado claramente.
- [ ] **Uso correcto de patrones:** Si se aplicó un patrón de diseño, está justificado y bien implementado (p.ej., Strategy realmente encapsula variantes, no es innecesario). *¿El patrón simplifica el código o lo enreda más?*
- [ ] **Concurrencia segura (si aplica):** Para código concurrente, sin dataraces evidentes: variables compartidas protegidas, estructuras atómicas o inmutables. *¿Se consideró la sincronización/ backpressure?* No hay operaciones críticas sin protección.
- [ ] **Logs y métricas adecuados:** Los logs añadidos tienen nivel apropiado, no exponen datos sensibles, y están desacoplados (no lógica de negocio mezclada). *¿Ayudarán estos logs a depurar en prod sin inundar?*
- [ ] **Ley de Demeter y encapsulamiento:** Las interacciones entre objetos no violan principios de bajo acoplamiento (no se ven cadenas largas de accesos). *¿Los módulos se comunican mediante interfaces claras o están hurgando en estructuras internas de otros?*
- [ ] **Documentación actualizada:** Si el PR cambia comportamiento notable o añade módulos, hay comentarios/docstrings actualizados. En especial para utilidades públicas: *¿Explica cómo usarlo o por qué se hizo así?*
- [ ] **Feature toggles limpios (si hay):** Si se introdujo un flag, el código bajo el toggle está correctamente encapsulado y el valor por defecto respeta la retrocompatibilidad. *¿Se planea cómo y cuándo quitar el toggle?* (Que no quede permanente).

Este checklist rápido ayuda a cubrir la mayoría de puntos clave discutidos. No reemplaza un análisis a fondo, pero asegura que nada crítico pase desapercibido durante la revisión.

Riesgos de sobreingeniería en esta guía

Finalmente, es importante mencionar que *incluso las guías de buenas prácticas pueden inducir sobreingeniería si se aplican sin juicio*. Los desarrolladores podrían obsesionarse con cumplir cada principio al pie de la letra, complicando soluciones simples. **Mitigaciones:**

- **Contexto sobre dogma:** Usa estas recomendaciones como herramientas, no obligaciones. Si una regla no encaja en tu caso, está bien romperla justificadamente. Por ejemplo, DRY es valioso, pero ya vimos que duplicar puede ser mejor en ciertos momentos.
- **Iteración y YAGNI:** No intentes implementar todos los patrones/diseños sugeridos de una vez en cada proyecto. Comienza simple; mejora el diseño cuando los requerimientos lo demanden o los

problemas aparezcan (refactor continuo). Esta guía ofrece un panorama de posibles mejoras, pero no todas aplican simultáneamente.

- **Costo-beneficio:** Para cada patrón o principio, considera el costo de implementarlo vs el beneficio en tu situación. ¿Merece la pena un Saga completo por una transacción que casi nunca falla? ¿Necesitas event sourcing para un módulo sencillo? Si la respuesta es no, no te sientas obligado por la teoría.
- **Revisiones colectivas:** Comparte con el equipo (aunque seas uno ahora, en el futuro puede crecer) las decisiones. Un código muy "ingenioso" que solo el autor entiende va contra la mantenibilidad. Asegúrate de que las soluciones se puedan explicar fácilmente a otro desarrollador.
- **Simplicidad como meta:** Recuerda el objetivo central: código limpio, robusto y mantenible. Si en nombre de "la guía" el código se vuelve más difícil de entender, retrocede y reevalúa. A veces, *lo más simple que podría funcionar* es la mejor elección.

En conclusión, aplicar buenas prácticas es un acto de equilibrio. Esta guía debe servir de mapa, pero el pilotaje en terreno real requiere criterio. Priorizando claridad y simpleza, e introduciendo mejoras graduales con base en problemas concretos, lograrás un código de alta calidad sin incurrir en sobreingeniería. ¡Happy coding! 4 5

---

1 Law of Demeter - Wikipedia
https://en.wikipedia.org/wiki/Law_of_Demeter

2 Law of Demeter in Java - Principle of Least Knowledge - GeeksforGeeks
https://www.geeksforgeeks.org/java/law-of-demeter-in-java-principle-of-least-knowledge/

3 4 16 24 How to avoid over-engineering | RST Software
https://www.rst.software/blog/how-to-avoid-over-engineering

5 You aren't gonna need it - Wikipedia
https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it

6 23 Functional Error Handling in .NET With the Result Pattern
https://www.milanjovanovic.tech/blog/functional-error-handling-in-dotnet-with-the-result-pattern

7 17 18 21 22 31 The key points of Working Effectively with Legacy Code | Understand Legacy Code
https://understandlegacycode.com/blog/key-points-of-working-effectively-with-legacy-code/

8 14 15 The Wrong Abstraction — Sandi Metz
https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction

9 10 32 Design Patterns Explained – Dependency Injection- Stackify
https://stackify.com/dependency-injection/

11 12 What is Value Object? | Dremio
https://www.dremio.com/wiki/value-object/

13 46 47 EventBridge best practice: why you need a custom event envelope
https://theburningmonk.com/2024/11/eventbridge-best-practice-why-you-should-wrap-events-in-event-envelopes/

19 20 What is Mutation Testing? | Definition from TechTarget
https://www.techtarget.com/searchitoperations/definition/mutation-testing

25 44 45 Pattern: Transactional outbox
https://microservices.io/patterns/data/transactional-outbox.html

26 How immutable objects help to prevent race conditions
https://stackoverflow.com/questions/44062300/how-immutable-objects-help-to-prevent-race-conditions

27 28 Back Pressure in Distributed Systems - GeeksforGeeks
https://www.geeksforgeeks.org/computer-networks/back-pressure-in-distributed-systems/

29 Law of Demeter with examples in Kotlin | by Fabri Di Napoli
https://proandroiddev.com/law-of-demeter-with-examples-in-kotlin-6e1cf75e3f94

30 Mastering the Principle of Least Knowledge (PLK) - Medium
https://medium.com/@Nelsonalfonso/mastering-the-principle-of-least-knowledge-plk-enhancing-software-modularity-fa3f8eddfd0e

33 36 Sequential coupling - Wikipedia
https://en.wikipedia.org/wiki/Sequential_coupling

34 35 37 38 Acoplamiento secuencial - Wikipedia, la enciclopedia libre
https://es.wikipedia.org/wiki/Acoplamiento_secuencial

39 40 41 42 43 Pattern: Saga
https://microservices.io/patterns/data/saga.html

48 Feature Toggles (aka Feature Flags)
https://martinfowler.com/articles/feature-toggles.html