

Ingeniería de Software

Complejidad, módulos profundos y conceptos

Ousterhout, Parnas y Jackson · Acoplamiento y cohesión · Refactoring

Gerson Garrido

Semana 5

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A

1. Qué es la complejidad
2. Programación táctica y estratégica
3. Módulos profundos
4. Ocultamiento de información

Sesión B

5. Conceptos
6. Acoplamiento y cohesión
7. Refactorizar sin cambiar el comportamiento

Lecturas

Ousterhout, *A Philosophy of Software Design*, 2.ª ed., caps. 2–5 (o su charla en Talks at Google).

Jackson, *The Essence of Software*: *Software = concepts*, *Concept purposes*, *Operational principles*.

Complementarias: Parnas (1972); Ousterhout, caps. 6, 7 y 9; Jackson, *Concept modularity* y *Concept dependencies and subsets*.

SECCIÓN 01

Qué es la complejidad

01

Una definición que mira los cambios

> **Complejidad *(complexity)*, según Ousterhout**
>
> Todo lo que, en la estructura de un sistema de software, lo hace difícil de entender y de modificar.

- **No es el tamaño.** Un sistema grande puede ser fácil de cambiar; uno de 300 líneas, imposible.
- **Cuentan los cambios que de verdad se hacen.** Complejidad de cada parte $\times$ tiempo que se pasa en ella.
- **La nota quien lee, no quien escribe.** La prueba es otra persona, o uno mismo en tres meses.

Tres síntomas

Amplificación de cambios

un cambio simple obliga a tocar muchos lugares

Carga cognitiva

cuánto hay que saber para cambiar sin romper

Incógnitas desconocidas

no se sabe qué hay que tocar, ni que hay algo que saber

la peor

Amplificación de cambios: cerrar a las 23:00

src/domain/booking.ts

```ts
export const OPENING_HOUR = 7;
export const CLOSING_HOUR = 22;
```

El horario está en un solo lugar.
¿Basta con cambiar esa línea? En vez de suponerlo, se mide en una copia.

copia de Kallpa

```
$ sed -i "s/CLOSING_HOUR = 22;/CLOSING_HOUR = 23;/" \
    src/domain/booking.ts
$ node --test "test/**/*.test.ts"
...
i tests 40
i suites 6
i pass 38
i fail 2
...
✗ failing tests:

test at test/booking.test.ts:18:3
✗ rechaza una reserva que termina después del cierre (0.554583ms)
...
test at test/http.test.ts:26:3
✗ una cancha sin reservas está libre todo el día (2.865917ms)
```

¿Por qué fallan dos pruebas?

```
node --test

✗ una cancha sin reservas está libre todo el día (2.865917ms)
  AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:

  16 !== 15
```

Una espera 15 horas libres; la otra usa 21:00 + 2 horas como ejemplo de “después del cierre”.

- El código se enteró solo: validación, mensaje de error y disponibilidad usan la constante.
- Una prueba es una especificación: cuando la regla cambia, está bien que falle.
- Cambio completo: una constante, dos pruebas y el README.

Incógnita desconocida: las claves foráneas de SQLite

- SQLite no revisa las claves foráneas si cada conexión no lo pide.
- Quien abra la base con `new DatabaseSync(...)` en otro archivo acepta reservas de canchas que no existen, y nada le avisa.

kallpa

```
$ grep -rn "PRAGMA foreign_keys\|new DatabaseSync" src test
src/store/database.ts:41:  const db = new DatabaseSync(path);
src/store/database.ts:43:  db.exec('PRAGMA foreign_keys = ON');
```

Una sola puerta

`openDatabase` abre, activa la revisión y migra. El dato oscuro solo lo necesita quien mantiene `database.ts`.

Dos causas

Dependencia

Un trozo de código no se puede entender ni cambiar solo.

- No se eliminan: se buscan pocas y claras.
- Kallpa: los handlers reciben un `Clock`; no leen la hora del sistema.

Oscuridad *(obscurity)*

Información importante que no es evidente.

- Una variable `time` sin unidades; un número mágico.
- Kallpa: `priceCents`, no `price`.

Se acumula

Ningún cambio solo vuelve complejo un sistema: es una copia aquí, un booleano allá. Por eso, cero tolerancia en cada cambio.

SECCIÓN 02

Programación táctica y estratégica

02

Que funcione no basta

Táctica

- Objetivo: que la tarea de hoy funcione, ya.
- Cada atajo es pequeño y razonable.
- Así se acumula la complejidad.
- El *(tactical tornado)*: entrega rápido y deja a otros el costo.

Estratégica

- Objetivo: un buen diseño que, además, funciona.
- Invertir un poco en cada cambio: pensar la interfaz, probar dos diseños, arreglar lo que se encuentra.
- Ousterhout propone 10–20 % del tiempo: una recomendación de su experiencia, no un resultado medido.

Deuda técnica, bien y mal usada

- Metáfora de Ward Cunningham (1992): entregar código que todavía no refleja lo que se entendió es pedir un préstamo; se paga reescribiéndolo pronto.
- Los intereses: el costo extra de cada cambio mientras la deuda siga ahí.

Error común

Llamar “deuda técnica que pagaremos luego” a diseño que nadie va a revisar. Una deuda tiene **monto, responsable y fecha**: un issue, una persona y un momento.

En la industria y con IA

En la industria

Ousterhout (cap. 3): Facebook pasó de *Move fast and break things* a *Move fast with solid infrastructure*. Google y VMware, con una cultura que valoraba el diseño.

Es un argumento a partir de casos, sin datos de productividad.

Con IA

Un agente trabaja de forma táctica por omisión: resuelve lo pedido donde le es cómodo, a veces con una capa que solo reenvía o una copia de algo que ya existía.

Revisen el diseño, no solo si las pruebas pasan.

SECCIÓN 03

Módulos profundos

03

Módulo, interfaz e implementación

Módulo

Cualquier unidad con una interfaz (lo que hay que saber para usarla) y una implementación: función, clase, archivo, servicio.

Interfaz formal

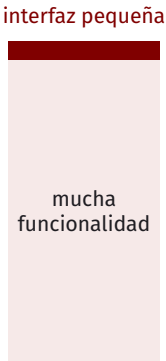
- La revisa el compilador.
- `limaTime(instant: Date): LimaTime`

Interfaz informal

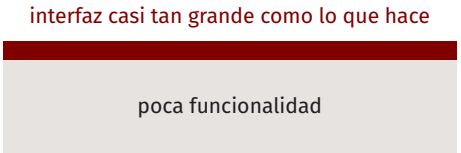
- Qué hace, en qué orden se llama, qué efectos tiene.
- El contrato de la semana 4. Suele ser más grande.

Una **abstracción** deja fuera lo que no importa. Falla si incluye de más (carga cognitiva) o si omite algo importante (abstracción falsa).

Profundo y superficial



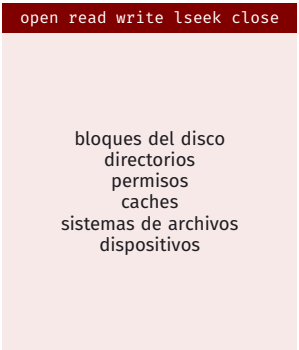
profundo



superficial

Beneficio: lo que hace por quien lo usa.
Costo: la interfaz que hay que aprender.
La profundidad no se mide en líneas.

El ejemplo clásico: archivos en Unix



- Cinco llamadas al sistema.
- Por debajo cambió casi todo en cincuenta años; la interfaz, casi nada.
- Otro ejemplo del libro: el garbage collector. Interfaz casi nula, funcionalidad enorme.

Un módulo profundo pequeño: `clock.ts`

src/domain/clock.ts

```ts
const LIMA_FORMAT = new Intl.DateTimeFormat('en-CA', {
  timeZone: 'America/Lima',
  year: 'numeric', month: '2-digit', day: '2-digit',
  hour: '2-digit', minute: '2-digit',
  hourCycle: 'h23',
});

export function limaTime(instant: Date): LimaTime {
  // formatToParts -> { date, hour, minute }
}
```

Interfaz: un `Date` entra, `{ date, hour, minute }` sale.

Esconde:

- la zona `America/Lima`, en vez de restar 5 horas;
- la medianoche con `h23`;
- armar la fecha con `formatToParts`, parte por parte, sin depender del orden regional.

Nadie más lo sabe

kallpa

```
$ grep -rn "Intl\|America/Lima" src
src/domain/clock.ts:24:const LIMA_FORMAT = new Intl.DateTimeFormat('en-CA', {
src/domain/clock.ts:25:  timeZone: 'America/Lima',
```

- Si mañana se reemplaza `Intl.DateTimeFormat` por la API `Temporal`, cambia un archivo.
- Eso es esconder una decisión de diseño: los que llaman a `limaTime` no se enteran.

Un ejemplo para el capítulo: préstamo de implementos

- La administración presta pelotas y redes. Tres reglas:
  1. un implemento prestado no se presta otra vez hasta que lo devuelvan;
  2. un vecino no tiene dos implementos a la vez;
  3. un préstamo se devuelve una sola vez.
- Tabla `loans`: `returned_at` queda en `NULL` mientras el préstamo está abierto.
- Dos versiones, `loans/shallow.ts` y `loans/deep.ts`, que responden exactamente lo mismo.

> **Por qué un ejemplo aparte**
>
> Tiene la misma forma que un módulo de Kallpa que van a refactorizar en el laboratorio.

La versión superficial: un servicio que reenvía

loans/shallow.ts

```ts
// La "capa de servicios": cada método reenvía la llamada al repositorio.
export class LoanService {
  #repo: LoanRepository;
  listItems(): ItemRow[] {
    return this.#repo.listItems();
  }
  findItem(id: number): ItemRow | undefined {
    return this.#repo.findItem(id);
  }
  // ... cinco métodos más con la misma forma
}
```

Método que solo reenvía *(pass-through method)*: llama a otro con los mismos parámetros, o casi, y no hace nada más (Ousterhout, cap. 7).

¿Dónde quedó la lógica? En el handler

loans/shallow.ts

```ts
const item = service.findItem(itemId);
if (!item) return error(404, 'ITEM_NOT_FOUND', `No existe el implemento ${itemId}.`);
if (service.loansForItem(itemId).some((l) => l.returned_at === null)) {
  return error(409, 'ITEM_LENT', 'El implemento está prestado.');
}
if (service.loansForMember(memberId).some((l) => !l.returned_at)) {
  return error(409, 'MEMBER_HAS_LOAN', 'Primero hay que devolver el implemento anterior.');
}
const id = service.saveLoan(itemId, memberId, now());
```

La misma regla, cuatro veces

"Un préstamo está abierto si no tiene `returned_at`": aquí dos, de dos formas; otra al listar y otra al devolver. Mañana, un estado "perdido"…

Contar los reenvíos con el compilador de TypeScript

code/week-05

```
$ node pass-through.ts loans/shallow.ts loans/deep.ts
...
shallow.ts: clase LoanService, 35 líneas, 7 métodos públicos
  listItems()                       reenvía a this.#repo.listItems
  findItem(id)                      reenvía a this.#repo.findItem
  findLoan(id)                      reenvía a this.#repo.findLoan
  loansForItem(itemId)              reenvía a this.#repo.loansForItem
  loansForMember(memberId)          reenvía a this.#repo.loansForMember
  saveLoan(itemId, memberId, lentAt)   reenvía a this.#repo.insert
  markReturned(id, returnedAt)      reenvía a this.#repo.setReturnedAt
  7 de 7 métodos solo reenvían la llamada
deep.ts: clase Loans, 53 líneas, 3 métodos públicos
  items()                           hace algo más
  lend(itemId, memberId)            hace algo más
  giveBack(loanId)                  hace algo más
  0 de 3 métodos solo reenvían la llamada
```

La versión profunda

loans/deep.ts

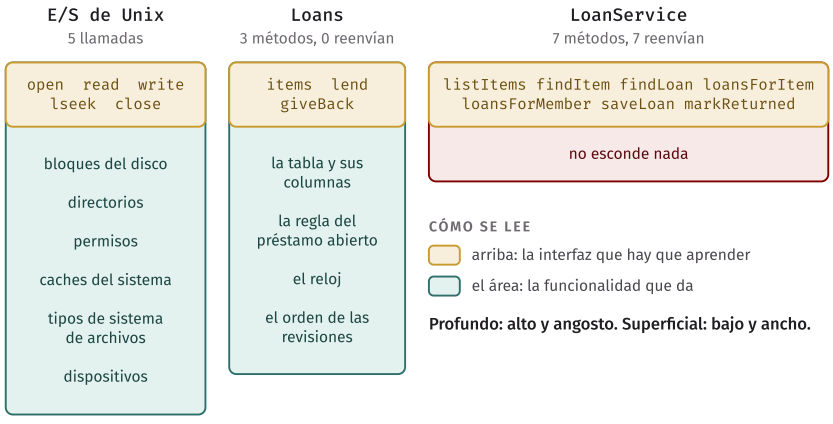
```ts
export type LoanResult = { ok: true; loan: Loan } | { ok: false; error: LoanError };

// Esconde la tabla y sus columnas, la regla de qué préstamo está abierto,
// el reloj y el orden en que se revisan las reglas.
export class Loans {
  items(): ItemStatus[] { /* ... */ }
  lend(itemId: number, memberId: number): LoanResult { /* ... */ }
  giveBack(loanId: number): LoanResult { /* ... */ }
}

const OPEN = 'returned_at IS NULL'; // la regla, en un solo lugar
```

- Interfaz más chica: dice lo que quiere quien llama, no cómo se guarda.
- Errores del negocio (`ITEM_LENT`), no de HTTP: el handler elige el 409.
- `loans.test.ts`: el mismo guion de 12 solicitudes, respuestas idénticas.

Tres módulos, medidos con el mismo rectángulo



¿Siempre está mal reenviar?

No, si el módulo hace algo

- Un despachador que elige a quién llamar.
- Un decorator que agrega un log.
- Una interfaz con varias implementaciones: `Notifier` (consola en producción, lista en las pruebas).

La pregunta: ¿qué sabe este módulo que los demás no tienen que saber?

Error común: ***(classitis)***

Creer que muchas clases pequeñas son buen diseño. Cada clase trae una interfaz.

Java: `FileInputStream` dentro de `BufferedInputStream` dentro de `ObjectInputStream`. Si olvidan el buffer, no hay error: solo lentitud.

SECCIÓN 04

Ocultamiento de información

04

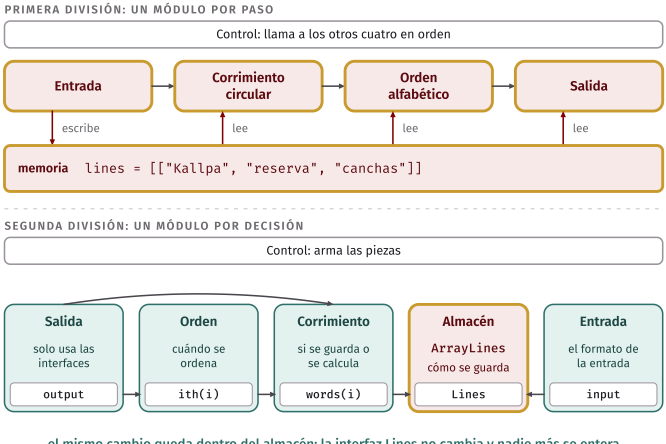
Parnas, 1972: el índice KWIC

- Entran líneas; cada línea se “rota” pasando la primera palabra al final.
- Salen todas las rotaciones de todas las líneas, en orden alfabético.



- Un buen programador lo hace en una o dos semanas. Parnas lo trata como si fuera grande para discutir el criterio de división.

Dos formas de dividir el mismo programa



Lo primero que dice Parnas: los dos esquemas funcionan. Podrían dar el mismo programa ensamblado; la diferencia está en cambiarlos.

Cinco decisiones que pueden cambiar

Cambio	1. Por pasos	2. Por decisiones
Formato de la entrada	un módulo	un módulo
Todas las líneas en memoria	todos	un módulo
Empaquetar 4 caracteres por palabra	todos	un módulo
Índice de rotaciones en vez de copias	tres módulos	un módulo
Ordenar todo de una vez	difícil	un módulo

Parnas compara también el **desarrollo independiente** (formatos de tablas frente a nombres de funciones) y la **comprensibilidad** (para entender la salida de la división 1 hay que entender las demás).

El criterio

Ocultamiento de información ***(information hiding)***

"Casi siempre es incorrecto empezar la descomposición de un sistema en módulos a partir de un diagrama de flujo. Proponemos, en cambio, empezar por una lista de las decisiones de diseño difíciles o que probablemente cambien. Cada módulo se diseña entonces para esconder una de esas decisiones de los demás."

D. Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, CACM 15(12), 1972. Traducción nuestra.

KWIC en TypeScript: el formato, en todas partes

code/week-05

```
$ grep -n "lineStart\|END" kwic/flowchart.ts
6:// terminada en END; lineStart[i] es dónde empieza la línea i en ese texto.
7:export const END = '\u0000';
9:export type Store = { chars: string; lineStart: number[] };
17:  const lineStart: number[] = [];
19:    lineStart.push(chars.length);
20:    for (const word of line.trim().split(/\s+/)) chars += word + END;
22:  return { chars, lineStart };
28:  store.lineStart.forEach((start, line) => {
29:    const end = store.lineStart[line + 1] ?? store.chars.length;
31:    for (let i = start; i < end - 1; i++) if (store.chars[i] === END) shifts.push([line, i + 1]);
39:  const lineBegin = store.lineStart[line];
40:  const lineEnd = store.lineStart[line + 1] ?? store.chars.length;
42:  return rotated.split(END).filter(Boolean).join(' ');
56:// Módulo 4, salida: vuelve a leer chars y lineStart para imprimir.
```

La entrada lo escribe (17–22), el corrimiento lo recorre (28–31), el orden y la salida lo rearman (39–42).

KWIC en TypeScript: el formato, escondido

kwic/hiding.ts

```ts
export type Lines = {
  addLine(words: string[]): void;
  lineCount(): number;
  wordCount(line: number): number;
  word(line: number, w: number): string;
};
```

`PackedLines`: el formato de `flowchart.ts`, dentro de una clase. `ArrayLines`: un arreglo por línea.

Misma interfaz, misma salida.

make test

```
➤ índice KWIC
  ✓ lista cada corrimiento circular en orden alfabético (0.774208ms)
  ✓ las dos divisiones en módulos dan exactamente la misma salida (0.460083ms)
  ✓ cambiar cómo se guardan las líneas no cambia la salida (0.357458ms)
✓ índice KWIC (2.896042ms)
```

¿Qué decisión esconde cada módulo de Kallpa?

Módulo	Decisión que esconde
`domain/clock.ts`	Cómo se pasa de un instante a la hora de Lima
`domain/calendar.ts`	Fechas como texto `'YYYY-MM-DD'`, cuentas en UTC
`domain/pricing.ts`	Tarifas, hora punta y descuento
`notifications/notifier.ts`	El canal del aviso
`store/database.ts`	Esquema, migraciones y el `PRAGMA`
`http/router.ts`	Cómo se emparejan las rutas con `:id`

Para el laboratorio

¿Qué esconde el módulo de `src/services/`? ¿Y `handlers.ts`, que debería esconder solo cómo se traduce HTTP?

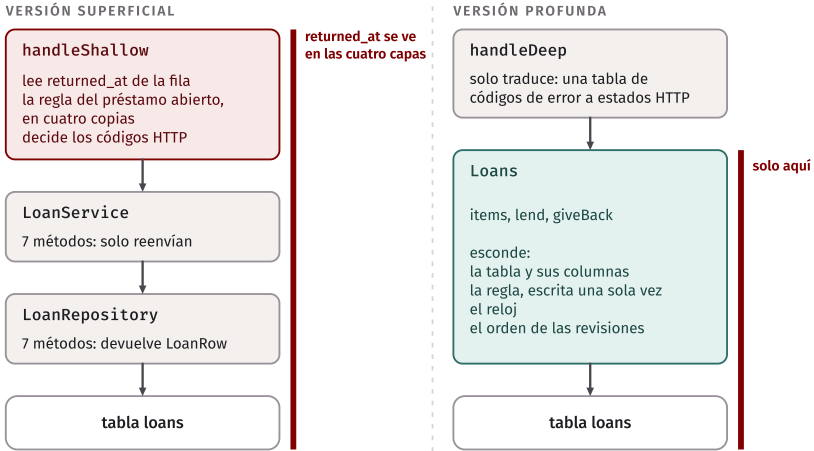
Abstracción que se filtra

Filtración de información ***(information leakage)***

La misma decisión de diseño aparece en varios módulos: cambiarla obliga a tocarlos todos (Ousterhout, cap. 5).

- Los nombres de las columnas de una tabla que llegan hasta la respuesta HTTP.
- Un formato que conocen por separado quien lo escribe y quien lo lee (por la puerta de atrás: ninguna interfaz lo muestra).
- Una consulta que devuelve de más: “acuérdense de filtrar las canceladas”.
- Un `HttpError` lanzado desde una capa que no debería saber de HTTP.

Hasta dónde llega una decisión



Descomposición temporal

- Dividir según el orden en que pasan las cosas, no según lo que cada parte sabe.
- Ejemplo de Ousterhout: leer un archivo, modificarlo y escribirlo, una clase por paso. El lector y el escritor conocen el formato.
- Solución: un módulo que sepa el formato, con leer y escribir.

Pregunten qué **conocimiento** necesita cada paso, no en qué momento corre.

Error común

Creer que declarar campos privados es ocultar información. Con un getter y un setter por campo, quien usa la clase sigue conociendo la representación.

Con IA

"Agrega validación de X" suele terminar en una copia de una regla que ya existía. Búsquenla con `grep` antes de aceptar.

SECCIÓN 05

Conceptos

05

Qué es un concepto

Concepto (Jackson)

Una unidad de funcionalidad que el usuario reconoce y que tiene sentido por sí sola.

Propósito	**Estado**	**Acciones**	**Principio operacional**
para qué existe	lo que recuerda	lo que se puede hacer	el escenario típico

- **No es una *(feature)***: se entiende sin los demás y pasa de una aplicación a otra.
- **No es una clase**: involucra varias colecciones y sus relaciones. Jackson prefiere “un servicio muy pequeño”.
- **Es genérico**: *reserva* sirve para canchas, mesas o salas.

El principio operacional: “si…, entonces…”

- Una historia corta que muestra cómo el concepto cumple su propósito.
- No es un caso de uso: no busca cubrir todos los caminos, solo el esencial.

Papelera (Jackson)

Si borro un archivo, lo puedo recuperar sacándolo de la papelera. Si borro y vacío la papelera, el archivo desaparece.

Propósito: deshacer un borrado, no borrar.

Reserva (Kallpa)

Si reservo la cancha de 19:00 a 21:00, a esa hora la cancha es mía y nadie más la puede reservar.

Propósito: uso exclusivo de un recurso durante un tiempo.

El propósito

Tres criterios

- **Centrado en una necesidad**: un *like* sirve para ordenar contenido, no para “expresar aprobación”.
- **Específico**: “aumentar los usuarios” vale para cualquier concepto.
- **Evaluable**: sirve para medir el diseño.

Un solo propósito

La reacción de Facebook sirve para decidir qué se muestra y para mandar una emoción. “Me enoja” cuenta como voto a favor.

Modularidad de conceptos: **especificidad** (un propósito), **completitud** (lo cumple entero) e **independencia** (se entiende sin otros).

Kallpa como conjunto de conceptos

Concepto	Propósito	Principio operacional
Reserva	Uso exclusivo de una cancha unas horas	Si reservo de 19:00 a 21:00, nadie más la puede reservar
Membresía	Premiar a quien paga la cuota	Si soy socio, mis reservas cuestan 10 % menos
Tarifa con hora punta	Cobrar más las horas que todos quieren	Una hora de fulbito un martes: S/ 90 a las 10:00, S/ 112.50 a las 19:00
Cancelación con plazo	Arrepentirse sin dejar la cancha vacía a último momento	Reservé para las 19:00: cancelo hasta las 16:00; desde las 16:01, ya no
Notificación	La confirmación sin entrar a la página	Si reservo, me llega un mensaje con cancha, hora y total

Se combinan con **sincronizaciones**: cuando se confirma una reserva, se calcula el precio, se registra el plazo y se avisa.

Un tipo, dos propósitos

src/domain/member.ts

```ts
export type Member = {
  id: number;
  name: string;
  phone: string;
  plan: MemberPlan;
};
```

- `name` y `phone`: saber quién reserva y cómo avisarle (**registro de vecinos**).
- `plan`: el descuento (**membresía**).
- Hoy no molesta. “Socios solo de vóley” o “la membresía vence” tocan algo que también usa el aviso por WhatsApp.

Un concepto en código

concepts/reservation.ts

```
// concepto Reserva [Recurso, Usuario]
//
// propósito  asegurar a alguien el uso exclusivo de un recurso durante un
//            tiempo
// principio  si reservo un recurso para un bloque de horas, en esas horas el
//            recurso es mío: nadie más lo puede reservar; si cancelo, el
//            bloque vuelve a estar libre
// estado     reservas: Id -> (recurso, usuario, día, desde, hasta, activa)
// acciones   reserve(usuario, recurso, día, desde, hasta) -> Id | ocupado
//            cancel(id)
//            holder(recurso, día, hora) -> Usuario | nadie
```

`Reservation<R, U>` es genérica: no sabe qué es una cancha ni qué es un vecino. `deadline.ts` y `notification.ts` tampoco saben de reservas.

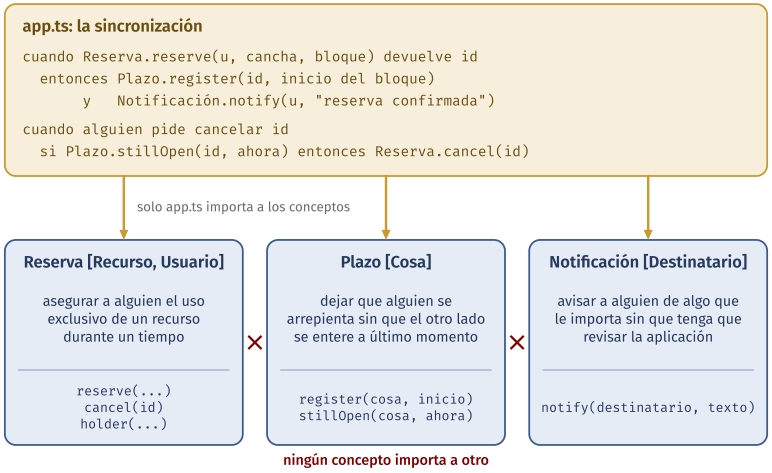
Sincronizaciones

concepts/app.ts

```
1  //   cuando Reserva.reserve(u, cancha, bloque) devuelve id
2  //     entonces Plazo.register(id, inicio del bloque)
3  //          y   Notificación.notify(u, "reserva confirmada")
4  //   cuando alguien pide cancelar id
5  //     si Plazo.stillOpen(id, ahora) entonces Reserva.cancel(id)
```

- Ningún concepto importa a otro: solo `app.ts` los conoce y los combina.
- Es una forma de llevar los conceptos al código, no la única. Lo que importa es la propiedad: cada uno se entiende y se prueba solo.

Conceptos y sincronización



Los principios operacionales, como pruebas

```
make test

➤ principios operacionales
  ✓ Reserva: si reservo de 19 a 21, a esa hora nadie más puede reservar; si cancelo, queda libre (0.771541ms)
  ✓ Reserva es genérica: sirve igual para la parrilla del club (0.08975ms)
  ✓ Plazo: con 3 horas de aviso, a las 15:00 se deshace algo de las 18:00; a las 15:01, ya no (0.114625ms)
✓ principios operacionales (2.074708ms)
➤ la aplicación: tres conceptos y sus sincronizaciones
  ✓ al reservar se avisa; al cancelar a tiempo se libera la cancha (0.210292ms)
  ✓ ningún concepto importa a otro (1.96175ms)
✓ la aplicación: tres conceptos y sus sincronizaciones (2.286291ms)
```

Un principio operacional describe el comportamiento esencial con una historia corta: es una buena prueba.

Turnos fijos: ¿concepto nuevo o forma de usar *reserva*?

- **¿Propósito propio?** “Ahorrarse diez reservas” es un botón *repetir*. “Asegurar a un equipo su horario de la temporada” es otro propósito.
- **¿Principio propio?** “Si un jueves no vienen, esa fecha queda libre, pero el turno sigue.”
- **¿Estado y acciones propios?** Una regla (día, hora, cancha, desde, hasta); crear, saltar una fecha, terminar.

La respuesta cambia todo

Modelo de datos, endpoints y reglas: ¿saltar una fecha tiene plazo? ¿Precio por hora o por temporada? ¿Y si la fecha ya estaba reservada?

Lo decide la cliente, **antes** de programar, y queda en el design doc.

SECCIÓN 06

Acoplamiento y cohesión

06

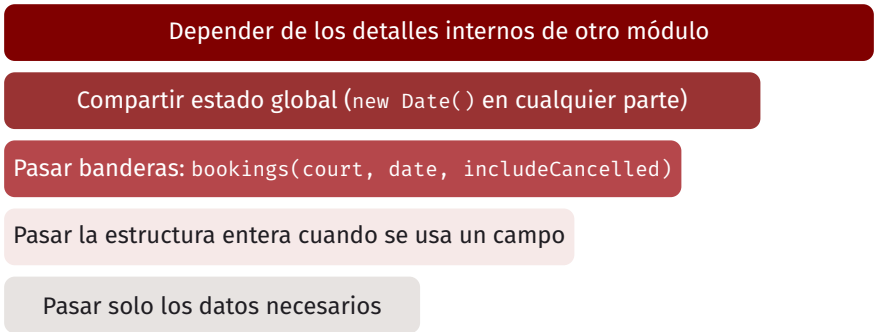
Dos medidas de lo mismo

Acoplamiento *(coupling)*
Cuánto obliga un cambio en un módulo a cambiar otro.

Cohesión *(cohesion)*
Cuánto pertenecen juntas las partes de un módulo.

- Diseño estructurado: Stevens, Myers y Constantine (1974). Se busca bajo acoplamiento y alta cohesión.
- Heurística de Robert C. Martin: una sola razón para cambiar. Si un archivo cambia por una tarifa y por el formato de una respuesta, junta dos cosas.
- La medida es la de la Sesión A: ¿cuánto cuesta el cambio que de verdad va a pasar?

Formas de acoplamiento, de más a menos dañina



Y el acoplamiento oculto: dos módulos que cambian juntos por un supuesto que no está escrito.

Solo los datos necesarios: el dominio de Kallpa

src/domain/booking.ts

```ts
export function isCancellableAt(slot: Pick<TimeSlot, 'date' | 'startHour'>, now:
    LimaTime): boolean {
  return minutesUntilStart(slot, now) >= CANCELLATION_NOTICE_HOURS * 60;
}
```

- Recibe la fecha y la hora de inicio, no la reserva entera.
- El precio recibe el `plan` del vecino, no el vecino con su nombre y su celular.
- Se prueban con objetos de dos campos y no cambian si `Booking` o `Member` ganan un campo.

Acoplamiento oculto… y documentado

src/domain/court.ts

```ts
// Deportes que se juegan en el complejo. Estos textos son los que se guardan
// en la base de datos y los que viajan en la API.
export const SPORTS = ['fulbito', 'voley', 'basquet'] as const;
```

src/store/database.ts, primera migración

```
    sport TEXT NOT NULL CHECK (sport IN ('fulbito', 'voley', 'basquet'))
```

¿Que la migración use `SPORTS`? No.

- Una migración es un registro histórico: nunca se edita una que ya corrió.
- Agregar `'tenis'` exige una migración nueva de todas formas.
- La duplicación es deliberada; el comentario hace visible el acoplamiento.

DRY es sobre conocimiento, no sobre texto

> **DRY (Hunt y Thomas, *The Pragmatic Programmer*, 1999)**
> Cada pieza de conocimiento debe tener una representación única, sin ambigüedad y con autoridad dentro del sistema.

src/domain/booking.ts

```ts
export const MAX_HOURS_PER_BOOKING = 3;
export const CANCELLATION_NOTICE_HOURS = 3;
```

Mismo valor, dos reglas. Un `THREE_HOURS` sería una abstracción falsa.

Una regla, tres textos:

```
l.returned_at === null
!l.returned_at
returned_at IS NULL
```

En Kallpa: `grep -rn "confirmed" src` es un buen comienzo (laboratorio).

Juntar o separar

Se juntan (Ousterhout, cap. 9)

- si comparten información;
- si siempre se usan juntos;
- si es difícil entender uno sin el otro;
- si así se simplifica la interfaz o se elimina una duplicación.

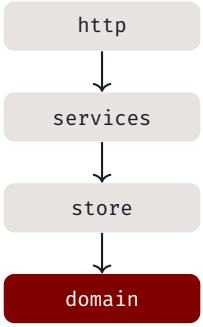
Se separa

- lo de propósito general de lo de un caso particular: `overlaps` es general; responder `SLOT_TAKEN`, particular.

Error común

Juntar código solo porque se parece, o partir una función solo por su largo (*(conjoined methods)*).

La dirección de las dependencias, medida



no depende de nada

```
make capas

$ make capas
...
http → services           2 imports (1 solo tipos)
http → notifications      2 imports (1 solo tipos)
http → store              1 import (solo tipos)
http → domain             6 imports
services → store          3 imports
services → domain         2 imports (solo tipos)
notifications → domain    1 import
store → domain            6 imports (4 solo tipos)
Ninguna dependencia va hacia arriba.
```

Se omiten las tres líneas de `(raíz)` (`main.ts`, `seed.ts`). `layers.test.ts` lo convierte en prueba.

¿Qué tipo cruza de `store` a `http`? (laboratorio)

SECCIÓN 07

Refactorizar sin cambiar el comportamiento

07



Qué es un refactoring

Refactoring (Fowler, refactoring.com)

Una técnica disciplinada para reestructurar código existente, cambiando su estructura interna sin cambiar su comportamiento externo. Una serie de transformaciones pequeñas, y el sistema funciona después de cada una.

- **Dos sombreros** (Kent Beck, en el libro de Fowler): o agregan funcionalidad o reestructuran; nunca las dos a la vez.
- **Comportamiento** es lo observable: respuestas, mensajes de error, datos guardados, avisos. Ley de Hyrum (semana 1).

Un refactoring pequeño: extraer `requireDate`

refactor-fecha.diff

```
1  -  const date = url.searchParams.get('
       date');
2  -  if (!isValidDate(date)) {
3  -    throw new HttpError(400, '
       INVALID_DATE', ...);
4  -  }
5  +  const date = requireDate(url);
```

make refactor

```
 src/http/handlers.ts | 20 ++++++++++++--------
 1 file changed, 12 insertions(+), 8 deletions(-)
make --no-print-directory -C build/refactor check
tsc -p .
eslint .
node --test "test/**/*.test.ts"
...
i tests 40
i suites 6
i pass 40
i fail 0
...
test/ sin cambios
```

La validación de `?date=` estaba copiada en dos handlers. El mensaje queda idéntico.

Ciclo: un paso pequeño, `make check`, un commit. Si falla, se deshace el paso.

Pruebas en verde no prueban que nada cambió

Error común

"Las pruebas pasan, así que no cambió nada." Las pruebas solo revisan lo que alguien pensó en revisar. Las de Kallpa no cubren todos los códigos de error del README ni el orden de las revisiones.

- **Prueba de caracterización** (Feathers, 2004): fija lo que el código hace hoy, para cambiarlo sin cambiar eso.
- `loans.test.ts` es una: el mismo guion a las dos versiones.
- En Kallpa: un guion de `curl` por los errores del README, antes y después. `diff antes.txt despues.txt` vacío.

Si encuentran un error, no lo corrijan aquí

- Abran un issue con la evidencia; terminen el refactoring; arreglen en otro pull request, con su prueba.
- El revisor tiene que poder comprobar que el refactoring no cambia nada.
- “Parece un error” a veces es una regla que nadie documentó: se le pregunta a la cliente.

Con IA

Buenos para refactorings mecánicos. Revisen siempre: `git diff –stat – test` vacío; nada “arreglado” de paso; mensajes idénticos; un módulo profundo y no una capa más.

Un cambio como los que suele proponer un agente: una interfaz nueva y una clase con los mismos métodos.

SECCIÓN 08

Laboratorio

08

Refactorizar un módulo superficial de Kallpa

1. Medir (`make check` en verde).
2. Diagnóstico con evidencia en `docs/diseno-reservas.md`.
3. Diseñar la interfaz antes de implementar.
4. Implementar en `src/services/`; `createServer` no cambia.
5. Pasos pequeños: un commit y `make check` por paso.
6. Medir de nuevo.
7. Pull request con el antes y el después.

Regla: nada cambia en `test/`.

```
make medir

197 src/http/handlers.ts
 59 src/services/booking-service.ts
256 total
```

Herramientas: `node pass-through.ts <archivo>`, `make capas`, `grep -rn "confirmed" src`.

Preguntas para revisar la interfaz de otra pareja

- ¿Qué tiene que saber un handler para usarla? Si incluye “las columnas”, “filtrar las canceladas” o “qué hora es en Lima”, todavía filtra.
- ¿Los errores del negocio salen como resultado tipado o error del dominio? Si el módulo lanza `HttpError`, sabe de HTTP.
- ¿Cuántas operaciones tiene, comparadas con los métodos que reemplaza?
- Que el total de líneas suba un poco no es malo: lo que debe bajar es lo que la capa HTTP necesita saber.

Para la próxima clase

- Terminar el laboratorio: pull request con el antes y el después, y la revisión del de otra pareja.
- **P1**: entrega (cliente, entrevistas, historias, backlog y prototipo).
- **P2**: publicado. Diseño y primer sprint; entrega en la semana 8.
- Semana 6, APIs y patrones de diseño. Lecturas: Google, *API Design Guide* (*Resource-oriented design*, *Standard methods*, *Errors*) y las AIP-121, 136, 158 y 180; Microsoft, *Azure REST API Guidelines*; RFC 9110, secs. 9.2.1 y 9.2.2.

Pregunta

En su proyecto, ¿qué decisión de diseño es la más probable que cambie? ¿Qué módulo la esconde hoy?