

UNIVERSIDAD NACIONAL DE INGENIERÍA

Ingeniería de Software

Pruebas unitarias, dobles de prueba, TDD y BDD

Tamaño y alcance · Dobles de prueba · Cobertura · Rojo, verde, refactorizar

Gerson Garrido

Semana 9

Hoy

Sesión A

1. Para qué se prueba
2. Tamaño y alcance
3. Pruebas que sobreviven a un refactoring
4. Pruebas claras
5. Dobles de prueba
6. Cobertura

Sesión B

7. TDD
8. BDD
9. TDD con un agente

Lecturas

A: SWEG, caps. 11, 12 y 13. Complementarias: Fowler, *TestDouble* y *Mocks Aren't Stubs*; Vocke, *The Practical Test Pyramid*; Google Testing Blog, *Code Coverage Best Practices*; Node.js, *Test runner*.

B: Beck, *Canon TDD*; Fowler, *TestDrivenDevelopment* y *GivenWhenThen*; Cucumber, *Behaviour-Driven Development*. Complementarias: Cucumber, *Gherkin reference*; Node.js, *Test runner*.

SECCIÓN 01

Para qué se prueba

01

No para “ver si funciona hoy”

> **Suite de pruebas** *(test suite)*
>
> Programas que ejercitan otro programa y comparan lo que ocurre con lo que debería ocurrir.

La pregunta en un sistema que vive años: cuando alguien lo cambie dentro de seis meses, ¿cómo va a saber que no rompió nada?

- **Detecta errores** cerca de quien los introdujo.
- **Deja cambiar el código**: es lo que permitió refactorizar en la semana 5.
- **Documenta lo que se promete**: `los socios tienen 10% de descuento`.

SWEG, cap. 11: en Google Web Server, tras exigir pruebas en todo cambio nuevo, los pases de emergencia bajaron a la mitad en un año. Lo cuenta la propia empresa, sin grupo de comparación.

La regla de Beyoncé

> **SWEG, cap. 11**
>
> *"If you liked it, then you shoulda put a test on it."*
> Si te importaba, le hubieras puesto una prueba.

- Es una regla sobre responsabilidades: quien cambia una dependencia no tiene por qué adivinar un comportamiento que nadie probó.
- Si todas las pruebas pasan y el producto se rompe, faltaba una prueba.
- Se junta con la ley de Hyrum (semana 1): alguien depende de todo lo observable, y ninguna suite lo cubre todo. ¿Qué les importa a ustedes y a su cliente?

Dos condiciones, dos lados

Falla cuando el comportamiento cambió

si no, deja pasar el error

No falla cuando no cambió

si no, gasta tiempo y pierde credibilidad

Pruebas flaky

Pasan o fallan sin que nada cambie. SWEG, cap. 11: en Google rondan el 0.15 %, y al acercarse al 1 % las pruebas empiezan a perder valor.

SECCIÓN 02

Tamaño y alcance

02

Dos preguntas distintas

Tamaño: qué recursos usa

Pequeña	un proceso; sin red, disco ni `sleep`
Mediana	una máquina; red a `localhost`
Grande	lo que necesite

Alcance: cuánto código ejercita

Unitaria	una unidad
Integración	varias juntas
Punta a punta	el sistema

Por qué le importa el tamaño a SWEG

De los recursos vienen la lentitud y el no determinismo. Una prueba sin llamadas bloqueantes es difícil que se vuelva flaky por accidente.

Las pruebas de Kallpa

Prueba	Qué usa	Tamaño	Alcance
`test/pricing.test.ts`	funciones del dominio	pequeña	unitaria
`week-09/booking-store.test.ts`	repositorio + SQLite `:memory:`	pequeña	integración
`test/http.test.ts`	servidor en `127.0.0.1`	mediana	integración

- Pequeña no es lo mismo que unitaria: `:memory:` corre en el proceso, sin disco ni red.
- `http.test.ts` corre en un solo proceso. Es mediana porque usa red a localhost.
- ¿Qué tendría que usar para dejar de ser mediana? Otra máquina: un WhatsApp de verdad.

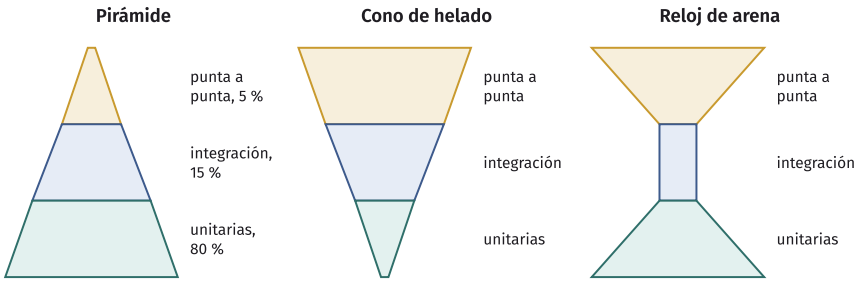
Tamaño y alcance, en una cuadrícula



En la misma fila, distinto alcance: una prueba pequeña no es lo mismo que una unitaria.

En la misma columna, distinto tamaño: http.test.ts es mediana porque hace llamadas de red, aunque sean a localhost.

La pirámide



muchas unitarias, menos de integración, pocas de punta a punta

sale de prototipos que pasaron rápido a producción; suites lentas y poco confiables

el acoplamiento hace difícil levantar las dependencias por separado

80, 15 y 5 son una guía muy aproximada de SWEG, y hablan de alcance, no de tamaño.

Arriba: más confianza en el sistema completo, pero más lentas, más caras y con fallas ajenas al código.

Cómo leer la pirámide

- Alcance: arriba, más confianza en el sistema, y más lentas y frágiles.
- SWEG: cerca de 80 / 15 / 5, “*a very rough guideline*”, y es de alcance, no de tamaño.
- Lo que hay que evitar: el **cono de helado** (casi todo de punta a punta) y el **reloj de arena** (pocas de integración).

La pregunta, no el porcentaje

Para este comportamiento, ¿cuál es la prueba más pequeña que lo verifica de verdad?

SECCIÓN 03

Pruebas que sobreviven a un refactoring

03



Frágil y ciega

Prueba frágil *(brittle)*, SWEG cap. 12

Falla ante un cambio que no introdujo ningún error real.

Prueba ciega

Pasa aunque el comportamiento esté roto.

Idea clave

Una prueba que falla cuando el comportamiento no cambió es tan cara como una que no falla cuando sí cambió.

Dos pruebas para reservar dos horas (las dos pasan hoy)

Por dentro: interacción

```
const saveBooking = t.mock.fn(() => 7);
const service = {
  findCourt: () => ({ /* ... */ }),
  findMember: () => ({ /* ... */ }),
  bookingsForCourt: () => [],
  saveBooking,
  findBooking: () => ({ /* ... */ }),
};
// ... req y res de mentira
await createBooking(deps)({ req, res, url,
      params: {} });

assert.equal(saveBooking.mock.callCount(),
      1);
```

Por fuera: comportamiento

```
const server = await startTestServer();
t.after(() => server.close());

await postJson(`${server.url}/api/bookings
    `, {
  courtId: 1, memberId: 1, date: '
      2026-03-10',
  startHour: 19, hours: 2, teamName: 'Los
      Pumas',
});

const { body } = await getJson(
  `${server.url}/api/courts/1/availability
      ?date=2026-03-10`);
const taken = /* horas ocupadas */;
assert.deepEqual(taken, [19, 20]);
```

Cambio 1: un refactoring

`saveBooking` pasa a llamarse `insertBooking`. Ningún comportamiento cambia.

in-kallpa/run.sh

```
== refactoring: saveBooking pasa a llamarse insertBooking
 src/http/handlers.ts           | 2 +-
 src/services/booking-service.ts | 2 +-
 2 files changed, 2 insertions(+), 2 deletions(-)
ℹ pass 9
ℹ fail 0
✓ una reserva de dos horas ocupa las dos horas
✗ createBooking llama a service.saveBooking
ℹ tests 2
ℹ pass 1
ℹ fail 1
✗ failing tests:
✗ createBooking llama a service.saveBooking
  TypeError [Error]: deps.service.insertBooking is not a function
```

Falsa alarma: las 9 pruebas de `http.test.ts` siguen en verde y solo se rompe la de interacción.

Cambio 2: un error de verdad

El repositorio guarda siempre 1 hora: la cancha queda libre a las 20:00.

in-kallpa/run.sh

```
== error: el repositorio guarda siempre 1 hora
 src/store/booking-repository.ts | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
✗ una reserva de dos horas ocupa las dos horas
✓ createBooking llama a service.saveBooking
i pass 1
i fail 1
✗ failing tests:
✗ una reserva de dos horas ocupa las dos horas
  AssertionError [ERR_ASSERTION]: Expected values to be strictly deep-equal:
    [
      19,
  -   20
    ]
```

Error no detectado: la prueba de interacción reemplazó justo la parte que estaba mal.

El resultado de los dos experimentos

	Refactoring (nada se rompió)	**Error real** (el negocio se rompió)
Prueba de interacción	falla	pasa
Prueba de comportamiento	pasa	falla

- **A través de la interfaz pública**: como la llaman sus usuarios.
- **Comportamientos, no métodos**: una prueba por regla, no `test('quotePriceCents')`.
- **El estado antes que las llamadas**: qué quedó, no cómo se llegó.

SECCIÓN 04

Pruebas claras

04

DAMP antes que DRY

Demasiado DRY

```
1  test('no deja cancelar',
       async () => {
2    const id = await
       bookDefault(server);
3    await expectCancelError(
       server, id);
4  });
```

¿Qué reserva? ¿A qué hora? ¿Por qué debería fallar?

DAMP

```
1  test('una hora antes ya no se puede cancelar', async
       () => {
2    const created = await postJson(`${server.url}/api/
       bookings`, {
3      courtId: 1, memberId: 1, date: '2026-03-09',
4      startHour: 11, hours: 1, teamName: 'Los Pumas',
5    });
6
7    const { status, body } = await postJson(
8      `${server.url}/api/bookings/${created.body.booking
         .id}/cancel`);
9
10   assert.equal(status, 422);
11   assert.equal(body.error.code, 'TOO_LATE_TO_CANCEL');
12 });
```

Los helpers son buenos (`postJson`) si no esconden lo que hace que el resultado sea el esperado.

Sin lógica en las pruebas

Dos aserciones sobre el mismo precio

```
1  // Mal: la prueba calcula lo mismo que el código.
2  assert.equal(quotePriceCents(quote), HOURLY_RATE_CENTS.fulbito * 1.25);
3
4  // Bien: el valor sale de la tabla de precios del README.
5  assert.equal(quotePriceCents(quote), 11250);
```

- Un `for`, un `if` o un cálculo son código que también puede estar mal, y nadie prueba las pruebas.
- El nombre dice el comportamiento y la condición.
- Preparar, ejecutar, revisar, con una línea en blanco entre ellos.

Aisladas: una prueba que depende de otra

demos/order-dependent.test.ts

```ts
const db = openDatabase(':memory:');
seedCatalog(db);
const repository = new
      BookingRepository(db);
test('guarda una reserva', () => {
  const id = repository.insert({ /*
      ... */ });
  assert.equal(repository.findById(
      id)?.start_hour, 19);
});
test('lista las reservas del día',
      () => {
  assert.equal(repository.listByDate
      ('2026-03-10').length, 1);
});
```

corriendo solo la segunda

```
$ node --test --test-name-pattern="lista las reservas del día" \
    demos/order-dependent.test.ts
✗ lista las reservas del día (1.55325ms)
ℹ tests 1
ℹ pass 0
ℹ fail 1
...
  AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:
  0 !== 1
```

Con todo el archivo, pasan las dos. El código no tiene ningún error.

Aislar: cada prueba prepara lo que necesita

demos/order-independent.test.ts

```ts
let repository: BookingRepository;

beforeEach(() => {
  const db = openDatabase(':memory:');
  seedCatalog(db);
  repository = new BookingRepository(db);
});

test('lista las reservas del día', () => {
  repository.insert(PUMAS);
  assert.equal(repository.listByDate('
      2026-03-10').length, 1);
});
```

con otro orden

```
$ node --test --test-randomize --test-random-seed=1 \
    demos/order-dependent.test.ts
✗ lista las reservas del día (1.543708ms)
✓ guarda una reserva (0.3215ms)
ℹ Randomized test order seed: 1
ℹ tests 2
ℹ suites 0
ℹ pass 1
ℹ fail 1
```

`--test-randomize` sortea el orden y `--test-random-seed` lo repite. Con `beforeEach` la prueba pasa sola y con todas.

De dónde sale el oráculo

Oráculo: decide si un resultado es correcto. Viene de la especificación, no del código.

demos/oracle.test.ts: formatSoles tiene un error, muestra un solo decimal

```
1  // copia la fórmula del código
2  assert.equal(formatSoles(11250), `S/ ${(11250 / 100).toFixed(1)}`);
3  // copia el valor que devolvió el código
4  assert.equal(formatSoles(11250), 'S/ 112.5');
5  // usa el valor que dice la especificación
6  assert.equal(formatSoles(11250), 'S/ 112.50');
```

node --test demos/oracle.test.ts

```
✓ copia la fórmula del código (0.566834ms)
✓ copia el valor que devolvió el código (0.098208ms)
✗ usa el valor que dice la especificación (0.76875ms)
```

SECCIÓN 05

Dobles de prueba

05



Costuras: dónde poner un doble

Doble de prueba *(test double)* y costura *(seam)*

Un objeto que reemplaza a una dependencia durante una prueba, y el punto del código donde se puede cambiarla sin tocarlo.

src/http/server.ts

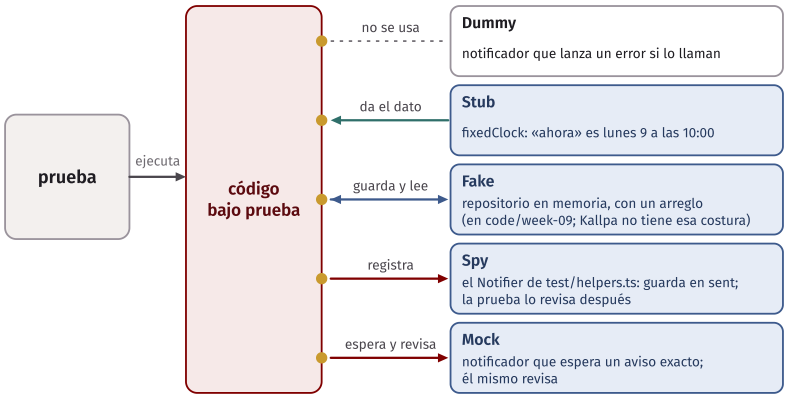
```ts
export type ServerOptions = {
  db: DatabaseSync;
  clock?: Clock;
  notifier?: Notifier;
  log?: (line: string) => void;
};
```

Con *dependency injection*, Kallpa recibe el reloj y el notificador en `createServer` en lugar de llamar a `new Date()` o a WhatsApp desde el handler.

Las cinco clases de Meszaros

Clase	Qué hace	En Kallpa
Dummy	Se pasa y nunca se usa	Notificador que lanza un error, en una prueba que no reserva
Stub	Da respuestas fijas	`fixedClock`: “ahora” siempre es el lunes 9 a las 10:00
Fake	Funciona, con un atajo que no sirve en producción	Repositorio de reservas con un arreglo
Spy	Stub que registra cómo lo llamaron	`Notifier` de `test/helpers.ts`, guarda en `sent`
Mock	Trae programadas las llamadas que espera	Notificador que espera un aviso exacto

Qué controla cada doble



costura: el punto donde se cambia la dependencia sin tocar el código

La clase la define para qué se usa, no con qué se crea: t.mock.fn() puede ser un stub o un spy.

Dummy y stub

dummy

```
1  const dummy: Notifier = {
2    bookingConfirmed() {
3      throw new Error('este doble no deber
         ía usarse');
4    },
5  };
6  const server = await startServer({
       notifier: dummy });
7
8  const response = await fetch(`${server.
       url}/api/courts`);
9  assert.equal(response.status, 200);
```

El `log: () => {}` de los helpers no es dummy: el servidor lo llama en cada pedido.

stub

```
1  const server = await startServer({
2    clock: fixedClock('2026-03-09T15:00:00Z')
       , // 10:00
3  });
4
5  const response = await postBooking(server.
       url, AT_NINE);
6
7  assert.equal(response.status, 422); //
       IN_THE_PAST
```

Con `2026-03-09T13:00:00Z` (8:00), el mismo pedido responde 200. La prueba no depende de la hora en que se corra.

Spy con `t.mock.fn()`

doubles.test.ts

```ts
test('spy: el log registra cada pedido con su código de respuesta', async (t) => {
  const log = t.mock.fn<(line: string) => void>();
  const server = await startServer({ log, notifier: { bookingConfirmed() {} } });
  t.after(() => server.close());
  const tomorrow = { ...AT_NINE, date: '2026-03-10' };

  await postBooking(server.url, tomorrow);
  await postBooking(server.url, tomorrow);

  assert.deepEqual(
    log.mock.calls.map((call) => call.arguments[0]),
    ['POST /api/bookings 200', 'POST /api/bookings 409'],
  );
});
```

El nombre engaña: `t.mock.fn` crea un spy porque se revisa *después*. Con un valor fijo y sin revisar llamadas, sería un stub. **La clase la decide el uso**, no la herramienta.

Mock: sabe de antemano qué espera

doubles.test.ts: escrito a mano, node:test no trae uno

```ts
function mockNotifier(expected: BookingConfirmation): Notifier & { verify(): void } {
  let received = 0;
  return {
    bookingConfirmed(confirmation) {
      assert.deepEqual(confirmation, expected);
      received++;
    },
    verify() {
      assert.equal(received, 1, `se esperaba un aviso y hubo ${received}`);
    },
  };
}
```

- Con el spy se revisa después; el mock revisa él mismo, en la llamada y en `verify()`.
- Si les piden “ponle un mock”: ¿controlar una respuesta o revisar una llamada?

Un fake y su contrato

booking-store.test.ts

```ts
bookingStoreContract('SQLite en memoria', () => {
  const db = openDatabase(':memory:');
  seedCatalog(db);
  return new BookingRepository(db);
});

bookingStoreContract('fake', () => new FakeBookingRepository());
```

Prueba de contrato: las mismas pruebas sobre la real y el fake, para cuidar la **fidelidad**.

un fake que dejó de ordenar

```
$ node --test demos/fake-drift.test.ts
...
ℹ pass 3
ℹ fail 1
...
✗ las reservas de una cancha salen ordenadas por hora de inicio (0.955291ms)
  AssertionError [ERR_ASSERTION]: Expected values to be strictly deep-equal:
  + actual - expected
    [
  +   19,
      9,
  -   19
    ]
```

Solo cubre lo que revisa: este fake no conoce las claves foráneas.

Cuándo usar cada uno (SWEG, cap. 13)

1. **La implementación real**, si es rápida y determinista: *prefer realism over isolation*.
2. **Un fake**, mantenido junto a lo que imita, si la real no se puede usar.
3. **Un stub**, con moderación: cada uno es una copia de lo que uno cree que hace la dependencia.
4. **Interaction testing**, solo en funciones que cambian estado y cuyo efecto no se puede observar de otra forma.

En Kallpa

Avisar por WhatsApp no deja rastro: la llamada es el comportamiento. `saveBooking` también cambia estado, pero la reserva se ve en la disponibilidad: mejor mirar eso.

¿`:memory:` es un fake?

SWEG da “una base de datos en memoria” como ejemplo de fake

Se refiere a una implementación **aparte**, más liviana, de la misma interfaz.

- En Kallpa, `:memory:` es el mismo motor, con el mismo SQL y las mismas restricciones. Solo cambia dónde se guardan los datos: es la implementación real.
- Nadie escribió una segunda implementación que pueda alejarse de la primera.
- Si en producción fuera PostgreSQL y en las pruebas SQLite, sí tendría los problemas de fidelidad de un fake.

La costura del reloj

no-seam.ts: sin costura

```ts
export function hasStarted(date: string,
    hour: number) {
  const now = limaTime(new Date());
  return date < now.date ||
    (date === now.date && hour <= now.
      hour);
}
```

La misma llamada da `false` a las 10:00 y `true` a las 13:00 (`seam.test.ts` lo muestra con `t.mock.timers`).

con costura: now es un argumento

```ts
const now: LimaTime =
  { date: '2026-03-09', hour: 10, minute:
      0 };
assert.equal(checkBookingRules(
  { date: '2026-03-09', startHour: 12,
      hours: 1 }, now),
  null);
```

Los temporizadores simulados sirven para código que no se puede cambiar. La costura es mejor: la dependencia queda en la firma y no se altera nada global.

Código difícil de probar suele ser una señal de diseño.

SECCIÓN 06

Cobertura

06

Qué se mide

De **líneas**, de **ramas** (`if`, `?:`, `&&`), de **funciones**.

```
node --test --experimental-test-coverage "test/**/*.test.ts"
```

```
ℹ file                  | line %  | branch %  | funcs %  | uncovered lines
ℹ   booking.ts          |  95.56  |    87.50  |  100.00  | 50-51 53-54
ℹ   calendar.ts         | 100.00  |   100.00  |  100.00  |
ℹ   clock.ts            | 100.00  |   100.00  |   80.00  |
ℹ   pricing.ts          | 100.00  |   100.00  |  100.00  |
ℹ   report.ts           | 100.00  |   100.00  |  100.00  |
ℹ   handlers.ts         |  86.80  |    77.55  |   85.71  | 54 89-90 135-136 139-140 149-151 158-163 170-171 178-179 182-187
ℹ   json.ts             |  91.30  |    75.00  |  100.00  | 24-25 31-32
ℹ   router.ts           | 100.00  |   100.00  |  100.00  |
ℹ   server.ts           |  92.96  |    55.56  |  100.00  | 57-58 64-66
ℹ   static-files.ts     |  81.82  |    66.67  |  100.00  | 18-21
ℹ   notifier.ts         |  86.67  |   100.00  |    0.00  | 20-22 28
ℹ   booking-service.ts  |  89.83  |   100.00  |   75.00  | 29-30 41-42 45-46
ℹ   booking-repository.ts |  90.16  |   100.00  |   75.00  | 39-40 43-46
ℹ   court-repository.ts |  100.00  |   100.00  |  100.00  |
ℹ   database.ts         |  95.08  |    80.00  |  100.00  | 57-59
ℹ   member-repository.ts |  92.00  |   100.00  |   80.00  | 12-13
ℹ   seed.ts             |  76.84  |   100.00  |   33.33  | 72-90 93-95
ℹ all files             |  91.54  |    86.91  |   86.41  |
```

La columna más útil no tiene porcentajes: **uncovered lines** señala qué mirar.

Líneas y ramas no son lo mismo

src/domain/pricing.ts, línea 27

```
1  return isPeakHour(date, hour) ? base + (base * PEAK_SURCHARGE_PERCENT) / 100 : base;
```

Con una sola prueba, `hourlyRateCents('fulbito', '2026-03-10', 10)`, de una hora normal:

solo pricing.ts

```
i pass 1
i fail 0
i   pricing.ts |  83.67 |    75.00 |   50.00 | 39-44 48-49
```

La línea 27 no aparece entre las sin cubrir: se ejecutó. Pero el recargo de hora punta nunca se calculó, y eso solo se ve en las ramas.

Una prueba sin aserciones: 100 %

in-kallpa/no-assert.test.ts

```ts
test('precios', () => {
  quotePriceCents({ sport: 'fulbito',
    date: '2026-03-10', startHour: 10,
      hours: 1,
    plan: 'regular' });
  quotePriceCents({ sport: 'fulbito',
    date: '2026-03-10', startHour: 20,
      hours: 1,
    plan: 'socio' });
  formatSoles(11250);
});
```

in-kallpa/run.sh

```
== sin aserciones, pricing.ts correcto
i pass 1
i fail 0
i   pricing.ts | 100.00 |   100.00 |  100.00 |
== sin aserciones, descuento de socio de 50 %
i pass 1
i fail 0
i   pricing.ts | 100.00 |   100.00 |  100.00 |
== las pruebas de pricing.test.ts con el mismo error
  ✗ los socios tienen 10 % de descuento
i fail 1
i pass 5
```

Una línea cubierta solo dice que se ejecutó, no que una aserción revisó su resultado. La misma cifra puede ser una suite que verifica mucho o una que no verifica nada.

Qué archivos cuenta

`src/main.ts`, `src/config.ts` y `src/seed.ts` no estaban en la tabla. No es que tengan 100 %: ninguna prueba los carga, y Node solo reporta los que se cargaron.

```
node --test --experimental-test-coverage --test-coverage-include='src/**'
--test-coverage-include-all "test/**/*.test.ts"

ℹ config.ts             |  28.57 |  100.00 |  100.00 | 1-2 7-14
ℹ main.ts               |   0.00 |  100.00 |  100.00 | 1-25
ℹ seed.ts               |   0.00 |  100.00 |  100.00 | 1-22
ℹ  seed.ts              |  76.84 |  100.00 |   33.33 | 72-90 93-95
ℹ all files             |  85.79 |   86.91 |   86.41 |
```

De 91.54 % a 85.79 %, con las mismas pruebas y el mismo código. El número depende de qué se decide contar.

Un mínimo, no un objetivo

kallpa

```
$ node --test --experimental-test-coverage --test-coverage-lines=95 "test/**/*.test.ts"
...
ℹ Error: 91.54% line coverage does not meet threshold of 95%.
$ echo $?
1
```

- **Como piso**, sirve: evita que la suite se deteriore sin que nadie lo note. P3 lo usa con 70 %.
- **Como meta**, se infla (ley de Goodhart): cuando una medida se vuelve objetivo, deja de ser buena medida.
- SWEG, cap. 11: un mínimo de 80 % termina funcionando como techo.
- Google Testing Blog (2020): 60 % “aceptable”, 75 % “encomiable”, 90 % “ejemplar”; son lineamientos de una empresa, no un estudio, y cada equipo elige. Lo más útil: lo que no está cubierto dice más que lo que sí.

Con IA: pruebas generadas

Una herramienta que genera pruebas a partir del código tiende a:

- describir lo que el código hace, incluidos sus errores (el oráculo copiado);
- usar un mock de la misma unidad que prueba;
- escribir pruebas sin aserciones útiles (`assert.ok(result)`). La cobertura sube igual.

La pregunta

Por cada prueba generada: ¿qué error del código haría que falle? Si no hay respuesta, no verifica nada. Para comprobarlo, introduzcan ese error a mano.

SECCIÓN 07

TDD

07



Canon TDD, según Beck

1. Escribir una **lista** de escenarios.
2. Convertir **exactamente uno** en una prueba concreta.
3. Cambiar el código para que esa prueba y todas las anteriores pasen.
4. **Opcionalmente**, refactorizar.
5. Volver al paso 2 hasta vaciar la lista.

Lo que no es TDD

Escribir las pruebas después. Convertir toda la lista en pruebas de una vez. Mezclar el refactoring con hacer pasar la prueba.

Por qué funciona, cuando funciona

- **La prueba obliga a diseñar la interfaz** desde quien la llama.
- **Cada paso es chico**: si algo se rompe, está en las últimas líneas.

La evidencia es mixta:

- Nagappan et al. (2008): 3 equipos en Microsoft y 1 en IBM frente a proyectos parecidos. Densidad de defectos antes del release, entre 40 % y 90 % menor; tiempo inicial, entre 15 % y 35 % mayor *según la apreciación de los equipos*. Sin asignación al azar.
- Fucci et al. (IEEE TSE): 82 observaciones de 39 profesionales. Calidad y productividad se asocian con ciclos cortos y parejos; el orden (prueba antes o después) no tuvo influencia importante. Correlación, en tareas acotadas.

Lo que nadie discute: queda una suite que permite refactorizar, si se escribió bien.

El rojo importa

Una prueba que nunca se vio fallar puede no estar probando nada.

node –test test/busiest-hour.test.ts, sin el módulo: no sirve todavía

```
Error [ERR_MODULE_NOT_FOUND]: Cannot find module '/work/code/week-09/build/tdd/src/domain/busiest-hour.ts' imported from ...
✗ test/busiest-hour.test.ts (59.633208ms)
```

Un error de tipeo en el import se vería igual. Se agrega lo mínimo para que el import funcione:

el mismo comando: ahora sí, por la aserción

```
✗ sin reservas no hay hora más pedida (0.798458ms)
ℹ tests 1
ℹ pass 0
ℹ fail 1
...
  AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:

  -1 !== null
```

Tres formas de llegar al verde

Fake it

devolver la constante que hace pasar la prueba

Triangular

un segundo ejemplo obliga a generalizar

Implementación obvia

si el camino es claro, se escribe

La velocidad se regula: pasos chicos cuando hay dudas, más grandes cuando no. (Beck, *Test-Driven Development: By Example*, 2002.)

El pedido y la lista

> **La administración**
>
> La caja del día tiene que decir cuál fue la hora más pedida, para saber cuándo conviene poner a alguien más en la entrada.

La lista (comportamiento, no implementación):

1. Sin reservas, no hay hora más pedida.
2. Con una reserva, la más pedida es su hora de inicio.
3. Una reserva de varias horas cuenta en cada una de sus horas.
4. Si hay empate, gana la hora más temprana.
5. Las canceladas no cuentan.

`make tdd` repite todos los pasos en `build/tdd`.

Ciclo 1: fingir

rojo 1

```
1  test('sin reservas no hay hora más pedida', () => {
2    assert.equal(busiestHour([]), null);
3  });
```

verde 1: fake it

```
1  export function busiestHour(bookings: readonly { startHour: number; hours: number }[]): number |
       null {
2    return null;
3  }
```

El rojo de esta prueba es el de “El rojo importa”. Este verde parece trampa, pero confirma que la prueba funciona.

Ciclo 2: otro ejemplo

rojo 2

```
1  test('con una reserva, la más pedida es su hora de inicio', () => {
2    assert.equal(busiestHour([{ startHour: 19, hours: 1 }]), 19);
3  });
```

rojo por la razón correcta

```
✗ con una reserva, la más pedida es su hora de inicio (0.519791ms)
  AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:
  null !== 19
```

verde 2: el cambio más simple

```
1  return bookings.length === 0 ? null : bookings[0].startHour;
```

Ciclo 3: triangular

rojo 3

```
test('una reserva de varias horas
      cuenta en cada una', () => {
  const bookings = [
    { startHour: 9, hours: 1 },
    { startHour: 19, hours: 2 },
    { startHour: 20, hours: 1 },
  ];
  assert.equal(busiestHour(bookings),
      20);
});
```

la constante ya no alcanza

```
9 !== 20
```

verde 3: ahora sí hay que contar

```
const counts = new Map<number, number>();
for (const booking of bookings) {
  for (let hour = booking.startHour;
       hour < booking.startHour + booking.
      hours; hour++) {
    counts.set(hour, (counts.get(hour) ?? 0)
      + 1);
  }
}
let best: number | null = null;
let bestCount = 0;
for (const [hour, count] of counts) {
  if (count > bestCount) { best = hour;
      bestCount = count; }
}
return best;
```

Ciclo 4: el empate

rojo 4

```
1  test('si hay empate gana la hora más temprana', () => {
2    assert.equal(busiestHour([{ startHour: 19, hours: 1 }, { startHour: 9, hours: 1 }]), 9);
3  });
```

rojo

```
✓ sin reservas no hay hora más pedida (0.466292ms)
✓ con una reserva, la más pedida es su hora de inicio (0.083167ms)
✓ una reserva de varias horas cuenta en cada una de sus horas (0.09125ms)
✗ si hay empate gana la hora más temprana (0.495542ms)
...
  19 !== 9
```

Se hace pasar con una condición más en el `if`. Después, con el verde asegurado, se refactoriza.

Refactorizar, sin tocar ninguna prueba

src/domain/busiest-hour.ts

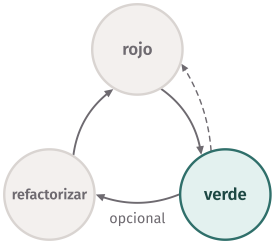
```ts
export function busiestHour(bookings: readonly Block[]): number | null {
  const counts = [...countPerHour(bookings)].sort(
    ([hourA, countA], [hourB, countB]) => countB - countA || hourA - hourB);
  return counts.length === 0 ? null : counts[0][0];
}
```

node --test test/busiest-hour.test.ts

```
✓ sin reservas no hay hora más pedida (0.483875ms)
✓ con una reserva, la más pedida es su hora de inicio (0.086375ms)
✓ una reserva de varias horas cuenta en cada una de sus horas (0.091375ms)
✓ si hay empate gana la hora más temprana (0.069083ms)
ℹ tests 4
ℹ pass 4
```

Hacerla pasar y refactorizar son dos sombreros. Primero uno, luego el otro.

Los cuatro ciclos



DE VUELTA EN LA PRUEBA DE AFUERA

prueba

la caja del día dice cuál fue la hora más pedida, sin contar las canceladas

```
busiestHour: busiestHour(confirmed),
```

El punto 5 no le toca a la función: lo verifica la prueba de aceptación.

LA LISTA

- ✓ 1. Sin reservas, no hay hora más pedida
- ✓ 2. Con una reserva, su hora de inicio
- ✓ 3. Varias horas: cuenta en cada una
- ✓ 4. En un empate, la hora más temprana
- ✓ 5. Las canceladas no cuentan

GIT LOG, UN COMMIT POR CICLO

8b41d83	Kallpa publicada
3c34b43	Sin reservas no hay hora más pedida
6fc5441	Con una reserva, la más pedida es su hora de inicio
6dd8a84	Cuenta cada hora de las reservas largas
b5435ef	En un empate gana la hora más temprana
5e3ff9d	La caja del día dice la hora más pedida

El ritmo se ve en la historia

```
git log --reverse --format='%h %s'

8b41d83 Kallpa publicada
3c34b43 Sin reservas no hay hora más pedida
6fc5441 Con una reserva, la más pedida es su hora de inicio
6dd8a84 Cuenta cada hora de las reservas largas
b5435ef En un empate gana la hora más temprana
5e3ff9d La caja del día dice la hora más pedida
```

- Un commit por ciclo, con todas las pruebas en verde.
- ¿Y el punto 5 de la lista? No le toca a esta función: `dailySummary` ya tiene las confirmadas. Lo cubre la prueba de aceptación.

SECCIÓN 08

BDD

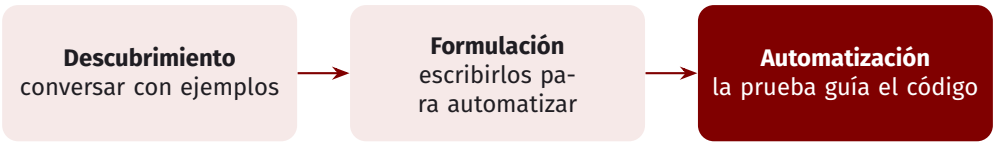
08



Empezar por una conversación

BDD ***(behaviour-driven development)***

Cerrar la brecha entre el negocio y lo técnico trabajando sobre ejemplos concretos. No es una herramienta.



Ustedes ya hicieron las dos primeras en la semana 3. BDD agrega el último paso: que se ejecuten.

De un criterio a una prueba

Dado que hoy es lunes 9 a las 10:00 y Rosa reservó la cancha 1 para hoy a las 11:00, **cuando** cancela, **entonces** la respuesta es 422 con `TOO_LATE_TO_CANCEL`.

acceptance.test.ts

```ts
test('una hora antes de empezar ya no se puede cancelar', async () => {
  // Dado que hoy es lunes 9 a las 10:00 y Rosa reservó la cancha 1 para hoy a las 11:00
  const created = await postJson(`${server.url}/api/bookings`, {
    courtId: 1, memberId: 1, date: '2026-03-09', startHour: 11, hours: 1, teamName: 'Los Pumas de Comas',
  });
  // Cuando cancela
  const { status, body } = await postJson(`${server.url}/api/bookings/${created.body.booking.id}/cancel`);
  // Entonces la respuesta es 422 con TOO_LATE_TO_CANCEL
  assert.equal(status, 422);
  assert.equal(body.error.code, 'TOO_LATE_TO_CANCEL');
  // ... y la cancha sigue ocupada a las 11:00
});
```

Gherkin y Cucumber

Un escenario en Gherkin

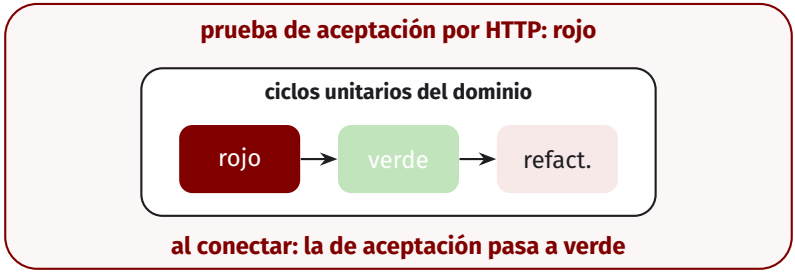
```
Escenario: Una hora antes ya no se puede cancelar
  Dado que hoy es lunes 9 de marzo a las 10:00
  Y que Rosa reservó la cancha 1 para hoy a las
11:00
  Cuando Rosa cancela la reserva
  Entonces la respuesta es 422 con el código
    "TOO_LATE_TO_CANCEL"
```

- Texto que lee el cliente; Cucumber conecta cada línea con una función.
- Esa capa cuesta. Vale si el cliente de verdad lee los escenarios.
- En el curso: `describe` y `test`, con nombres en español y comentarios Dado/Cuando/Entonces.

Error común

Decir “hacemos BDD” porque se usa Cucumber, sin conversar con el cliente. Lo contrario sí es BDD: una conversación con ejemplos y pruebas en `node:test`.

De afuera hacia adentro



- La de aceptación dice que el cliente recibe lo que pidió; las unitarias cubren los casos finos sin levantar un servidor por cada uno.
- Empezar por fuera evita construir un dominio impecable que nadie llama.

La hora más pedida, de afuera hacia adentro

test/busiest-hour-http.test.ts

```
1  // Dado que el martes hay dos reservas a las 19:00, una a las 8:00 y otra a las 8:00 que se cancel
       ó
2  // ... cuatro POST y una cancelación
3  // Cuando la administración pide la caja de ese día
4  const { status, body } = await getJson(`${server.url}/api/reports/daily?date=2026-03-10`);
5  // Entonces la hora más pedida es las 19:00
6  assert.equal(status, 200);
7  assert.equal(body.busiestHour, 19);
```

antes de los cuatro ciclos

```
AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:
undefined !== 19
```

Si se contara la cancelada, las 8:00 empatarían y ganarían. Al final: tres líneas en `report.ts` y `make check` con 45 pruebas.

SECCIÓN 09

TDD con un agente

09

TDD como protocolo con un agente

Reparto de roles

Ustedes escriben las pruebas: la especificación.

El agente escribe el código que las hace pasar.

Dos riesgos en el diff:

1. Que cambie o borre las pruebas para que pasen.
   `git diff –stat main – test/`
2. Que haga pasar justo los ejemplos.

Que una prueba cambie no siempre es malo: si el pedido cambia la forma de una respuesta, tiene que cambiar. Pero la razón tiene que salir del pedido, no de que estaba en rojo.

Hecha a la medida de los ejemplos

como los que suele proponer un agente

```
1  function busiestHour(bookings) {
2    if (bookings.length === 0) return
       null;
3    if (bookings.length === 3) return
       20;
4    return Math.min(
5      ...bookings.map((b) => b.
       startHour));
6  }
```

node –test demos/overfit.test.ts

```
✓ sin reservas no hay hora más pedida (0.484417ms)
✓ con una reserva, la más pedida es su hora de inicio (0.089916ms)
✓ una reserva de varias horas cuenta en cada una de sus horas (0.075541ms)
✓ si hay empate gana la hora más temprana (0.074125ms)
✗ ejemplo nuevo: dos reservas que comparten la hora de las 10:00 (0.5935ms)
ℹ tests 5
ℹ pass 4
ℹ fail 1
...
  9 !== 10
```

Ilustración escrita para la clase. No cuenta horas y pasa los cuatro ejemplos.
Pocos ejemplos se pueden pasar con condiciones especiales. La semana 10 cierra ese hueco.

SECCIÓN 10

Laboratorio

10

Pruebas unitarias de Kallpa con `node:test`

1. Medir: cada rango sin cubrir de `handlers.ts` y `json.ts` es una regla, una ruta o un código.
2. Correr una por una las pruebas de `test/http.test.ts` y arreglar las que dependan de otra.
3. Clasificar los dobles; prueba con `t.mock.fn()` como notificador.
4. Aceptación: al menos 6 códigos de error y 3 rutas, primero en Dado/Cuando/Entonces.
5. TDD de `occupancyPercent`, un commit por ciclo.
6. Un mínimo de cobertura en el `Makefile`.
7. Pull request con la cobertura antes y después.

Para la revisión

La otra pareja elige dos pruebas nuevas y dice qué error del código haría fallar cada una.

Rama nueva, en parejas, un commit por paso. `startTestServer` no recibe un notificador: extiendan el helper o llamen a `createServer`. Para cuerpos que no son JSON, `fetch`.

Para la próxima clase

- Terminar el laboratorio: pull request con la cobertura antes y después, y la revisión de otra pareja.
- **P3** (segundo sprint: calidad) está en curso; se entrega en la semana 11. Las pruebas de este laboratorio son el modelo para las del proyecto.
- Semana 10, pruebas basadas en propiedades, fuzzing y mutación. Lecturas: fast-check, *Introduction* y *Arbitraries*; MacIver, *What is Property Based Testing?*; Claessen y Hughes, *QuickCheck* (la introducción); StrykerJS, *Introduction* y *Tap Runner*; Petrović e Ivanković, *State of Mutation Testing at Google*.

Pregunta

Tomen la prueba de la que más se fían en su proyecto. ¿Qué error del código la haría fallar? Y una que debería fallar con ese error, ¿existe?