

Ingeniería de Software

Especificaciones y design docs

Contratos, tipos y especificaciones para personas y agentes

Gerson Garrido

Semana 4

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A

1. Qué es una especificación
2. Contratos: precondición, postcondición, invariante
3. Quién revisa: parcial o total
4. Postcondiciones completas
5. Los tipos como especificación

Sesión B

6. Design docs
7. La documentación como código
8. Desarrollo guiado por especificaciones
9. Laboratorio: de la especificación al código

Lecturas

A: Meyer, *Applying "Design by Contract"* (1992); King, *Parse, don't validate* (2019); TypeScript Handbook, *Narrowing*.
B: Ubl, *Design Docs at Google* (2020); SWEG, cap. 10 *Documentation*; Böckeler, *Understanding Spec-Driven-Development* (2025). Complementarias: Handbook, *Everyday Types*; Stanford CS146S, *The Modern Software Developer*.

SECCIÓN 01

Qué es una especificación

01



Qué, no cómo

Especificación *(specification)*

Dice **qué** hace un módulo sin decir **cómo** lo hace.

- *"Devuelve `true` si los dos bloques comparten al menos una hora".*
- Muchas implementaciones la cumplen: comparar bordes, armar conjuntos, recorrer horas.
- Quien llama no necesita saber cuál se eligió.

src/domain/booking.ts

```ts
// Una de muchas implementaciones posibles.
export function overlaps(a, b): boolean {
  return a.startHour < b.startHour + b.hours
      && b.startHour < a.startHour + a.hours;
}
```

Un acuerdo entre dos lados



Sin especificación, el acuerdo es el código

Ley de Hyrum (semana 1): con suficientes usuarios, todo comportamiento observable termina siendo usado por alguien.

Requisito, especificación, prueba

	Habla de	Ejemplo en Kallpa
Requisito	El sistema visto desde afuera	Se cancela hasta 3 horas antes de que empiece
Especificación	Un módulo: qué recibe, qué garantiza	`isCancellableAt(slot, now)` es `true` si y solo si faltan al menos 180 minutos
Prueba	Un caso concreto	Lunes 10:00: la reserva del lunes a las 14:00 se puede cancelar

Las pruebas muestran casos; la especificación dice la regla.

SECCIÓN 02

Contratos

02

Diseño por contrato (Meyer, 1992)

	Obligación	Beneficio
Cliente	Paquete chico, pagar	Llega en 4 horas o menos
Empresa	Entregar en 4 horas	Nada de paquetes enormes ni impagos

Lo que es obligación para uno es beneficio para el otro.

Tres aserciones

Precondición: lo que **quien llama** garantiza antes de la llamada.

Postcondición: lo que **la rutina** garantiza al terminar, si se cumplió la precondición.

Invariante: lo que vale siempre entre operaciones.

El contrato de `overlaps`

	Obligación	Beneficio
Quien llama	Pasar dos bloques de la misma cancha y el mismo día	Saber si se cruzan
`overlaps`	`true` si y solo si comparten al menos una hora	No mirar fechas ni canchas

Comentario o contrato

"Cada uno empieza antes de que el otro termine" explica *cómo*. El contrato dice *qué promete*.

La postcondición se puede comprobar

overlaps.test.ts

```ts
// El oráculo, escrito de la forma más tonta posible.
function sharesAnHour(a: Block, b: Block): boolean {
  for (let h = a.startHour; h < a.startHour + a.hours;
       h++) {
    if (h >= b.startHour && h < b.startHour + b.hours)
      return true;
  }
  return false;
}

for (const a of blocks)       // los 42 bloques de un
     dia
  for (const b of blocks)
    assert.equal(overlaps(a, b), sharesAnHour(a, b));
```

- 15 bloques de 1 hora, 14 de 2, 13 de 3.
- El oráculo está escrito de otra forma: una copia de `overlaps` pasaría siempre.
- Semana 10: pruebas basadas en propiedades, cuando los casos no caben.

Los 1764 pares

```
node --test overlaps.test.ts

✓ overlaps cumple su postcondición en los 1764 pares de bloques posibles (1.95075ms)
✓ fuera de la precondición, la respuesta no significa nada (0.242417ms)
ℹ tests 2
ℹ suites 0
ℹ pass 2
ℹ fail 0
```

Una postcondición bien escrita se traduce en una prueba

"`true` si y solo si comparten al menos una hora": para cada par, comparar con la respuesta calculada de la forma más simple.

Fuera de la precondición, no hay promesa

overlaps.test.ts

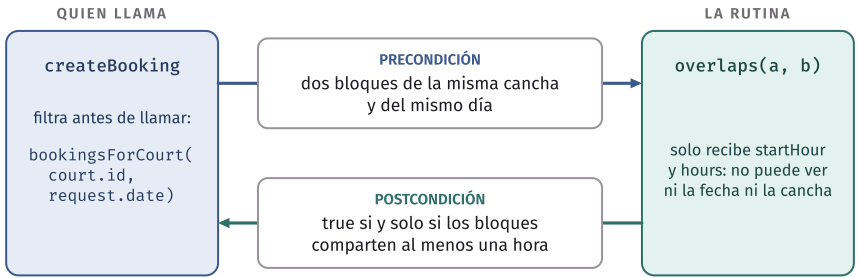
```
1  // Martes de 18:00 a 20:00 y miércoles de 19:00
2  // a 20:00: no comparten nada, pero overlaps no
3  // recibe la fecha y responde true.
4  const tuesday =
5    { date: '2026-03-10', startHour: 18, hours: 2
       };
6  const wednesday =
7    { date: '2026-03-11', startHour: 19, hours: 1
       };
8  assert.equal(overlaps(tuesday, wednesday), true)
       ;
```

- Si la precondición no se cumple, la rutina no está obligada a nada.
- No tiene que lanzar un error: el contrato no dice nada para ese caso.

¿De quién es el bug?

Precondición rota: de **quien llama**.
Postcondición rota: de **la rutina**.

Quién responde por qué



Si se rompe la precondición, el bug es de quien llama.

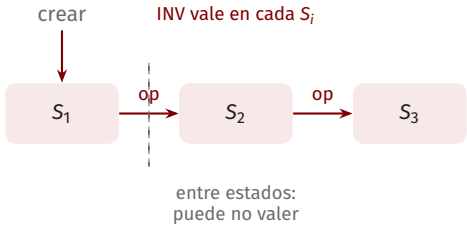
Si se rompe la postcondición, el bug es de la rutina.

FUERA DE LA PRECONDICIÓN

`overlaps(martes de 18:00 a 20:00, miércoles de 19:00 a 20:00)` **true: no significa nada**

Invariante de clase (Meyer)

- Vale para todos los objetos en sus **estados observables**.
- Se cumple al crear el objeto.
- Cada operación pública que lo encuentra cumplido lo deja cumplido.
- A mitad de una operación puede romperse un momento.



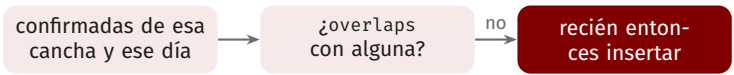
En la práctica

El invariante se agrega a la precondición y a la postcondición de cada operación pública.

El invariante del sistema en Kallpa

Extensión de la idea de Meyer a un sistema

Una cancha no tiene dos reservas **confirmadas** que se crucen (mismo `courtId`, misma `date`, bloques que comparten una hora).



`createBooking`, en `src/http/handlers.ts`

- Crear puede romperlo: revisa. Cancelar no puede: solo quita una confirmada.
- ¿Quién más lo protege? Nadie. La tabla no tiene ninguna restricción.

La base de datos acepta una fila que lo rompe

invariant.test.ts

```ts
// Real Barranco tiene la cancha 1 hoy de 19 a 21; esta fila no pasa por el handler.
repo.insert({ court_id: 1, member_id: 3, date: TODAY, start_hour: 20, hours: 1, /* ... */ });
const pairs = overlappingPairs(allBookings(repo, addDays(TODAY, -3), 10));
assert.equal(pairs.length, 1);
```

node --test invariant.test.ts

```
✓ los datos de ejemplo cumplen el invariante (2.083917ms)
✓ la base de datos acepta una fila que lo rompe: nadie más lo revisa (0.950916ms)
ℹ tests 2
ℹ suites 0
ℹ pass 2
ℹ fail 0
```

`overlappingPairs` no hace cumplir el invariante: dice si se cumple. Semana 7: dos instancias sobre el mismo archivo.

SECCIÓN 03

Quién revisa

03

Precondición fuerte o débil

Fuerte: la rutina exige

No revisa; quien llama garantiza. La rutina es **parcial**. Meyer: estilo *demanding*.

Ejemplo: `overlaps`.

Débil: la rutina revisa

Responde algo definido si no se cumple. Si vale para toda entrada, es **total**. Meyer: *tolerant*.

Ejemplo: `checkBookingRules` devuelve una violación o `null`.

Las dos son válidas

Criterio de Meyer: la arquitectura más simple. Lo obligatorio es que la especificación **diga cuál se eligió**.

`formatSoles` fuera de lo que se pensó

src/domain/pricing.ts

```ts
// 11250 -> 'S/ 112.50'
export function formatSoles(cents: number): string {
  return `S/ ${(cents / 100).toFixed(2)}`;
}
```

node

```
11250 S/ 112.50
-50 S/ -0.50
12.5 S/ 0.13
0.5 S/ 0.01
NaN S/ NaN
```

- El comentario da un ejemplo, no un contrato.
- `S/ 0.13` parece un precio razonable.
- Es el síntoma de un bug en otro lado: alguien hizo una cuenta con decimales.
- ¿Le toca a `formatSoles` detectarlo? **Depende del contrato.**

Tres contratos para lo mismo

1. Parcial

```
// Exige un entero >= 0.
// Si no, no promete nada.
return formatSoles(cents);
```

2. Revisa y lanza

```
if (!Number.isSafeInteger(cents) || cents < 0) {
  throw new RangeError(...);
}
return formatSoles(cents);
```

3. Total, error en el tipo

```
type FormatResult =
  | { ok: true; text: string }
  | { ok: false; code: 'NOT_AN_INTEGER' | 'NEGATIVE' };
```

- La excepción es parte del contrato, pero el tipo `string` no la muestra.
- El resultado con `ok` obliga a mirarlo antes de usar el texto.
- Para montos que salen de la base de datos, la parcial con el contrato escrito es razonable.

Meyer: nunca en los dos

- Programación defensiva: revisar todo en todas partes, “por si acaso”.
- Meyer: más código, más complejidad, menos confiabilidad.
- Una condición va en la precondición **o** en un `if` del cuerpo, nunca en los dos.
- Con un contrato preciso, cada condición tiene **un** responsable.

Dos aclaraciones del artículo

Garantizar no es hacer un `if` antes de cada llamada: a veces ya lo garantiza el contexto.

Si todos los que llaman hacen el mismo tratamiento especial, el contrato es demasiado exigente: cámbienlo.

Kallpa revisa `hours` dos veces

src/http/handlers.ts: parseBookingRequest

```
if (typeof value !== 'number' || !Number.isInteger(value)) {
  throw new HttpError(400, 'VALIDATION_ERROR', `El campo ${name} debe ser un número entero.`);
}
```

src/domain/booking.ts: checkBookingRules

```
if (!Number.isInteger(slot.hours) || slot.hours < 1) {
  return { code: 'INVALID_HOURS', message: 'La cantidad de horas debe ser un entero mayor que cero.' };
}
```

- ¿Redundancia? No necesariamente: **forma** del cuerpo HTTP frente a función de dominio **total**.
- Lo que falta: que esos contratos estén escritos. Si no, alguien borra uno “porque ya se revisa en el otro lado”.

Error común: tres capas o ninguna

“por si acaso”		“lo revisa otro”	
la página	**revisa**	la página	—
el handler	**revisa**	el handler	—
el dominio	**revisa**	el dominio	—

Las dos pasan por lo mismo

Nadie escribió quién es responsable de cada condición. Decidan, y escríbanlo en el contrato.

SECCIÓN 04

Postcondiciones completas

04

“Devuelve las horas libres”

free-hours.ts

```ts
// Cumple "devuelve horas libres":
// todo lo que devuelve está libre.
export function freeHoursNone(_b): number[] {
  return [];
}

// También: la primera hora libre está libre.
export function freeHoursFirstOnly(b): number
    [] {
  return freeHours(b).slice(0, 1);
}
```

- La frase solo dice que lo devuelto es **correcto**.
- No dice que esté todo.

Postcondición completa

Devuelve **todas** las horas sin reserva confirmada, **y solo esas**, **sin repetir** y **de menor a mayor**.

Lo mismo pasa en las pruebas

```
node --test free-hours.test.ts

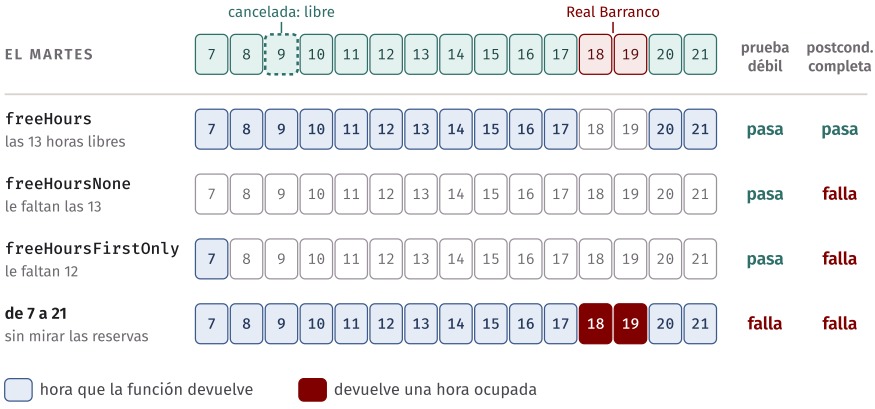
➤ la prueba débil: "todo lo que devuelve está libre"
  ✓ freeHours la pasa (0.47625ms)
  ✓ freeHoursNone la pasa (0.089959ms)
  ✓ freeHoursFirstOnly la pasa (0.083667ms)
✓ la prueba débil: "todo lo que devuelve está libre" (2.269708ms)
➤ la postcondición completa
  ✓ freeHours la cumple (0.636375ms)
  ✓ freeHoursNone no: le faltan las 13 horas libres (0.117292ms)
  ✓ freeHoursFirstOnly no: le faltan 12 (0.079291ms)
  ✓ el revisor también ve horas repetidas o desordenadas (0.081167ms)
✓ la postcondición completa (1.139083ms)
ℹ tests 7
ℹ suites 2
ℹ pass 7
ℹ fail 0
```

- Martes: Real Barranco de 18 a 20; una cancelada a las 9.
- Las tres pasan la prueba débil.
- La completa compara la lista entera, calculada de otra forma.

Error común

Probar solo que cada hora devuelta está libre: pasa con `[]`.

Cuatro implementaciones, dos pruebas



Seis preguntas para una postcondición

1. ¿Todo lo que devuelve cumple? — corrección
2. ¿Devuelve todo lo que cumple? — completitud
3. ¿En qué orden? ¿Con repetidos?
4. ¿Qué pasa en los bordes? — primera y última hora, día vacío, día lleno
5. ¿Qué no cambia? — ¿ordena la lista que le pasaron?
6. ¿Una implementación tonta la cumple? — `return []`, `return null`

> **La más útil es la sexta**
>
> Imaginen la implementación más perezosa posible. Si cumple lo que escribieron, falta una frase.

SECCIÓN 05

Los tipos como especificación

05



Una especificación que se revisa sola

- `(cents: number): string` es una especificación parcial.
- `tsc` la revisa en **cada llamada**, en cada compilación (`make typecheck`).
- Gratis, y sin escribir pruebas.

Los tipos se borran

Node quita los tipos antes de ejecutar: en tiempo de ejecución no queda nada.

Solo saben lo que se les dice

Si un `unknown` recibe un tipo sin revisarlo, `tsc` lo cree.

Uniones de literales en lugar de `string`

src/domain/booking.ts

```ts
export const BOOKING_STATUSES = ['confirmed', 'cancelled'] as const;

export type BookingStatus = (typeof BOOKING_STATUSES)[number];  // 'confirmed' | 'cancelled'
```

- `as const`: de solo lectura, con tipos literales. Una sola fuente: lista en tiempo de ejecución y tipo para `tsc`.
- Un error de ortografía ya no compila:

tsc -p bad

```
bad/typo-status.ts(5,10): error TS2367: This comparison appears to be unintentional because the types '"confirmed" | "cancelled"' and '"canceled"' have no overlap.
```

Por qué no `enum`

- Un `enum` genera código (un objeto con claves y valores); Node solo quita los tipos.
- En su lugar: `as const` + unión de literales. Lo mismo vale para `namespace` y los parámetros de constructor con `private`.

```
tsc -p bad
```

```
bad/enum.ts(3,6): error TS1294: This syntax is not allowed when 'erasableSyntaxOnly' is enabled.
```

```
node bad/enum.ts
```

```
file:///work/code/week-04/bad/enum.ts:3
    // genera código, y Node solo quita los tipos.
  > enum Status {
      Confirmed,
      Cancelled,
  > }

SyntaxError [ERR_UNSUPPORTED_TYPESCRIPT_SYNTAX]: TypeScript enum is not supported in strip-only mode
```

Narrowing: `tsc` aprende en cada rama

Forma	Ejemplo	Dentro de la rama
`typeof`	`typeof value === 'string'`	`string`
Veracidad	`if (booking)` con `Booking | undefined`	`Booking`
Igualdad	`value === 1 || value === 2`	`1 | 2`
`in`	`'booking' in result`	las variantes con `booking`
`instanceof`	`error instanceof HttpError`	`HttpError`
Predicado de tipo	`isValidDate(value)`, `value is string`	`string`

Es análisis de flujo: después de `if (typeof x !== 'number') throw ...`, `x` es `number` en el resto de la función.

Uniones discriminadas

booking-result.ts

```ts
export type BookResult =
  | { ok: true; booking: Booking }
  | { ok: false; code: 'SLOT_TAKEN' }
  | { ok: false; code: 'OUTSIDE_OPENING_HOURS'; open: number; close: number };
```

- El **discriminante** (`ok`, `code`) dice cuál variante es. Cada una lleva solo sus datos.
- `tsc` no deja leer `booking` sin mirar `ok`:

tsc -p bad

```
bad/no-narrowing.ts(5,17): error TS2339: Property 'booking' does not exist on type 'BookResult'.
  Property 'booking' does not exist on type '{ ok: false; code: "SLOT_TAKEN"; }'.
```

Con una excepción, el tipo sería `Booking` y nada obligaría a pensar en el error.

`switch` exhaustivo con `never`

booking-result.ts

```ts
export function assertNever(value: never): never {
  throw new Error(`Caso no previsto: ${JSON.stringify(value)}`);
}

export function httpStatus(result: BookResult): number {
  if (result.ok) return 200;
  switch (result.code) {
    case 'SLOT_TAKEN': return 409;
    case 'OUTSIDE_OPENING_HOURS': return 422;
    default: return assertNever(result);  // aquí result: never
  }
}
```

- `never`: el tipo sin valores.
- Descartadas todas las variantes, en el `default` queda `never`.
- Si llega un valor igual (por un `as`), `assertNever` lanza.

Agregar un caso y olvidarse de manejarlo

bad/missing-case.ts

```
    | { ok: false; code: 'OUTSIDE_OPENING_HOURS'; open: number; close: number }
    | { ok: false; code: 'TOO_FAR_AHEAD'; maxDays: number };   // nuevo
```

tsc -p bad

```
bad/missing-case.ts(20,26): error TS2345: Argument of type '{ ok: false; code: "TOO_FAR_AHEAD"; maxDays: number; }'
      is not assignable to parameter of type 'never'.
```

El error aparece donde falta el cambio

Con diez `switch` sobre el mismo resultado: una lista de errores de compilación, no una búsqueda ni un bug en producción.

Parse, don't validate (King, 2019)

Validar

Responde sí o no y **se olvida** de lo que comprobó. El valor sigue con el tipo de antes.

Parsear

Convierte un tipo poco preciso en uno **más preciso**, o falla. Lo comprobado queda en el tipo.

- Empujen la carga de la prueba lo más arriba posible: a la **frontera** del sistema.
- Una sola vez, antes de actuar sobre los datos.
- Una sola fuente de verdad: si un dato está en dos lugares, "no coinciden" es un estado posible.

Validar frente a parsear, en código

hours.ts

```ts
type BookingHours = 1 | 2 | 3;
function isValidHours(value: unknown): boolean {
  return typeof value === 'number' && Number.
      isInteger(value)
    && value >= 1 && value <=
      MAX_HOURS_PER_BOOKING;
}
function parseHours(value: unknown): BookingHours
    {
  if (value === 1 || value === 2 || value === 3)
      return value;
  throw new RangeError(`... llegó ${JSON.stringify
      (value)}.`);
}
function endHour(startHour: number, hours:
    BookingHours) { ... }
```

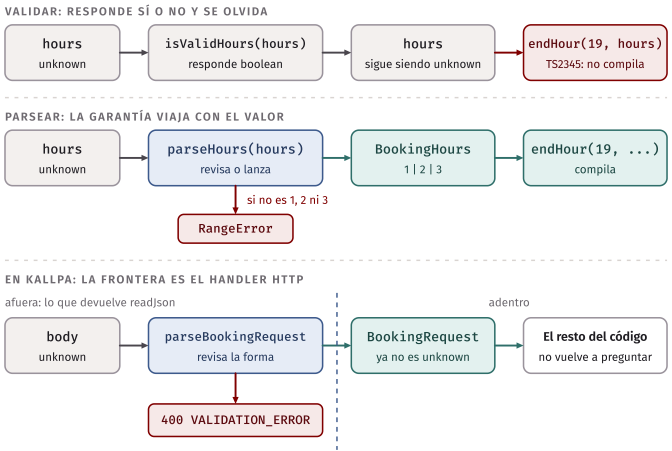
bad/hours.ts

```ts
if (isValidHours(hours)) {
  endHour(19, hours);      // lí
      nea 8
}
endHour(19, parseHours(hours));
```

tsc -p bad

```
bad/hours.ts(8,15): error TS2345: Argument
      of type 'unknown' is not assignable
      to parameter of type 'BookingHours
      '.
```

La garantía viaja con el valor



En Kallpa: la frontera es el handler

src/http/handlers.ts

```ts
// Revisa la forma del cuerpo; las reglas del
// negocio se revisan después.
function parseBookingRequest(body: unknown):
    BookingRequest {
  if (typeof body !== 'object' || body === null
      || Array.isArray(body)) {
    throw new HttpError(400, 'VALIDATION_ERROR',
      ...);
  }
  const fields = body as Record<string, unknown>;
  ...
```

- `unknown` entra; `BookingRequest` sale. El resto no vuelve a revisar la forma.
- Es un parse parcial: `hours: number`, `date: string`.
- Las reglas las revisa `checkBookingRules`, que devuelve `null`, no un tipo más preciso.
- Nada impide usar un `BookingRequest` que no pasó por las reglas.

`as` no revisa nada

types.test.ts

```ts
test('`as` no revisa nada: 7 as BookingHours pasa y da una hora de fin imposible', () => {
  assert.equal(endHour(19, 7 as BookingHours), 26);
});
```

src/http/handlers.ts: toBookingJson

```ts
    status: row.status as BookingStatus,   // la fila de SQLite trae status: string
```

- Una **afirmación que nadie comprueba**. Si alguien guarda `'canceled'`, entra al tipo sin que nadie lo note.
- Un `as` solo se justifica junto a la revisión que lo respalda (o dentro de la función que revisa).

Error común: `JSON.parse(body) as Booking`. El error de compilación desaparece; el problema, no.

Tipos marcados

cents.ts

```ts
export type Cents = number & { readonly __brand: 'Cents' };
export type Soles = number & { readonly __brand: 'Soles' };

// El único `as Cents`: después de revisar.
export function toCents(value: number): Cents {
  if (!Number.isSafeInteger(value)) {
    throw new RangeError(`... llegó ${value}.`);
  }
  return value as Cents;
}
```

- Para `tsc`, `Cents` es un `number` con una marca que ningún `number` común tiene.
- Un `number` común no pasa como `Cents`.
- Un `Soles` tiene otra marca: tampoco.

Mezclar céntimos con soles no compila

bad/brands.ts

```ts
const price = toCents(9000);
addCents(price, toSoles(112.5));
addCents(price, 11250);
```

tsc -p bad

```
bad/brands.ts(5,17): error TS2345: Argument of type 'Soles' is not assignable to parameter of type 'Cents'.
  Type 'Soles' is not assignable to type '{ readonly __brand: "Cents"; }'.
    Types of property '__brand' are incompatible.
      Type '"Soles"' is not assignable to type '"Cents"'.
bad/brands.ts(6,17): error TS2345: Argument of type 'number' is not assignable to parameter of type 'Cents'.
  Type 'number' is not assignable to type '{ readonly __brand: "Cents"; }'.
```

Lo que la marca no garantiza

1. **No existe en tiempo de ejecución.** `typeof` dice `'number'`; `JSON.stringify` escribe un número.
2. **Necesita un `as`**, y solo uno, dentro de la función que revisa. King: un “constructor inteligente” como única puerta.
3. **No garantiza que sea entero.** `112.5 as Cents` compila; `types.test.ts` lo comprueba.

> **Lo que sí da**
>
> No mezclar céntimos con soles, ni con un `number` cualquiera, sin que `tsc` lo note.

Lo que los tipos no dicen

Propiedad	¿tsc?	Dónde va
`status` es `'confirmed'` o `'cancelled'`	Sí (sin `as`)	El tipo
Mirar el error antes de usar el éxito	Sí	Unión discriminada
`hours` es 1, 2 o 3	Con `BookingHours`	El tipo o una revisión
Entre las 7:00 y las 22:00	No	Especificación, `checkBookingRules`, pruebas
`priceCents` es entero	No	`toCents`, pruebas
No se cruza con otra reserva	No	Invariante escrito, handler, pruebas
Se cancela hasta 3 horas antes	No	`isCancellableAt`, pruebas

Empujen hacia los tipos todo lo que se pueda; lo demás, escrito y probado.

Con IA: el camino más corto puede ser un `as`

- Ante *“Argument of type ’unknown’ is not assignable…”*, un cambio como los que suele proponer un agente es `JSON.parse(body) as BookingRequest`.
- El error desaparece, y la revisión también.
- Al revisar, busquen los `as`, `any` y `// @ts-ignore` nuevos.
- Para cada uno, pidan la revisión que lo justifica.

SECCIÓN 06

Design docs

06

Qué es un design doc

Design doc (Ubl, 2020)

Documento **informal** que se escribe **antes** de programar: estrategia a alto nivel y decisiones de diseño, con énfasis en los **trade-offs**.

"Nuestro trabajo no es producir código, sino resolver problemas."

Para qué sirve

- Encontrar problemas cuando cambiar es barato
- Llegar a un consenso
- No olvidar seguridad, privacidad, observabilidad
- Llevar la experiencia de los seniors al resto
- Memoria de por qué se decidió

Las secciones habituales

Contexto y alcance	hechos objetivos, breves; no es un documento de requisitos
Objetivos y **no objetivos**	un no objetivo podría ser objetivo, y se elige no perseguirlo
El diseño	vista general, diagrama de contexto, API y datos sin pegar esquemas
Alternativas consideradas	trade-offs de cada una y por qué se descartó
Temas transversales	seguridad, privacidad, observabilidad

Regla número uno: la forma que más sentido tenga. Largo: 10 a 20 páginas un proyecto grande; mini design doc de 1 a 3. P2 pide además *Riesgos*.

Turnos fijos: objetivos y no objetivos

Objetivos

- La administración crea un turno: cancha, día, horario, temporada.
- Disponibilidad y caja ven esas horas como ocupadas.
- Saltar una semana sin cancelar todo el turno.
- El invariante se mantiene, con los turnos.

No objetivos

- Que los vecinos creen turnos desde la página.
- Cobrar la temporada por adelantado.
- Repeticiones que no sean semanales.

Un no objetivo no es un objetivo negado

“El sistema no debe caerse” no es un no objetivo. “Los vecinos no crean turnos” sí: alguien podría esperarlo.

Las alternativas son la parte más útil

Crear las reservas de la temporada

Una reserva por semana, con `seriesId`.

+ Todo lo que existe sigue funcionando.
+ El invariante sigue siendo "no hay dos filas cruzadas".
– Cambiar el turno es cambiar muchas filas.

Guardar la regla

"Jueves 19–21, de marzo a junio" y calcular al vuelo.

+ Cambiar el turno es editar una fila.
– Disponibilidad, cruces y caja miran dos fuentes.
– Saltar una semana pide una tabla de excepciones.

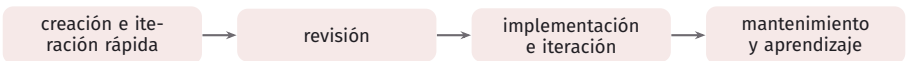
Una es más simple de **consultar**; la otra, de **cambiar**. Cuál conviene depende de qué pasa más seguido: una pregunta para el cliente.

Cuándo no escribirlo, y cuándo es tarde

- Cuesta escribirlo y revisarlo.
- Vale la pena si la solución es **ambigua**: por el problema, por la solución o por las dos.
- Señal de que sobra: un manual de implementación, sin trade-offs ni alternativas.
- “Lo probé y funciona”: un prototipo es de los mejores argumentos.

Error común

Escribirlo **después** de construir, para cumplir. Ya no puede cambiar ninguna decisión, y las alternativas se inventan.



SECCIÓN 07

La documentación como código

07

La documentación como código (SWEG, cap. 10)

- Reglas internas que se siguen
- **Control de versiones**
- Un **owner** claro
- **Revisión** de cambios, junto con el código
- Problemas registrados como bugs
- Evaluada periódicamente

Lo que pasó en Google

Una wiki sin owners: documentos obsoletos y duplicados (de 7 a 10 sobre cómo configurar un mismo sistema). Al moverlos al control de versiones, con owners y revisión, mejoraron.

En su equipo: `docs/` en el repositorio; el PR que cambia una función cambia su especificación.

Un público y un tipo por documento

Tipo	Para qué	En Kallpa
Referencia	Cómo se usa el código	El contrato de `checkBookingRules`; la API del README
Design doc	Proponer un diseño antes de construir	Turnos fijos
Tutorial	Llevar a alguien nuevo a hacer algo real	“Cómo correrla”
Conceptual	Explicar una idea	“Por qué las fechas son texto”
Página de entrada	Dirigir a los demás	El índice de `docs/`

No mezclen tipos: las decisiones de diseño no van en la referencia de una API.

SECCIÓN 08

Desarrollo guiado por especificaciones

08

Desarrollo guiado por especificaciones

Spec-driven development (Böckeler, 2025)

Escribir una especificación **antes** de escribir código con IA. La especificación es la fuente de verdad para la persona y para la IA. La definición todavía está en movimiento.

Spec-first
se escribe antes y se usa para esa tarea

Spec-anchored
se conserva para mantener la funcionalidad

Spec-as-source
solo se edita la especificación; el código se regenera

Tres herramientas, según Böckeler (septiembre de 2025)

Herramienta	Lo que observó	Nivel
Kiro	Tres documentos: requisitos (*given, when, then*), diseño, tareas	spec-first
spec-kit	Una “constitución” y *specify, plan, tasks*; muchos archivos; una rama por especificación	spec-first
Tessl	Beta privada; especificación y archivo de código uno a uno	spec-anchored; explora spec-as-source

Cambian rápido: esto es lo que ella vio entonces, no una descripción actual.

Lo que gana y lo que cuesta

Gana

- Un acuerdo explícito en lugar de adivinar.
- Se revisa **antes** de que exista el código.

Cuesta (observaciones de Böckeler)

- Un bug pequeño: 4 historias con 16 criterios.
- Mucho Markdown repetitivo que revisar.
- Falsa sensación de control: el agente no siguió todas las instrucciones, y siguió otras con demasiado celo.
- Como model-driven development, pero sin un lenguaje que se pueda revisar, y con no determinismo.

Cómo leerlo

La experiencia de una ingeniera, con tres herramientas, en un momento dado. No es un estudio con mediciones.

Donde la especificación calla, el agente elige

Especificación: *“devuelve las próximas reservas del vecino”*. Escrita para la clase, como ilustración:

upcoming.ts

```ts
export function upcomingBookingsGuess(bookings: readonly Booking[], now: LimaTime):
    Booking[] {
  return bookings
    .filter((b) => b.date >= now.date)              // "próximas": de hoy en adelante
    .sort((a, b) => a.date.localeCompare(b.date));  // por fecha
}
```

- Incluye las canceladas: no filtra por estado.
- Incluye una de hoy que ya empezó: compara solo la fecha.
- En la misma fecha, no ordena por hora.
- Ninguna de las tres decisiones se anuncia.

Las pruebas de aceptación lo muestran

Especificación completa: las **confirmadas** que **todavía no empiezan**, por **fecha y hora**.

```
node --test demos/upcoming-guess.test.ts

✓ devuelve las reservas futuras ordenadas por fecha (6.83175ms)
✗ no devuelve las reservas canceladas (0.716625ms)
✗ no devuelve una reserva de hoy que ya empezó (0.1385ms)
✗ en la misma fecha, ordena por hora de inicio (0.151792ms)
ℹ tests 4
ℹ suites 0
ℹ pass 1
ℹ fail 3
```

Error común

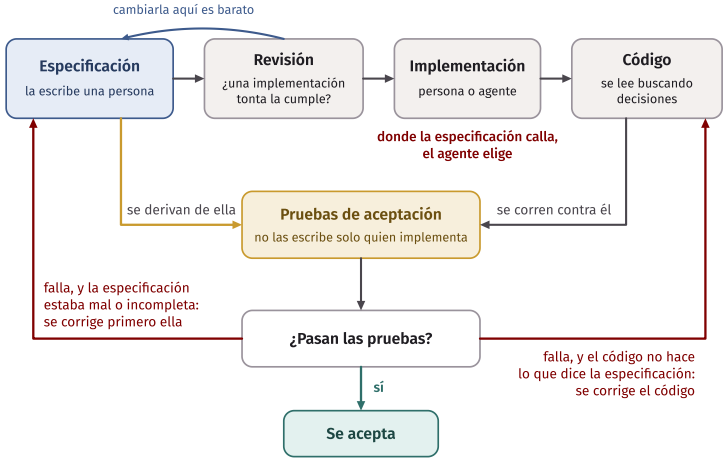
Aceptar el código porque “cumple la especificación”. Solo pasa el caso feliz: las tres decisiones habrían llegado a producción sin que nadie las tomara.

Cómo se comprueba

1. **Revisen la especificación antes que el código.** ¿Una implementación tonta la cumple?
2. **Las pruebas de aceptación no las escribe solo quien implementa**, persona o agente. Si salen de la misma lectura, confirman esa lectura.
3. **Lean el código buscando decisiones**: cada `if`, `filter`, `sort`. ¿Lo dice la especificación?
4. **Cuando una prueba falla, decidan quién tiene razón**: el código o la especificación. Primero la especificación.

> **Con IA**
>
> La especificación cambia *dónde* se pone el esfuerzo, no cuánto. No comprueba nada sola.

El ciclo de la especificación



SECCIÓN 09

Laboratorio

09

El pedido de la administración

Quien quiere 2 o 3 horas busca a mano dónde caben. La administración pide:

src/domain/possible-start-hours.ts

```ts
export function possibleStartHours(
  bookings: readonly Booking[],   // las de esa cancha ese día, con las canceladas
  date: string,
  hours: number,
  now: LimaTime,
): number[];
```

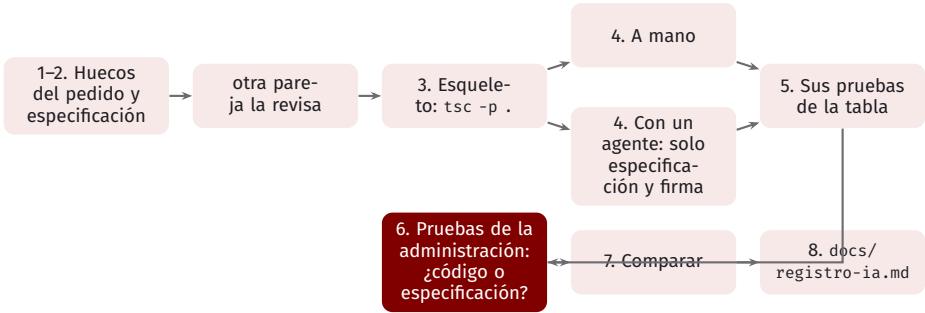
Decidió la administración

- Las canceladas no cuentan.
- Lo que ya empezó hoy no se ofrece.
- De menor a mayor, sin repetir.

Deciden ustedes

- ¿`hours` igual a 0, 4 o 2.5?
- ¿Una fecha fuera de las dos semanas?
- ¿Revisar o exigir?

De la especificación al código, con y sin agente



Guía y template: `code/week-04/horas-posibles.md`. Al final, `make check` en verde.

Para la próxima clase

- Leer Ousterhout, *A Philosophy of Software Design*, caps. 2–5 (o la charla de Ousterhout en Google).
- Terminar la especificación y el registro del laboratorio.
- P1 (requisitos) se entrega la semana 5.

Pregunta

Un agente implementa una función a partir de su especificación y todas las pruebas que generó pasan. ¿Qué tendría que ser cierto para que confíen en ese código?