Design patterns as functions

You solve recurring problems in the workshop shop with design patterns as functions and recognize them again in Express, React, and Apollo.

Mike Bild

Exercises: github.com/MikeBild/introduction-design-patterns

Connecting and chaining

Problem first, then the pattern · Ports and Adapters without classes · Facade simplifies, proxy controls · Every step decides whether to continue

What is a pattern?

- A named solution for a recurring problem
- A shared word for code reviews and architecture discussions
- A tool for a problem, never the goal of a change

Why functions instead of classes?

```
// From the textbook: interface, class, method
interface Notifier {
  send(order: Order): Promise<void>
}
class EmailNotifier implements Notifier {
  async send(order: Order) {
    await sendEmail(order.customerId)
  }
}

// In TypeScript: a function is a value
type Notify = (order: Order) => Promise<void>
const notifyByEmail: Notify = (order) => sendEmail(order.customerId)

const placeAndNotify = async (order: Order, notify: Notify) => {
  await save(order)
  await notify(order)
}
```

Ten problems, ten patterns

Connect and chain

- External API → Adapter
- Bundle steps → Facade
- Check access → Proxy
- Chain steps → Chain

Swap and create

- Enrich → Decorator
- Swap → Strategy
- Call as value → Command
- Build → Factory

Share and react

- Share value → Provider
- React → Observer

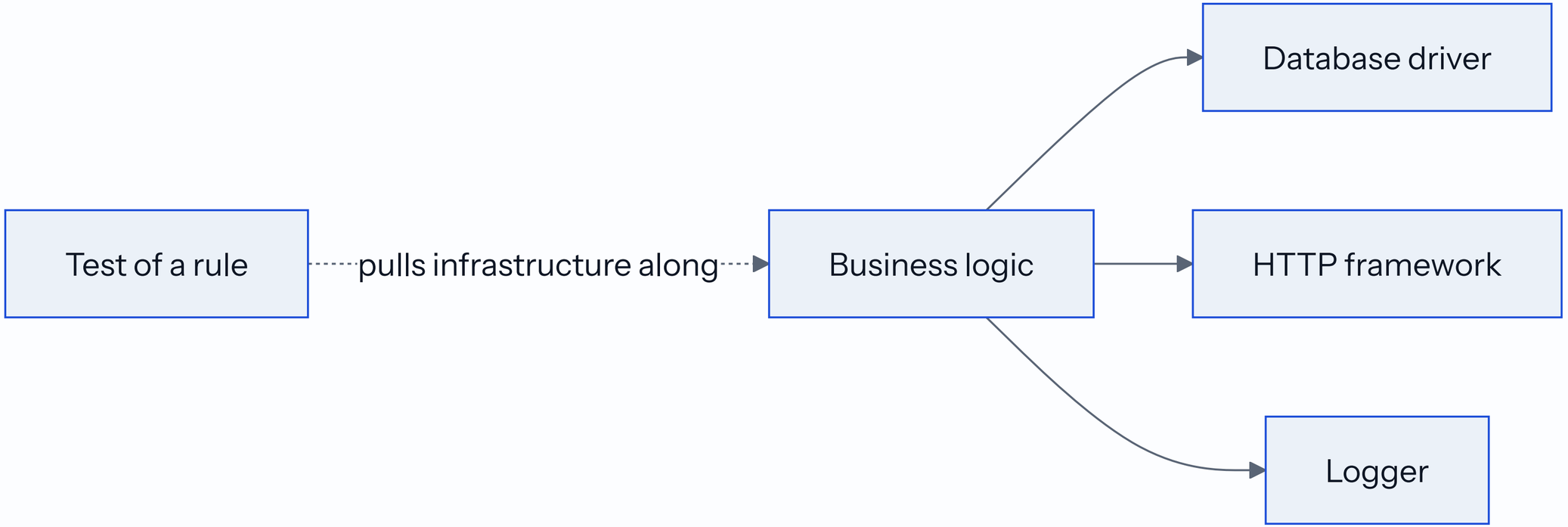
What is deliberately missing

- Singleton, because a module export is already unique
- Visitor, because Union Types with switch cover all cases
- Memento, Flyweight, and Abstract Factory, because they are rarely needed here

You already use patterns

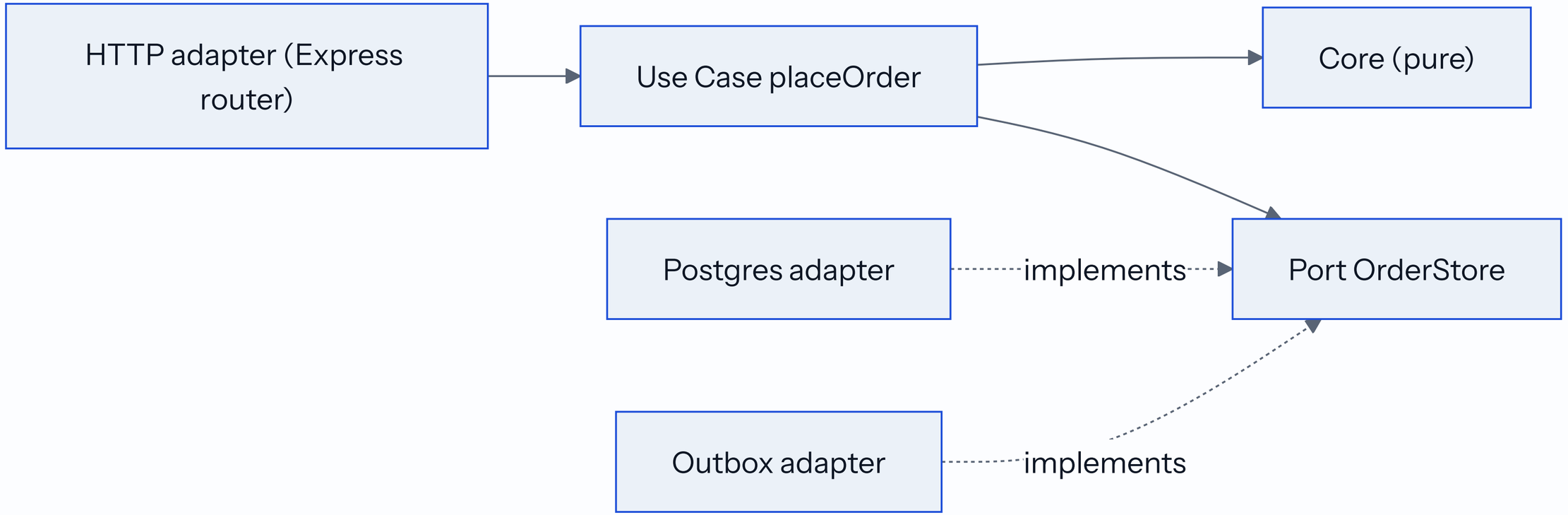
- Express Middleware is a Chain with next()
- React Context is a Provider for a whole tree
- A Postgres store is an Adapter behind a port

When everything depends on everything



Ports and Adapters

- The core defines what it needs: ports
- Adapters fulfill ports with concrete technology
- Dependencies always point inward



Implemented functionally

```ts
// ports.ts
export type OrderStore = {
  loadCatalog: (productIds: string[]) => Promise<Catalog>
  saveOrder: (order: NewOrder) => Promise<Order>
}

// use case
export const createPlaceOrder = (store: OrderStore, clock: () => Date) =>
  async (input: PlaceOrderInput) => {
    const catalog = await store.loadCatalog(input.items.map((item) => item.productId))
    const decision = placeOrder(input, catalog, clock())
    if (decision.kind === 'rejected') return decision
    return { kind: 'placed', order: await store.saveOrder(decision.order) } as const
  }
```

Wire in main

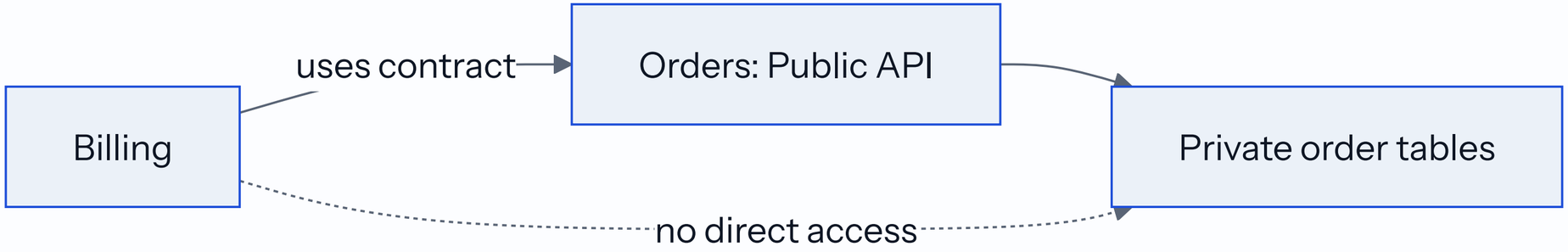
```
// main.ts - Composition Root
const config = loadConfig(process.env)
const logger = createLogger(config.LOG_LEVEL)
const sql = createDatabase(config.DATABASE_URL)

const placeOrder = createPlaceOrder(createPostgresOrderStore(sql), () => new Date())

const app = express()
app.use(express.json({ limit: '100kb' }), requestLogger(logger))
app.use('/orders', createOrderRoutes({ placeOrder }))
app.use(createHealthRoutes(() => shuttingDown, () => pingDatabase(sql)))
app.use(notFound, errorHandler(logger))

const server = app.listen(config.PORT)
```

Modules and trade-offs



Two callers, and one forgets the stock

- Every caller must know the order of the services
- Every copy can lose a step
- If payment fails, the stock stays reserved

```
// Web checkout
if (!(await stock.reserve(items))) return outOfStock
const totalCents = await sumPrices(prices, items)
if (!(await payments.charge(customerId, totalCents))) return declined

// Reorder, copied later: stock.reserve is missing
const totalCents = await sumPrices(prices, items)
if (!(await payments.charge(customerId, totalCents))) return declined
```

A Facade stands in front of three services

```
export const createPlaceOrder =
  ({ stock, prices, payments }: Services) =>
  async (input: OrderInput): Promise<OrderResult> => {
    if (!(await stock.reserve(input.items)))
      return { kind: 'rejected', reason: 'out-of-stock' }

    const totalCents = await sumPrices(prices, input.items)
    if (await payments.charge(input.customerId, totalCents))
      return { kind: 'placed', totalCents }

    await stock.release(input.items)
    return { kind: 'rejected', reason: 'payment-declined' }
  }
```

A Proxy keeps the contract and caches first

```
export const createCachedPrices = (
  prices: PriceService,
): PriceService => {
  const known = new Map<string, number>()

  return {
    priceOf: async (productId) => {
      const cached = known.get(productId)
      if (cached !== undefined) return cached

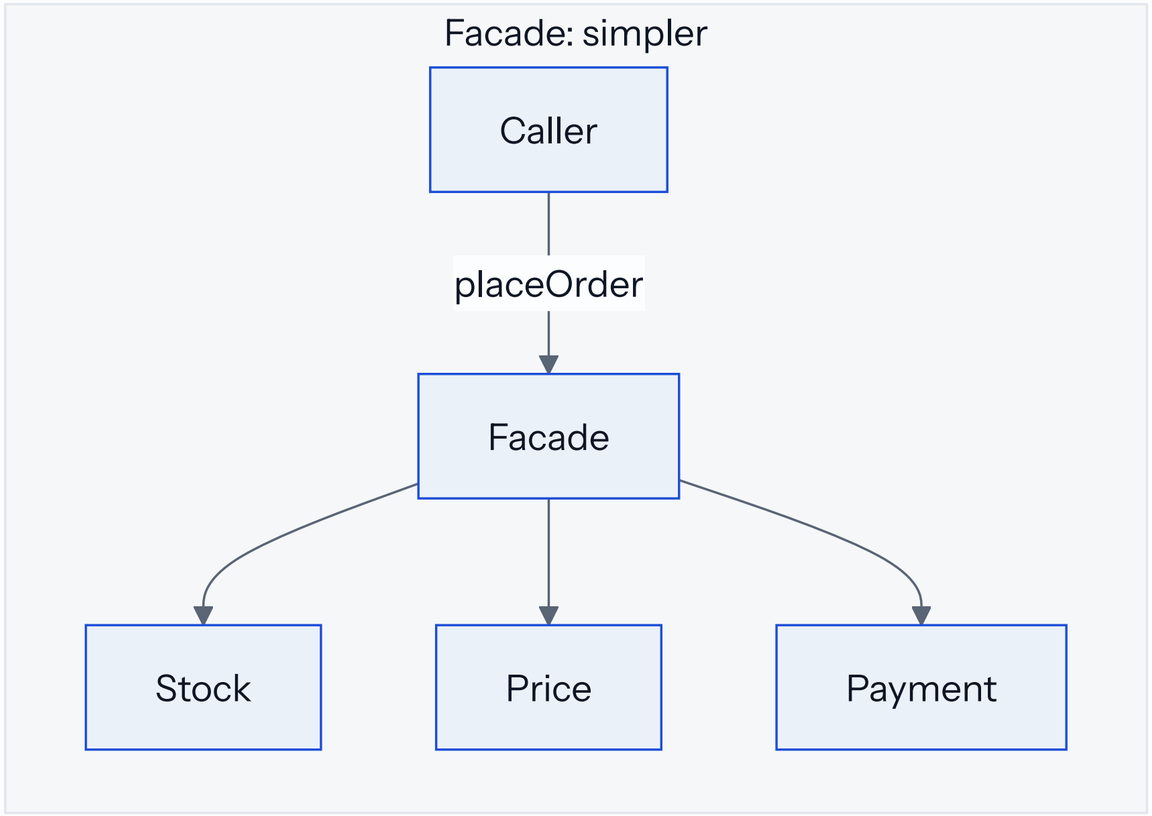
      const price = await prices.priceOf(productId)
      known.set(productId, price)
      return price
    },
  }
}
```

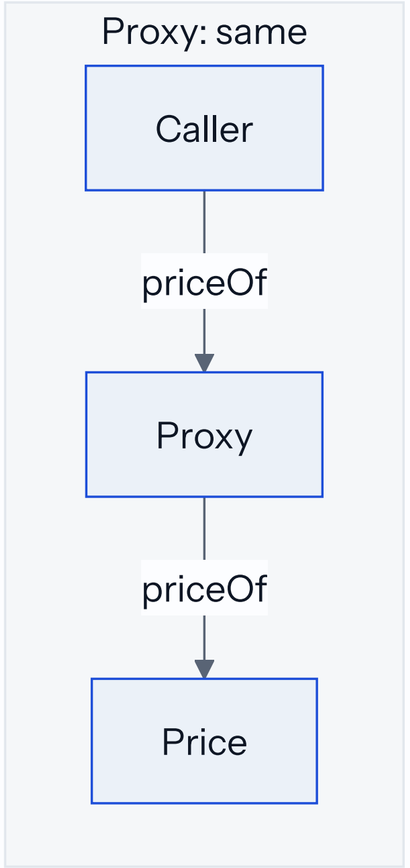
A Proxy can also refuse

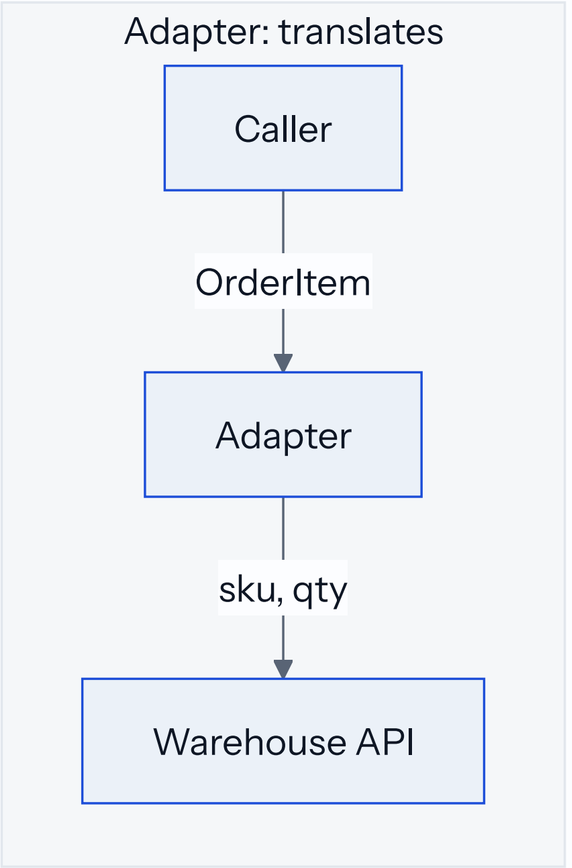
- Cache controls whether the service needs to work at all
- Access protection controls whether the call is allowed
- Both keep the service signature

```
export const createGuardedPayments = (
  payments: PaymentService,
  canPay: (customerId: string) => boolean,
): PaymentService => ({
  charge: async (customerId, amountCents) => {
    if (!canPay(customerId))
      throw new Error('Melde dich mit dem Konto dieser Bestellung an.')
    return payments.charge(customerId, amountCents)
  },
})
```

Facade, Proxy, or Adapter?







Where you already use both

- GraphQL schema as a Facade in front of REST services
- new Proxy as a language feature with the same object shape before an object
- Reverse Proxy with the same HTTP contract in front of your service

A handler that cannot stop

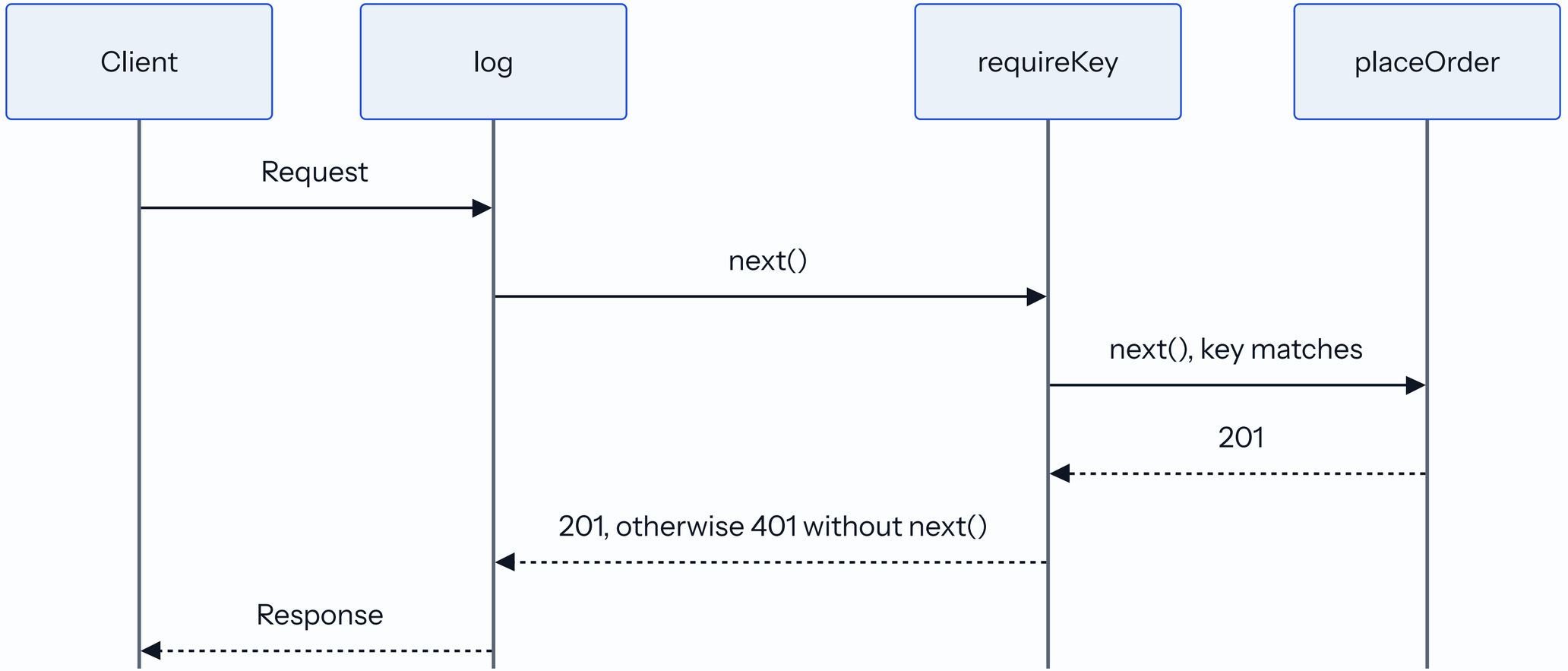
- Invalid key: status 401, but the order is still created
- Invalid order: status 500 as an HTML page
- New route: the check is missing because no one copied it

```
app.post('/orders', (req, res) => {
  if (req.get('x-api-key') !== apiKey) res.status(401)

  const { customerId, items } = req.body
  if (!customerId || !items?.length) throw new Error('invalid')

  const order = placeOrder(customerId, items)
  res.json(order)
})
```

Continue or stop



A chain in a few lines

```ts
type Reply = { status: number; body?: unknown }
type Next = () => Promise<Reply>
type Handler<Req> = (request: Req, next: Next) => Promise<Reply>

const chain =
  <Req>(...handlers: Handler<Req>[]) =>
  (request: Req): Promise<Reply> => {
    const run = async (index: number): Promise<Reply> => {
      const handler = handlers[index]
      return handler
        ? handler(request, () => run(index + 1))
        : { status: 404 }
    }
    return run(0)
  }
```

Express Middleware is the same chain

```
const requireApiKey: RequestHandler = (req, res, next) => {
  if (req.get('x-api-key') === apiKey) return next()
  res.status(401).json({ error: 'Prüf deinen API-Schlüssel.' })
}

const validateOrder: RequestHandler = (req, res, next) => {
  if (req.body.items?.length) return next()
  next(httpError(400, 'Eine Bestellung braucht Positionen.'))
}

ordersRouter.use(requireApiKey)
ordersRouter.post('/', validateOrder, postOrder)
app.use(errorHandler) // vier Parameter: (err, req, res, next)
```

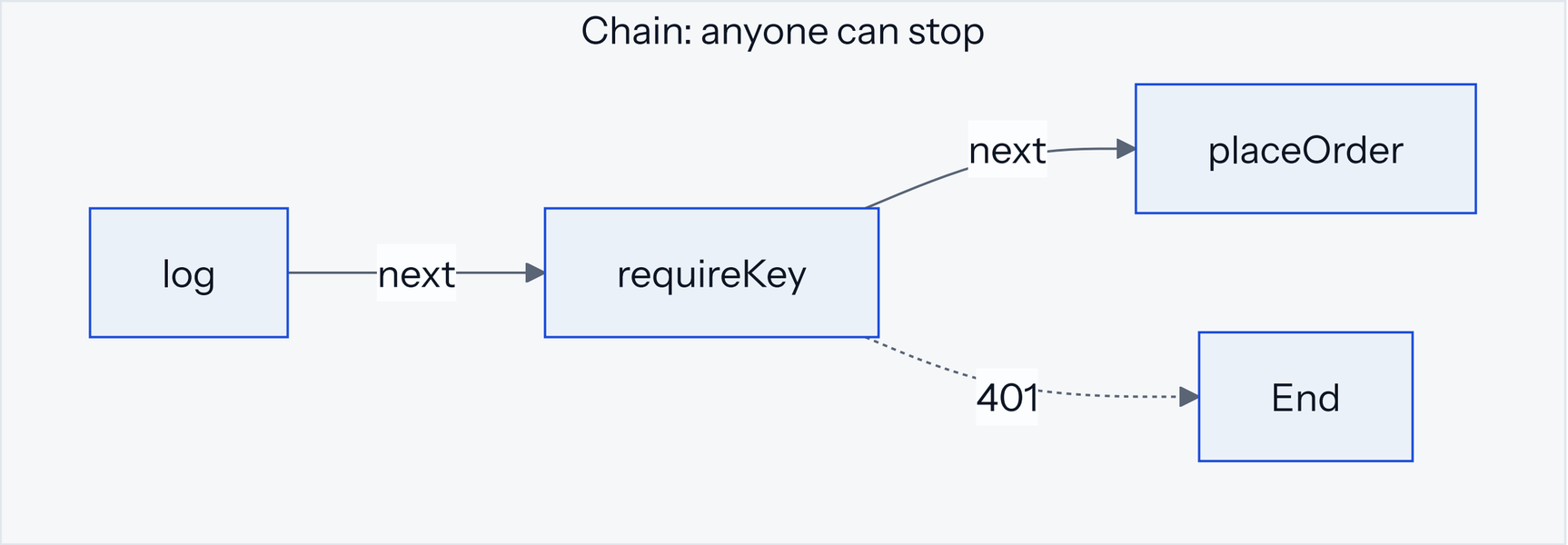
ApolloLink uses forward instead of next

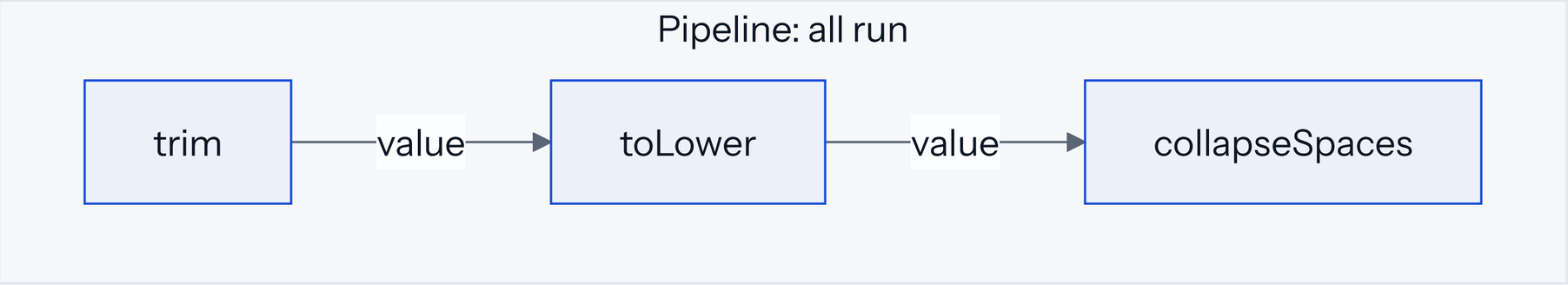
- forward(operation) corresponds to calling next()
- HttpLink is the Terminating Link and does not call forward
- forward returns an Observable; the response flows back through it

```
const logOperation = new ApolloLink((operation, forward) => {
  console.log(operation.operationName)
  return forward(operation)
})

const link = ApolloLink.from([
  logOperation,
  new HttpLink({ uri: '/graphql' }),
])
```

Chain or pipeline?





Swapping and creating

Enrich behavior without touching it · Behavior is an argument · One function builds, the rest uses it

A function with three jobs

- Declined card: it runs three times
- Failed call: it is missing from the measurement
- Second function: it cannot reach retry or measurement

```
export const createChargeOrder = (gateway, record, now) =>
  async (orderId: string, amountCents: number) => {
    const start = now()
    for (let attempt = 1; ; attempt += 1) {
      try {
        const receipt = await gateway(orderId, amountCents)
        record({ name: 'chargeOrder', ms: now() - start })
        return receipt
      } catch (error) {
        if (attempt === 3) throw error
      }
    }
  }
```

Same signature in and out

```
type AsyncFn<Args extends unknown[], Result> =
  (...args: Args) => Promise<Result>

const withTiming =
  <Args extends unknown[], Result>(
    fn: AsyncFn<Args, Result>,
    name: string,
    record: (measure: Measure) => void,
  ): AsyncFn<Args, Result> =>
  async (...args) => {
    const start = performance.now()
    try {
      return await fn(...args)
    } finally {
      record({ name, ms: performance.now() - start })
    }
  }
```

Layers from outside to inside



Every Decorator has the same shape

```
const withCache = <Args extends unknown[], Result>(
  fn: AsyncFn<Args, Result>,
): AsyncFn<Args, Result> => {
  const known = new Map<string, Result>()

  return async (...args) => {
    const key = JSON.stringify(args)
    if (known.has(key)) return known.get(key) as Result

    const result = await fn(...args)
    known.set(key, result)
    return result
  }
}

const loadPrice = withCache(withRetry(priceApi, retry))
```

Decorator or Proxy?

- Proxy controls access to a contract
- Decorator adds behavior around a function
- The code often stays the same, the intent decides the name

Handler wrapper in Express

- An Express handler is a function, so the Decorator fits
- The Promise stays intact, errors reach the error handler
- Express recognizes error handlers by four parameters

```
const postOrder = async (req: Request, res: Response) => {
  res.status(201).json(await placeOrder(req.body))
}

app.post('/orders', withTiming(postOrder, 'postOrder', record))
app.use(errorHandler) // do not wrap

const rest = async (...args: unknown[]) => {}
rest.length // 0, not 4
```

Why does this switch keep growing?

- Every new shipping method needs a new case
- The default swallows typos and gives away shipping
- Special terms end up in the same block

```
export const shippingCents = (order: Order, method: string): number => {
  const subtotal = subtotalCents(order)
  switch (method) {
    case 'standard':
      return subtotal >= 5_000 ? 0 : 490
    case 'express':
      return 990
    case 'pickup':
      return 0
    default:
      return 0
  }
}
```

A Strategy is just a function

```ts
export type ShippingStrategy = (subtotalCents: number) => number

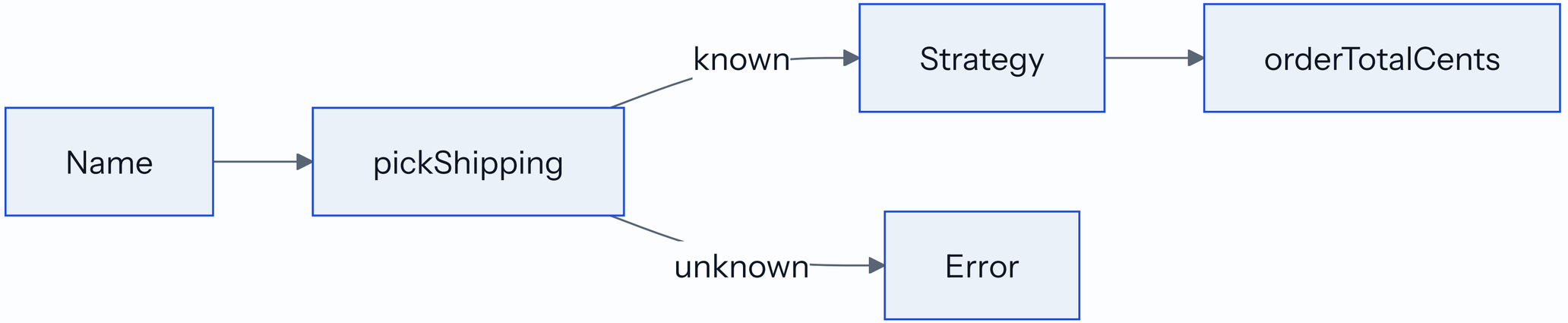
export const standardShipping: ShippingStrategy = (subtotal) =>
  subtotal >= 5_000 ? 0 : 490
export const expressShipping: ShippingStrategy = () => 990
export const pickupShipping: ShippingStrategy = () => 0

export const orderTotalCents = (order: Order, shipping: ShippingStrategy) => {
  const subtotal = subtotalCents(order)
  return subtotal + shipping(subtotal)
}

orderTotalCents(order, expressShipping)
orderTotalCents(order, () => 0) // Special case without a new case
```

A table replaces the switch

- A new shipping method is a new row
- satisfies checks every row against the function type
- Object.hasOwn rejects unknown external names



A Command turns a call into a value

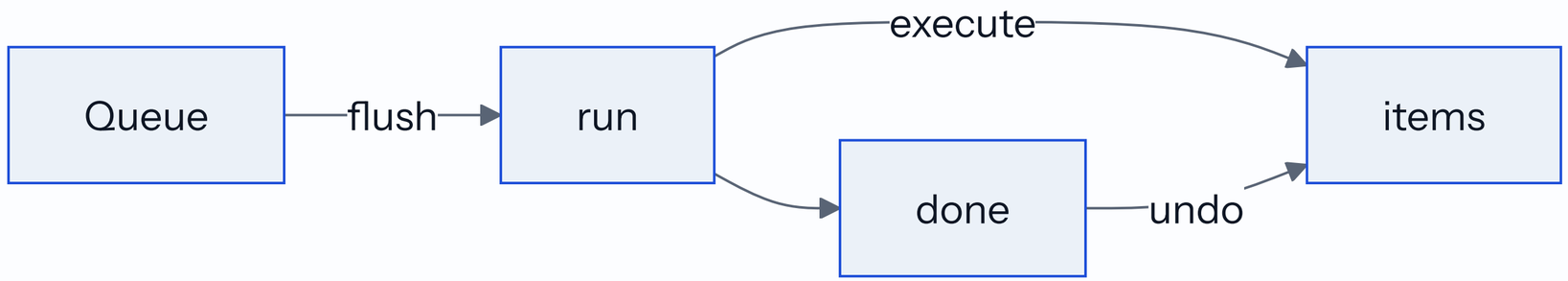
```
export type CartCommand = Readonly<{
  label: string
  execute: (items: Items) => Items
  undo: (items: Items) => Items
}>

export const addItem = (item: OrderItem): CartCommand => {
  let index = -1
  return {
    label: `add ${item.productId}`,
    execute: (items) => {
      index = items.length
      return [...items, item]
    },
    undo: (items) => items.toSpliced(index, 1),
  }
}

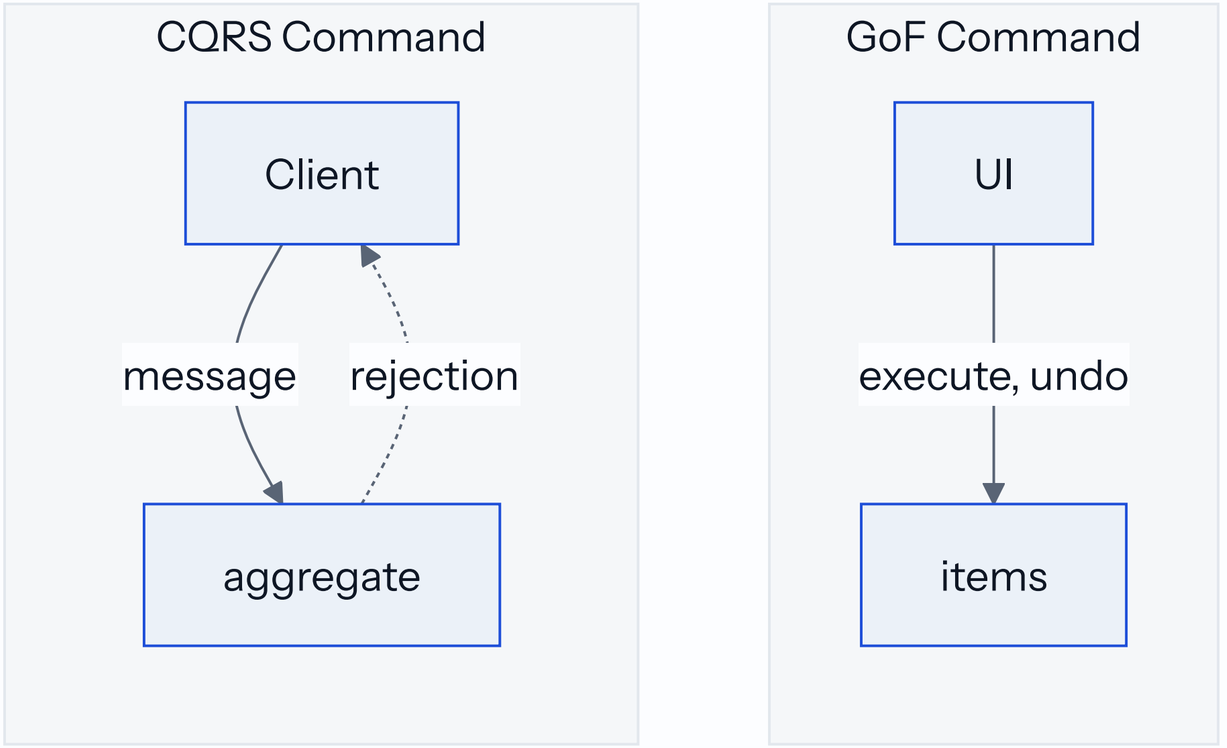
cart.run(addItem(hammer))
cart.undo()
```

What more can a call as a value do?

- Undo: process the list backward
- Queue: enqueue now, execute later
- Log: every change carries a label



A GoF Command is not a CQRS Command



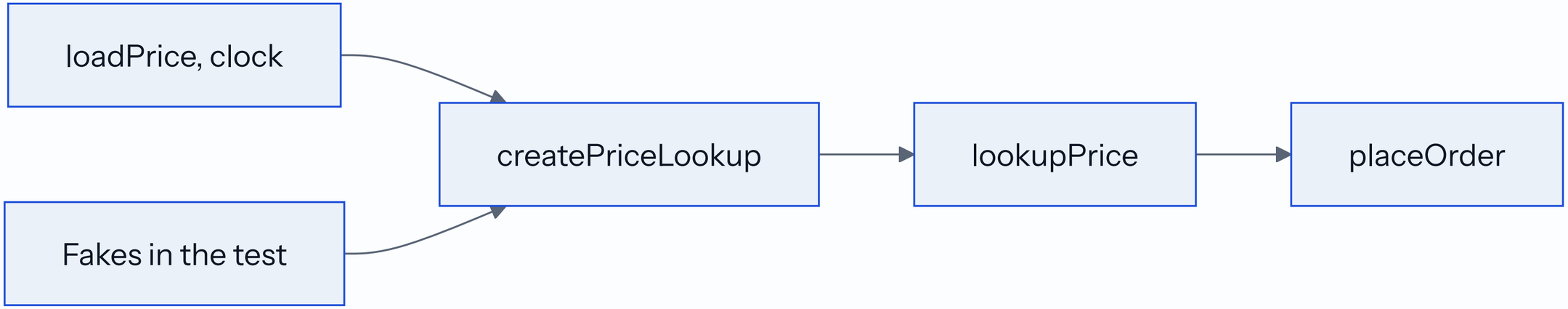
What is wrong with let at module level?

- Every test inherits the previous test's state
- Two shops share one counter
- Clock and data source cannot be swapped

```
const priceCache = new Map<string, number>()
let lastNumber = 1_000

export const placeOrder = (input: OrderInput): PlacedOrder => ({
  number: `WS-${++lastNumber}`,
  totalCents: input.items.reduce(
    (sum, item) => sum + item.quantity * lookupPrice(item.productId),
    0,
  ),
  createdAt: new Date().toISOString(),
})
```

createX gets dependencies and returns behavior



Each instance has its own state

```
export const createOrderNumbers = ({ prefix, start }: {
  prefix: string
  start: number
}) => {
  let last = start
  return () => `${prefix}-${++last}`
}

const workshop = createOrderNumbers({ prefix: 'WS', start: 1_000 })
const outlet = createOrderNumbers({ prefix: 'OUT', start: 0 })

workshop() // 'WS-1001'
workshop() // 'WS-1002'
outlet()   // 'OUT-1'
```

Dependencies are passed as arguments

```ts
export const createPriceLookup = ({ loadPrice, ttlMs, clock }: {
  loadPrice: (productId: string) => number
  ttlMs: number
  clock: () => Date
}) => {
  const cache = new Map<string, { priceCents: number; expiresAt: number }>()
  return (productId: string): number => {
    const now = clock().getTime()
    const cached = cache.get(productId)
    if (cached && cached.expiresAt > now) return cached.priceCents
    const priceCents = loadPrice(productId)
    cache.set(productId, { priceCents, expiresAt: now + ttlMs })
    return priceCents
  }
}
```

Three factories from your stack

- createLoaders builds fresh loaders for every request
- createPlaceOrder binds the Use Case to its ports
- express.Router() returns a router without new

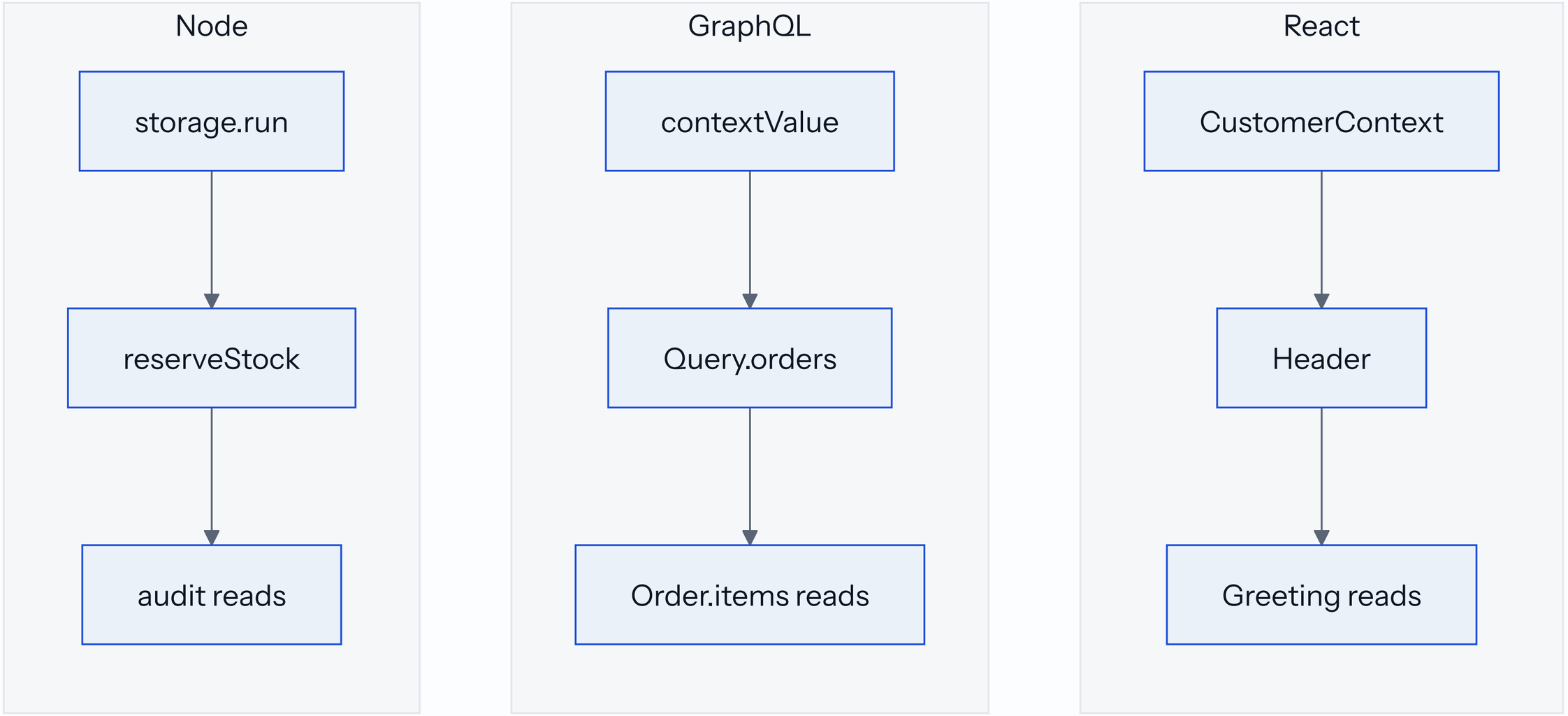
Where factories go wrong

- Built too often: a new pool for every request
- Built too rarely: one cache across all requests
- Built without need: a factory without state or dependencies

Sharing and reacting

One value for the whole tree or request · Whoever subscribes gets notified · When is a pattern too much?

The same pattern in three places



Props first, then children, then Context

- Props make visible who uses which data
- children move layers out of the way
- Context stays for values many things far below need

In React 19, the Context itself is the provider

```
const CustomerContext = createContext<Customer | null>(null)

export const useCustomer = (): Customer => {
  const customer = use(CustomerContext)
  if (!customer) throw new Error('useCustomer braucht einen CustomerContext')
  return customer
}

export const Shop = ({ customer }: { customer: Customer }) => (
  <CustomerContext value={customer}>
    <Header />
  </CustomerContext>
)
```

In GraphQL, the request is the scope

```
// Provider: the context function runs once per request
context: async ({ req }) => ({ customerId: await authenticate(req) }),

// Consumer: each resolver reads the value in the third parameter
me: (_parent, _args, { customerId }: Context) => customers.get(customerId),

// Danger: a module variable is shared by all requests
let currentCustomerId: string
```

In Node, AsyncLocalStorage carries the value

```
const storage = new AsyncLocalStorage<{ requestId: string }>()

export const audit = (message: string) => {
  const request = storage.getStore()
  if (!request) throw new Error('audit braucht einen laufenden Request')
  return `[${request.requestId}] ${message}`
}

export const handleOrder = (requestId: string, orderId: string) =>
  storage.run({ requestId }, () => reserveStock(orderId))
```

Where does the scope end?

- React: the subtree under the provider
- GraphQL: exactly one request
- Node: everything that starts from run

Why does placeOrder know all receivers?

- Every new receiver changes the Use Case
- The Use Case waits for tasks that are not its own
- Tests need a fake for every receiver

```
export const placeOrder = async (input: PlaceOrderInput) => {
  const order = await saveOrder(input)
  await sendConfirmationMail(order)
  await reserveStock(order)
  await updateDashboard(order)
  return order
}
```

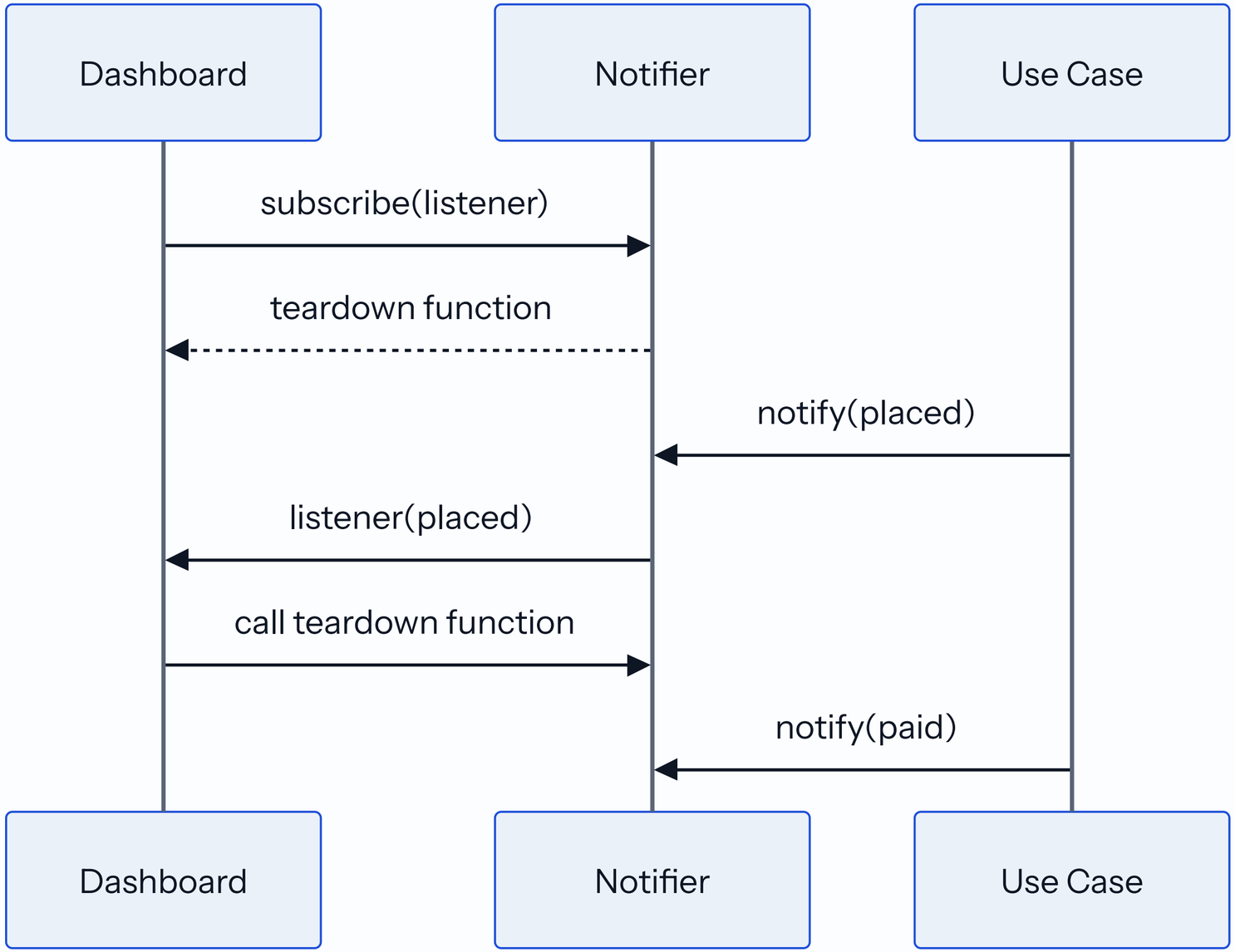
An Observer is a set of functions

```ts
export type Listener<T> = (value: T) => void
export type Teardown = () => void

export const createNotifier = <T>() => {
  const listeners = new Set<Listener<T>>()

  return {
    subscribe: (listener: Listener<T>): Teardown => {
      listeners.add(listener)
      return () => listeners.delete(listener)
    },
    notify: (value: T) => {
      for (const listener of [...listeners]) listener(value)
    },
  }
}
```

subscribe returns the teardown function



EventEmitter and Rx are also Observers

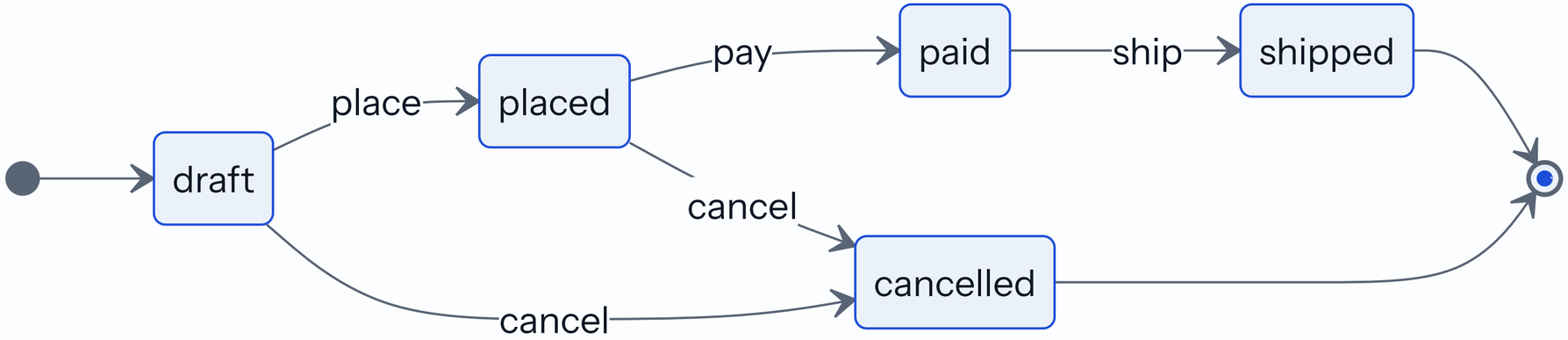
- EventEmitter notifies synchronously, in subscription order
- A throw in a listener stops all later listeners
- Rx adds completion, errors, and operators

```
import { EventEmitter } from 'node:events'

const orders = new EventEmitter()
const onPlaced = (order: Order) => sendConfirmationMail(order)

orders.on('placed', onPlaced)
orders.emit('placed', order)
orders.off('placed', onPlaced) // the reference is needed here
```

Which transitions are allowed at all?



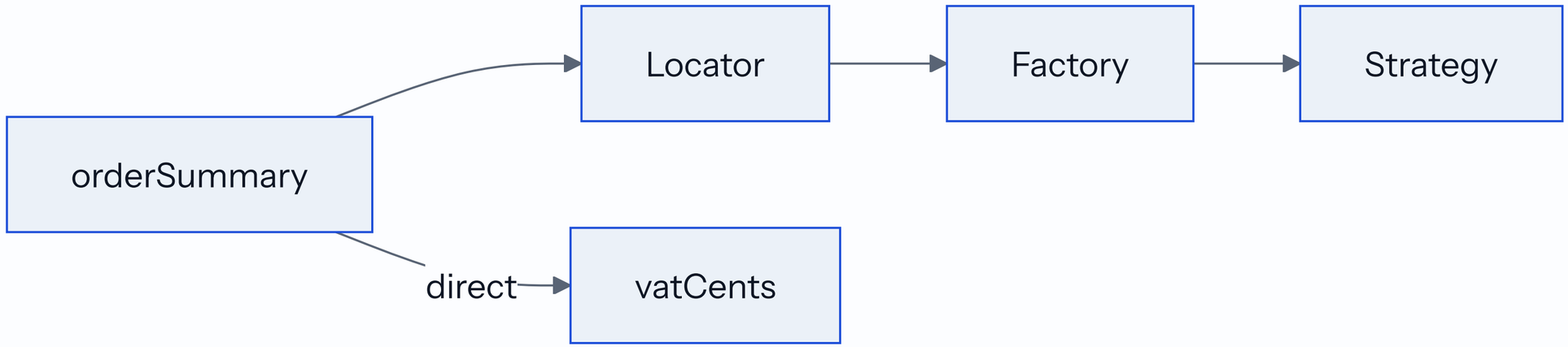
The table decides, the Observer notifies

```
export const transitions: Record<
  OrderStatus, Partial<Record<OrderAction, OrderStatus>>
> = {
  draft: { place: 'placed', cancel: 'cancelled' },
  placed: { pay: 'paid', cancel: 'cancelled' },
  paid: { ship: 'shipped' },
  shipped: {},
  cancelled: {},
}

// in createOrderMachine
send: (action: OrderAction): SendResult => {
  const next = transitions[status][action]
  if (!next) return { kind: 'rejected', status, action }
  status = next
  changes.notify(next)
  return { kind: 'moved', status }
},
```

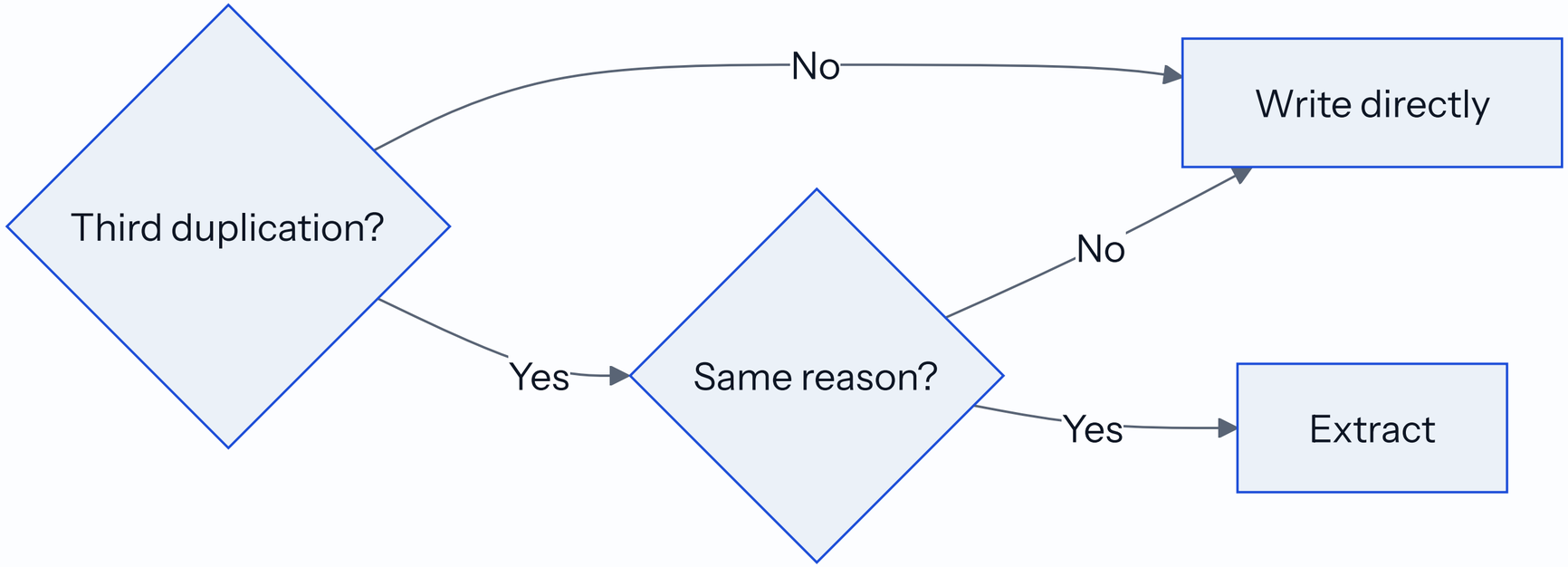
What does a pattern cost?

- Every layer is a jump while reading
- Every name must be learned and maintained
- The value appears only with the next change



The rule of three slows abstraction

- The first time, you write it directly
- The second time, you tolerate the duplication
- The third time, you extract the abstraction



One implementation does not need a Strategy

```
// Before: Strategy and Factory for exactly one rule
export type TaxStrategy = (netCents: number) => number
const germanVat: TaxStrategy = (netCents) => Math.round((netCents * 19) / 100)
const createTaxStrategyFactory = () => ({ create: (): TaxStrategy => germanVat })

// After: a named function
const vatCents = (netCents: number) => Math.round((netCents * 19) / 100)

// If a second rate is added, a parameter is enough
const vatCents = (netCents: number, percent = 19) =>
  Math.round((netCents * percent) / 100)
```

A module is already a Singleton

- An ES module runs once, and all imports share it
- You need neither getInstance nor a registry for that
- Only mutable state in the module is critical

```
// Before: registry to ensure there is only one instance
const services = new Map<string, unknown>()
export const getLogger = () => {
  if (!services.has('logger')) services.set('logger', createLogger())
  return services.get('logger') as Logger
}

// After: the module handles this
export const logger = createLogger()
```

The Service Locator hides dependencies

- The signature hides what the function needs
- A typo in the name only fails at runtime
- One test changes the service for all others

```
// Hidden: dependency is not in the signature
export const orderSummary = (items: LineItem[]) => {
  const factory = serviceLocator.get<TaxFactory>('taxStrategyFactory')
  return summarize(items, factory.create())
}

// Visible: The compiler knows every dependency
export const orderSummary = (items: LineItem[], tax: Tax) =>
  summarize(items, tax)
```

Remove it in small steps

- First characterization tests, then the first step
- One pattern per step, then all tests are green
- No value for changes means the pattern can go

Questions?

Mike Bild · mike.bild@gmail.com · mikebild.dev

Workshops für Teams, die Architekturentscheidungen selbst treffen