

UNIVERSIDAD NACIONAL DE INGENIERÍA

Ingeniería de Software

Revisión de código y análisis estático

El estándar para aprobar · Falsos positivos efectivos · TypeScript estricto · Revisar código escrito con IA

Gerson Garrido

Semana 11

Hoy

Sesión A

1. Para qué se revisa
2. El estándar para aprobar
3. Qué mirar y en qué orden
4. Cambios pequeños y comentarios útiles

Sesión B

5. Análisis estático: por qué todo aproxima
6. Tricorder: falsos positivos efectivos
7. TypeScript estricto y linting con tipos
8. Revisar código escrito con IA

Lecturas

SWEG, caps. 9 *Code Review* y 20 *Static Analysis*.

Google Engineering Practices, *How to do a code review* y *The Standard of Code Review*.

typescript-eslint, *Linting with Type Information* y *Shared Configs*.

TypeScript, `noUncheckedIndexedAccess` y `exactOptionalPropertyTypes`.

Complementarias: Sadowski y otros (ICSE SEIP 2018); *The CL author's guide*; *Small CLs*; Conventional Comments; GitHub, *About pull request reviews*; SWEG, caps. 8 y 19; UMD, CMSC 398Z.

SECCIÓN 01

Para qué se revisa

01

Revisión de código

Revisión de código *(code review)*

La lectura de un cambio por una persona distinta de quien lo escribió, antes de que entre a la rama principal.

- Se revisa un **diff** y una **descripción**: en GitHub, un pull request; en Google, un *CL*.
- *Antes*: tiene poder de decisión. Después del merge, es una auditoría.
- Las semanas 9 y 10 ejecutaron el programa. Hoy, dos formas de encontrar problemas sin ejecutarlo.

Solo uno de seis beneficios es encontrar errores

Corrección
hace lo que se pretendía

Comprensión
otra persona lo entiende

Consistencia
se parece al resto del código

Ownership del equipo
el código no es “de alguien”

Conocimiento
dos personas conocen esa parte

Registro histórico
por qué se decidió, años después

Si quien revisa no lo entiende, ya es un hallazgo

Quien lo mantenga en dos años tampoco lo va a entender. SWEG: en comprensión, “el cliente siempre tiene la razón”.

SWEG, cap. 9

Lo que midió un estudio en Google

Qué hicieron

- 12 entrevistas y una encuesta con 44 respuestas.
- Registros de 9 millones de cambios revisados.

Qué esperan quienes revisan

- Educación, mantener normas, control de acceso, prevención de accidentes.
- Encontrar defectos es bienvenido, no es el único foco.
- Depende de la relación entre autor y revisor.

En los registros (medianas)

24 líneas modificadas por cambio.
1 revisor.
Menos de 4 horas de latencia.

Límites

Exploratorio, una sola empresa. Las expectativas son lo que la gente dice valorar, no cuánto aprende de verdad.

Sadowski y otros, ICSE SEIP 2018

Tres permisos en Google, y su equivalente en GitHub

Google	Revisa	GitHub
LGTM de otra persona	¿es correcto y se entiende?	revisión *Approve* obligatoria
Owner (`OWNERS`)	¿va en esta parte del sistema?	`CODEOWNERS`
Readability	¿sigue las prácticas del lenguaje?	(no existe)
Presubmit	análisis automáticos	revisiones de estado del pipeline

- Normalmente una sola persona cumple los tres papeles.
- Quien es owner y tiene readability solo necesita un LGTM. Pero **siempre** lee otra persona.
- En P3: todo pull request integrado a `main` tiene la revisión de alguien distinto de quien lo abrió.

SWEG, cap. 9

SECCIÓN 02

El estándar para aprobar

02

Se puede fallar de dos formas

Demasiado exigente

se detiene el trabajo y quien propone mejoras deja de hacerlo

Se aprueba todo

cada cambio es solo un poco peor, y ninguno parece el culpable

El estándar ***(Google Engineering Practices)***

In general, reviewers should favor approving a CL once it is in a state where it definitely improves the overall code health of the system being worked on, even if the CL isn't perfect.

Se aprueba cuando mejora con seguridad la salud general del código, aunque no sea perfecto.

No existe el código perfecto, solo el código mejor

- Lo que se podría mejorar se comenta igual, con una marca: `Nit:`. No bloquea.
- **Hechos** y datos pesan más que opiniones y preferencias.
- En **estilo**, manda la guía de estilo. Sin regla, es preferencia personal.
- El **diseño** casi nunca es solo estilo: se decide con principios.
- Si nada aplica: que sea consistente con lo que ya hay.

Dos límites

Se puede negar la aprobación a una funcionalidad bien escrita que no se quiere en el sistema.
Nada justifica integrar lo que **empeora** la salud del código, salvo una emergencia.

¿Se aprueba? Cuatro casos en Kallpa

El cambio	Decisión
Agrega `GET /api/courts/:id` con buenas pruebas. Un nombre podría ser mejor	**Sí**, con un `Nit:`
Corrige un error de precios; deja una función de 40 líneas que ya estaba así	**Sí**, y una tarea aparte: no empeoró
Agrega un reporte copiando `dailySummary` con otra forma de redondear	**No**: dos versiones de una regla
Funcionalidad que el cliente no pidió, bien escrita y con pruebas	**No, por ahora**: primero el backlog

Error común: el LGTM de cortesía

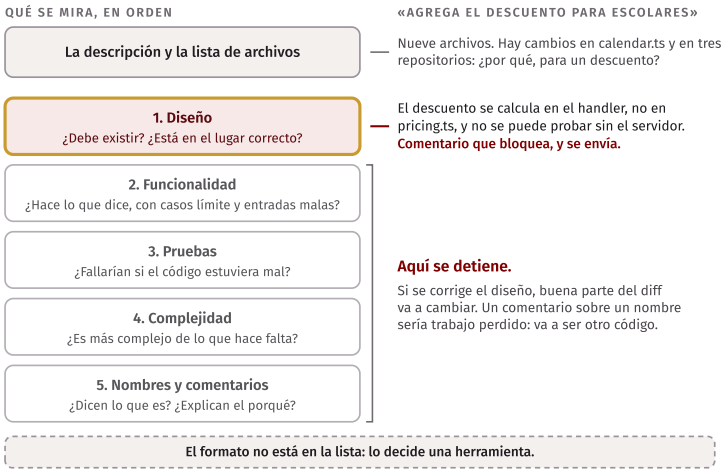
Aprobar sin correr ni entender. Es peor que no revisar: da una seguridad que nadie ganó.

SECCIÓN 03

Qué mirar

03

De lo grande a lo chico



Por qué ese orden

Si el diseño está mal, todo lo demás va a cambiar. **El formato no lo revisan las personas**: lo decide una herramienta.

Lo que hay que buscar a propósito

Funcionalidad

- La lista vacía, el texto vacío, el apóstrofe, el `null`, el id que no existe.
- Quien revisa también puede sacar la rama y usar `curl`.

Complejidad

- Una configuración para un caso que no existe.
- Una interfaz con una sola implementación “por si acaso”.

Pruebas

- ¿Qué mutante mataría? (semana 10)
- Lo que cambió en las pruebas que ya existían.

La señal más importante

Una prueba borrada o aflojada en el mismo pull request. El diff se ve como una limpieza.

Un error común

Doce comentarios sobre comillas y espacios.

Ninguno sobre el `if` que tiene la condición al revés.

Qué hacer

El estilo se ve sin pensar; la lógica cuesta. Si en su equipo los comentarios de formato son frecuentes, no necesitan revisores más estrictos: necesitan un formateador en el pipeline. SWEG, cap. 20: señalar errores de formato no es un buen uso del tiempo de quien revisa.

SECCIÓN 04

Cambios pequeños y comentarios

04

Cambios pequeños

Orden de magnitud

100 líneas suele ser razonable.
1000 líneas suele ser demasiado.
No es una regla exacta: importa también cuántos archivos toca.

- Se revisa mejor.
- Se integra antes: menos conflictos.
- Se revierte y se entiende con facilidad.

Cómo se divide

- Por capas, cada una con `make check` en verde.
- Por slices verticales: un caso completo primero.
- Refactoring aparte, antes de la funcionalidad.
- Pruebas y código, juntos.

Google, *Small CLs*

Lo que le toca a cada uno

Quien escribe

Un cambio chico, con un solo propósito.

Una descripción: qué cambia y por qué.

Se revisa a sí mismo: lee su propio diff.

Corre `make check` antes de pedir revisión.

Quien revisa

Responde en un día hábil como máximo.

Responder rápido no es aprobar rápido.

Lee en orden y corre el cambio.

Responde por lo que aprueba.

Si no se ponen de acuerdo

Google: consenso, luego una conversación, luego escalar. Regla del curso (nuestra): si no se resuelve en dos idas y vueltas, se conversa.

Cuatro reglas para un comentario

1. **Sobre el código, no sobre la persona.**
   “Esta función no cierra la conexión”, no “te olvidaste de cerrarla”.
2. **Con el porqué.** Si no, solo se puede obedecer o preguntar.
3. **Con una sugerencia concreta**, cuando se puede.
4. **Con la severidad marcada.** ¿Bloquea o es una idea? Si no se dice, se asume lo peor.

Conventional Comments

`<etiqueta> [decoraciones]: <asunto>`

`issue`, `suggestion`, `question`, `nitpick`, `praise`, `todo` · `(blocking)`, `(non-blocking)`, `(security)`

El mismo hallazgo, dos comentarios

inútil

```
Esto está mal, así no se hace.
```

útil

```
issue (blocking): este SQL arma la consulta con el texto del usuario.

Con ?team=' UNION SELECT ... se puede leer cualquier tabla (lo probé con curl en
la rama). La guía del curso pide sentencias preparadas. Sugerencia:

    .prepare("... WHERE team_name LIKE '%' || ? || '%'").all(team)
```

Dice dónde, qué pasa, cómo se comprobó, qué regla aplica, qué hacer y que bloquea.

Con IA: un revisor automático es un revisor más

- Hay herramientas que comentan pull requests automáticamente.
- Encuentran ciertos errores, y producen también comentarios equivocados o irrelevantes.
- Se miden como cualquier analizador: ¿alguien hizo algo con el aviso?
- Cada comentario es una pregunta que hay que comprobar, no una decisión.

No reemplazan la aprobación

Quien aprueba entiende el cambio y responde por él. Una herramienta no responde por nada. En P3, cuenta la revisión de una persona.

SECCIÓN 05

Análisis estático

05

Razonar sin ejecutar

Análisis estático

Programas que analizan el código fuente para encontrar problemas posibles, sin ejecutarlo.

Nivel	Mira	Encuentra
Formateador	el texto	nada: reescribe
Linter	patrones locales	una variable sin usar, un `catch` vacío
Type checker	tipos del programa	un método que no existe
Linter con tipos	patrones y tipos	una promesa que nadie espera
Flujo de datos	cómo viaja un valor	texto del usuario que llega a SQL

Mira todos los caminos, incluso los que ninguna prueba recorre. Pero no sabe con certeza qué va a pasar al correr.

Por qué todas aproximan

Teorema de Rice (1953)

Toda propiedad no trivial del comportamiento de los programas es indecidible: no existe un algoritmo que, para todo programa, termine y responda correctamente si la cumple.

¿llega a lanzar el error, con todo entero positivo n?

```
function reaches(n: number): void {
  while (n !== 1) n = n % 2 === 0 ? n / 2 : 3 * n + 1;
  throw new Error('llegó');
}
```

Responder exige saber si ese ciclo siempre termina: la conjetura de Collatz, un problema abierto. Y en general, “¿llega a esta línea?” equivale a “¿este programa termina?”, que Turing demostró indecidible en 1936.

Cada herramienta elige de qué lado equivocarse

	avisa	no avisa
hay error	verdadero positivo	**falso negativo**
no hay error	**falso positivo**	verdadero negativo

- **Sólida** *(sound)*: sin falsos negativos. **Completa**: sin falsos positivos. Una herramienta que siempre termina no puede ser las dos cosas.
- TypeScript no es sólido a propósito: `any` y `as` le dicen que no revise.
- "No dijo nada" significa: no encontró lo que sabe buscar, de la forma en que lo sabe buscar.

Los dos errores en un mismo archivo

ejemplos/first-item.ts

```ts
export function firstDate(slots: readonly Slot[]): string {
  return slots[0].date;              // línea 7: se cae con []
}
export function firstPart(text: string): number {
  return text.split('-')[0].length;  // línea 13: nunca se cae
}
```

Con `strict`, `tsc` no dice nada: falso negativo en `firstDate`.

con noUncheckedIndexedAccess

```
ejemplos/first-item.ts(7,10): error TS2532: Object is possibly 'undefined'.
ejemplos/first-item.ts(13,10): error TS2532: Object is possibly 'undefined'.
```

La línea 13 es un falso positivo: `split` siempre da al menos un elemento.

Una regla hecha a mano

eslint.sql.js

```
selector: "CallExpression[callee.property.name='prepare'] > TemplateLiteral[expressions.length>0]",
message: 'SQL armado con un template: usen parámetros (?).',
```

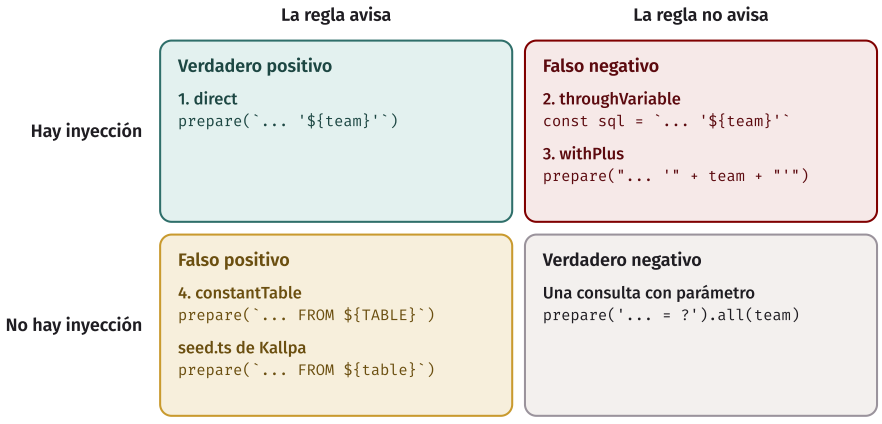
eslint -c eslint.sql.js ejemplos/sql-shapes.ts

```
 9:21  error  SQL armado con un template: usen parámetros (?)  no-restricted-syntax
26:21  error  SQL armado con un template: usen parámetros (?)  no-restricted-syntax
```

Función	¿Inyección?	¿Avisa?
`direct`	sí	sí
`throughVariable` (`const sql = ...`)	sí	no: falso negativo
`withPlus` (`"..." + team`)	sí	no: falso negativo
`constantTable` (`${TABLE}`)	no	sí: falso positivo

Mira la **forma**, no a dónde llega el dato. En Kallpa publicada da 1 aviso, falso: `src/seed.ts`.

La regla mira la forma



SECCIÓN 06

Tricorder

06

La medida que importa

Falso positivo efectivo

Un aviso con el que quien desarrolla no hace nada, sea cual sea el motivo.

Error real, mensaje confuso
Técnicamente correcto. Nadie hace nada.
Sí es falso positivo efectivo.

No es error, pero aclara el código
Técnicamente falso. Lo corrigen igual.
No es falso positivo efectivo.

Por qué importa

Si la tasa es alta, se aprende que los avisos no valen el tiempo y se dejan de leer **todos**, incluidos los verdaderos.

SWEG, cap. 20

Lo que decide es la columna

FALSO POSITIVO EFECTIVO

	Quien desarrolla actúa	Quien desarrolla no hace nada
El aviso señala un error real	**El aviso sirvió** Se corrige el error.	**Tenía razón y no sirvió** El mensaje confunde, o el problema no importa.
El aviso no señala un error técnico	**También sirvió** containsValue en lugar de contains: el código queda más claro.	**No tenía razón y no sirvió** Es falso de las dos formas.

Lo que Google aprendió con Tricorder

Para que un chequeo entre

1. Se entiende.
2. Se puede actuar, y se corrige con facilidad.
3. Menos de 10 % de falsos positivos efectivos.
4. Puede tener un impacto significativo.

Cómo se muestra

- En la revisión de código, no en un reporte.
- Los avisos nuevos, en las líneas que se tocan.
- Con una corrección sugerida.
- Con un botón de “no es útil” que llega a quien mantiene la regla.

Lo que se traslada: el método, no el número

Reportan una tasa global de algo menos de 5 %: cifras de Google sobre sí mismo, sin experimento que compare. Lo útil es medir si la gente actúa.

SWEG, cap. 20

SECCIÓN 07

TypeScript estricto

07



Más allá de `strict`

- `"strict": true` activa un grupo, entre ellas `strictNullChecks` y `noImplicitAny`.
- Hay opciones más estrictas fuera del grupo. Se piden una por una.

`noUncheckedIndexedAccess`

`items[0]` es `T | undefined`, porque la lista puede estar vacía. También los `Record` y la desestructuración.

`exactOptionalPropertyTypes`

`note?: string` no acepta `undefined` explícito. `Object.keys` y `{...a, ...b}` sí distinguen no estar de valer `undefined`.

En Kallpa: 9 errores en 5 archivos

```
con noUncheckedIndexedAccess: make typecheck 2>&1 | grep 'error TS' | cut -c1-110

src/domain/calendar.ts(15,19): error TS2769: No overload matches this call.
src/domain/calendar.ts(15,25): error TS18048: 'month' is possibly 'undefined'.
src/http/router.ts(59,9): error TS18048: 'expected' is possibly 'undefined'.
src/http/router.ts(60,14): error TS18048: 'expected' is possibly 'undefined'.
src/http/router.ts(60,54): error TS2345: Argument of type 'string | undefined' is not assignable to parameter
src/store/database.ts(53,15): error TS2345: Argument of type 'string | undefined' is not assignable to paramet
src/store/seed.ts(85,45): error TS18048: 'court' is possibly 'undefined'.
src/store/seed.ts(85,116): error TS18048: 'member' is possibly 'undefined'.
test/http.test.ts(54,18): error TS2532: Object is possibly 'undefined'.
```

¿Hay errores reales?

Casi ninguno se manifiesta hoy. Pero cada uno es un lugar donde el código supone algo sin decirlo, y mañana esa suposición puede dejar de ser cierta. Con `exactOptionalPropertyTypes` sale 1 más: `test/helpers.ts(59,37)`.

Tres formas de corregir, y una que no

corregir firstDate

```ts
// 1. Una guarda.
const first = slots[0];
if (first === undefined) throw new Error('No hay horarios.');
return first.date;

// 2. Un valor por omisión, si hay uno razonable.
return slots[0]?.date ?? 'sin fecha';

// 3. Otra estructura: ya no se indexa.
return slots.map((slot) => slot.date);
```

Esto no corrige nada

`return slots[0]!.date;` no comprueba nada: es un "confía en mí" que se borra al quitar los tipos. `firstDate([])` se sigue cayendo.

Linting con tipos

eslint.config.js

```
tseslint.configs.strictTypeChecked,
{
  languageOptions: {
    parserOptions: { projectService: true, tsconfigRootDir: import.meta.dirname },
  },
},
{ files: ['**/*.js'], extends: [tseslint.configs.disableTypeChecked] },
```

- Usa la información de tipos: ve qué devuelve una función de otro archivo.
- **Costo:** es como hacer un build antes de correr las reglas.
- `strictTypeChecked` no es estable según semver: sus reglas pueden cambiar sin una versión mayor.
- Se recomienda si una parte importante del equipo domina TypeScript.

Lo que `tsc` deja pasar

ejemplos/promises.ts

```ts
// Nadie espera a saveAudit: su error no pasa por este catch.
export function confirm(line: string, saved: string[]): string {
  try { saveAudit(line, saved); return 'confirmada'; } catch { return 'falló'; }
}
```

node: confirm(", [

```
)]
confirmada
file:///work/code/week-11/ejemplos/promises.ts:4
  if (line === '') throw new Error('Línea vacía.');
                   ^

Error: Línea vacía.
```

`tsc` y ESLint sin tipos no dicen nada; `no-floating-promises` avisa. Y `no-misused-promises` ve `lines.forEach(async ...)`, que no espera a nadie. (Aquí, el código compactado.)

En Kallpa: 105 errores, primero se agrupan

make medir: strictTypeChecked, por regla

```
46 @typescript-eslint/no-floating-promises
21 @typescript-eslint/restrict-template-expressions
18 @typescript-eslint/no-unsafe-member-access
 8 @typescript-eslint/no-unsafe-assignment
 4 @typescript-eslint/no-unsafe-call
 2 @typescript-eslint/no-unnecessary-condition
 2 @typescript-eslint/no-non-null-assertion
 1 @typescript-eslint/restrict-plus-operands
 1 @typescript-eslint/no-unsafe-return
 1 @typescript-eslint/no-unsafe-argument
 1 @typescript-eslint/no-misused-promises
```

26 en `src/` y 79 en `test/`

Los 46 de `no-floating-promises` son **todos** de `test()` y `describe()`, cuyas promesas ya maneja el runner: falsos positivos. Sin agrupar, se ve una pared de errores falsos y se apaga la regla también en `src/`.

Un falso positivo causado por un tipo optimista

src/http/static-files.ts

```ts
const entry = FILES[pathname];
if (entry === undefined) return false;
```

eslint, sin noUncheckedIndexedAccess

```
17:7  error  Unnecessary conditional, the types have no overlap  @typescript-eslint/no-unnecessary-condition
```

- El tipo dice que indexar siempre da un valor. Es falso: `pathname` puede ser cualquier ruta.
- Si se borra esa línea, `entry.file` lanza un error y la API entera responde 500.
- Con `noUncheckedIndexedAccess`, el aviso desaparece: el tipo ya dice la verdad.
- Aparecen 5 avisos nuevos (`params.id`, las partes de la fecha): 109 en vez de 105.

Por eso, primero los tipos: al mentir menos, muestran lugares nuevos que revisar.

Una trampa al configurar

15 de los 19 avisos de `restrict-template-expressions` en `src/` son números en un texto. Parece razonable permitirlos.

Configuración de la regla	**Avisos en `src/`**
La de `strictTypeChecked`	19
`{ allowNumber: true }`	**0**
`allowNumber: true`, y las demás `allow*` en `false`	4

Las opciones reemplazan

Las que se pasan reemplazan a las de la configuración compartida, y las demás vuelven a su valor por omisión, que es permisivo. Los 4 que quedan son valores que pueden ser `undefined`. **Después de configurar, midan.**

Las fronteras del programa

ejemplos/cast-lie.ts

```ts
export function parseWithCast(text: string): Draft {
  return JSON.parse(text) as Draft;       // courtId puede ser "uno", y el tipo dice number
}

export function parseChecked(text: string): Draft {
  const value: unknown = JSON.parse(text);
  if (typeof value !== 'object' || value === null) throw new Error('Se esperaba un objeto.');
  if (!('courtId' in value) || typeof value.courtId !== 'number') throw new Error('Falta courtId.');
  if (!('date' in value) || typeof value.date !== 'string') throw new Error('Falta date.');
  return { courtId: value.courtId, date: value.date };
}
```

- Lo que llega de afuera (JSON, filas, entorno) se tipa como `unknown` y se revisa.
- Kallpa lo hace bien en HTTP. En `src/` hay 14 `as`, la mayoría en los repositorios.
- Ante cada `as`, `!`, `any` o `eslint-disable` nuevo, pregunten: ¿qué lo garantiza?

Adoptar reglas sin romper el proyecto

1. Una opción o una regla a la vez, en un pull request que no cambia el comportamiento.
2. Agrupar por regla: los números grandes suelen ser una sola decisión. En P3: `--max-warnings 0`.
3. Por cada regla, una decisión por escrito:

Decisión	Cuándo	Ejemplo en Kallpa
Corregir	Señala algo real	`no-misused-promises` en `server.ts`
Configurar	Tiene sentido, no aquí o no así	`no-floating-promises` en `test/`
Desactivar	Sus avisos no llevan a nada útil	discútanlo, con el porqué

Los dos extremos

Silenciar el aviso sin entenderlo (`!`, `as`, `eslint-disable` sin razón). Y activar cien reglas de golpe: nadie lee cien avisos.

SECCIÓN 08

Revisar código escrito con IA

08

Falla de maneras propias

Falla	Quién la detecta primero
Código plausible que no hace lo pedido	La lectura y las pruebas de aceptación
APIs o paquetes que no existen	`tsc`; `npm view` (semana 12)
Cambios fuera del alcance	La lectura, si el diff es chico
Pruebas cambiadas para que pasen	La lectura, y Stryker en parte
Consultas armadas con texto	La lectura, y flujo de datos si lo tienen
Un estilo ajeno al proyecto	La lectura, y el formateador
Diffs grandes	La lista de archivos
Afirmaciones sin comprobar	Comprobarlas

Miren la columna de la derecha

Las herramientas estáticas son el primer filtro, indispensable, y cubren poco de esta lista.

Node solo quita los tipos

```
make errores
```

```
ejemplos/invented-method.ts(6,56): error TS2339: Property 'getAll' does not exist on type 'StatementSync'.
```

```
node ejemplos/invented-method.ts
```

```
file:///work/code/week-11/ejemplos/invented-method.ts:6
export const rows = db.prepare('SELECT * FROM courts').getAll();
                                                       ^

TypeError: db.prepare(...).getAll is not a function
    at file:///work/code/week-11/ejemplos/invented-method.ts:6:56
```

Node arranca, y falla al llegar a esa línea

Si ninguna prueba pasa por ahí, las pruebas quedan en verde con un método que no existe. “Pasan las pruebas” no dice si compila. Por eso `make check` corre `tsc` primero.

Un ejemplo: búsqueda por equipo

busqueda-agente.diff

```
Tarea: "Agreguen GET /api/bookings/search?team=texto: reservas cuyo nombre de
equipo contiene el texto."
Descripción: "Agrega la búsqueda de reservas por equipo, con su prueba. make check pasa."
```

make busqueda

```
tsc -p .
eslint .
node --test "test/**/*.test.ts"
ℹ tests 41
ℹ suites 6
ℹ pass 41
ℹ fail 0
```

Es cierto: pasa. Ahora se lee. Y se empieza por las pruebas.

Ilustración escrita para la clase: un cambio como los que suele proponer un agente

Primero, las pruebas

test/search.test.ts

```
1  test('busca reservas por equipo', async () => {
2    const { status } = await getJson(`${server.url}/api/bookings/search?team=Pumas`);
3    assert.equal(status, 200);
4  });
```

- No crea ninguna reserva. No mira el resultado.
- ¿Qué mutante mataría? Quitamos el `WHERE` de la consulta: sigue en verde.
- No prueba la búsqueda: prueba que la ruta existe.

Después, el código, y se comprueba

src/store/booking-repository.ts

```
    .prepare('SELECT * FROM bookings WHERE team_name LIKE '%${team}%' ORDER BY date, start_hour')
    .all() as BookingRow[];
```

con el servidor corriendo y los datos de ejemplo

```
$ curl -s "localhost:3000/api/bookings/search?team=Pumas" | jq -c '.[] | {teamName, memberId}'
{"teamName":"Los Pumas de Comas","memberId":1}
{"teamName":"Los Pumas de Comas","memberId":1}
```

un apóstrofe en el nombre

```
$ curl -s -G "localhost:3000/api/bookings/search" --data-urlencode "team=O'Higgins"
{"error":{"code":"INTERNAL_ERROR","message":"Ocurrió un error inesperado."}}
```

Y SQL después del apóstrofe

con el servidor corriendo y los datos de ejemplo

```
$ curl -s -G "localhost:3000/api/bookings/search" --data-urlencode \
    "team=' UNION SELECT id, 1, 1, name, 7, 1, phone, 0, plan, '', NULL FROM members -- " \
    | jq -c '.[] | select(.status != "confirmed" and .status != "cancelled")
                 | {nombre: .date, telefono: .teamName, plan: .status}'
{"nombre":"Rosa Quispe Mamani","telefono":"987654321","plan":"regular"}
{"nombre":"Luis Huamán Torres","telefono":"912345678","plan":"socio"}
{"nombre":"Carmen Flores Rojas","telefono":"945112233","plan":"regular"}
```

(Son 10 filas, una por vecino; aquí, las tres primeras.) Salieron las filas de otra tabla por una ruta que debía buscar reservas. El mismo truco sobre `sqlite_master` lista todas las tablas de la base.

Seamos precisos con lo que demuestra

- `GET /api/members` **ya devuelve esos teléfonos**, y Kallpa no tiene autenticación. No es eso lo que bloquea el cambio.
- Lo que lo bloquea: el texto del usuario se volvió código. Toda tabla futura queda al alcance de esa línea.
- `'; DROP TABLE bookings; --` responde 200 y la tabla sigue ahí: `prepare` de `node:sqlite` solo compila la primera sentencia. Depende de la biblioteca. No es una protección.

¿Y las herramientas?

`make check` pasó completo. `strictTypeChecked` no da ningún aviso en ese archivo: `team` es un `string` y el template string es válido.

La revisión escrita

comentarios del pull request

```
issue (blocking, security): la consulta se arma con el texto del usuario.
Con ?team=O'Higgins responde 500, y con un UNION lee otras tablas (curl en la rama).
Usen un parámetro: LIKE '%' || ? || '%' con .all(team).

issue (blocking, test): la prueba no crea reservas ni mira el resultado. Sigue en verde si
se quita el WHERE. Pido una con dos equipos, comprobando que solo sale el que coincide, y
otra con un apóstrofe.

question: ¿qué debe pasar con ?team= vacío? Hoy devuelve todas las reservas. El README
define VALIDATION_ERROR para un dato que falta.

question (non-blocking): con parámetros, un  %  en el texto sigue siendo comodín de LIKE.
¿Se acepta? Para esta búsqueda, me parece que sí, pero quisiera que sea una decisión.

praise: la ruta y el servicio siguen la estructura del resto de los handlers.

todo: falta la ruta en la tabla de la API del README.
```

Decisión: pedir cambios. No por perfección: empeora la salud del código. Después de corregirlo (`busqueda-corregida.diff`), la misma inyección devuelve `[]`, y quitar el `WHERE` hace fallar las pruebas.

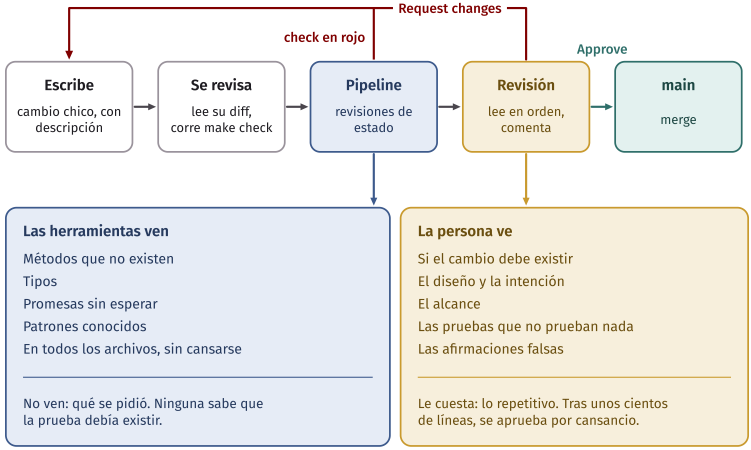
Un procedimiento

1. **Pedir cambios chicos.** La unidad es lo que ustedes pueden revisar, no lo que la herramienta puede escribir.
2. **`make check` antes de leer** (`make -k check` muestra los tres pasos, aunque falle el primero).
3. **Leer primero lo que cambió en las pruebas.** Después, las nuevas: ¿qué mutante matan?
4. **Mirar la lista de archivos.** Lo que no es parte de la tarea se saca.
5. **Comprobar en la documentación** cada API que no conozcan.
6. **Revisar cada dependencia nueva** (semana 12).
7. **Comprobar la descripción.** “Validado”, “seguro”, “probado” son afirmaciones.

La responsabilidad no cambia

Quien abre el pull request responde por todo el código, lo haya escrito a mano o con un agente, y tiene que poder explicarlo. Si no puede, el pull request no está listo.

Pipeline primero, persona después



SECCIÓN 09

Laboratorio

09

Laboratorio: revisar

1. Dos pull requests de otro equipo: tres comentarios en líneas, con etiquetas, y una decisión con el estándar.

contenedor

```
$ gh pr checkout <número> -R <owner>/<repositorio>
$ make check
```

2. `pr-generado.diff`, ilustración escrita para la clase: primero sin correr nada, comentarios en `revision.md`.

/work/kallpa-semana10

```
$ git switch -c pr-generado main
$ git apply /work/code/week-11/pr-generado.diff
$ make -k check
```

Individual. ¿Qué encontraron las herramientas que ustedes no vieron? ¿Qué vieron ustedes que ninguna reporta?

Laboratorio: endurecer

3. **En parejas.** `noUncheckedIndexedAccess`, luego `exactOptionalPropertyTypes`: cada error con una guarda, un valor por omisión o un cambio de estructura. Sin `!` ni `as`. `make test` sigue igual.
4. `strictTypeChecked` con `projectService`, y agrupar:

su repositorio

```
$ eslint . -f json | jq -r '.[].messages[].ruleId' | sort | uniq -c | sort -rn
```

5. Regla por regla: corregir, configurar o desactivar. Todos los de `src/`. Qué hacer con `test()`, `describe()` y el `any` de `test/helpers.ts`. Un comentario con el porqué.
6. `make check` en verde y un pull request con una tabla: regla, cuántos avisos, qué se decidió y por qué. Lo revisa la otra pareja.

Para la próxima clase

- Terminar el laboratorio: `revision.md` y el pull request con la tabla de reglas.
- **P3**: entrega y demo del sprint 2.
- **P4**: publicado. Pipeline de CI/CD, despliegue, observabilidad, revisión de seguridad y postmortem; entrega en la semana 14.
- Semana 12, integración continua y dependencias. Lecturas: SWEG, caps. 23 *Continuous Integration* y 18 *Build Systems and Build Philosophy*, hasta *Modern Build Systems*; SWEG, cap. 21 *Dependency Management*; Russ Cox, *Our Software Dependency Problem*.

Pregunta

En su proyecto, ¿qué regla de análisis estático habría encontrado un error real que ya tuvieron? ¿Con cuántos de sus avisos harían algo?