

UNIVERSIDAD NACIONAL DE INGENIERÍA

Ingeniería de Software

Pruebas basadas en propiedades, fuzzing, mutación y oráculos

fast-check · Fuzzing con oráculo implícito · Stryker · Oráculos para código generado

Gerson Garrido

Semana 10

Hoy

Sesión A

1. De los ejemplos a las propiedades
2. Encontrar, reducir, reproducir
3. Clases de propiedades
4. Generadores
5. Fuzzing

Sesión B

6. Pruebas de mutación
7. El problema del oráculo
8. El laboratorio

Lecturas

fast-check, *Introduction* y *Arbitraries*. MacIver, *What is Property Based Testing?* Claessen y Hughes, *QuickCheck* (ICFP 2000), la introducción.

StrykerJS, *Introduction* y *Tap Runner*. Petrović e Ivanković, *State of Mutation Testing at Google* (ICSE SEIP 2018).

Complementarias: fast-check, *Model-based testing*; *The Fuzzing Book*, *Fuzzing: Breaking Things with Random Inputs*; SWEG, cap. 14; StrykerJS, *Supported mutators* y *Configuration*; CMU 17-313.

Dos preguntas que quedaron de la semana pasada

1. ¿Qué pasa con todas las entradas que a nadie se le ocurrió escribir?
   → propiedades y fuzzing
2. Si el código tuviera un error aquí, ¿alguna prueba fallaría?
   → pruebas de mutación

Y la pregunta de fondo

¿Quién sabe cuál es la respuesta correcta? Sobre todo cuando la misma herramienta escribe el código y sus pruebas.

SECCIÓN 01

De los ejemplos a las propiedades

01

Ejemplo y propiedad

una prueba de ejemplo

```
1  assert.deepEqual(splitCents(9000, 3), [3000, 3000, 3000]);
```

Propiedad *(property)*

Una afirmación sobre todo un dominio: para todo entero $t \geq 0$ y todo $n \geq 1$, las partes de `splitCents(t, n)` suman t.

Prueba basada en propiedades *(property-based testing)*

Un programa que intenta refutarla generando entradas. Sin **contraejemplo**, queda sin refutar, no demostrada.

Los ejemplos eligen los casos fáciles

split.ts

```ts
// Todos pagan el promedio redondeado.
export function splitCentsNaive(totalCents: number, players: number): number[] {
  return Array.from({ length: players }, () => Math.round(totalCents / players));
}
```

split.test.ts

```ts
assert.deepEqual(splitCentsNaive(9000, 3), [3000, 3000, 3000]);
assert.deepEqual(splitCentsNaive(11250, 2), [5625, 5625]);
```

Las dos pasan. En las dos, la división es exacta. Nadie lo planeó: son los números fáciles de calcular a mano para escribir la prueba.

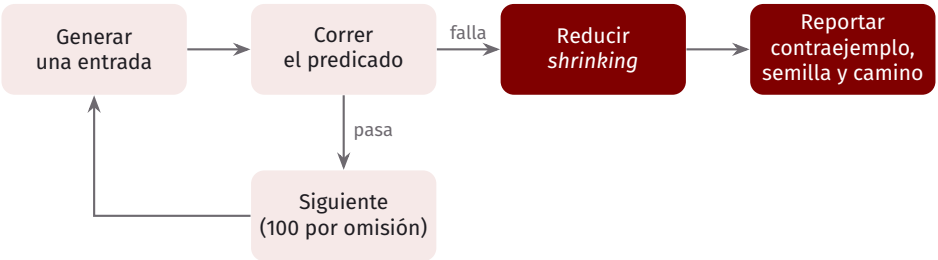
Anatomía de una propiedad en fast-check

demos/split-naive.demo.test.ts

```ts
test('las partes suman el total', () => {
  fc.assert(
    fc.property(
      fc.integer({ min: 0, max: 100_000 }), fc.integer({ min: 1, max: 22 }),
      (total, players) => {
        const parts = splitCentsNaive(total, players);
        assert.equal(parts.reduce((a, b) => a + b, 0), total);
      }),
    { seed: 2026 }, // solo para repetir la salida en clase
  );
});
```

Arbitraries: de dónde salen las entradas. **Predicado**: falla si lanza un error o devuelve `false`. `fc.assert`: corre 100 entradas por omisión y se detiene en el primer contraejemplo. Sin él, no se revisa nada.

Qué hace el runner



QuickCheck (Claessen y Hughes, 2000) inició la técnica; Hypothesis y fast-check vienen de ahí. MacIver: el azar no es lo esencial. Es “lo que uno hace, no lo que hace la computadora”.

SECCIÓN 02

Encontrar, reducir, reproducir

02

Leer un contraejemplo

```
make demo D=split-naive

✗ las partes suman el total (3.535417ms)
  Error: Property failed after 3 tests
  { seed: 2026, path: "2:1:0:0:0:0:1:0:0:0:0:0:0:0:0:0:0", endOnFailure: true }
  Counterexample: [1564,3]
  Shrunk 16 time(s)
...
    1563 !== 1564
```

- `Counterexample`: los valores, en el orden de los arbitraries.
- `Shrunk 16 time(s)`: el contraejemplo original era otro.
- Se comprueba a mano: $1564/3 = 521.33 \to 521$, y $3 \times 521 = 1563$. Falta un céntimo.

Siempre a mano: puede ser un error del código, de la propiedad, o una entrada que no debió generarse.

Shrinking: qué valores probó

verbose: 1

```
Encountered failures were:
- [99991,3]
- [49996,3]
- [24998,3]
- [12499,3]
- [6250,3]
- [3125,3]
- [2344,3]
- [1954,3]
...
- [1567,3]
- [1565,3]
- [1564,3]
```

Shrinking

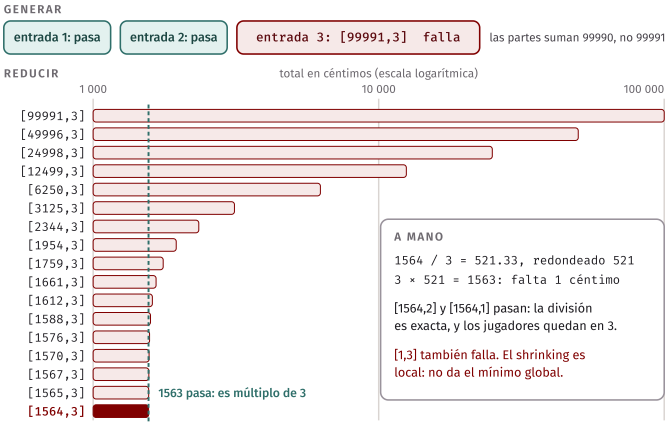
Probar versiones más simples de la entrada mientras sigan fallando. Reporta lo último que falló.

Busca por mitades. La mitad de 3125 es 1563, múltiplo de 3: **pasa**. Sigue entre 1563 y 3125, y termina pegado al valor que pasa.

No es el mínimo

`[1,3]` también falla, y es más simple. El shrinking es una búsqueda local con reglas fijas.

Shrinking, paso a paso



se detiene en [1564,3], después de 16 reducciones: ninguno de sus pasos falla ya

Semilla y camino: repetir exactamente

demos/split-naive-replay.demo.test.ts

```
1  { seed: 2026, path: '2:1:0:0:0:0:1:0:0:0:0:0:0:0:0:0', endOnFailure: true }
```

make demo D=split-naive-replay

```
Error: Property failed after 1 tests
{ seed: 2026, path: "2:1:0:0:0:0:1:0:0:0:0:0:0:0:0:0", endOnFailure: true }
Counterexample: [1564,3]
Shrunk 0 time(s)
```

- Si falla en la integración continua, copien esa línea del log.
- **Conviertan el contraejemplo en una prueba de ejemplo**: la próxima corrida generará otras entradas.
- En sus pruebas, no fijen la semilla: cada corrida explora entradas nuevas.

Cuántas corridas

Si el error aparece en una fracción p de lo que genera el generador, la probabilidad de no verlo en n corridas es $(1 - p)^n$.

p	sin verlo en 100	sin verlo en 1000
0.10 (1 de 10)	0.003 %	≈ 0
0.01 (1 de 100)	36.6 %	0.004 %
0.005 (1 de 200)	60.6 %	0.7 %
0.001 (1 de 1000)	90.5 %	36.8 %

La mejor palanca es p

Subir `numRuns` ayuda y es barato en funciones puras. Pero mejorar el generador sube p, y eso vale más.

SECCIÓN 03

Clases de propiedades

03

¿Qué afirmar, si no se conoce el resultado?

Pregunta	Clase	En Kallpa
¿Qué es siempre cierto de la salida?	Invariante	El precio es un entero, de S/ 45 para arriba
¿Qué deshace esta operación?	Ida y vuelta	`addDays(d, n)` y `addDays(_, -n)`
¿Hay una forma lenta y obvia de calcularlo?	Modelo	Dos bloques: contar hora por hora
¿Cómo cambia la salida si cambio la entrada así?	Metamórfica	Una semana después, la misma tarifa
¿Hay otra versión que deba dar lo mismo?	Equivalencia	`formatSoles` antes y después

Todas salen del README, no del código: `code/week-10/properties.test.ts`.

Invariante e ida y vuelta

properties.test.ts

```ts
const dateArb = fc.integer({ min: 0, max: 364 }).map((d) => addDays('2026-01-01', d));

fc.property(sportArb, planArb, dateArb, fc.integer({ min: 7, max: 21 }),
  (sport, plan, date, startHour) => {
    const price = quotePriceCents({ sport, plan, date, startHour, hours: 1 });
    assert.ok(Number.isInteger(price));
    assert.ok(price >= 4500); // vóley sin hora punta, con descuento de socio
  });

fc.property(dateArb, fc.integer({ min: -400, max: 400 }), (date, n) => {
  assert.equal(addDays(addDays(date, n), -n), date);
});
```

Los invariantes son fáciles y débiles: una función que siempre devuelve 4500 pasa el primero. Aun así detectan un `NaN` o un total que no cuadra.

Comparación con un modelo

model.ts

```ts
function sharesAnHour(a, b) {
  const own = hoursOf(a);
  return hoursOf(b).some((h) => own.
      includes(h));
}
```

le falta un caso

```ts
function overlapsForgetful(a, b) {
  return a.startHour <= b.startHour &&
    b.startHour < a.startHour + a.hours;
}
```

make demo D=overlaps-model

```
✓ los dos ejemplos de test/booking.test.ts pasan (0.424916ms)
✗ coincide con el modelo: comparten alguna hora (3.263209ms)
...
  Counterexample: [{"startHour":19,"hours":1},{"startHour":18,"hours":2}]
...
    false !== true
```

Si copian la función, se compara consigo misma.

Relaciones metamórficas

Relación metamórfica

No compara una salida con una respuesta: dice cómo tiene que cambiar la salida cuando cambia la entrada.

properties.test.ts

```ts
// Agregar una reserva cancelada no cambia la caja.
fc.property(fc.array(bookingArb(), { maxLength: 30 }), bookingArb(), (bookings, extra) => {
  const before = dailySummary('2026-03-10', COURTS, bookings);
  const after = dailySummary('2026-03-10', COURTS, [...bookings, { ...extra, status: 'cancelled' }]);
  assert.deepEqual(after, before);
});
```

La prueba no sabe cuánto debe dar la caja, y aun así nota si alguien olvida filtrar las canceladas. Sirve donde no hay una respuesta fácil de calcular: agregar un filtro no da más resultados.

Equivalencia con la versión anterior

differential.ts

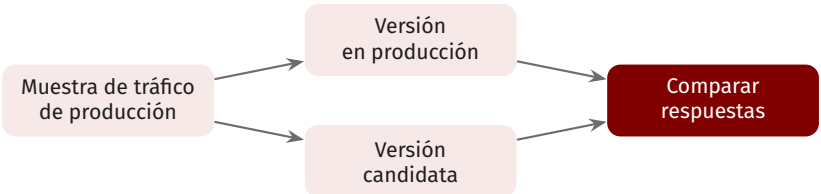
```ts
export function formatSolesNoFloat(cents: number): string {
  const soles = Math.trunc(cents / 100);
  const rest = String(cents % 100).padStart(2, '0');
  return `S/ ${soles}.${rest}`;
}
```

make demo D=differential: con enteros negativos

```
  Counterexample: [-1]
...
    + 'S/ 0.-1'
    - 'S/ -0.01'
```

- Con céntimos no negativos dan lo mismo. ¿Un monto negativo existe en el dominio? Alguien decide.
- Una comparación diferencial no dice qué está bien; dice **qué cambió**.

En la industria: pruebas A/B diff



- SWEG, cap. 14: posiblemente la forma más común de prueba grande en Google.
- Sin aserciones: una persona revisa si las diferencias son intencionales.
- Buscan **fidelidad**: qué tan bien refleja la prueba el comportamiento real. A cambio, son caras.
- Es Google a gran escala, no una receta para dos personas. Lo que se lleva: la versión anterior es un oráculo.

Una más: secuencias de operaciones

- Muchos errores solo aparecen después de cierta secuencia: reservar, cancelar y volver a reservar.
- **Model-based testing**: generar secuencias y, tras cada operación, comparar el sistema con un modelo en memoria.
- En fast-check: `fc.commands` y `fc.modelRun`.
- El shrinking reduce la secuencia: de treinta operaciones quedan las tres que importan.

Lectura complementaria y ejercicio 11 del capítulo. No se usa en el laboratorio.

SECCIÓN 04

Generadores

04

Construir en vez de filtrar

no: generar texto y descartar

```
1  fc.property(fc.string(), (text) => { fc.pre(isValidDate(text)); /* ... */ });
```

make demo D=too-many-discards

```
Error: Failed to run property, too many pre-condition failures encountered
{ seed: 2026 }

Ran 0 time(s)
Skipped 10001 time(s)
...
```

sí: construir la entrada válida

```
1  const dateArb = fc.integer({ min: 0, max: 364 }).map((d) => addDays('2026-01-01', d));
```

`fc.pre` sirve si descarta **pocas**: `startHour + hours <= 22`.

Mirar qué se genera: `fc.statistics`

```
make generadores

Fechas de un mes: tipo de día
lunes a viernes..79.60 %
fin de semana....20.40 %

Fechas de un mes: mes
2026-01..82.80 %
2026-02..17.20 %

Fechas de todo el año: tipo de día
lunes a viernes..72.50 %
fin de semana....27.50 %
```

El generador de un mes nunca pasa de febrero. Una regla que dependa del mes no se revisa.

Es la misma pregunta que en la cobertura, “¿por dónde pasó?”, pero sobre el **dominio**, no sobre el código.

Error común

El generador que nunca produce el caso que importa. Sale en verde.

Propiedades que no prueban nada

tautología

```
assert.equal(isPeakHour(date, hour), isWeekend(date) || hour >= PEAK_START_HOUR);
```

Pasa siempre: es la implementación otra vez. Si cambian la constante a las 12:00, sigue pasando.

desde el README: las fechas y los montos, no las constantes

```
const saturdayArb = fc.integer({ min: 0, max: 51 }).map((k) => addDays('2026-01-03', 7 * k));

fc.property(saturdayArb, fc.integer({ min: 7, max: 21 }), (date, hour) => {
  assert.equal(hourlyRateCents('fulbito', date, hour), 11250);
});
```

Control: ¿la pueden hacer fallar? Cambien una condición a mano y vean si lo nota.

Con IA: una propiedad no se pasa con casos especiales

split.ts: ilustración escrita para la clase

```ts
export function splitCentsFake(totalCents: number, players: number): number[] {
  if (totalCents === 9000 && players === 3) return [3000, 3000, 3000];
  if (totalCents === 11250 && players === 2) return [5625, 5625];
  if (totalCents === 100 && players === 3) return [34, 33, 33];
  return [totalCents];
}
```

make demo D=split-fake

```
  Counterexample: [0,2]
...
    1 !== 2
```

- Los tres ejemplos pasan. “Una parte por jugador” falla a la primera.
- Al revés: una propiedad escrita por una herramienta se lee como cualquier prueba.

SECCIÓN 05

Fuzzing

05



Fuzzing

Fuzzing

Alimentar un programa con muchas entradas generadas, a menudo malformadas, buscando que falle.

Oráculo implícito

Lo que vale para cualquier programa: no se cae, no se cuelga, no responde 500.

- Barton Miller, 1988: una tormenta metía ruido en su línea telefónica de 1200 baudios y los comandos de UNIX se caían.
- 1989: cerca de un tercio de las utilidades de UNIX probadas se cayeron, se colgaron o fallaron (Miller y otros, 1990).

Un fuzzer pequeño

parse-range.ts: "18-20" es de 18:00 a 20:00

```ts
export function parseHourRangeNaive(text: string): HourRange {
  const [start, end] = text.split('-');
  return { startHour: Number(start), hours: Number(end.trim()) - Number(start) };
}
```

make demo D=parse-range

```
Error: Property failed after 1 tests
{ seed: 2026, path: "0:0", endOnFailure: true }
Counterexample: [""]
Shrunk 1 time(s)
...
   [cause]: TypeError [Error]: Cannot read properties of undefined (reading 'trim')
```

Lo que no ve: con `"abc-def"` devuelve `NaN` sin caerse. Un programa que no se cae puede estar calculando cualquier cosa.

El generador decide hasta dónde se llega

```
make fuzz: POST /api/bookings

Valores de cualquier tipo: { '400 VALIDATION_ERROR': 1000 }
Tipos correctos, valores cerca de los bordes: {
  '422 INVALID_HOURS': 83,
  '200 reserva creada': 113,
  '404 COURT_NOT_FOUND': 273,
  '400 VALIDATION_ERROR': 116,
  '422 TOO_MANY_HOURS': 154,
  '422 IN_THE_PAST': 35,
  '422 OUTSIDE_OPENING_HOURS': 76,
  '404 MEMBER_NOT_FOUND': 99,
  '409 SLOT_TAKEN': 32,
  '422 TOO_FAR_AHEAD': 19
}
```

Ninguno encontró un 500. Pero con valores de cualquier tipo, las **1000** se quedaron en la primera validación.

> **Idea clave**
>
> Un fuzzer que no encuentra nada no demuestra que no haya nada. Miren hasta dónde llegaron las entradas.

Contra una base de datos de prueba, nunca la real.

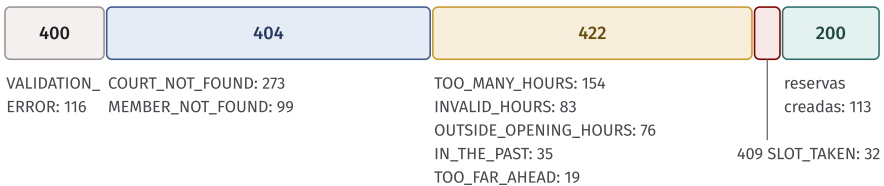
Hasta dónde llegó cada generador

Valores de cualquier tipo

400 VALIDATION_ERROR: las 1000

Nada pasa de la primera validación: las reglas, el precio y la base de datos ni corren.

Tipos correctos, valores cerca de los bordes



Cada barra son 1000 solicitudes. Ninguna respondió 500.

Generadores o guiado por cobertura

Basado en generadores

- Las entradas salen de una descripción de su forma.
- No mira qué hace el programa.
- Simple: una propiedad de fast-check sobre las rutas de una API.

Guiado por cobertura (AFL, libFuzzer)

- Instrumenta el programa.
- Guarda la entrada que llegó a código nuevo y la muta.
- Avanza solo por las capas. Corre horas o días.

En la industria: OSS-Fuzz (Google)

Según su página, hasta agosto de 2023 ayudó a corregir más de 10 000 vulnerabilidades y 36 000 errores en 1000 proyectos de código abierto. Es la cuenta que publica el propio proyecto: sirve como orden de magnitud.

Qué hacer con un hallazgo

1. **Reproducirlo fuera de la prueba**: con `curl`, contra el servidor.
2. **Leer el log**: el 500 dice que algo falló; la pila de llamadas dice dónde.
3. **Decidir cuál es la respuesta correcta**: “no 500” no lo dice. Vayan a la tabla de errores y justifiquen.
4. **Convertir la entrada en una prueba de ejemplo**, con la respuesta esperada.
5. **Corregir** y volver a correr todo.

SECCIÓN 06

Pruebas de mutación

06

La pregunta que la cobertura no responde

mutation/voucher.weak.test.ts

```ts
test('calcula el monto a pagar', () => {
  assert.equal(typeof amountDueCents(9000, 1000), 'number');
  assert.equal(typeof amountDueCents(1000, 9000), 'number');
  assert.equal(typeof applyVouchers(9000, [1000, 500]), 'number');
});
```

node –test –experimental-test-coverage voucher.weak.test.ts

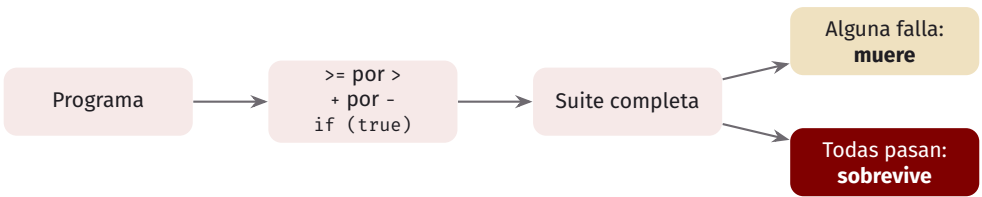
```
i file       | line % | branch % | funcs % | uncovered lines
i -------------------------------------------------------------
i voucher.ts | 100.00 |   100.00 |  100.00 |
```

La cobertura dice por dónde pasó la ejecución. No puede decir si alguien miró el resultado.

Qué es un mutante

Mutante

Una copia del programa con un cambio pequeño. Se corre la suite contra cada uno.



Un sobreviviente es una frase concreta: este cambio de comportamiento no lo fija ninguna prueba.

Operadores de Stryker

Operador	Original	Mutado
Aritmético	`a + b`, `a * b`	`a - b`, `a / b`
Igualdad	`a < b`	`a <= b`, `a >= b`
Igualdad	`a >= b`	`a > b`, `a < b`
Igualdad	`a === b`	`a !== b`
Lógico	`a && b`	`a || b`
Condicional	`if (a > b)`	`if (true)`, `if (false)`
Bloque	`{ ...código }`	`{}`
Texto	`'hola'`	`""`
Incremento	`i++`	`i--`

Lista completa: *Supported mutators*, en la documentación de StrykerJS.

Configurar Stryker

mutation/stryker.weak.json

```
{
  "testRunner": "tap",
  "tap": { "testFiles": ["voucher.weak.test.ts"], "nodeArgs": ["--test-reporter=tap"]
      },
  "mutate": ["voucher.ts"],
  "reporters": ["clear-text", "json"],
  "jsonReporter": { "fileName": "reports/weak.json" },
  "plugins": ["@stryker-mutator/tap-runner"]
}
```

- `tap`: el runner de Node escribe sus resultados en TAP, y Stryker los lee.
- `mutate`: solo código de producción, nunca las pruebas.
- Copia el proyecto a `.stryker-tmp`, hace una corrida inicial con todas las pruebas y, si alguna falla, no arranca.

100 % de cobertura, 25 % de mutation score

make mutantes S=weak (2 de los 9 sobrevivientes)

```
[Survived] ArithmeticOperator
voucher.ts:5:10
-     return totalCents - voucherCents;
+     return totalCents + voucherCents;
...
[Survived] BlockStatement
voucher.ts:11:45
-     for (let i = 0; i < vouchers.length; i++) {
-       due = amountDueCents(due, vouchers[i]);
-     }
+     for (let i = 0; i < vouchers.length; i++) {}
```

la tabla final

```
File        |  total | covered | # killed | # timeout | # survived | # no cov | # errors |
------------|--------|---------|----------|-----------|------------|----------|----------|
All files   |  25.00 |   25.00 |        2 |         1 |          9 |        0 |        0 |
```

Estados y score

Killed	alguna prueba falló
Timeout	se colgó (`i--`): cuenta como detectado
Survived	las pruebas pasaron por ahí, y todas pasaron
No coverage	ninguna prueba ejecuta esa línea
Ignored	marcado: fuera del cálculo

Mutation score

detectados = killed + timeout

total = detectados / (detectados + survived + no coverage)

covered = detectados / (detectados + survived)

`total` mucho menor que `covered`: falta ejecutar código. `covered` también bajo: faltan aserciones.

Con valores esperados: queda uno

```
make mutantes S=strong

[Survived] EqualityOperator
voucher.ts:4:7
-     if (voucherCents >= totalCents) return 0;
+     if (voucherCents > totalCents) return 0;
...
All files   |  91.67 |   91.67 |       10 |         1 |          1 |        0 |        0 |
```

1. Solo difieren si `voucherCents === totalCents`.
2. Original: devuelve 0. Mutante: `totalCents - voucherCents`, también 0.

Mutante equivalente

Con céntimos enteros, no cambia nada observable. Dentro del dominio no se puede matar, y no indica un hueco.

El mismo código, dos suites

MUTANTE: ORIGINAL Y CAMBIO	prueba débil	valores esperados
`- if (voucherCents >= totalCents)` `+ if (true)`	sobrevive	muere
`- if (voucherCents >= totalCents)` `+ if (voucherCents > totalCents)`	sobrevive	sobrevive
`- return totalCents - voucherCents;` `+ return totalCents + voucherCents;`	sobrevive	muere
`- { due = amountDueCents(due, vouchers[i]); }` `+ {}`	sobrevive	muere
`- i++` `+ i--`	timeout	timeout

LOS 12 MUTANTES DE CADA CORRIDA



Con la prueba débil, la cobertura es 100 %.

Qué hacer con cada sobreviviente

1. Buscar una entrada donde mutante y original den resultados distintos. Casi siempre está en el borde exacto.
2. Si existe: es un **hueco**. Escribir la prueba. El valor esperado sale de la especificación.
3. Si no: escribir el argumento. Decidir la equivalencia es indecidible en general.
4. Marcar y seguir:

mutation/voucher-marked.ts

```ts
// Stryker disable next-line EqualityOperator: con voucher === total, el resto ya da 0.
if (voucherCents >= totalCents) return 0;
```

Ojo: ese comentario apaga también `voucherCents < totalCents`, que no era equivalente. Revisen qué quedó apagado.

Error común: matar sin mirar la especificación

Matar un mutante con la aserción que el código hoy satisface

Sube el score y congela lo que hay, incluidos sus errores.

- Si no pueden decir de qué línea del README sale el valor esperado, primero averigüen cuál es la regla.
- Tampoco todo sobreviviente merece prueba: un mensaje de error cambiado por `""` sobrevive si nadie mira los mensajes. Si el texto no es parte del contrato, no lo fijen.

Por qué funciona

Hipótesis del programador competente

Los errores reales suelen estar a un cambio pequeño del programa correcto.

Efecto de acoplamiento

Las pruebas que detectan errores simples suelen detectar también los compuestos.

La evidencia (Just y otros, FSE 2014): 357 errores reales en 5 aplicaciones Java.

- 73 % acoplados a mutantes comunes; 10 % requerirían operadores nuevos; 17 % a ninguno.
- Detectar mutantes se correlaciona con detectar errores reales, más que la cobertura de sentencias.
- Es una correlación en cinco programas, no una ley. No dice qué score alcanza.

Costo

- Hay que correr la suite muchas veces. Stryker corre, para cada mutante, solo las pruebas que pasan por esa línea (`perTest`).
- Con el runner `tap`, **una prueba es un archivo**, y cada archivo es un proceso.
- Por eso el fuzzing no va en `tap.testFiles`: haría cientos de solicitudes HTTP por cada mutante.
- Un módulo de dominio toma unos segundos; un monolito grande, horas.

> **Para su equipo**
>
> Sobre los módulos con reglas de negocio. Lean los sobrevivientes de lo que cambiaron y decidan caso por caso.

En la industria: Google (Petrović e Ivanković, 2018)

- Sin score global: inviable por costo, y no encontraron cómo mostrarlo para que llevara a actuar.
- Se muestra **en la revisión de código**: solo las líneas del cambio, a lo más un mutante por línea, sin las líneas “áridas” (por ejemplo, logs).
- Un clic marca un hallazgo como no útil, y con eso afinaron los filtros.

Lo que reportan: 6000 ingenieros, más de 70 000 diffs, 1.1 millones de mutantes, 150 000 hallazgos; la utilidad reportada subió de 20 % a 80 %.

Los límites: lo escribe quien construyó el sistema; no mide si llegan menos errores a producción.

Lo que se lleva

Un comentario concreto sobre un cambio, no un número para la gerencia.

SECCIÓN 07

El problema del oráculo

07

Quién sabe la respuesta

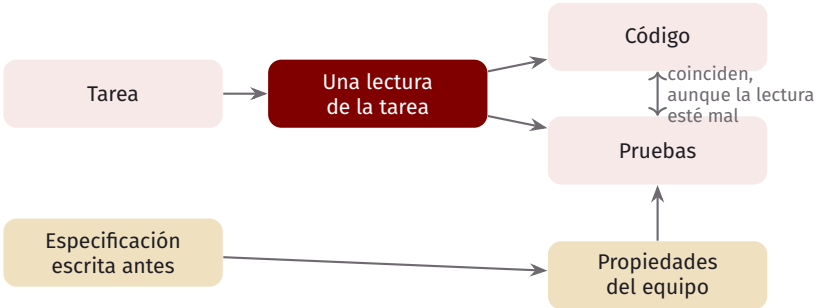
Oráculo ***(oracle)***

Un mecanismo que decide si la salida de una prueba es la correcta. Conseguir uno es el **problema del oráculo**; Weyuker (1982) llamó *non-testable* a los programas sin uno practicable.

Fuente	Fuerza	Límite
Ejemplos de la especificación	Exacto, lo acuerda el cliente	Pocos casos
Propiedades e invariantes	Todo el dominio	Solo parte del resultado
Modelo	Respuesta completa	Si existe uno simple
Otra versión (diferencial)	No hay que entender la regla	No dice cuál está bien
Relaciones metamórficas	Sin saber la respuesta	Muchos errados la cumplen
Implícito (“no 500”)	Gratis y universal	Solo fallas abruptas

La mutación no es un oráculo: mide cuánto revisan los que ya tienen.

Código y pruebas de la misma lectura



Si un agente escribe el código y sus pruebas, las pruebas comparten sus malentendidos: **no son un oráculo independiente**.

Un caso escrito para la clase

Un cambio como los que suele proponer un agente ante “valida la anticipación máxima de 14 días”.

mutation/booking-window.ts

```ts
return daysAhead >= 0 && daysAhead < WINDOW_DAYS;
```

mutation/booking-window.test.ts

```ts
assert.equal(isWithinBookingWindow(0), true);    assert.equal(isWithinBookingWindow(-1), false);
assert.equal(isWithinBookingWindow(13), true);   assert.equal(isWithinBookingWindow(14), false);
```

make mutantes S=window

```
File              |  total | covered | # killed | # timeout | # survived | # no cov | # errors |
------------------|--------|---------|----------|-----------|------------|----------|----------|
All files         | 100.00 |  100.00 |       10 |         0 |          0 |        0 |        0 |
```

100 % de cobertura, 100 % de mutation score, y está mal

El README: no se puede reservar “con más de 14 días de anticipación”. A 14 días exactos, sí.

demos/window-readme.demo.test.ts

```
1  test('a 14 días se puede reservar; a 15, no', () => {
2    assert.equal(isWithinBookingWindow(14), true);
3    assert.equal(isWithinBookingWindow(15), false);
4  });
```

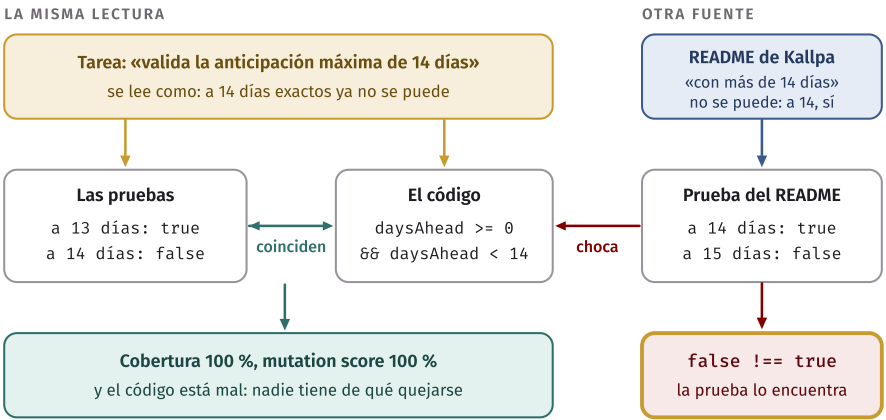
make demo D=window-readme

```
✗ a 14 días se puede reservar; a 15, no (0.804334ms)
  AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:

  false !== true
```

Cobertura: por dónde se pasó. Mutación: si las pruebas notan cambios. El malentendido solo lo ve algo que **no salió de la misma lectura**.

Lo encuentra otra fuente



Qué es independiente

Sí

- La especificación escrita **antes**.
- Las propiedades que escribe el equipo.
- La versión anterior, si dice “no cambia el comportamiento”.
- La mutación, para medir si sus pruebas verifican algo. Score bajo: no. Score alto: no dice que se haya entendido bien.

No

- Las pruebas que escribió junto con el código.
- “Corrí las pruebas y todas pasan.”
- Una especificación generada a partir del código.
- Una segunda pasada que no mira la especificación.

Error común

Perseguir 100 %, o leerlo como “el código es correcto”. No dice nada de un requisito mal entendido.

Con IA: antes de aceptar un cambio de un agente

1. Leer el diff. ¿Tocó pruebas que no debía?
2. Correr la suite de ustedes, no solo las pruebas nuevas.
3. Buscar en la especificación los bordes de la regla que tocó, y escribir ustedes esos casos.
4. Si dice que no cambia el comportamiento, compararlo con la versión anterior.
5. Correr Stryker y leer los sobrevivientes de las líneas que cambiaron.

Comprobar es parte del trabajo

“Corrí las pruebas y pasan” solo dice que las pruebas que existían no lo vieron.

SECCIÓN 08

El laboratorio

08

Laboratorio: pasos 1 a 4

paso 1: una copia limpia

```
cp -r /work/kallpa /work/kallpa-semana10 && cd /work/kallpa-semana10
git init -q && git add -A && git commit -qm "Kallpa publicada" && git tag base
```

2. `test/pricing.property.test.ts`: dos reglas del README. “Cada hora se cobra con la tarifa que le corresponde” y “los socios tienen 10 % de descuento”.
3. `test/calendar.property.test.ts`: `daysBetween(d, addDays(d, n)) === n`.
4. `test/api.fuzz.test.ts`: 300 textos en `/api/courts/<texto>/availability`. Solo se afirma: **no es 500**.

Si algo falla

Contraejemplo, README, prueba de ejemplo, y recién corregir. Si pasa, córranla varias veces y midan qué se generó.

Laboratorio: pasos 5 a 7

5. `stryker run`, con sus archivos de los pasos 2 y 3 en `tap.testFiles` (el de fuzzing no). Para todos los sobrevivientes de `pricing.ts` y al menos tres de `booking.ts`: la prueba que lo mata, con el valor sacado del README, o el argumento de por qué es equivalente.
6. Tres diffs en `code/week-10/agente-*.diff`: ilustraciones escritas para la clase, con su tarea y su descripción. Una rama por cada uno: leer, aplicar, correr. Cada uno tiene al menos un error, y las 40 pruebas publicadas pasan con los tres.
7. Pull request con propiedades, fuzzing, pruebas que matan mutantes y la configuración, y una tabla: error, técnica que lo encontró, prueba que lo evita ahora.

Lo más importante

Cuando sus pruebas no encuentren un error, pregúntense qué clase de oráculo le falta a la suite, no solo qué caso.

Para la próxima clase

- Terminar el laboratorio: el pull request con la tabla y el registro de uso de IA (¿qué habrían aceptado sin estas pruebas?).
- **P3** se entrega la semana 11: al menos dos propiedades con fast-check sobre sus reglas de negocio, y, si corren Stryker, comentar el resultado en su estrategia de pruebas.
- Semana 11, revisión de código y análisis estático. Lecturas: SWEG, cap. 9; Google Engineering Practices, *How to do a code review* y *The Standard of Code Review*.

Pregunta

En el producto de su cliente, ¿qué regla de negocio dejaría más dinero o más confianza en juego si estuviera mal? ¿Qué propiedad la vigilaría, y de qué parte de la especificación sale?