

UNIVERSIDAD NACIONAL DE INGENIERÍA

Arquitectura de Software

Modularidad: límites, acoplamiento y dependencias

Parnas · Cohesión y acoplamiento · Ciclos · Ports y adapters · Reglas que se comprueban

Gerson Garrido

Semana 3

Hoy

Sesión A

1. Módulos y modificabilidad
2. El criterio de Parnas
3. Ocultamiento, cohesión y acoplamiento

Sesión B

4. El grafo de dependencias
5. Inversión de dependencias
6. Ports y adapters
7. Reglas que se comprueban

Lecturas

A: Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules* (1972), completo; *Software Architecture in Practice*, cap. 8 *Modifiability*.

B: Cockburn, *Hexagonal Architecture*; Schuchert, *DIP in the Wild*.

Complementarias: *Fundamentals of Software Architecture*, cap. 3 y 8; connascence.io; dependency-cruiser, *Rules reference*; *SAiP*, cap. 15.

De la semana 2: un escenario cuyo estímulo es un cambio

Fuente	El área comercial
Estímulo	Pide aceptar un medio de pago nuevo
Artefacto	El código de Tinkuy
Entorno	En desarrollo, con el sistema ya en producción
Respuesta	El cambio se hace, se prueba y se despliega sin modificar la compra ni el catálogo
Medida	Toca un solo módulo y toma menos de dos días de una persona

¿Cómo se divide un sistema para que esto sea verdad? ¿Y cómo se comprueba que lo sigue siendo?

SECCIÓN 01

Módulos y modificabilidad

01



Qué es un módulo

Módulo ***(module)***

Una unidad de código con una responsabilidad y una interfaz.

Parnas: una asignación de responsabilidad antes que un subprograma. Lo que una persona o un equipo puede hacer sin preguntarle a otro.

Lo que se espera de dividir bien

1. Desarrollo en paralelo
2. Cambiar un módulo sin cambiar los demás
3. Entender el sistema un módulo a la vez

Los tres dependen de cuánto tiene que saber un módulo sobre los otros.

La pregunta de la semana

- La estructura de módulos responde: ¿qué toca un cambio? y ¿quién puede trabajar en paralelo?
- Tinkuy es un solo proceso con cinco módulos. Esta semana miramos los cinco, no el proceso.

> **La medida**
>
> Cuando llegue este cambio, que es probable, ¿cuántos módulos hay que tocar? Una división es buena o mala respecto de una lista de cambios, no en abstracto.

Ninguna división aguanta todos los cambios. Dividir es apostar por cuáles van a llegar.

SECCIÓN 02

El criterio de Parnas

02

Parnas (1972): un índice KWIC

- Entran líneas; cada línea es una lista de palabras.
- Una línea se rota: la primera palabra pasa al final.
- Sale el listado de todas las rotaciones, en orden alfabético.
- Pequeño a propósito: se trata *como si* hubiera un equipo por módulo.

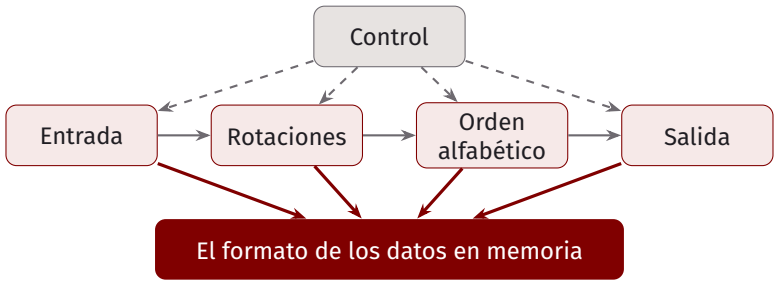
kwic/titles.txt

```
La ciudad y los perros
Los ríos profundos
El mundo es ancho y ajeno
```

node kwic/decisions/main.ts

```
ajeno El mundo es ancho y
ancho y ajeno El mundo es
ciudad y los perros La
El mundo es ancho y ajeno
es ancho y ajeno El mundo
La ciudad y los perros
los perros La ciudad y
Los ríos profundos
mundo es ancho y ajeno El
perros La ciudad y los
profundos Los ríos
ríos profundos Los
y ajeno El mundo es ancho
y los perros La ciudad
```

Primera descomposición: por pasos



- Sale del diagrama de flujo. Cada paso es pequeño y claro.
- Antes de repartir el trabajo hay que acordar cómo quedan los datos. Ese acuerdo es la interfaz entre los cuatro.

El acuerdo, en `kwic/steps`

kwic/steps/formats.ts

```ts
// Todos los caracteres en un solo texto, con un espacio entre palabras.
// lineStart[i] es la posición donde empieza la línea i.
export type Lines = { chars: string; lineStart: number[] };
// Una rotación: la línea original y dónde empieza, dentro de chars.
export type Shift = { line: number; start: number };
```

kwic/steps/main.ts

```ts
const lines = readLines(text);
const shifts = circularShifts(lines);
const sorted = alphabetize(lines, shifts);
return render(lines, sorted);
```

- `alphabetize` y `render` también reciben `lines`: para escribir una rotación hay que saber cómo se guardan los caracteres.

Segunda descomposición: por decisiones

Módulo	La decisión que esconde
`line-storage.ts`	Cómo se guardan las líneas en memoria
`input.ts`	El formato de la entrada
`circular-shifter.ts`	Cómo se obtienen las rotaciones: guardadas, con índice o al vuelo
`alphabetizer.ts`	Qué es “orden alfabético” y cuándo se ordena
`output.ts`	El formato del listado
`main.ts`	Cómo se arma todo

- No sale del diagrama de flujo. Sale de preguntar: ¿qué decisiones podrían cambiar?
- El almacén de líneas no es un paso: lo usan casi todos.

Una interfaz que no delata

kwic/decisions/line-storage.ts

```ts
export interface LineStorage {
  addLine(words: readonly string[]): void;
  lineCount(): number;
  wordCount(line: number): number;
  word(line: number, index: number): string;
}
```

- Pregunta por posición lógica (línea, palabra). No dice dónde ni cómo se guarda.
- Por dentro guarda lo mismo que la otra versión: un texto y un índice. Ahora son variables locales.
- El módulo de rotaciones se ve como “otro almacén”, y no promete ningún orden.

Las dos funcionan

- Producen el mismo listado (hay una prueba).
- Parnas: podrían ser idénticas una vez compiladas, con los mismos algoritmos y estructuras.

Idea clave

Dos divisiones del mismo programa pueden correr igual y costar muy distinto de cambiar. La estructura de módulos no es una propiedad de lo que el sistema hace: es una propiedad de cómo está escrito.

Llega un cambio: las líneas se guardan de otra manera

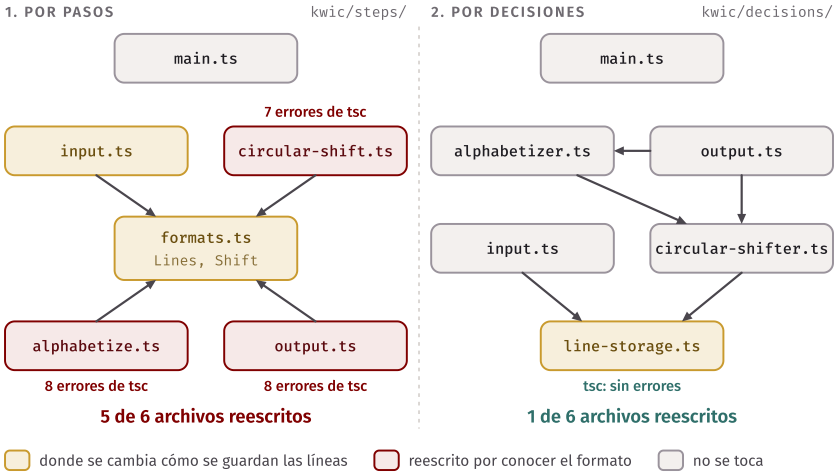
```
make kwic-change

Descomposición 1, por pasos (kwic/steps/, 6 archivos)
  1. Cambia la forma de guardar las líneas: formats.ts, input.ts
  2. tsc encuentra errores en otros módulos:
       alphabetize.ts: 8
       circular-shift.ts: 7
       output.ts: 8
  3. Archivos reescritos para completar el cambio: 5 de 6 (alphabetize.ts, circular-shift.ts, formats.ts, input.ts, output.ts)
  4. El listado sale igual que antes: sí

Descomposición 2, por decisiones (kwic/decisions/, 6 archivos)
  1. Cambia la forma de guardar las líneas: line-storage.ts
  2. tsc: sin errores
  3. Archivos reescritos para completar el cambio: 1 de 6 (line-storage.ts)
  4. El listado sale igual que antes: sí
```

- El cambio 3 de Parnas, en TypeScript: de un texto con un índice a una lista de palabras por línea.

Cinco de seis contra uno de seis



Cinco de seis contra uno de seis

- Por pasos: quien cambia el formato rompe a los otros tres equipos. Solo se salva el control, que pasa valores sin mirarlos.
- Por decisiones: los demás solo conocían `wordCount` y `word`.
- No es mejor por tener interfaces. Es mejor para este cambio, porque alguien lo anticipó.

Los cinco cambios del artículo

Decisión que cambia	Por pasos	Por decisiones
1. Formato de la entrada	Un módulo	Un módulo
2. Tener todas las líneas en memoria	Todos	Almacén de líneas
3. Cómo se empaquetan los caracteres	Todos	Almacén de líneas
4. Índice de rotaciones en vez de guardarlas	Rotaciones, orden y salida	Rotaciones
5. Ordenar todo de una vez	Difícil: la salida supone el índice completo	Orden alfabético

Son decisiones de representación o de momento. Por pasos quedaron en las interfaces; por decisiones, adentro de un módulo.

El criterio

Parnas, 1972

Casi siempre es incorrecto empezar la división a partir de un diagrama de flujo. Se empieza con la lista de decisiones difíciles o que es probable que cambien; cada módulo se diseña para esconder una de las demás.

- Se divide por lo que puede cambiar, no por el orden en que se ejecuta.
- Tiene costo, y el artículo lo dice: más llamadas entre módulos, y puede ser más lento.

Trade-off

Esconder una decisión cuesta indirección. Se cobra cuando la decisión cambia. Si el cambio no era probable, se pagó y no se cobró.

Dos cosas más del artículo

Encuentra un error en su propio diseño

- Su módulo de rotaciones prometía un orden.
- Nadie lo necesitaba.
- La interfaz reveló más de lo necesario: impide una implementación que ya las entregue ordenadas.

Jerarquía no es lo mismo que buena división

- La relación “usa a” de su segunda versión no tiene vueltas.
- Se puede cortar arriba y lo de abajo sigue sirviendo.
- Si los de abajo usan a los de arriba, “nivel” deja de significar algo.

Lo segundo es el problema de los ciclos: Sesión B.

SECCIÓN 03

Cohesión y acoplamiento

03

Ocultamiento de información

Interfaz

Todo lo que los demás módulos pueden suponer sobre este. Lo que no está en la interfaz se puede cambiar sin avisar.

- Es más que las funciones exportadas: tipos, errores, orden de las llamadas, si es rápido o lento.
- La prueba: para cada decisión interna, *si mañana cambia, ¿algún otro módulo tiene que cambiar?*

Tres maneras de no esconder nada

Exponer sus tablas · exponer sus tipos internos · exponer su orden de llamadas

En Tinkuy: el reporte de ventas

tinkuy/src/reports/sales-report.ts

```
1  import { findEvent } from '../catalog/events.ts';
2  import { withClient } from '../db.ts';
3  import { notFound } from '../errors.ts';
4  import { ORDER_STATUS } from '../orders/order-repository.ts';
```

- Entra a dos archivos internos, sin pasar por el `index.ts` de cada módulo.
- Más abajo hace sus propios `JOIN` sobre `sections`, `seats`, `order_items` y `orders`.
- `reports` conoce el esquema de las tablas de `orders`, su constante de “pagada” y un archivo de `catalog`. Tres decisiones ajenas que ya no se pueden cambiar sin abrirlo.

Cohesión

> **Cohesión** ***(cohesion)***
> Un módulo es cohesivo cuando sus partes cambian juntas y por la misma razón.

Señales de que falta

- El nombre no dice de qué se ocupa: `utils`, `common`, `helpers`, `manager`.
- Los cambios tocan solo una mitad.
- Quienes lo usan usan pedazos distintos.
- Para describirlo hay que decir “y”.

Parnas dice qué separar (decisiones distintas); la cohesión dice qué dejar junto.

Error común: dividir por capa técnica y llamarlo modular

División	“Las órdenes llevan un código de descuento” toca
`controllers/`, `services/`, `repositories/`	Las tres carpetas
`orders/`, `catalog/`, `payments/`	Una: `orders/`

- Es la descomposición por pasos con otros nombres: recibir, procesar, guardar.
- Las capas no están mal: responden a *otro* cambio (otra base de datos, otro protocolo), que llega menos veces.
- Se combinan: módulos por tema y, dentro de cada uno, capas.

Acoplamiento: importa de qué se depende

Acoplamiento *(coupling)*

Cuánto debe saber un módulo de otro. Están acoplados si un cambio en uno puede obligar a cambiar el otro.

`reports` depende de…	Lo rompe	Costo
Una función pública y estable	Que cambie su firma	Bajo
Un tipo interno de `orders`	Reorganizar los tipos	Medio
El esquema de sus tablas	Renombrar una columna	Alto
El orden en que hace las cosas	Nada falla: cambian los números	Alto

En un diagrama, las cuatro son la misma flecha.

Connascence: un vocabulario

Dos elementos tienen connascence si un cambio en uno exige un cambio en el otro.

Estáticas (se ven en el código)

De nombre	cómo se llama
De tipo	qué tipo tiene
De significado	2 quiere decir visitante
De posición	el orden de los argumentos
De algoritmo	la misma fórmula en dos lados

Dinámicas (solo al ejecutar)

De ejecución	el orden de las llamadas
De tiempo	cuándo ocurre cada cosa
De valor	valores que cambian juntos
De identidad	que sea el mismo objeto

Fuerza, localidad y grado

De arriba abajo, más fuerte: más difícil de encontrar y corregir. Cuanto más lejos están dos elementos, más débil debe ser lo que los ata.

De posición y de significado a de nombre

coupling/connascence.ts

```ts
export function lateFeeByPosition(dueOn: string, returnedOn: string, category: number): number
```

coupling/connascence.test.ts

```ts
assert.equal(lateFeeByPosition('2026-03-16', '2026-03-20', 1), 200);
assert.equal(lateFeeByPosition('2026-03-20', '2026-03-16', 1), 0);
```

coupling/connascence.ts

```ts
export type MemberCategory = 'student' | 'visitor';
export function lateFeeCents(loan: { dueOn: string; returnedOn: string; category: MemberCategory
    }): number
```

En Tinkuy: ¿cuántos módulos toca un medio de pago nuevo?

```
grep -rn "=== 'card'\|=== 'yape'" src tools

src/payments/gateway-client.ts:12:  if (payment.method === 'card') {
src/payments/gateway-client.ts:14:  } else if (payment.method === 'yape') {
src/orders/validation.ts:46:  if (method === 'card') {
src/orders/validation.ts:55:  } else if (method === 'yape') {
src/orders/fees.ts:9:  if (method === 'card') {
src/orders/fees.ts:11:  } else if (method === 'yape') {
tools/fake-gateway.ts:50:  if (method === 'card') {
tools/fake-gateway.ts:55:  if (method === 'yape') {
```

- El mismo `if` en tres archivos de dos módulos, más los tipos de `payments/types.ts`.
- El escenario dice “un módulo”. El sistema dice dos. Esa distancia es P1.
- El escenario se comprueba contando, antes de discutir ningún diseño.

Error común: más módulos no es menos acoplamiento

Error común

Creer que dividir en más módulos reduce el acoplamiento.

- Dividir no quita dependencias: convierte dependencias internas en dependencias entre módulos.
- Si las dos mitades siguen cambiando juntas, solo se agregó una frontera que cruzar.
- La prueba está en el historial: dos módulos que aparecen juntos en casi todos los commits son uno.
- El KWIC por pasos tenía cinco módulos y el otro seis. La diferencia nunca estuvo en la cantidad.

SECCIÓN 04

El grafo de dependencias

04

De los imports al grafo

Grafo de dependencias

Un nodo por módulo y una flecha de A a B si A importa algo de B.

- “Si B cambia, A puede tener que cambiar.”
- No se dibuja: se calcula.
- El de la pizarra es lo que el equipo cree; el de los imports, lo que el sistema es.

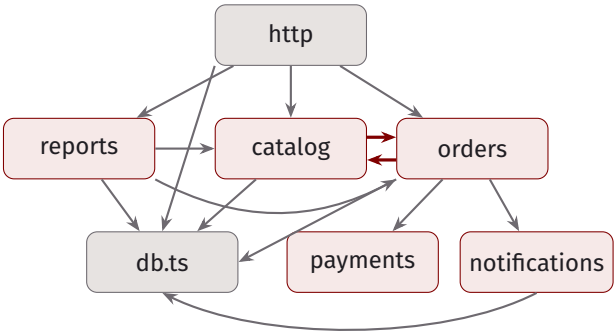
make graph

```
src/adapters → src/loans
src/adapters → src/members
src/loans → src/members
src/main.ts → src/adapters
src/main.ts → src/loans
src/main.ts → src/members
```

el comando

```
depcruise src --config .dependency-cruiser.cjs --include-only '^src' --collapse '^src/[^/]+' --output-type text
```

El grafo de módulos de Tinkuy



`make tinkuy-graph`: 30 flechas. Aquí, sin `config`, `errors` ni `log`.

- Cinco llegan a `db.ts`.
- `catalog` → `orders` y `orders` → `catalog`.

Ciclos

Ciclo

Siguiendo las flechas se puede salir de un módulo y volver a él.

Si A y B se necesitan mutuamente:

- No se pueden entender por separado.
- No se pueden probar por separado.
- No se pueden reutilizar ni desplegar por separado. En P4 un módulo de Tinkuy sale del monolito.

En la práctica

Dos módulos en ciclo son un solo módulo con dos nombres. Y no se anuncian: en TypeScript un ciclo suele funcionar al ejecutarse.

Dirección: estabilidad

La regla

Lo que cambia mucho depende de lo que cambia poco, no al revés.

- Un módulo del que muchos dependen es caro de cambiar.
- Conviene que sea el más general: reglas e interfaces, no detalles.
- Inestabilidad: $I = C_e/(C_a + C_e)$.

`make metrics`, carpetas de la biblioteca

	C_a	C_e	I
`src/main.ts`	0	9	100 %
`src/adapters`	7	6	46 %
`src/loans`	5	1	17 %
`src/members`	4	0	0 %

“Inestable” no es “malo”: es donde un cambio no molesta a nadie. El problema es algo que cambia mucho *y* tiene muchos dependientes.

SECCIÓN 05

Inversión de dependencias

05



Un ejemplo pequeño: la biblioteca

- `members`: los socios y su carné.
- `loans`: prestar y devolver. Máximo tres libros, 14 días, nada con un préstamo vencido.
- `lend(member, copyCode)`: `loans` → `members`.

Pedido nuevo: el carné muestra cuántos libros tiene el socio.



errors/cycle/src/members/member-desk.ts

```ts
import type { LoanDesk } from '../loans/index.ts';

export function createMemberDesk(deps: {
  members: MemberRepository;
  activeLoans: LoanDesk;
}): MemberDesk {
  // ...
  async card(memberId) {
    const member = await find(memberId);
    return { member, activeLoans:
      await deps.activeLoans.countActiveLoans(
      memberId) };
  },
```

La herramienta lo encuentra

```
make errors

== errors/cycle

  error no-circular: src/loans/index.ts →
      src/loans/loan-desk.ts →
      src/members/index.ts →
      src/members/member-desk.ts →
      src/loans/index.ts

x 1 dependency violations (1 errors, 0 warnings). 18 modules, 37 dependencies cruised.

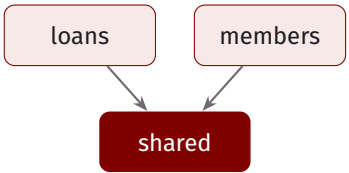
depcruise terminó con código 1
```

- Los dos imports son `import type`. El ciclo existe igual.
- Para probar que el carné muestra un número hay que construir un mostrador de préstamos completo.

Cuatro maneras de romper un ciclo

	Cuándo	Cuesta
1. Fusionar los módulos	Siempre cambian juntos	Un módulo con dos responsabilidades
2. Subir la función que cruza	La operación necesita a los dos y no es de ninguno	Ya no está donde uno la buscaría
3. Mover lo común a `shared`	Lo común es independiente y cambia poco	Casi siempre esconde el problema
4. Invertir una dependencia	El de abajo necesita la operación de verdad	Una interfaz y un lugar que conecta

Error común: el módulo `shared`



la consulta sobre la tabla de préstamos vive aquí

- El ciclo desaparece del grafo. La herramienta deja de quejarse.
- Pero `shared` contiene una decisión interna de `loans`.
- El acoplamiento es el mismo, y ya no se ve.
- `shared`: todos dependen de él y cambia cada vez que cambia cualquiera. Lo contrario de la regla de estabilidad.

Señal de alarma

Mover algo a `shared` *para que la herramienta pase.*

La inversión, paso a paso

1. ¿Qué flecha sobra? `members` es el estable: no debería conocer a `loans`.
2. `members` declara lo que necesita, en su carpeta y con sus palabras.
3. Usa esa interfaz en vez del otro módulo.
4. `LoanDesk` ya la cumple: tipos estructurales.
5. `main.ts` los conecta: `activeLoans: loanDesk`.

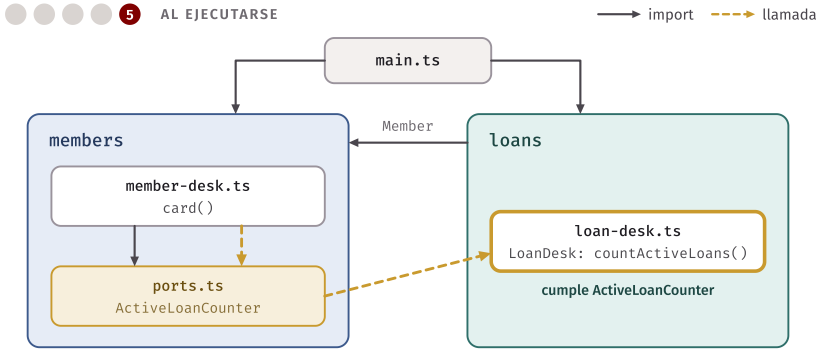
library/src/members/ports.ts

```ts
export interface ActiveLoanCounter {
  countActiveLoans(memberId: number): Promise<number>;
}
```

library/src/members/member-desk.ts

```ts
import type { ActiveLoanCounter, MemberRepository }
  from './ports.ts';
export function createMemberDesk(deps: {
  members: MemberRepository;
  activeLoans: ActiveLoanCounter }): MemberDesk {
```

Import hacia adentro, llamada hacia afuera



card() sigue terminando en el código de loans: la llamada no cambió.

El import apunta hacia members; la llamada sale de members.

Import hacia adentro, llamada hacia afuera

- La llamada no cambió de dirección; el `import` sí.
- La interfaz le pertenece a quien la usa. Se llama `ActiveLoanCounter`, no `LoanDeskLike`.
- La prueba del carné: `activeLoans: { countActiveLoans: async () => 2 }`.

SECCIÓN 06

Ports y adapters

06

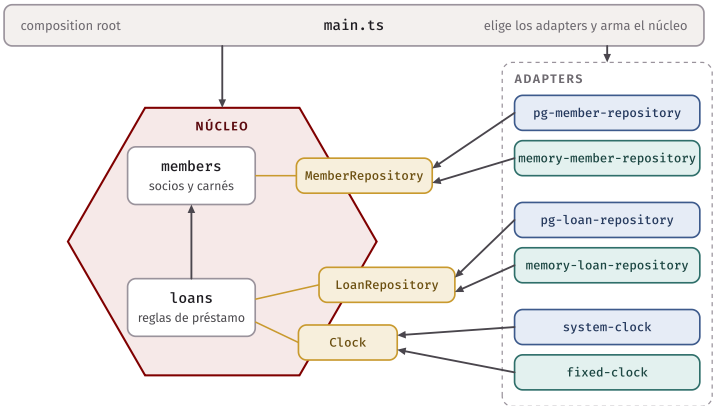


La idea de Cockburn (2005)

> **Intención del patrón**
>
> *Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.*

- La asimetría que importa no es arriba y abajo: es adentro y afuera.
- **Port**: una conversación con propósito entre el núcleo y algo de afuera. Una interfaz.
- **Adapter**: traduce entre un port y una tecnología. Varios por port; uno en memoria.
- El hexágono no es por el seis. ¿Cuántos ports? Pocos: dos, tres o cuatro.

La biblioteca como hexágono



Las flechas son imports: los adapters y main.ts conocen al núcleo; el núcleo no importa nada de afuera.

Los dos ports del núcleo de préstamos

library/src/loans/ports.ts

```ts
export interface LoanRepository {
  add(loan: NewLoan): Promise<Loan>;
  activeByCopy(copyCode: string): Promise<Loan | undefined>;
  activeByMember(memberId: number): Promise<Loan[]>;
  markReturned(loanId: number, returnedOn: Day): Promise<void>;
}
export interface Clock { today(): Day }
```

- No mencionan tablas, SQL ni conexiones. Tienen las operaciones que las reglas necesitan, ni una más.
- ¿Una interfaz para `new Date()`? Es el port que más rinde: sin él, probar “préstamo vencido” es esperar quince días.

El núcleo: reglas y nada más

library/src/loans/loan-desk.ts

```
async lend(member, copyCode) {
  if (member.status !== 'active') throw new LoanError('MEMBER_SUSPENDED', ...);
  if (await deps.loans.activeByCopy(copyCode)) throw new LoanError('COPY_ON_LOAN', ...);
  const today = deps.clock.today();
  const current = await deps.loans.activeByMember(member.id);
  if (current.some((loan) => loan.dueOn < today)) throw new LoanError('HAS_OVERDUE', ...);
  if (current.length >= MAX_ACTIVE_LOANS) throw new LoanError('LOAN_LIMIT', ...);
  return deps.loans.add({ memberId: member.id, copyCode, lentOn: today, dueOn: dueDay(today)
  });
},
```

- Las cuatro reglas, de corrido (aquí, abreviadas). Ni una línea de SQL, ningún `import` de `pg`.
- Lo que se sabe de la tabla `loans` está en un solo archivo: `pg-loan-repository.ts`.

Composition root y dependency injection

library/src/main.ts

```ts
function chooseAdapters(inMemory: boolean): Adapters {
  if (inMemory) {
    return { loans: createMemoryLoanRepository(), members: createMemoryMemberRepository(),
             clock: createFixedClock('2026-03-02'), close: async () => {} };
  }
  const pool = createPool();
  return { loans: createPgLoanRepository(pool), members: createPgMemberRepository(pool),
           clock: systemClock, close: () => pool.end() };
}
```

- **Dependency injection**: un objeto no crea lo que necesita; lo recibe.
- **Composition root**: el único archivo que conoce a todos. Sin reglas del negocio.
- No hace falta un framework: parámetros y un archivo que arma todo, que el compilador revisa.

Probar el núcleo sin base de datos

library/test/loan-desk.test.ts

```
1  test('un préstamo vence 14 días después, según el reloj que recibe el núcleo', async
       () => {
2    const { loanDesk, memberDesk } = memoryLibrary(); // el reloj marca 2026-03-02
3    const rosa = await memberDesk.register('Rosa Quispe');
4
5    const loan = await loanDesk.lend(rosa, 'YAW-001');
6
7    assert.equal(loan.dueOn, '2026-03-16');
8  });
```

- No crea una base de datos, no carga un esquema, no borra tablas, no choca con otra prueba.
- No depende del día en que se corre. Para un préstamo vencido: `clock.set('2026-03-17')`.

Cuánto tarda

```
make timing

200 préstamos con consulta de carné y devolución
  adapters en memoria:     1.8 ms
  adapters de PostgreSQL:  147.0 ms
  PostgreSQL tarda 84 veces más
```

En una laptop con Apple Silicon, dentro de Docker. En otras dos corridas: 54 y 81 veces.

- El número exacto cambia. El orden de magnitud se sostiene: cada operación cruza un socket.
- Con tres pruebas no se nota: 95 ms contra 122 ms por archivo; casi todo es arrancar Node.
- El costo mayor no es el tiempo: base de datos, esquema, vaciar tablas, un archivo a la vez.

> **Idea clave**
>
> No elimina las pruebas con base de datos: las reduce a las que prueban el adapter.

El contrato de un port: ¿los dos adapters se comportan igual?

library/test/repository-contract.ts

```ts
export function loanRepositoryContract(adapter: string, setup: ContractSetup): void {
  test(`${adapter}: un préstamo devuelto deja de estar vigente`, async () => {
    const { loans, memberId } = await setup();
    const saved = await loans.add({ memberId, copyCode: 'YAW-001', lentOn: '2026-03-02', dueOn: '
      2026-03-16' });
    await loans.markReturned(saved.id, '2026-03-10');
    assert.equal(await loans.activeByCopy('YAW-001'), undefined);
  });
```

make test

```
✓ memoria: un préstamo devuelto deja de estar vigente (0.1235ms)
✓ PostgreSQL: un préstamo devuelto deja de estar vigente (6.510292ms)
```

- Las mismas comprobaciones contra cada adapter. Límite: lo que a alguien se le ocurrió escribir.

Trade-off: cuándo un port no vale lo que cuesta

Se justifica si hay

- Un cambio probable del otro lado
- Una frontera que se quiere proteger
- Pruebas que lo aprovechan

No se justifica si

- Tiene una sola implementación
- Nadie espera otra
- Ninguna prueba usa un sustituto

> **Trade-off**
>
> Cada interfaz es indirección que alguien tiene que leer: un archivo, un adapter, una línea en el composition root. Eso se paga siempre; lo que se gana depende de que el port se use.

Schuchert: en software que va a vivir poco, depender directamente puede ser lo mejor.

SECCIÓN 07

Reglas que se comprueban

07

La interfaz pública: un `index.ts`

library/src/loans/index.ts

```ts
export { createLoanDesk } from './loan-desk.ts';
export type { LoanDesk } from './loan-desk.ts';
export type { Clock, LoanRepository } from './ports.ts';
export { LoanError } from './types.ts';
export type { Day, Loan, LoanErrorCode, NewLoan } from './types.ts';
```

- La interfaz del módulo, en una pantalla. `policy.ts` (14 días, tres libros) no aparece.
- "A un módulo se entra solo por su `index.ts`" vuelve comprobable el ocultamiento de información.
- Error común: `export * from` de todo. Cumple la regla y no esconde nada.

Cockburn, sobre las capas: se promete no cruzarlas y, sin un mecanismo que detecte la violación, años después están cruzadas.

Tres clases de reglas

library/.dependency-cruiser.cjs

```
const CORE = '^src/(loans|members)/';

{ name: 'no-circular',
  from: {},
  to: { circular: true } },

{ name: 'core-does-not-touch-infrastructure',
  from: { path: CORE },
  to: { path: ['^pg$', 'node_modules/pg/', '^src/adapters/'] } },

{ name: 'modules-only-through-index',
  from: { path: '^src/([^/]+)' },       // $1: la carpeta de quien importa
  to: { path: CORE, pathNot: ['^src/$1/', '/index\\.ts$'] } },
```

- `from`: quién está sujeto. `to`: qué no puede importar. `pathNot`: las excepciones.
- `tsPreCompilationDeps: true`: los `import type` también cuentan.

Las otras dos maneras de romperlas

```
make errors

== errors/driver-in-core

  error core-does-not-touch-infrastructure: src/loans/loan-desk.ts → ../../../../../../../opt/course/node_modules/pg/lib/index.js

x 1 dependency violations (1 errors, 0 warnings). 18 modules, 37 dependencies cruised.

depcruise terminó con código 1
== errors/deep-import

  error modules-only-through-index: src/adapters/overdue-notice.ts → src/loans/policy.ts

x 1 dependency violations (1 errors, 0 warnings). 19 modules, 38 dependencies cruised.

depcruise terminó con código 1
```

- Una consulta “rápida” escrita dentro del núcleo. Un adapter que necesitaba el plazo y lo tomó de un archivo interno.
- Las dos tienen el mismo origen: era el camino más corto.

Las cuatro violaciones de Tinkuy

```
make tinkuy-deps

  error no-circular: src/catalog/events.ts ⟶
      src/orders/index.ts ⟶
      src/orders/create-order.ts ⟶
      src/catalog/index.ts ⟶
      src/catalog/events.ts
  error modules-only-through-index: src/reports/sales-report.ts ⟶ src/orders/order-repository.ts
  error modules-only-through-index: src/reports/sales-report.ts ⟶ src/catalog/events.ts
  error http-does-not-touch-db: src/http/order-handlers.ts ⟶ src/db.ts

x 4 dependency violations (4 errors, 0 warnings). 35 modules, 86 dependencies cruised.
```

- Un ciclo, dos entradas a archivos internos y un handler que hace su propio SQL.
- Ninguna impide que Tinkuy funcione: `make check` pasa. Por eso `make deps` aún no es parte de `make check`.
- Una deja de violarse en P0. Desde P1, las reglas corren en cada build.

Escribir una regla nueva: verla fallar

1. **Escribirla.** Primero el `to` (qué se protege), después el `from` (quién no puede tocarlo).
2. **Comprobar que pasa** con el código como está.
3. **Comprobar que falla**: hacer el import prohibido a propósito y leer el mensaje.
4. **Quitar el import** y volver a correr.

El paso que más se omite

Una expresión regular mal escrita no coincide con nada, y una regla que no coincide con nada pasa siempre.

Lo que la herramienta no ve

Una regla comprobada se revisa en cada cambio y dice el archivo y la línea. Es la primera fitness function del curso (semana 15). Pero solo ve imports:

- Dos módulos que leen la misma tabla, cada uno con su SQL: ninguna flecha.
- El acoplamiento que pasa por un módulo permitido (`shared`).
- Si el `index.ts` exporta demasiado.
- Si la división es la correcta: el primer KWIC cumpliría todas sus reglas.
- Connascence dinámica: orden y momento no dejan rastro en los imports.

Idea clave

La regla comprueba que se respete un límite. Decidir dónde va el límite sigue siendo diseño: qué cambio se espera y cuántos módulos debería tocar.

En la industria: Shopify y Packwerk

- Un monolito grande en Ruby on Rails, ya ordenado en carpetas por dominio.
- No bastó: las llamadas seguían cruzando. No había paredes, solo “un acuerdo social fácil de romper”.
- Packwerk, análisis estático, comprueba dos límites entre paquetes:
  - de dependencia: un paquete solo usa a los que declaró;
  - de privacidad: desde afuera solo se usa lo que el paquete publica.

Maple Ong, *Enforcing Modularity in Rails Apps with Packwerk*, Shopify Engineering, 23 de septiembre de 2020.

SECCIÓN 08

Laboratorio

08

Laboratorio: reglas de dependencia y un ciclo que romper

En parejas, sobre su copia de Tinkuy, con el material de `code/week-03/`.

1. `make deps`: leer cada violación. ¿Qué cambio futuro la volvería un problema?
2. Leer `.dependency-cruiser.cjs`. Sacar el grafo como texto y compararlo con el de la semana 1.
3. Seguir el ciclo entre `catalog` y `orders`. ¿Cuál de las dos dependencias sobra, y por qué?
4. Romperlo con una inversión de dependencias. `make deps`: una violación menos. `make test`: todo pasa.
5. Biblioteca: completar `lab/loan-desk.test.ts` sin base de datos. `make timing`, tres veces.
6. Una regla nueva: solo `src/payments` importa el cliente HTTP de la pasarela. Verla fallar.
7. Corregir una violación de `modules-only-through-index` sin exportar todo.

> **Antes de escribir código en el paso 4**
>
> Repasen las cuatro maneras de romper un ciclo y digan por qué eligen esa.

Para la próxima clase

- Terminar el laboratorio. **P0** se entrega esta semana; se publica **P1 · Módulos con límites**.
- Leer: *Fundamentals of Software Architecture* (2.ª ed.), cap. 9 a 13; Fowler, *PresentationDomainDataLayering* y *MonolithFirst*.
- Semana 4: estilos monolíticos. Capas, modular monolith, pipeline y microkernel.

> **Pregunta**
>
> En el grafo de Tinkuy, ¿qué módulo es el más caro de cambiar hoy? ¿Es el que debería serlo?