

Arquitectura de Software

Estilos monolíticos

Capas · Modular monolith · Pipeline · Microkernel

Gerson Garrido

Semana 4

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A

1. Qué es un estilo
2. Capas
3. Por rol técnico o por dominio
4. Modular monolith

Sesión B

5. Pipeline
6. Microkernel
7. Combinar estilos · Big ball of mud

Lecturas

A: Richards y Ford, *Fundamentals of Software Architecture* (2.ª ed.), caps. 9, 10 y 11; Fowler, *PresentationDomainDataLayering* y *MonolithFirst*.

B: *Fundamentals*, caps. 12 y 13; AOSA vol. 1, *Eclipse*; Foote y Yoder, *Big Ball of Mud* (*Introduction* y *Forces*).

Complementarias: Westeinde, *Deconstructing the Monolith* (Shopify); *Software Architecture in Practice*, cap. 8; AOSA, *Graphite*; Parnas (1972).

De una flecha a la forma del sistema

- Semana 3: una dependencia. ¿Este módulo debería importar a aquel?
- Hoy: la forma del sistema completo. ¿Con qué criterio se divide? ¿Qué puede hablar con qué?
- El hilo: el escenario de modificabilidad. Cuando llega un cambio, ¿cuántos archivos, módulos y personas toca?

Cómo trabajamos hoy

Una aplicación pequeña, **Qhatu**, escrita de dos maneras. Le aplicamos el mismo cambio a las dos y contamos.

Tinkuy aparece como está: arreglarlo es P1.

SECCIÓN 01

Qué es un estilo

01

Estilo de arquitectura

Estilo de arquitectura *(architectural style)*

Un vocabulario de tipos de **elementos** y de **conectores**, más un conjunto de **restricciones** sobre cómo se combinan.

- "Cada capa usa solo la de abajo": la presentación no consulta la base de datos.
- "Un filtro solo conoce su entrada y su salida": un filtro no llama a otro.
- "El núcleo no importa a ningún plug-in": no hay `if` por variante.

Las propiedades salen de las restricciones

Un estilo que no se hace cumplir no da nada: cada estilo de hoy viene con sus reglas.

No es una receta

- Un estilo es un punto de partida con propiedades conocidas: qué va a ser fácil y qué va a doler.
- No dice cuántos módulos hacer ni cómo llamarlos.
- Elegir un estilo es elegir qué atributos se favorecen y cuáles se sacrifican.

Estilo y patrón

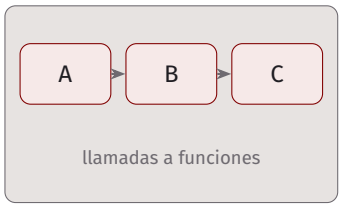
Estilo: la forma global del sistema o de una parte grande.

Patrón: resuelve un problema local dentro de esa forma (un repositorio, un registro, un circuit breaker).

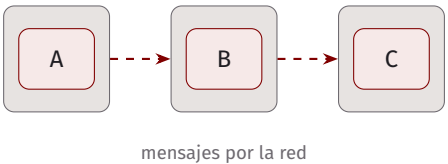
Muchos autores los usan como sinónimos.

La primera gran división

Monolítico: una unidad de despliegue



Distribuido: varias unidades



- La pregunta es sobre el despliegue, no sobre el orden interno.
- *Monolito* dice cuántas cosas se despliegan. *Big ball of mud* dice que nadie cuidó los límites. Son ejes distintos.

Lo que un monolito da y lo que cobra

Da

- Una base de código, un build.
- Llamadas locales: rápidas, y terminan o lanzan un error.
- Transacciones simples sobre una base de datos.
- Un solo despliegue.

Cobra

Todo se despliega junto.

Una parte lenta afecta a todas.

Se escala entero.

Sin disciplina, los límites internos se borran.

Los tres primeros costos se pagan cuando llega un escenario (de despliegue, de disponibilidad, de escalabilidad). El cuarto se paga desde el primer mes.

La transacción que viene gratis

shop/test/behaviour.ts

```
test('si un producto no alcanza, la orden entera no se guarda', async () => {
  // Del libro 8 queda un ejemplar; del libro 2 hay de sobra.
  const rejected = await post('/orders', {
    buyerEmail: 'luis.mamani@correo.pe',
    items: [
      { productId: 2, quantity: 1 },
      { productId: 8, quantity: 2 },
    ],
  });
  assert.equal(rejected.status, 409);
  assert.equal(rejected.body.error.code, 'OUT_OF_STOCK');
  // El libro 2 ya se había descontado dentro de la transacción: el ROLLBACK lo devuelve.
  assert.equal(await stockOf(2), 8);
  assert.equal(await stockOf(8), 1);
});
```

SECCIÓN 02

Capas

02



El ejemplo de la semana: Qhatu

Una librería en línea, en `code/week-04/shop/`.

- `GET /products?category=`
- `POST /orders` y `GET /orders/:id`
- `GET /reports/sales`

Escrita dos veces: `by-layer/` y `by-domain/`.

Las mismas pruebas

Las dos versiones pasan las mismas seis pruebas, que solo usan la API HTTP.

Un comprador no puede distinguirlas. Cambia la estructura de módulos y, con ella, el costo de cada cambio.

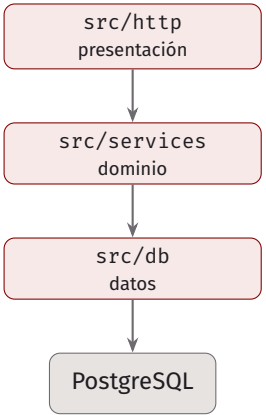
Capas: la regla

Arquitectura en capas

El sistema se divide por **rol técnico**.
Cada capa usa solo la que tiene debajo.

- **Presentación**: traduce HTTP y JSON.
- **Dominio**: las reglas del negocio.
- **Datos**: el SQL, y solo aquí.

Cada capa esconde una tecnología.



Capas cerradas y capas abiertas

- **Cerrada**: no se puede saltar. Aísla los cambios; cuesta funciones que solo pasan la llamada.
- **Abierta**: se permite saltarla. Ahorra esas funciones; las reglas de negocio pierden su lugar único.
- Si se abre, se abre a propósito: se escribe y se cambia la regla.

Un handler que llama directo al repositorio, y la regla de capas cerradas:

```
make cruces

== errors/handler-skips-service.ts en Qhatu por capas

  error http-only-uses-services: src/http/handler-skips-service.ts → src/db/product-repository.ts

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

depcruise terminó con código 1
```

El sumidero

src/services/product-service.ts

```ts
export async function listProducts(category
    ?: Category): Promise<Product[]> {
  return findProducts(category);
}
```

Sumidero *(architecture sinkhole)*

Peticiones que atraviesan las capas sin que ninguna les haga nada.

En Qhatu por capas, de las cuatro funciones del dominio, una pasa la llamada tal cual, dos agregan una línea y solo `placeOrder` tiene reglas.

Error común

Resolverlo abriendo todas las capas: las reglas de negocio terminan repartidas entre handlers y servicios.

Capas: balance

Hace bien

- Simple y conocida.
- Separa lo técnico: una tecnología, una carpeta.
- Reduce lo que hay que tener en la cabeza (Fowler).
- El dominio se prueba sin HTTP.

Hace mal

Parte por rol técnico, y los cambios que más llegan son de negocio.

Un campo nuevo se recibe, se valida y se guarda: **tres capas**.

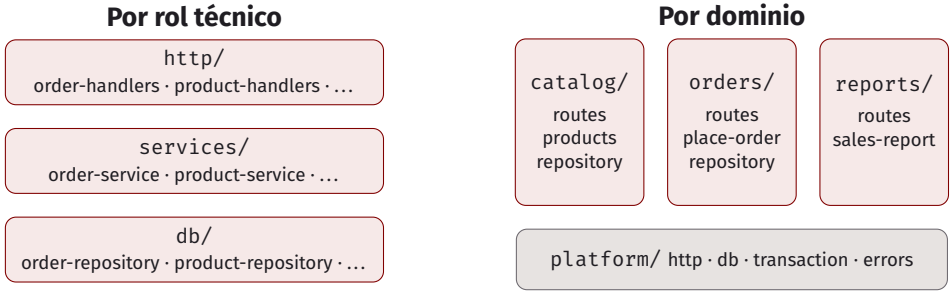
Sumideros: funciones que solo pasan la llamada.

SECCIÓN 03

Por rol técnico o por dominio

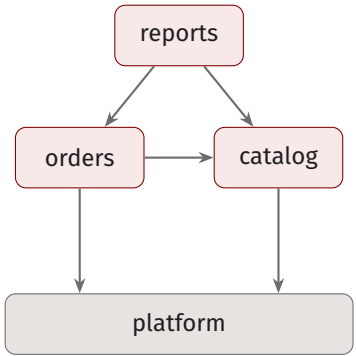
03

La misma aplicación, dos divisiones



Las mismas piezas, agrupadas con otro criterio. Partir por dominio no elimina las capas: las mete adentro de cada módulo.

El grafo ahora habla del negocio



- `orders` → `catalog`: para vender hay que saber qué hay.
- `reports` → `orders`, `reports` → `catalog`: el reporte combina los dos.
- Ninguna flecha en sentido contrario: el catálogo no sabe que existen las órdenes.
- Por capas, el grafo dice solo “`http` usa a `services`”: verdad en cualquier sistema.

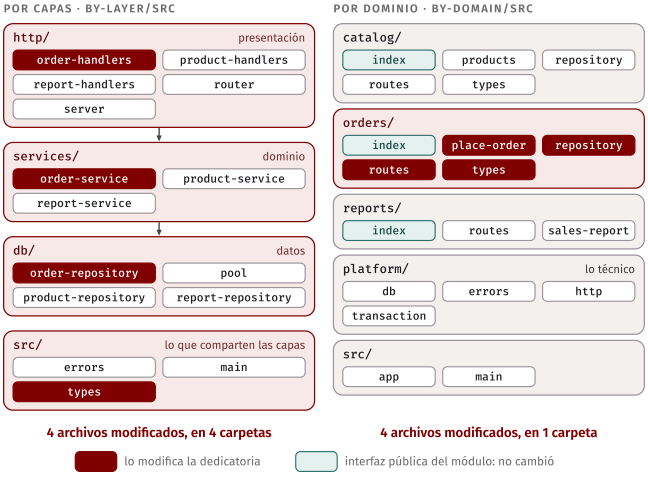
El mismo cambio, contado

“El comprador puede escribir una dedicatoria de hasta 140 caracteres, que se guarda con la orden y se muestra.” El mismo cambio, hecho en las dos versiones:

```
make cambio

== Por capas
 src/db/order-repository.ts    |   24 ++++++++++++++++++++--------
 src/http/order-handlers.ts    |    5 ++++-
 src/services/order-service.ts |    6 +++++-
 src/types.ts                  |    2 ++
 4 files changed, 27 insertions(+), 10 deletions(-)
== Por dominio
 src/orders/place-order.ts |    6 +++++-
 src/orders/repository.ts  |   19 ++++++++++++++------
 src/orders/routes.ts      |    5 ++++-
 src/orders/types.ts       |    2 ++
 4 files changed, 24 insertions(+), 8 deletions(-)
```

La dedicatoria en las dos Qhatu



Cuatro archivos y cuatro archivos

	Por capas	**Por dominio**
Archivos modificados	4	4
Carpetas tocadas	4 (`src/`, `http/`, `services/`, `db/`)	1 (`orders/`)
¿Cambió una interfaz pública?	No hay interfaces	No: `orders/index.ts` intacto

- Partir por dominio no reduce el código que hay que escribir.
- Cambia dónde cae: si cada carpeta tiene un owner, es un equipo o son tres.
- Fowler: los desarrolladores no tienen que ser full-stack, pero los equipos sí.

El cambio contrario

```
make sql

== Por capas: archivos con SQL
src/db/order-repository.ts
src/db/product-repository.ts
src/db/report-repository.ts
== Por dominio: archivos con SQL
src/catalog/repository.ts
src/orders/repository.ts
```

- Un cambio técnico (revisar todo el SQL) cae al revés.
- Por capas: una carpeta, de quien más sabe de bases de datos.
- Por dominio: un archivo por cada módulo con datos. Con cuarenta módulos, cuarenta owners.

Trade-off

Ninguna división vuelve baratos todos los cambios. Se elige cuál abaratar.

La pregunta que decide

¿Qué tipo de cambio llega más seguido?

¿Uno que afecta a una parte del negocio a través de todas sus capas, o uno que afecta a una tecnología a través de todo el negocio?

- En los sistemas de negocio, casi siempre el primero: “aceptar Plin”, “vender abonos”. Por eso se parte primero por dominio.
- Excepciones razonables:
  - un sistema pequeño con un equipo que cabe en una mesa;
  - un sistema cuyo cambio dominante sí es técnico;
  - un equipo que todavía no entiende el dominio.

SECCIÓN 04

Modular monolith

04



Tres condiciones que se pueden comprobar

Modular monolith

Un solo despliegue, dividido en módulos por dominio que cumplen:

1. cada módulo tiene una **interfaz pública**, y los demás lo usan solo por ella;
2. los **datos** de cada módulo son suyos: nadie más lee sus tablas con SQL propio;
3. las **dependencias** entre módulos están declaradas y no tienen ciclos.

Da buena parte de la modificabilidad de los microservicios sin la red; se sigue desplegando y escalando junto.

Error común

Llamar “modular” a un monolito porque tiene carpetas.

La interfaz pública y sus reglas

shop/by-domain/src/catalog/index.ts

```ts
// Catálogo: los libros, sus precios y su stock.
// Esto es todo lo que otro módulo puede usar de catalog.

export { getProducts, takeStock } from './products.ts';
export { catalogRoutes } from './routes.ts';
export { CATEGORIES } from './types.ts';
export type { Category, Product } from './types.ts';
```

Seis reglas de dependency-cruiser. Dos ya las conocen: `no-circular` y `modules-only-through-index`.

- `platform-does-not-know-modules` y `only-platform-db-uses-pg`: lo técnico no depende del negocio.
- `only-repository-queries` y `reports-has-no-data`: quién puede llegar a las tablas.

Los datos como límite

shop/db/schema-modules.sql

```sql
CREATE TABLE orders.order_lines (
  order_id         integer NOT NULL REFERENCES orders.orders (id),
  product_id       integer NOT NULL,
  title            text    NOT NULL,
  quantity         integer NOT NULL CHECK (quantity > 0),
  unit_price_cents integer NOT NULL CHECK (unit_price_cents > 0),
  PRIMARY KEY (order_id, product_id)
);
```

- Un esquema de PostgreSQL por módulo: `catalog.products`, `orders.orders`, `orders.order_lines`.
- `product_id` no tiene llave foránea hacia `catalog`.
- Se gana: cada módulo cambia sus tablas solo. Se paga: una garantía pasa de la base de datos al código.

Lo que dependency-cruiser no ve

Dos módulos que comparten una tabla no tienen ningún `import` entre ellos. `shop/data-ownership.ts` busca en el SQL de cada módulo tablas de otro esquema.

```
make cruces

== errors/report-with-join.ts en Qhatu por dominio

  error reports-has-no-data: src/reports/report-with-join.ts ⟶ src/platform/db.ts
  error only-repository-queries: src/reports/report-with-join.ts ⟶ src/platform/db.ts

x 2 dependency violations (2 errors, 0 warnings). 22 modules, 46 dependencies cruised.

depcruise terminó con código 2
src/reports/report-with-join.ts: el módulo reports nombra la tabla orders.order_lines, que no es suya
src/reports/report-with-join.ts: el módulo reports nombra la tabla catalog.products, que no es suya
2 tabla(s) ajena(s)
data-ownership terminó con código 1
```

El `JOIN` que cruza dos módulos

El reporte necesita lo vendido por producto (`orders`) y la categoría de cada producto (`catalog`).

Opción	Se gana	Se paga
Pedir por la interfaz	Cero acoplamiento por el esquema	Más consultas; combinar en memoria
Duplicar un dato estable	Leer sin depender del otro módulo	La copia puede quedar vieja
Vista de lectura declarada	El `JOIN` y su rendimiento	Una interfaz en SQL; ata los módulos a una misma base

Qhatu usa la primera para el reporte y la segunda para el título y el precio de cada línea: la orden debe mostrar el precio al que se vendió.

Cómo se comunican los módulos

Llamada a la interfaz

`orders` llama a `takeStock(tx, ...)` de `catalog`, dentro de su transacción.

Cuando quien llama **necesita el resultado** para seguir.

Trade-off: pasar `tx` solo tiene sentido mientras compartan base de datos.

Evento dentro del proceso

`orders` publica `OrderPlaced`; los interesados se suscriben. Quien publica no importa a quien escucha.

Cuando solo **avisa** de algo que ya pasó.

Cuesta: ¿quién se entera si un suscriptor falla? El evento vive en memoria: si el proceso se cae, se pierde.

Tinkuy: un monolito partido a medias

tinkuy: make deps

```
depcruise src --config .dependency-cruiser.cjs

  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.

make: *** [Makefile:51: deps] Error 4
```

Fallan la condición 3 (un ciclo) y la 1 (`reports` entra a archivos internos). Y la 2, sin que la herramienta lo vea: nueve tablas en un esquema, y el SQL de `reports` hace `JOIN` con tablas de dos módulos. P1: volverlo un modular monolith con límites comprobados.

Monolith first

Fowler (2015), sobre lo que veía en equipos reales:

- Casi todos los casos exitosos de microservicios empezaron como un monolito que se partió.
- Acertar los límites al comienzo es muy difícil; se descubren con el uso.
- Mover un límite entre módulos: un commit. Entre servicios: una migración y despliegues coordinados.
- Su advertencia: no cualquier monolito se puede partir después. El que sí, es el que mantuvo sus límites.

K. Westeinde, *Deconstructing the Monolith*, Shopify Engineering

En la industria: Shopify (2019)

De carpetas por concepto de software (modelos, vistas, controladores) a carpetas por concepto del negocio. Cada componente con interfaz pública y como owner exclusivo de sus datos. Una herramienta propia, Wedge, lista las violaciones en cada integración.

SECCIÓN 05

Pipeline

05

Pipes y filtros



Pipeline *(pipes and filters)*

Filtros que hacen una sola cosa con los datos y **pipes** que los llevan, en una dirección, de uno al siguiente. Un filtro no sabe nada de los demás: solo conoce el formato de lo que entra y de lo que sale.

- Los comandos de Unix y la barra `|`; un compilador; una carga de datos.
- **Verificador** *(tester)*: deja pasar un dato o lo descarta.

Ya lo usaban

code/week-04

```
$ tail -n +2 pipeline/data/supplier-books.csv | grep -v -e '^#' -e '^$' | cut -d';' -f3 | tr 'A-Z' 'a-z' | sort | uniq -c
      1 cuento
      3 ensayo
      5 novela
      3 poesia
```

- Seis programas que no se conocen entre sí resuelven “cuántos títulos hay por categoría”.
- Funciona porque todos respetan el mismo formato: líneas de texto.
- Y no valida nada: cuenta la fila repetida, la del campo sobrante y una categoría que la tienda no vende.

La lista de precios del distribuidor

pipeline/src/main.ts

```ts
const summary = await from(readLines(file))
  .through(parseRows(';'))
  .through(onlyValid((rejection) => rejected.
      push(rejection)))
  .through(toBook)
  .through(dropDuplicates((book) => duplicated
      .push(book.isbn)))
  .into(summarize);
```

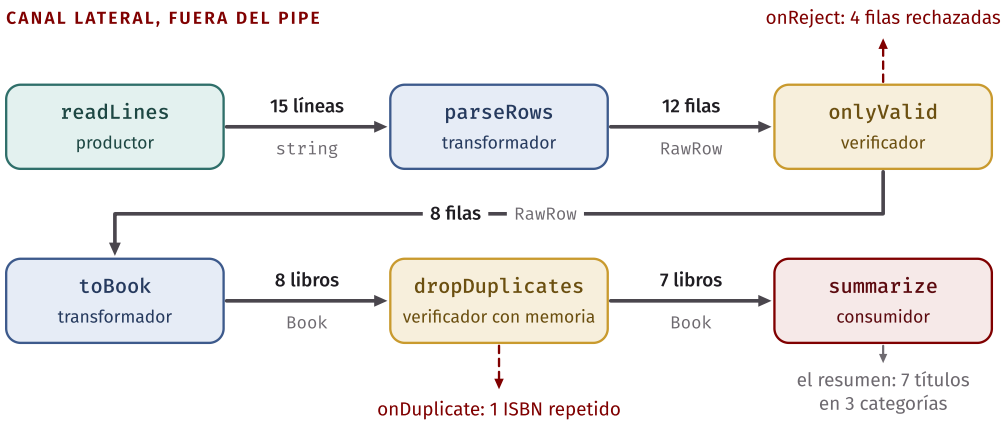
Un filtro es una función que recibe una secuencia y devuelve otra. El compilador exige que la salida de uno sea la entrada del siguiente.

node pipeline/src/main.ts

```
categoría  títulos  ejemplares  valor del stock
novela           3          10        S/ 446.00
poesia           2          12        S/ 347.70
ensayo           2           9        S/ 330.00

Filas rechazadas: 4
  línea 9: falta precio
  línea 12: stock no válido: -1
  línea 14: categoría desconocida: cuento
  línea 15: tiene 1 campo(s) de más
ISBN repetidos: 978-612-0002
```

Lo que pasa por cada pipe



Se prueban, se reemplazan y se reordenan

- **Se prueba solo**: a `toBook` se le da un arreglo de filas hecho a mano. Sin archivo, sin pipeline.
- **Se reemplaza**: otro productor (HTTP en vez de archivo), otro consumidor (cargar en vez de resumir).
- **Se reordena… con cuidado**:
  - `onlyInStock` y `onlyCategory` miran cada libro por separado: da igual cuál va primero.
  - `dropDuplicates` tiene memoria: “la primera aparición” depende de quién filtró antes.

Error común

Creer que cualquier filtro se puede mover a cualquier posición. El lugar de un filtro con memoria es una decisión, y debería estar en una prueba.

El formato es el acoplamiento

pipeline/

```
$ grep -rln "priceCents" src | sort
src/consumers/summarize.ts
src/filters/to-book.ts
src/formats.ts
```

Si el precio pasa a ser un monto con moneda, cambian tres archivos que no se conocen entre sí.

- Los filtros no se conocen, pero todos conocen el formato.
- Parnas (1972): su primera descomposición del KWIC **es un pipeline**, y un cambio en la representación tocaba todos los módulos.

Trade-off

Conviene cuando el formato es estable y los pasos cambian. No conviene cuando lo que cambia es la forma de los datos.

Dónde no sirve

- **Ida y vuelta**: los datos van en una dirección. Comprar una entrada no es un pipeline.
- **Errores**: ¿adónde va un dato rechazado? En el ejemplo, a un canal lateral (`onReject`).
- **Filtros que necesitan ver todo**: ordenar guarda todo en memoria antes de entregar.

En Tinkuy

Tienen forma de pipeline la **validación** de una orden (una cadena de verificadores) y el **reporte** de ventas (una cadena de transformaciones).

La compra completa no: tiene una llamada de ida y vuelta a la pasarela.

SECCIÓN 06

Microkernel

06



El problema: un `if` por variante

Qhatu entrega de dos formas: recojo en tienda (`pickup`) y envío a domicilio (`courier`).

microkernel/before/dispatch-note.ts

```ts
export function dispatchNote(method: MethodId, destination: Destination): string {
  if (method === 'pickup') {
    return `RECOJO EN TIENDA · ${STORES[destination.store ?? '']}`;
  } else if (method === 'courier') {
    return `DOMICILIO · ${(destination.address ?? '').trim()}, ${(destination.district ?? '').trim()}`;
  }
  throw new Error('Forma de entrega desconocida');
}
```

- El mismo `if` está en `validate-destination.ts` y en `shipping-cost.ts`; la lista de formas válidas, en `types.ts`.
- Para dos formas de entrega, es razonable.

Llega la tercera

Qhatu quiere entregar en casilleros (`locker`). El cambio, sobre la versión con `if`:

```
make tercer-plugin

== Con ifs (before/): archivos existentes que cambian
 before/dispatch-note.ts        |    2 ++
 before/shipping-cost.ts        |    2 ++
 before/types.ts                |    5 ++++-
 before/validate-destination.ts |    4 +++-
 4 files changed, 11 insertions(+), 2 deletions(-)
```

- Hay que saber que son cuatro. Se encuentran buscando `'courier'`.
- Los cuatro son del núcleo: quien agrega casilleros puede romper el envío a domicilio.
- Dos personas, dos formas nuevas: las mismas líneas de los mismos archivos.

En Tinkuy: los medios de pago

```
tinkuy: grep -rn "'card'\|'yape'" src

src/payments/types.ts:1:export type PaymentMethod = 'card' | 'yape';
src/payments/types.ts:3:export const PAYMENT_METHODS: readonly PaymentMethod[] = ['card', 'yape'];
src/payments/types.ts:7:  | { method: 'card'; cardNumber: string; holder: string }
src/payments/types.ts:8:  | { method: 'yape'; phone: string; approvalCode: string };
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') {
```

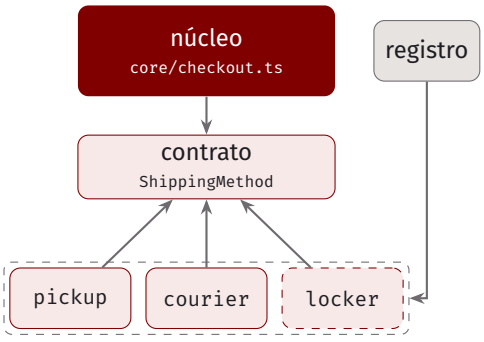
Cuatro archivos en dos módulos. Es el paso 1 del laboratorio, y la misma forma que `before/`.

Núcleo, contrato y registro

Microkernel *(plug-in)*

Un **núcleo** pequeño y estable con lo común, y **plug-ins** que agregan variantes.

- **Contrato**: la interfaz que el núcleo define y todo plug-in implementa.
- **Registro**: donde los plug-ins se inscriben y el núcleo los busca.
- El núcleo no conoce a ningún plug-in concreto.



El contrato

microkernel/src/core/shipping-method.ts

```ts
export interface ShippingMethod {
  /** Identificador estable, en minúsculas: es lo que llega en el pedido. */
  readonly id: string;
  /** El nombre que ve el comprador. */
  readonly label: string;

  /** Los problemas del destino, en español; una lista vacía quiere decir que es válido. */
  validate(destination: Destination): string[];

  /** Costo y plazo. El núcleo solo la llama con un destino que pasó validate. */
  quote(parcel: Parcel, destination: Destination): Quote;

  /** La línea que se imprime en la etiqueta del paquete. Igual: solo con un destino válido. */
  dispatchNote(destination: Destination): string;
}
```

Una operación por cada `if` de la versión anterior.

Un plug-in

microkernel/src/plugins/pickup.ts

```ts
const pickup: ShippingMethod = {
  id: 'pickup',
  label: 'Recojo en tienda',

  validate(destination) {
    const store = destination.store ?? '';
    return store in STORES ? [] : [`store debe ser una de: ${Object.keys(STORES).join(', ')}`];
  },

  quote() {
    return { costCents: 0, businessDays: 1 };
  },

  dispatchNote(destination) {
    return `RECOJO EN TIENDA · ${STORES[destination.store ?? '']}`;
  },
};
```

El registro

microkernel/src/registry.ts

```ts
import { createRegistry } from './core/method-registry.ts';
import courier from './plugins/courier.ts';
import pickup from './plugins/pickup.ts';

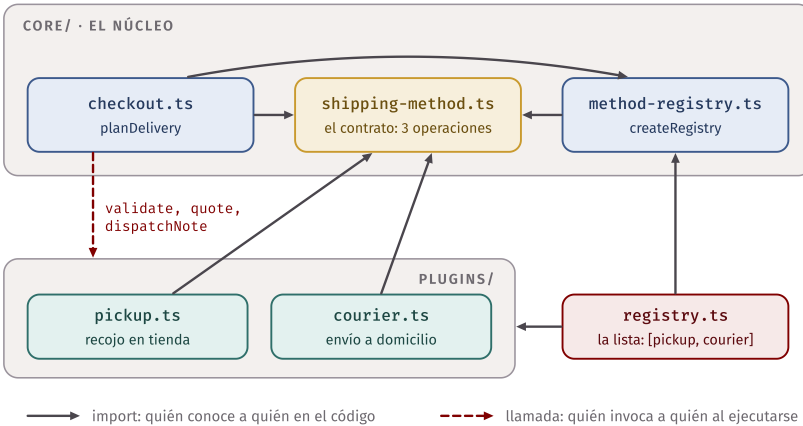
// El registro estático: la lista de formas de entrega que trae la aplicación.
// Es el único archivo que conoce a los plug-ins por su nombre.
export const registry = createRegistry([pickup, courier]);
```

make cotizar

```
Paquete de 2300 g, compra de S/ 97.00
  pickup   Recojo en tienda     envío  S/ 0.00  total  S/ 97.00  1 día(s)  [RECOJO EN TIENDA · Av. Larco 345, Miraflores]
  courier  Envío a domicilio    envío S/ 16.00  total S/ 113.00  3 día(s)  [DOMICILIO · Av. Túpac Amaru 1234, Comas]
```

Una prueba compara esta versión con la de los `if` en 54 pedidos: responden igual.

Quién llama a quién



Quién llama a quién

- Al ejecutarse, el núcleo llama al plug-in.
- En el código, el `import` va al revés: el plug-in importa el contrato.
- Es la inversión de dependencias de la semana 3, convertida en estilo.
- `registry.ts` es el *composition root*: el único que conoce a los dos lados.

Cuando el `if` vuelve a entrar

```
make cruces

== errors/plugin-reads-db.ts y errors/core-knows-plugin.ts en el microkernel

  error plugins-do-no-io: src/plugins/plugin-reads-db.ts → ../../../../../../opt/course/node_modules/pg/lib/index.js
  error only-the-registry-imports-plugins: src/core/core-knows-plugin.ts → src/plugins/courier.ts
  error core-does-not-know-plugins: src/core/core-knows-plugin.ts → src/plugins/courier.ts

x 3 dependency violations (3 errors, 0 warnings). 14 modules, 21 dependencies cruised.

depcruise terminó con código 3
```

- Lo que un plug-in tiene prohibido es tan parte del contrato como sus operaciones.
- “Solo para courier, avisar si tarda más de dos días”: así se deshace un microkernel, un caso especial a la vez.
- La pregunta correcta: ¿qué le falta al contrato para que el núcleo no pregunte quién es el plug-in?

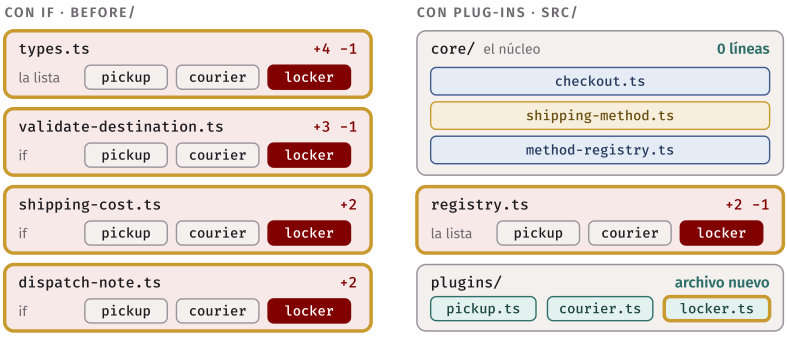
La tercera forma de entrega, con plug-ins

```
make tercer-plugin

== Con plug-ins (src/): un archivo nuevo, y los existentes que cambian
 src/registry.ts |    3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
== La copia con el tercer plug-in
Paquete de 2300 g, compra de S/ 97.00
  pickup   Recojo en tienda     envío  S/ 0.00  total  S/ 97.00  1 día(s)  [RECOJO EN TIENDA · Av. Larco 345, Miraflores]
  courier  Envío a domicilio    envío S/ 16.00  total S/ 113.00  3 día(s)  [DOMICILIO · Av. Túpac Amaru 1234, Comas]
  locker   Casillero            envío  S/ 5.00  total S/ 102.00  2 día(s)  [CASILLERO · LK-0420]
```

Agregar `locker`	Con `if`	Con plug-ins
Archivos nuevos	0	1 (`plugins/locker.ts`)
Archivos existentes modificados	4	1 (`registry.ts`)
Líneas tocadas del núcleo	11 agregadas, 2 quitadas	0

Agregar `locker`, en las dos versiones



El mismo cambio: cuatro archivos existentes modificados, o uno.

Pruebas del contrato

```
make tercer-plugin (final)

✓ locker: tiene un destino válido de ejemplo (0.109042ms)
✓ locker: un destino vacío se rechaza con un mensaje, sin lanzar (0.127667ms)
✓ locker: cotiza un costo entero no negativo y al menos un día (0.0745ms)
✓ locker: no modifica lo que recibe y responde lo mismo dos veces (0.060625ms)
i tests 12
i pass 12
i fail 0
```

- Nadie escribió estas pruebas para `locker`.
- Lo que el núcleo supone de cualquier plug-in se escribe una vez y corre contra todos los registrados.
- El contrato deja de ser un comentario: falla si alguien lo incumple.

Registro estático o descubrimiento

	Estático	Descubrimiento
Archivos existentes que toca un plug-in nuevo	1	0
¿El compilador comprueba el contrato?	Sí	No: se revisa al arrancar
¿Dónde se ve qué plug-ins hay?	En el código	En la carpeta
¿Puede agregar un plug-in alguien sin acceso al núcleo?	No	Sí

El descubrimiento se justifica cuando los plug-ins los escribe otra gente. Si son del mismo equipo y viven en el mismo repositorio, una línea en una lista es un precio bajo por tener al compilador de su lado.

En la industria: Eclipse

- Todo es un plug-in: lo que trae la plataforma y lo que escribe un tercero.
- Cada plug-in declara en un manifiesto qué puntos de extensión ofrece y a cuáles contribuye.
- Al arrancar, la plataforma lee los manifiestos y arma el registro en memoria: descubrimiento.
- El código de un plug-in se carga cuando se usa *(lazy activation)*.

"API is forever"

Otros construyen sobre esos contratos. Al cambiar el runtime en la versión 3.0, hubo que entregar una capa de compatibilidad para los plug-ins existentes.

Kim Moir, *Eclipse*, en *The Architecture of Open Source Applications*, vol. 1

Cómo se diseña un contrato

1. **¿Qué puede pedirle el núcleo?** Una operación por cada `if` que hoy pregunta por la variante.
2. **¿Qué datos cruzan?** Solo lo necesario: cada dato entregado ya no se puede cambiar libremente.
3. **¿Qué le está prohibido?** Aquí: importar el núcleo, otro plug-in, o hacer entrada y salida.
4. **¿Cómo se registra?**

¿Qué cambio lo rompería?

Una forma de entrega no disponible para ciertos paquetes: se agrega una operación opcional, sin romper nada.

Una tarifa que viene de un servicio externo: `quote` pasa a ser asíncrona, y cambian el núcleo y **todos** los plug-ins.

Barato y caro

Idea clave

En un microkernel, agregar un plug-in es barato y cambiar el contrato es caro. El diseño consiste en poner dentro del contrato lo que no va a cambiar y dejar fuera, en los plug-ins, lo que sí.

- Un plug-in se puede reescribir entero sin que nadie se entere.
- Un cambio en el contrato toca el núcleo y todos los plug-ins, incluidos los que escribió otra gente.
- Si un escenario anuncia el cambio, se decide antes de que existan diez plug-ins. Si no, se anota como limitación conocida.

Lo que cuesta un punto de extensión

- **Indirección**: del núcleo al registro, del registro al plug-in.
- Un contrato **difícil de cambiar**.
- Solo ayuda en **el eje elegido**: un cambio que afecta a todas las variantes toca todos los plug-ins.
- Hay que **adivinar el eje** antes de que lleguen las variantes.

Error común

Convertir todo en plug-in “por si acaso”. La extensibilidad se compra para el cambio que un escenario dice que va a llegar, con su medida.

Para dos variantes y ninguna tercera a la vista, el `if` es el diseño correcto.

SECCIÓN 07

Combinar estilos · Big ball of mud

07

Los estilos se combinan

Estilo	Divide por	Hace barato	Hace caro
Capas	Rol técnico	Cambiar una tecnología	Un cambio de negocio
Modular monolith	Dominio, con interfaces y datos propios	Un cambio de negocio; mover un límite	Lo que cruza módulos: reportes, cambios técnicos
Pipeline	Paso del procesamiento	Probar, reemplazar y agregar pasos	Cambiar el formato; ida y vuelta
Microkernel	Núcleo y variantes	Agregar una variante	Cambiar el contrato

Un estilo se elige por parte del sistema y por atributo: un modular monolith cuyo módulo de envíos es un microkernel y cuya carga de datos es un pipeline, con capas dentro de cada módulo.

Big ball of mud

Foote y Yoder (1997)

Un sistema estructurado de cualquier manera, cuya organización la dicta la urgencia más que el diseño. Es la arquitectura que más se despliega.

Fuerzas: tiempo, costo, experiencia, habilidad, visibilidad, complejidad, cambio, escala.

- Nadie decide construirlo. Se llega por muchas decisiones pequeñas, cada una razonable.
- Un `import` interno “solo por esta vez”. Un `JOIN` porque el reporte era para mañana. Un `if` más.
- El límite que se cruzó una vez ya no parece un límite.

Idea clave

Es lo que queda cuando ningún estilo se hace cumplir.

Laboratorio: un medio de pago como plug-in

En parejas, sobre su copia de Tinkuy de la semana 3. El ejemplo de las formas de entrega muestra la técnica, no la respuesta.

1. Encontrar dónde pregunta Tinkuy por el medio de pago. Tabla: archivo, función, qué decide cada `if`.
2. Escribir el escenario de modificabilidad, con su medida, antes de cambiar nada.
3. Estudiar el microkernel de `code/week-04/`: contrato, registro, quién importa a quién.
4. Diseñar `PaymentMethod` como interfaz de TypeScript.
5. Mover `card` y `yape` a plug-ins. `make test` pasa sin cambiar ninguna prueba.
6. Agregar `plin` y contar los archivos existentes modificados.
7. Dos reglas de dependency-cruiser; verlas fallar.
8. ¿Qué cambio obligaría a modificar el contrato? ¿Vale la pena preverlo?

Para la próxima clase

- Terminar el laboratorio: es la segunda mitad de **P1**, que se entrega en la semana 5.
- Semana 5: estilos distribuidos. Leer Richards y Ford, caps. 14 *Service-Based Architecture Style* y 18 *Microservices Architecture*, y Lewis y Fowler, *Microservices* (2014).
- Lecturas de hoy: *Fundamentals of Software Architecture*, caps. 9 a 13; Fowler, *PresentationDomainDataLayering* y *MonolithFirst*; AOSA, *Eclipse*; Foote y Yoder, *Big Ball of Mud*.

Pregunta

Hoy una compra en Qhatu descuenta el stock y guarda la orden en una sola transacción. Si el catálogo fuera otro proceso, en otra máquina, ¿qué puede pasar entre un paso y el otro?