

UNIVERSIDAD NACIONAL DE INGENIERÍA

Ingeniería de Software

Mantenimiento: código heredado, deuda técnica y deprecación

Costuras y caracterización · Cuadrante de Fowler · Strangler fig y parallel change · Presentaciones y defensa

Gerson Garrido

Semana 16

Hoy

Sesión A

1. Después de la primera entrega
2. Código heredado y caracterización
3. Deuda técnica
4. Migraciones graduales
5. Deprecación y cambios a gran escala

Sesión B

6. Presentaciones finales
7. Lo que necesita quien venga después

Laboratorio: defensa oral.

Lectura

SWEG, caps. 15 *Deprecation* y 22 *Large-Scale Changes*.

Enunciado de **P5**; SWEG, cap. 3 *Knowledge Sharing*.

Complementarias: Fowler, *Technical Debt Quadrant* y *StranglerFigApplication*; Sato, *Parallel Change*; RFC 9745 y RFC 8594; *Hyrum's Law*; SWEG, cap. 10.

SECCIÓN 01

Después de la primera entrega

01

Cuatro tipos de mantenimiento

Mantenimiento

Todo lo que se hace a un sistema después de su primera entrega.

Tipo	Lo dispara	En Kallpa
Correctivo	Algo no hace lo que debería	17:00 a 19:00 se cobra con una sola tarifa
Adaptativo	Cambió el entorno	Node nuevo; una ley de comprobantes electrónicos
Perfectivo	Quieren algo más o mejor	La caja del día con el porcentaje de ocupación
Preventivo	Nada falla, pero se ve venir	Extraer “solo cuentan las confirmadas”, repetida

Solo uno corrige errores

Un sistema sin un solo bug igual necesita mantenimiento: el mundo a su alrededor cambia.

El código es un pasivo

“code is a liability, not an asset”

Lo que tiene valor es la **funcionalidad**. El código es el medio, y sigue costando mientras exista.

- La misma funcionalidad en una línea entendible o en diez mil enredadas: se prefiere la primera.
- Quitar código puede ser de lo más útil que se hace en un mes.
- Porcentajes de “el mantenimiento es el X %”: suelen venir de estudios viejos y con otras definiciones. Si citan uno, vayan a la fuente y digan qué midió.
- **Con IA:** baja el costo de *escribir*, no el de *tener*. Antes de aceptar: ¿queremos ser owners de esto durante años?

SWEG, cap. 15

SECCIÓN 02

Código heredado

02

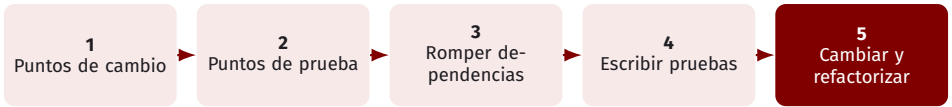


Qué es código heredado

> **Código heredado** *(Feathers, 2004)*
> Código sin pruebas.

- No importa quién lo escribió, cuándo ni qué tan bonito se ve.
- Importa una cosa: si lo cambian, ¿pueden saber si rompieron algo?
- **El dilema:** para cambiar con seguridad hacen falta pruebas; para poner pruebas hay que cambiar el código.
- La salida: esos primeros cambios, lo más pequeños y mecánicos posible.

El procedimiento



- Recién el quinto paso es lo que les pidieron.
- **Caso:** el cierre de caja de la administración. Sin pruebas, sin funciones, y Doris lo corre cada noche.
- Doris pide: “al final, que me diga cuántas reservas fueron de socios y cuántas de regulares”.

El cierre de caja

heredado/cierre-de-caja.ts

```
const db = new DatabaseSync(process.env.KALLPA_DB ?? 'data/kallpa.db');
const date = process.argv[2] ?? new Date().toISOString().slice(0, 10);
// ... una consulta con el nombre de la cancha y el plan de cada reserva ...
console.log(`CIERRE DE CAJA ${date}`);
let total = 0;
for (const row of rows) {
  if (row.status === 'cancelled') continue;
  let amount = row.price_cents / 100;
  if (row.plan === 'socio') amount = amount * 0.9;
  total += amount;
  console.log(`${row.court.padEnd(10)} ... S/ ${amount.toFixed(2)}`);
}
console.log(`TOTAL S/ ${total.toFixed(2)}`);
if (total > 500) console.log('Depositar en el banco');
```

Puntos de cambio: el bucle y el final. **Puntos de prueba:** solo lo que imprime. No hay función que llamar.

Costura

> **Costura** *(seam, Feathers)*
>
> Un lugar donde se puede cambiar el comportamiento de un programa sin editar en ese lugar.

- La mejor de Kallpa: el `Clock` que recibe `createServer`.
- El cierre ya trae dos, sin querer: `KALLPA_DB` elige la base de datos, `argv[2]` elige el día.
- Con ellas, una primera red sin tocar una línea: datos de ejemplo alrededor de un lunes fijo, correr el programa, guardar lo que imprime.
- `new Date()` y `console.log` no tienen costura. Esas son las dependencias que hay que romper.

Romper la dependencia, nada más

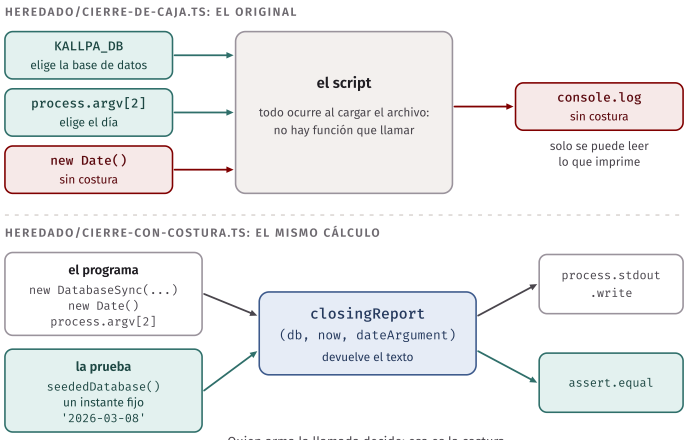
heredado/cierre-con-costura.ts

```ts
export function closingReport(db: DatabaseSync, now: Date, dateArgument?: string): string {
  const date = dateArgument ?? now.toISOString().slice(0, 10);
  // ... lo mismo, con lines.push en lugar de console.log ...
  return lines.join('\n') + '\n';
}
if (import.meta.main) {
  const db = new DatabaseSync(process.env.KALLPA_DB ?? 'data/kallpa.db');
  process.stdout.write(closingReport(db, new Date(), process.argv[2]));
}
```

- No cambió: ni los nombres, ni el `0.9`, ni el `toISOString`.
- La red: el original y el nuevo, como programas, contra los archivos guardados.

Mínimo y mecánico. Se rompe solo lo que estorba, y se deja lo demás, aunque moleste.

El mismo cálculo, con costura



Quien arma la llamada decide: esa es la costura.

Prueba de caracterización

Prueba de caracterización

Registra lo que el código hace hoy, no lo que debería hacer.

	Con TDD	Caracterización
Cuándo se escribe	Antes que el código	Después, a veces años
El valor esperado sale de	La especificación	El propio código, al correrlo
Si falla	El código está mal	Cambió algo: ¿se quería?
Afirma	Lo que debe pasar	Lo que pasa, errores incluidos

La semana 9 decía que copiar lo que hace el código no sirve como oráculo. Aquí se hace a propósito: la pregunta no es “¿está bien?”, sino “¿cambié algo sin querer?”.

La receta: que el fallo cuente qué hace

demos/primera-caracterizacion.check.ts

```
1  assert.equal(closingReport(db, new Date('2026-03-09T15:00:00Z'), '2026-03-08'), '?');
```

make primera

```
✗ ¿qué dice el cierre del 8 de marzo? (2.316708ms)
  AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:
...
  + 'CIERRE DE CAJA 2026-03-08\n' +
  +   'Fulbito 1  20:00 Deportivo Rímac           S/ 91.13\n' +
  +   'Vóley      18:00 Vóley Femenino San Juan   S/ 101.25\n' +
  +   'TOTAL S/ 192.38\n'
  - '?'
```

Se copia el resultado y queda fijado. Ejemplos que recorran las ramas: socios, cancelación, día vacío.

Lo que aparece cuando se mira

- **El día equivocado.** `toISOString` da UTC: desde las 19:00 en Lima, para el script ya es mañana. Por eso Doris pasa la fecha a mano. Lo sabe ella y nadie más.
- **El descuento doble.** `price_cents` ya trae el 10 %, y el script lo quita otra vez. Cierre: S/ 467.25. Caja de la API: S/ 487.50.
- **Fracciones de céntimo.** 101.25 × 0.9 = 91.125, que se imprime 91.13.

¿Qué se hace con eso? Nada, en este cambio

La prueba se queda con el valor de hoy, con “posible bug” en el nombre.

Se abre un issue con la evidencia.

El arreglo va en otro pull request, con su prueba y **después de preguntar**.

Por la ley de Hyrum: quizás el contador lleva años corrigiendo S/ 20 de memoria. Si lo arreglan sin avisar, el error es de ustedes.

Recién ahora, el cambio

heredado/cierre-por-plan.ts

```ts
export function closingReportByPlan(db: DatabaseSync, now: Date, dateArgument?: string): string {
  const old = closingReport(db, now, dateArgument);
  const date = dateArgument ?? now.toISOString().slice(0, 10);
  const counts = bookingsByPlan(db, date).map((c) => `${c.plan}: ${c.bookings}`);
  return `${old}RESERVAS ${counts.join(', ') || 'ninguna'}\n`;
}
```

cierre.test.ts

```ts
test('el cambio agrega una línea y deja igual todo lo anterior', () => {
  const report = closingReportByPlan(db, MORNING, '2026-03-09');
  assert.equal(report, `${golden('2026-03-09')}RESERVAS regular: 2, socio: 1\n`);
});
```

Código nuevo **al lado** del viejo (Feathers: *sprout*), con sus propias pruebas.

En Kallpa: fijar la caja del día

golden/caja.json, con jq: fecha, reservas, horas y céntimos

```
2026-03-06  1  2  22500
2026-03-08  2  3  21375
2026-03-09  3  5  48750
2026-03-10  2  3  21250
2026-03-11  2  3  22125
2026-03-14  1  2  11250
2026-03-20  0  0  0
```

- Siete días elegidos: dos pasados, hoy, una cancelación, dos horas que cruzan las 18:00, un sábado, un día vacío.
- El 11 de marzo, el básquet de 17:00 a 19:00 sale en 12000. El README dice S/ 135: el error de la semana 10. La caracterización lo fija: no dice qué está bien, dice qué hay.
- Quiere calcular la caja en SQL: un refactoring. Tiene que responder lo mismo.

Un cambio como los que suele proponer un agente

caja/report-sql-agente.ts

```
‘SELECT c.id AS courtId, c.name AS courtName, COUNT(*) AS bookings,
        SUM(b.hours) AS hours, SUM(b.price_cents) AS revenueCents
 FROM bookings b JOIN courts c ON c.id = b.court_id
 WHERE b.date = ? AND b.status = 'confirmed'
 GROUP BY c.id ORDER BY c.id‘
```

make check, con caja/agente.diff aplicado

```
i pass 40
i fail 0
```

- Pasa: ninguna prueba de Kallpa pide la caja por HTTP.
- Los totales están bien. ¿Lo aprobarían?

Ilustración escrita para la clase

La caracterización lo encuentra

```
make caja-agente

✗ la caja del 20 de marzo, un día sin reservas (2.741ms)
  AssertionError [ERR_ASSERTION]: Expected values to be strictly deep-equal:
...
      bookings: 0,
    + courts: [],
    - courts: [
    -   {
...
```

- Las canchas sin reservas desaparecen: el `JOIN` parte de las reservas.
- El orden cambió: antes por nombre, ahora por `id`. Nadie lo prometió, pero si un panel lee `courts[0]`, se rompió.
- Solo protege lo que sus ejemplos recorren. Sin un día vacío, el primer error pasaba.

SECCIÓN 03

Deuda técnica

03

La metáfora

> **“Shipping first time code is like going into debt”**
>
> “A little debt speeds development so long as it is paid back promptly with a rewrite.”

- **Deuda técnica:** un atajo que hace más rápido el cambio de hoy y más caros los siguientes.
- **Intereses:** el costo extra en cada cambio que pasa por ahí.
- **Capital:** lo que costaría arreglarlo.

> **Lo que decide es el interés, no lo feo que se ve**
>
> Un módulo enredado que no se toca casi no cobra interés. Una deuda prudente *“may not be worth paying down if the interest payments are sufficiently small”* (Fowler).

Ward Cunningham, *The WyCash Portfolio Management System*, 1992

El cuadrante de Fowler



El cuadrante no se deduce del código: se deduce de la decisión.

El cuadrante, en dos ideas

- Abajo a la derecha: inevitable en un buen equipo. Mientras se programa, se aprende.
- Lo que se ve en el código es el interés.

En su proyecto

- **Deliberada e imprudente.** “No hay tiempo para pruebas, ya después”. Casi siempre se equivocan en la cuenta: el buen diseño existe para ir más rápido.
- **Deliberada y prudente.** “La demo es el viernes. Guardamos en un archivo JSON, sabemos que no aguanta dos instancias, y lo cambiamos en el próximo sprint”. Y queda en un ADR.
- **Inadvertida e imprudente.** Las reglas del negocio dentro de los handlers, sin saber que eso se llama acoplamiento.
- **Inadvertida y prudente.** Lo mejor que se sabía; un año después se entiende cuál debió ser el diseño.

> **Con IA**
>
> Un repositorio lleno de código que pasa las pruebas y que nadie entendió es deuda inadvertida en el sentido estricto: nadie decidió tomarla.

Hacerla visible

“El código está feo” no es un issue. La deuda que no está escrita no se puede pagar.

```
grep -n "'confirmed'" src/http/handlers.ts src/domain/report.ts

src/http/handlers.ts:66:    const taken = deps.service.bookingsForCourt(courtId, date).filter((b) => b.status === 'confirmed');
src/http/handlers.ts:92:    const sameDay = deps.service.bookingsForCourt(court.id, request.date).filter((b) => b.status === 'confirmed');
src/domain/report.ts:24:  const confirmed = bookings.filter((b) => b.date === date && b.status === 'confirmed');
```

- **Un issue,** con evidencia, cuánto cuesta, una propuesta y cuándo.
- **Lo consciente, en un ADR:** qué se aceptó y qué señal diría que ya hay que pagar.
- **Al entregar,** en `## Pendientes` de `docs/entrega.md`, con la prioridad que acordaron con el cliente.

¿Dónde está el interés? `git log –format= –name-only | sort | uniq -c | sort -rn | head`. Lo enredado que cambia cada semana es donde el pago rinde.

SECCIÓN 04

Migraciones graduales

04

Por qué no reescribir

“Esto es un desastre: lo reescribimos en tres meses”

Casi siempre sale mal.

1. **Se pierde el conocimiento guardado en el código.** *“Replacements seem easy to specify, but often it’s hard to figure out the details of existing behavior”* (Fowler).
2. **Tarda más.** Ese plan *“go down in flames most of the time”* (Fowler). En Google, migrar a sistemas enteramente nuevos es *“extremely expensive”*.
3. **Mientras tanto hay dos sistemas,** y el viejo sigue cambiando.

> **SWEG, cap. 15**
>
> *“Evolving a system in place is usually cheaper than replacing it with a new one, when turndown costs are included.”*

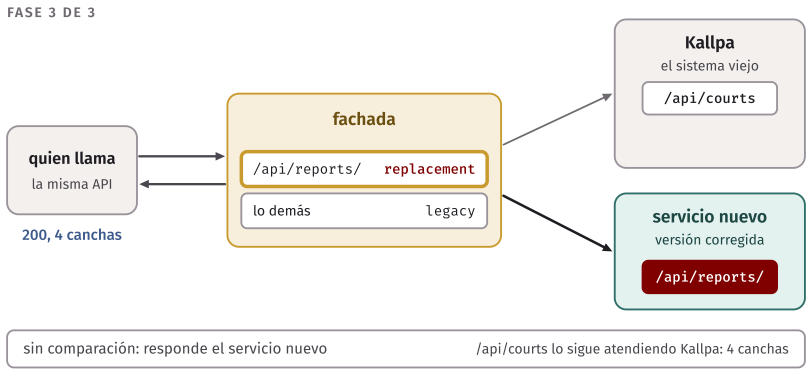
Strangler fig

Strangler fig *(Fowler)*

El sistema nuevo crece alrededor del viejo, detrás de la misma interfaz, y le quita funciones hasta que se puede apagar.

`legacy`	lo atiende el viejo
`shadow`	responde el viejo y se compara con el nuevo (solo lecturas)
`replacement`	lo atiende el nuevo

La fachada, fase por fase



Fase 3: atiende el servicio nuevo. Se deshace cambiando una línea de la tabla.

make facade

la caja del día se muda por fases

```
=== Fase 1: se compara la versión con errores (/api/reports/ en modo shadow)
SHADOW-DIFF /api/reports/daily?date=2026-03-06
  2026-03-06: 200, 4 canchas
...
=== Fase 2: se compara la versión corregida (/api/reports/ en modo shadow)
SHADOW-OK /api/reports/daily?date=2026-03-06
...
=== Fase 3: atiende el sistema nuevo (/api/reports/ en modo replacement)
  2026-03-06: 200, 4 canchas
```

- Fase 1: el nuevo está mal y quien llama sigue viendo 4 canchas. El error se encontró sin que nadie lo sufriera.
- Es la prueba diferencial de la semana 10, con tráfico real. Se deshace con una línea de la tabla.
- Una migración que nunca termina deja dos sistemas para siempre.

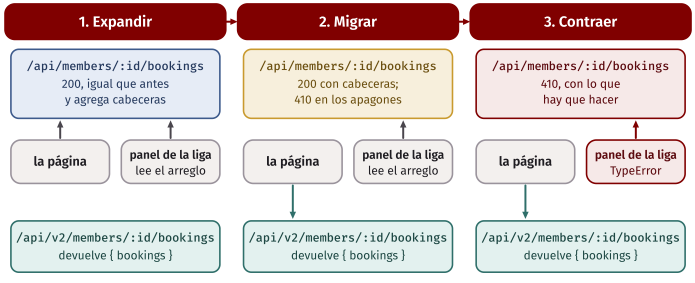
Parallel change

Expandir: se agrega lo nuevo, sin quitar lo viejo. **Migrar:** se pasa cada uso a lo nuevo. **Contraer:** se quita lo viejo.

- El sistema se puede liberar en cualquiera de las tres etapas.
- Con clientes externos, migrar es la etapa más larga.
- La semana 6 dejó planteado el caso: `GET /api/members/:id/bookings` devuelve un arreglo, y no se le puede agregar paginación. Ahí escribimos el primer paso. Hoy, hasta el final.

Danilo Sato, *Parallel Change*

Las tres etapas, en Kallpa



Cada etapa es un pull request que se puede integrar y desplegar por su cuenta.

Tres pull requests

Diff	Qué cambia	Pruebas
`1-expandir`	Nueva `/api/v2/...`; la vieja igual, con cabeceras; el log dice quién la usa	43
`2-migrar`	La página pasa a v2; apagones anunciados	46
`3-contraer`	La vieja responde `410` y dice qué hacer; se borra la transición	44

La etapa 3 agrega 19 líneas y quita 62: lo que paga el trabajo.

make etapas: el panel de la liga, un cliente que no está en el repositorio

```
=== etapa 1
GET /api/members/5/bookings → 200
panel de la liga: [ '2026-03-10 9:00 Las Matadoras' ]
...
=== etapa 3
GET /api/members/5/bookings → 410
panel de la liga: TypeError: bookings.filter is not a function
```

¿Cuándo se puede contraer?

etapa 1: una línea aparte cuando se sirve algo deprecado

```
if (res.hasHeader('Deprecation'))
  log('DEPRECATED /api/members/:id/bookings ua="${req.headers['user-agent'] ?? '?'}"'
    );
```

make etapas

```
=== quién sigue usando la ruta vieja (etapa 1)
   3 DEPRECATED /api/members/:id/bookings ua="panel-liga/1.2"
   2 DEPRECATED /api/members/:id/bookings ua="Mozilla/5.0"
   1 DEPRECATED /api/members/:id/bookings ua="script-contador"
```

- Sin el número de vecino: la pregunta no lo necesita (semana 15).
- `User-Agent` lo escribe quien llama. Con autenticación, se cuenta por cliente.
- Un mes en cero no prueba nada: un script de fin de año corre una vez al año.

Si se apuran

make apurados: la etapa 3, sin migrar la página

```
i tests 43
i pass 43
i fail 0
57:  const bookings = await api(‘/api/members/${memberSelect.value}/bookings‘);
```

- `make check` en verde y “mis reservas” rota para todos los vecinos. Ninguna prueba de Kallpa abre la página.
- Por eso la etapa 2 agrega una prueba que falla si la página vuelve a la ruta vieja.

El error al revés: quedarse en la etapa 2

Dos rutas y un `if` para siempre: se pagó la migración y no se cobró el beneficio. Fecha y owner para la tercera etapa.

La base de datos: la misma regla

Un rollback solo es posible si la versión anterior puede correr sobre el esquema nuevo. Las migraciones solo se agregan al final, y cada una es compatible con el código anterior.

make migraciones: el código publicado de Kallpa, sobre cada una

```
RENAME COLUMN: el código anterior falla: no such column: phone
ADD COLUMN: el código anterior lista 10 vecinos
user_version: 3; el código anterior solo conoce 2 migraciones
insertó el código anterior: { id: 11, name: 'Vecina del rollback', whatsapp: '900000001' }
insertó el código nuevo: { id: 12, name: 'Vecina nueva', whatsapp: '900000002' }
DROP COLUMN: el código anterior falla: no such column: phone
```

1. `ADD COLUMN whatsapp`, y se copia lo que hay.
2. El código escribe en las dos y lee la nueva, con respaldo: `COALESCE(whatsapp, phone)`.
3. Se rellenan las filas que quedaron en `NULL`. Luego, solo la nueva.
4. `DROP COLUMN phone`: al final, a propósito y con respaldo. Ahí se cierra la puerta.

Tres escalas de la misma idea

Técnica	Para	En cada paso
Strangler fig	un sistema	quien llama no se entera
Parallel change	una interfaz	conviven las dos formas
Migraciones que agregan	un esquema	el código anterior sigue funcionando

Nunca cambia todo a la vez

En cada paso el sistema funciona y se puede volver al anterior.

SECCIÓN 05

Deprecación

05

Deprecación

Deprecación *(SWEG, cap. 15)*

El proceso ordenado de migrar a los usuarios de un sistema obsoleto y finalmente retirarlo. No es marcarlo como viejo.

- **Ley de Hyrum:** *"removing a system can be thought of as the ultimate change"*.
- **Lo nuevo nunca es igual a lo viejo:** cada uso se evalúa de nuevo.
- **Cariño:** *"I like this code!"*. **Política:** quitar cuesta dinero visible; dejar, no.
- **Límite:** una organización aguanta pocas a la vez. Hay que elegir y terminar.
- Lo viejo no es lo obsoleto: LaTeX lleva décadas. Conviene cuando es *demostrablemente obsoleto* y *hay un reemplazo*.

Advisory y compulsory

	Advisory	**Compulsory**
Fecha límite	No tiene	Sí, y se hace cumplir
Recursos	Nadie dedicado a migrar	Un equipo que ayuda a migrar
Logra	Desanima usos nuevos	Termina con el sistema
Su riesgo	Nunca termina: “*Hope is not a strategy*”	Sin apoyo es un *unfunded mandate*

- En Google, una advertencia lleva a *(slightly)* menos usos nuevos y rara vez a que un equipo migre.
- Lo que escala: un solo equipo de expertos hace las migraciones de los demás.
- Y si el último usuario puede vetar la fecha, no era tan obligatoria.
- Sin **owners** no avanza. Sin **hitos intermedios**, el logro final no lo nota nadie.

Avisos que sirven

Una advertencia es **accionable** (se sabe el siguiente paso) y **relevante** (aparece cuando se puede actuar).

make etapas: la respuesta de la ruta vieja

```
deprecation: @1798761600
sunset: Sat, 01 Jan 2028 00:00:00 GMT
link: </api/v2/members/5/bookings>; rel="successor-version"
```

- `Deprecation` (RFC 9745): desde cuándo. *“does not change any behavior of the resource”*.
- `Sunset` (RFC 8594): cuándo se espera que deje de responder. No antes de `Deprecation`.
- `Link`: a dónde ir. Eso la hace accionable.

Son indicios *(hints)*

El panel de la liga no mira ninguna cabecera. **No reemplazan avisarle a la persona.**

Descubrir, migrar y prevenir el retroceso

- **Descubrir:** búsqueda de código, logs y apagones anunciados, cada vez más largos.

```
etapa 2: el Clock permite probarlos sin esperar a 2027

{ from: '2027-09-01T15:00:00Z', to: '2027-09-01T16:00:00Z' },
{ from: '2027-10-05T15:00:00Z', to: '2027-10-05T21:00:00Z' },
{ from: '2027-11-02T13:00:00Z', to: '2027-11-03T13:00:00Z' },
```

- **Prevenir el retroceso:** si alguien agrega usos mientras se migra, es *whack-a-mole*.

```
make -s retroceso

/work/code/week-16/ejemplos/pagina-nueva.js
  3:32  error  GET /api/members/:id/bookings está deprecada y deja de responder el 1 de enero de 2028. Usen /api/v2/members/:id/bookings ...
```

En TypeScript, `@deprecated` no molesta a `tsc`: lo tacha el editor. Si quieren que moleste, `@typescript-eslint/no-deprecated`.

Cambios a gran escala

Large-scale change

Cambios lógicamente relacionados que en la práctica no se pueden enviar como una unidad atómica.

- Mientras más crecen el código y el equipo, **más chico** es el cambio atómico más grande posible.
- Autorización, creación del cambio, manejo de los fragmentos y limpieza.
- En Google, Rosie parte por proyecto y por owner, en fragmentos que se prueban, revisan e integran por separado.
- Adam Bender, en un recuadro: entre 10 % y 20 % de los cambios de un proyecto vienen de un LSC. Sin buenas pruebas sería imposible: quienes los hacen no conocen el proyecto.

SWEG, cap. 22

El cambio mecánico más simple: renombrar `priceCents`

make renombrar, recortado

```
=== 1. Con sed, en todo el repositorio
  tsc: sin errores
  pruebas: 40 pasan, 0 fallan
  quedan con priceCents: ninguno

=== 2. Con el servicio de lenguaje de TypeScript
  tsc: sin errores
  pruebas: 39 pasan, 1 fallan
  quedan con priceCents: src/http/handlers.ts, src/notifications/notifier.ts, test/http.test.ts, public/app.js
```

- **1:** todo en verde y el cambio es incompatible. Renombró el JSON de la API; el panel de la liga se rompió, y nada lo dice.
- **2:** más preciso. No ve lo que no tiene tipos: un `any` en una prueba y `app.js`.
- **Un campo de la API no es un nombre interno:** es un contrato, y cambiarlo es un parallel change.

Con IA: 300 archivos

- Los agentes rinden en lo mecánico y repetido, si hay pruebas que caractericen el comportamiento. Sin ellas: un diff enorme que nadie puede verificar, y que además “pasa”.
- Antes de empezar: escribir lo que **no** debe cambiar (el JSON, por ejemplo).
- Partirlo: expandir, un pull request por módulo o por owner, cada uno con el check en verde, contraer. Si un fragmento no pasa solo, no es un fragmento.
- Revisar la regla de transformación y una muestra. Mejor si es un programa determinista: se revisa una vez y se vuelve a correr.

Lo que el equipo no entiende

El código que se aceptó sin entender es heredado desde el día en que se hizo commit: nadie sabe por qué está así ni qué se rompe si se cambia. Una suite que nadie leyó fija un comportamiento que nadie decidió.

SECCIÓN 06

Presentaciones finales

06

Sesión B: qué se presenta

No es un recorrido por las pantallas

El problema del cliente, lo que se construyó y **cómo se sabe que funciona**.

1. **El problema,** en palabras del cliente.
2. **La demo:** el producto desplegado, haciendo una tarea real.
3. **Una decisión de diseño** con las alternativas descartadas (un ADR).
4. **La evidencia de calidad:** el pipeline, las pruebas de la regla más difícil, el postmortem.
5. **El uso de IA, con datos:** qué se rechazó, y un caso incorrecto que pareció correcto.
6. **Qué se le entrega al cliente** para que alguien más lo mantenga.
7. **La retrospectiva:** qué harían distinto, con un ejemplo de cada sprint.

La demo y las preguntas

- **Datos inventados conocidos.** Nunca los del cliente.
- **Se prueba el día anterior,** en el mismo entorno.
- **Plan B:** un video corto, por si la red o el despliegue fallan en la sala.
- Si algo falla en vivo, muestren cómo responden: el log, el rollback, y siguen.
- **Cualquier integrante responde cualquier pregunta.**
- Lo que no salió bien se cuenta igual, con lo que se aprendió.

Lo que necesita quien venga después

- **Punto único de falla:** información crítica que solo tiene una persona. Suele nacer de “déjame hacerlo a mí, que es más rápido”.
- **Conocimiento tribal:** lo que hay entre lo que la gente sabe y lo que está escrito. “Sin fecha sale mal de noche”.
- **Islas de información:** partes que no se hablan, con sus propias versiones de cómo se hace.
- Un experto responde lo que el documento no previó; un documento llega a quien no conoce al experto. Se complementan. Y antes de todo, seguridad psicológica: preguntar no da vergüenza.

> **La prueba de P5**
>
> Que alguien de otro equipo siga el README desde cero, en el contenedor del curso, y anote dónde se trabó.

SWEG, cap. 3

SECCIÓN 07

La defensa oral

07

Laboratorio: defensa oral

> **No se prepara una presentación**
>
> Se prepara el repositorio y se conoce el código.

1. **Explicar una parte:** qué hace, por qué está así, qué alternativa se descartó y qué se rompería si se cambia.
2. **Un cambio en vivo:** en una rama, con su prueba, y `npm run check`. Con una herramienta de IA, se explica y se verifica cada línea.
3. **La operación:** dónde está desplegado, cómo se vuelve atrás, qué muestra el pipeline, qué dice el postmortem.
4. **Los límites del producto:** lo que está en `## Pendientes` lo eligieron ustedes.

Al terminar, cada integrante entrega su evaluación entre pares. “Lo generó la herramienta” no es una respuesta.

Antes: un clon limpio

```
make ensayo
```

```
=== En la laptop de quien lo escribió
ℹ pass 40
ℹ fail 0
=== git status
0
=== En un clon limpio
src/config.ts(16,26): error TS2307: Cannot find module './holidays.local.ts' ...
=== Después de subir lo que faltaba
ℹ pass 40
ℹ fail 0
=== El cambio en vivo: un nombre de equipo tiene al menos 3 caracteres
 3 files changed, 18 insertions(+), 2 deletions(-)
ℹ pass 41
ℹ fail 0
```

El archivo estaba en `.gitignore`: `git status` ni lo muestra. Clonen, `npm ci`, `npm run check`, la revisión de P5 y levanten el producto. Si algo no funciona, se arregla antes del día.

Para terminar

- **P5**: tag `p5` antes de la presentación final; presentación en la Sesión B y defensa oral en el laboratorio; evaluación entre pares al terminar.
- **Examen final**, en el período de exámenes: todo el curso, con énfasis en las semanas 9 a 16 (`exams/README.md`). Hay un **examen de práctica**: resuélvanlo con el reloj corriendo y sin herramientas de IA.
- Lecturas: SWEG, caps. 15, 22 y 3, y el enunciado de P5.

Pregunta

Dentro de un año, alguien hereda su proyecto. ¿Qué es lo primero que va a tener miedo de tocar? ¿Qué le dejarían para que no lo tenga?