

17-695: Design Patterns

Structural Patterns

Monday, September 14 • Lecture

What did we talk about last time?

Creation is a decision

Who chooses the implementation, when, and who owns the result.

Move construction to the edges

The service publishes. Startup decides which provider it publishes with.

A registry is shared state

It helps when independent teams add implementations. It costs when they do not.

Whoever constructs an object decides what it depends on

More on that this week.



Learning Goals

- Recognize when a boundary addresses a real problem and when it only adds a layer
- Argue a boundary decision against a credible alternative, including no boundary at all
- Critique a wrapper against the job it does: convert a shape you do not own, shrink a surface you own, add behavior, or control access

What does this change cost?

Let's see some code.

```
# orders/checkout.py
import stripe

stripe.api_key = config.STRIPE_SECRET_KEY

def checkout(user: User, order_id: OrderId) -> CheckoutResult:
    order = order_repo.get(order_id, owner=user)
    try:
        intent = stripe.PaymentIntent.create(
            amount=order.total.cents, currency=order.total.currency,
            customer=user.stripe_customer_id,
            payment_method=user.stripe_payment_method, confirm=True,
            idempotency_key=f"order-{order.id}",
        )
    except stripe.error.CardError as e:
        order.status = OrderStatus.PAYMENT_FAILED
        order.failure_code = e.error.decline_code or e.error.code
        order_repo.save(order)
        return CheckoutResult.declined(reason=e.user_message)

    order.stripe_payment_intent = intent.id
    order.status = OrderStatus.PENDING if intent.status == "requires_action" else OrderStatus.PAID
    order_repo.save(order)
    send_receipt(order, intent.charges.data[0].receipt_url)
    return CheckoutResult.confirmed(order)
```

Two things happen next quarter

An enterprise customer requires PayPal

Same storefront, same checkout, a second payment vendor.

This Stripe SDK version is deprecated

It stops working in six months. Every call in the codebase moves to the new version, on Stripe's schedule.

Talk to your neighbors, make new friends

What does this change cost, and why?

Some design issues with `checkout`

What

Where

Vendor status values drive the order's state

```
intent.status == "requires_action"
```

Vendor error codes stored as our data

```
order.failure_code = e.error.decline_code
```

Vendor identifiers in the schema

```
stripe_payment_intent, stripe_customer_id
```

Tests either call Stripe or fake Stripe's shape

Every test of `checkout`

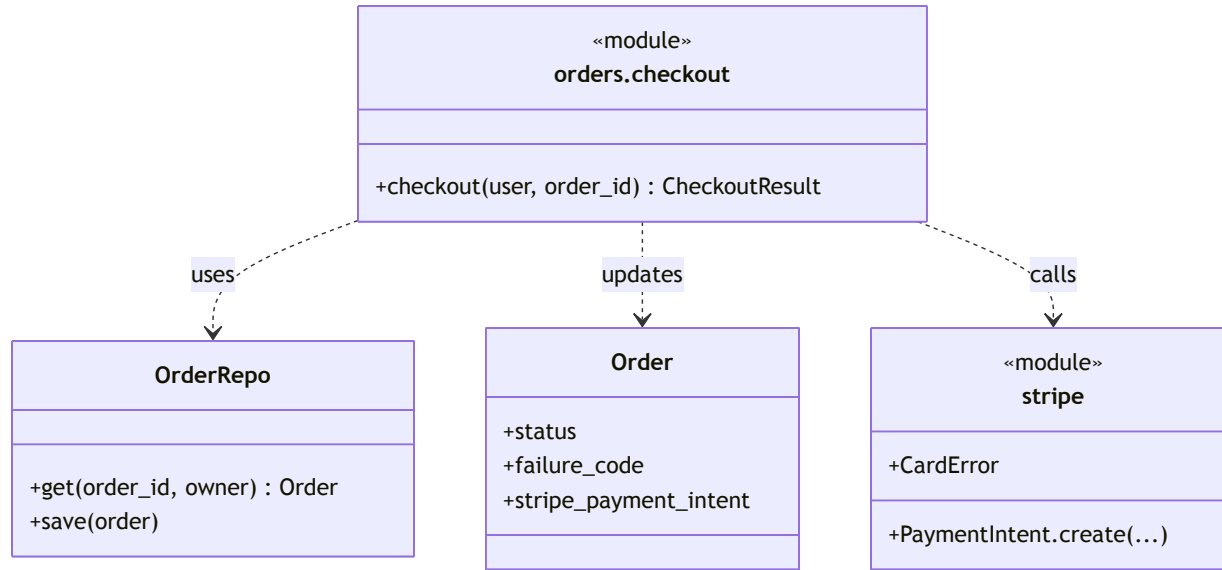
No single place talks to the vendor

This is one of several call sites

The principle

Coupling: what you must know about Stripe to change checkout. Here, basically everything.

The class diagram still looks fine, by the way



What the arrow doesn't show

One dashed arrow, like every other integration. It does not say how much of Stripe is on our side of it.

Coming up later

A test you can run on any codebase: **who imports the vendor?**

Convert a shape you do not own

Define the interface you wish the vendor had

```
class PaymentGateway(Protocol):  
    def charge(self, order: Order, customer: CustomerRef) -> ChargeResult: ...  
    def refund(self, charge: ChargeRef, amount: Money) -> RefundResult: ...  
    def status(self, charge: ChargeRef) -> ChargeStatus: ...  
  
class PaymentDeclined(Exception):  
    code: DeclineCode          # ours, not Stripe's  
    message: str
```

This interface is the port

The boundary our code calls through. Charge, refund, status. Our use cases, in our types.

The principle

Information hiding: the vendor is the decision most likely to change, so it lives behind an interface we own.

Then map Stripe to the interface

```
class StripeAdapter(PaymentGateway):
    def charge(self, order: Order, customer: CustomerRef) -> ChargeResult:
        try:
            intent = stripe.PaymentIntent.create(
                amount=order.total.cents, currency=order.total.currency,
                customer=self.vendor_id_for(customer),
                payment_method=self.vendor_method_for(customer), confirm=True,
                idempotency_key=f"order-{order.id}", expand=["latest_charge"],
            )
        except stripe.error.CardError as e:
            reason = e.error.decline_code or e.error.code
            raise PaymentDeclined(code=DECLINE_CODES.get(reason, DeclineCode.OTHER),
                                  message=e.user_message)
        charge = intent.latest_charge # None until 3-D Secure completes
        return ChargeResult(ref=ChargeRef(intent.id),
                             settled=intent.status != "requires_action",
                             receipt_url=charge.receipt_url if charge else None)
```

Where the vendor now lives

This class, and ideally nowhere else.

Where the defects now live

Miss one of Stripe's decline codes in `DECLINE_CODES` and the order records it as `OTHER`.

Did checkout get better?

```
def checkout(user: User, order_id: OrderId) -> CheckoutResult:
    order = order_repo.get(order_id, owner=user)
    try:
        result = gateway.charge(order, user.customer_ref)
    except PaymentDeclined as declined:
        order.mark_failed(declined.code)
        order_repo.save(order)
        return CheckoutResult.declined(reason=declined.message)
    order.mark_charged(result.ref, settled=result.settled)
    order_repo.save(order)
    send_receipt(order, result.receipt_url)
    return CheckoutResult.confirmed(order)
```

What changed

No `stripe`. No vendor status strings. The order's columns are ours. Somebody still has to choose gateway.

The principles

Open-closed: PayPal is a second adapter (extension). Checkout is untouched (no modification).

Dependency inversion: checkout depends on the port, not on `stripe`.

Let's diagram it

Draw the (informal) class diagram.

- Two arrow kinds: **implements** (dashed, hollow triangle) and **uses** (dashed, open arrowhead). Label which.
- Which dependency from the opening code is gone?
- Include your name on the page.

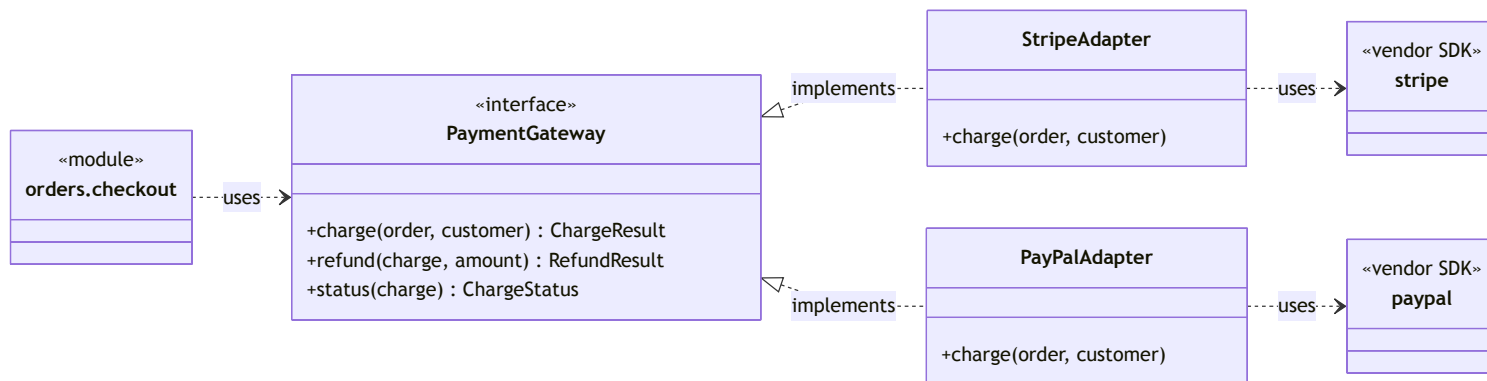
```
# orders/checkout.py
result = gateway.charge(order, user.customer_ref)

# payments/gateway.py
class PaymentGateway(Protocol):
    def charge(self, order, customer): ...

# payments/stripe_adapter.py
import stripe
class StripeAdapter(PaymentGateway):
    def charge(self, order, customer):
        intent = stripe.PaymentIntent.create(...)

# payments/paypal_adapter.py
import paypal
class PayPalAdapter(PaymentGateway):
    def charge(self, order, customer):
        # shape only, not the real SDK
        result = paypal.orders.create(...)
```

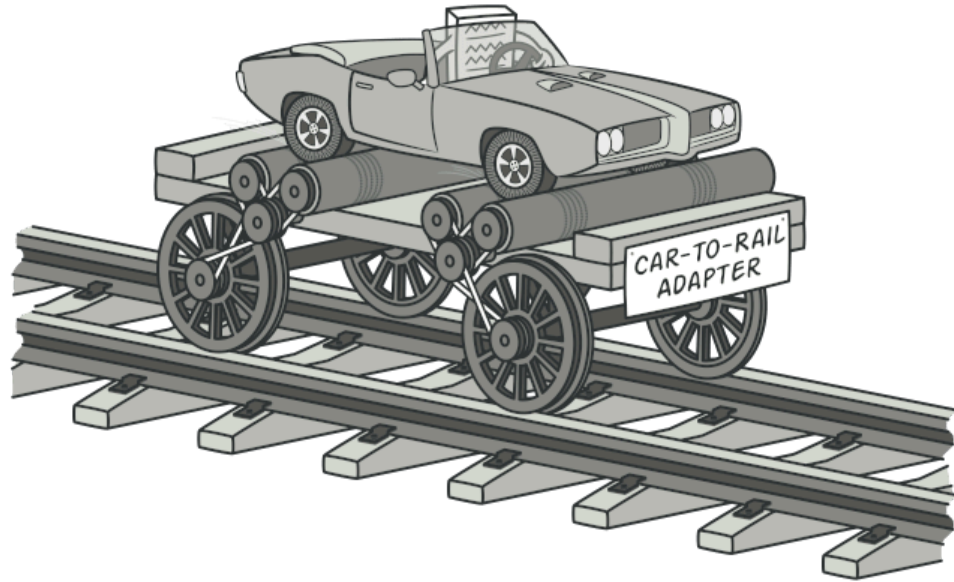
A candidate drawing



The dependency that is gone

checkout to stripe . It was the opening code. It is not on this diagram.

Adapter: convert a shape you do not own into the one you want



Adapter. Illustration: [Refactoring.Guru](https://refactoring.guru) / Dmitry Zhart

Who imports the vendor?

```
$ grep -rI "import stripe" src/ | wc -l
```

The opening code (no boundary)

41

Forty-one files import the vendor. The migration to the new SDK touches every one of them.

After the port

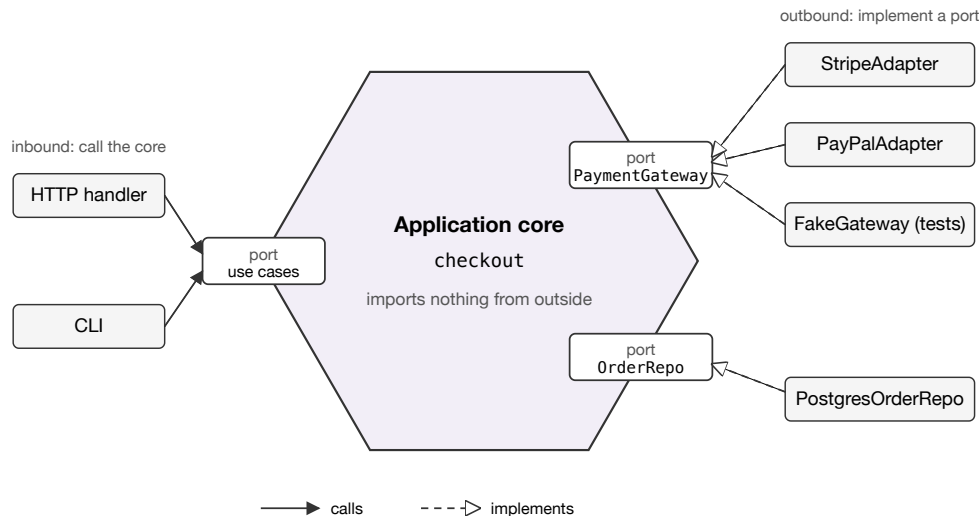
1

One package imports the vendor. The migration has an address.

You may have already used Adapter

Where	What it looks like
Storage backends	Django's <code>Storage</code> class. Local disk, S3, and GCS each implement it, and the app never imports <code>boto3</code> .
Go interfaces	The consumer declares the interface. The vendor's type satisfies it without knowing the interface exists, so the port needs no cooperation from the far side.
MCP (Model Context Protocol) servers	One tool interface the host speaks. Each server maps one vendor's API onto it, and the host never imports the vendor.
Anti-corruption layer	Domain-driven design's name for an adapter between your data model and another system's. It translates, so their entity names, statuses, and ids never enter your code.

A word on ports and adapters



An architectural pattern

Adapter applied to the whole system, not one vendor. Every external system sits behind an interface the core owns. The core imports nothing from outside.

Two directions

Inbound adapters call the core (HTTP, CLI). Outbound adapters are called by the core (payments, storage). Same shape both ways.

What if both sides can vary?

JDBC (Java's database API). The same three lines run against any database.

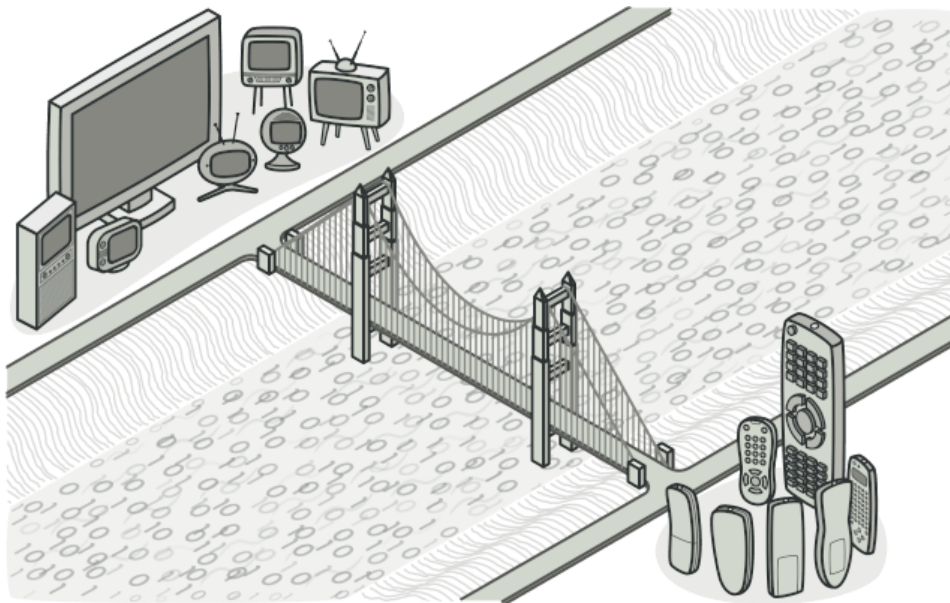
```
Connection c = DriverManager.getConnection(url);  
PreparedStatement s = c.prepareStatement(sql);  
ResultSet rs = s.executeQuery();
```

The API your code calls on one side. The Postgres, MySQL, and SQLite drivers that implement it (not shown) on the other. Either side can change without the other noticing.

What it is

An Adapter planned in advance, with an abstraction on **both** sides.

Bridge: an Adapter planned in advance, with an abstraction on both sides



Bridge. Illustration: [Refactoring.Guru](https://refactoring.guru/) / Dmitry Zhart

Is the port worth it?

What becomes cheaper

- Adding or swapping a vendor.
- Surviving an SDK major-version change.
- Testing. A fake gateway is one small class, not a sandbox account.

What it adds

- A mapping layer to maintain, per vendor.
- The port speaks only the intersection of what all vendors support. Anything vendor-specific needs an escape hatch, and escape hatches can leak.
- Vendors can disagree on idempotency and retry semantics. The port has to commit to one story and make every adapter honor it.

To Adapter or not to Adapter

1. Wrap `requests` behind `OurHttpClient`, in case the library changes.
2. Add a database port above SQLAlchemy, in case of a migration off Postgres.
3. A 2009-era billing service with a SOAP interface nobody can change.

	Verdict	Why
1	No	Call it directly. No credible change, and the wrapper costs every day.
2	No	SQLAlchemy already is the boundary. A second one adds a hop and absorbs nothing.
3	Yes	The far side is fixed and its model is probably unwanted.

The principle

YAGNI. A port is a bet on a change. Two of these bets have no credible change behind them.

Shrink a surface you own

One quarter later: three callers

```
# orders/checkout.py
result = gateway.charge(order, user.customer_ref)
ledger.record_sale(order, result.ref)
fraud.review(order, result)
receipts.send(order, result.receipt_url)

# admin/refunds.py
result = gateway.charge(order, customer)      # a re-charge after a partial refund
fraud.review(order, result)
ledger.record_sale(order, result.ref)
receipts.send(order, result.receipt_url)

# mobile/api.py
result = gateway.charge(order, customer)
fraud.review(order, result)
receipts.send(order, result.receipt_url)
```

Spot the difference

Mobile never reaches the ledger. Admin reviews fraud before the sale exists. Nobody decided either.

One front door

```
class CheckoutService:
    def complete_purchase(self, order, customer) -> CheckoutResult:
        result = self.gateway.charge(order, customer) # declines elided
        self.ledger.record_sale(order, result.ref)
        self.fraud.review(order, result)
        self.receipts.send(order, result.receipt_url)
        return CheckoutResult.confirmed(order)
```

```
# web, admin, mobile:
checkout_service.complete_purchase(order, customer)
```

What changed

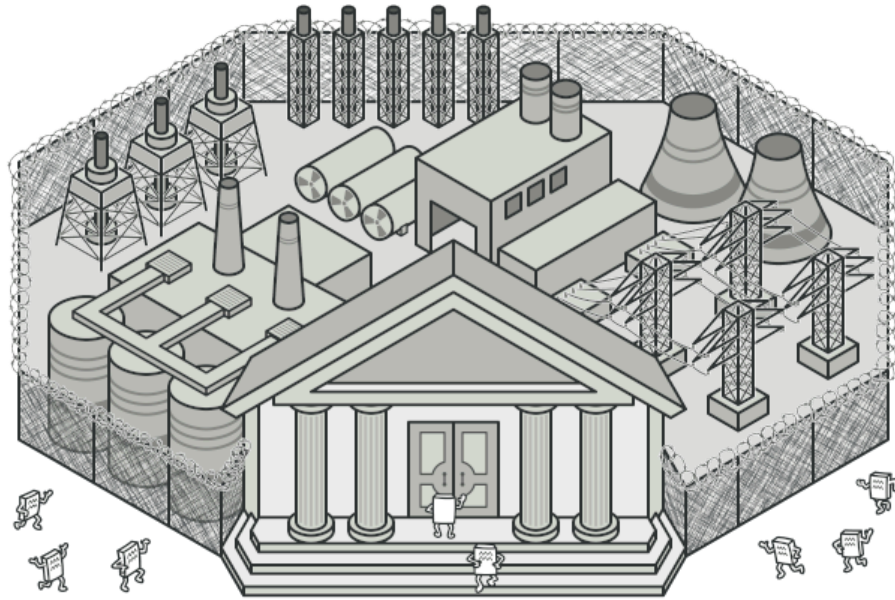
The choreography has one owner. Callers import one thing. In the diagram: one box in front of four.

The principles

Interface segregation: the interface a caller sees has the one method it uses, and none of the methods it does not.

Don't repeat yourself (DRY): the sequence exists once, so it can only be wrong once.

Facade: shrink a surface you own to what callers need



Facade. Illustration: Refactoring.Guru / Dmitry Zhart

Hyrum's law

“With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.

— HYRUM WRIGHT, HYRUMSLAW.COM

What it means for a surface

Every accidental detail callers can observe becomes part of the surface. Hiding is only as good as what actually stays hidden. (What if callers start depending on the order of `complete_purchase()` sending a receipt?)

You may have already used Facade

Where	What it hides
A vendor SDK	HTTP calls, retries, pagination. <code>stripe.PaymentIntent.create</code> is a facade over a REST API.
A service or application layer	The domain objects and repositories behind one use-case method.
An API gateway or a backend-for-frontend	Many services behind one endpoint shaped for one client.
A GraphQL server	A dozen services behind one query surface.
A package's <code>__init__.py</code> that only re-exports	The file layout inside the package. Callers write <code>from payments import PaymentGateway</code> , and files can move without them noticing.

Is the front door worth it?

What becomes cheaper

- Rearranging the subsystem behind the door.
- Onboarding. The surface says where to start reading.
- Getting the sequence right once. This also gives consistency.

What it adds

- One more hop when debugging.
- Unbounded growth. A method per caller per sprint, until the facade **is** the system.
- A standing temptation to move business logic into the facade, at which point the facade is an unnamed layer rather than a surface.

Keeping the two boundary patterns straight

Adapter converts a shape you do not own. Facade shrinks a surface you do.

To Facade or not to Facade

1. A module with two functions. Add a facade in front of it?
2. Every handler in the service starts by calling the same six internal services in the same order.
3. Wrap pandas behind a curated interface for the analytics team?

	Verdict	Why
1	No	The module already is one.
2	Yes	The repetition is the evidence.
3	It depends	Yes if it picks the few operations the team needs and sets the defaults. No if it only renames pandas.

One wrapper, two intents

Every charge now needs three more things

Retry, with the idempotency key

A key sent with each charge. Same key again, and the vendor charges once. Retries become safe.

Metrics

Latency and failure rate per vendor, on a dashboard.

An audit log

Who charged what, when, through which vendor. Compliance required.

First instinct

Add all three to `StripeAdapter`. Then `PayPalAdapter` needs the same three. Copy them, or subclass: `RetryingMetricsAuditedStripeAdapter`.

Wrap the same interface

```
class MetricsGateway(PaymentGateway):  
    def __init__(self, inner: PaymentGateway, metrics: Metrics):  
        self.inner, self.metrics = inner, metrics  
  
    def charge(self, order, customer):  
        with self.metrics.timer("payments.charge"):  
            return self.inner.charge(order, customer)  
  
    def refund(self, charge, amount):  
        with self.metrics.timer("payments.refund"):  
            return self.inner.refund(charge, amount)  
  
    def status(self, charge):  
        with self.metrics.timer("payments.status"):  
            return self.inner.status(charge)  
  
# at startup  
gateway = MetricsGateway(StripeAdapter(), metrics)
```

What changed

One class per concern. Each implements the port and holds one. Callers still use the PaymentGateway interface.

From last week

Whoever constructs the gateway decides what wraps it.

Retry and audit have the same shape.

What the forwarding hides

```
class PaymentGateway(Protocol):
    def charge(self, order, customer): ...
    def refund(self, charge, amount): ...
    def status(self, charge): ...
    def capture(self, charge): ...          # new

class StripeAdapter(PaymentGateway):
    def charge(self, order, customer):
        ...
        if self.status(ref).settled:      # self-call
            return ChargeResult(...)
```

What you bought

Add, remove, or reorder a concern per configuration, with no new class. The price is one more object per call and a stack trace to match.

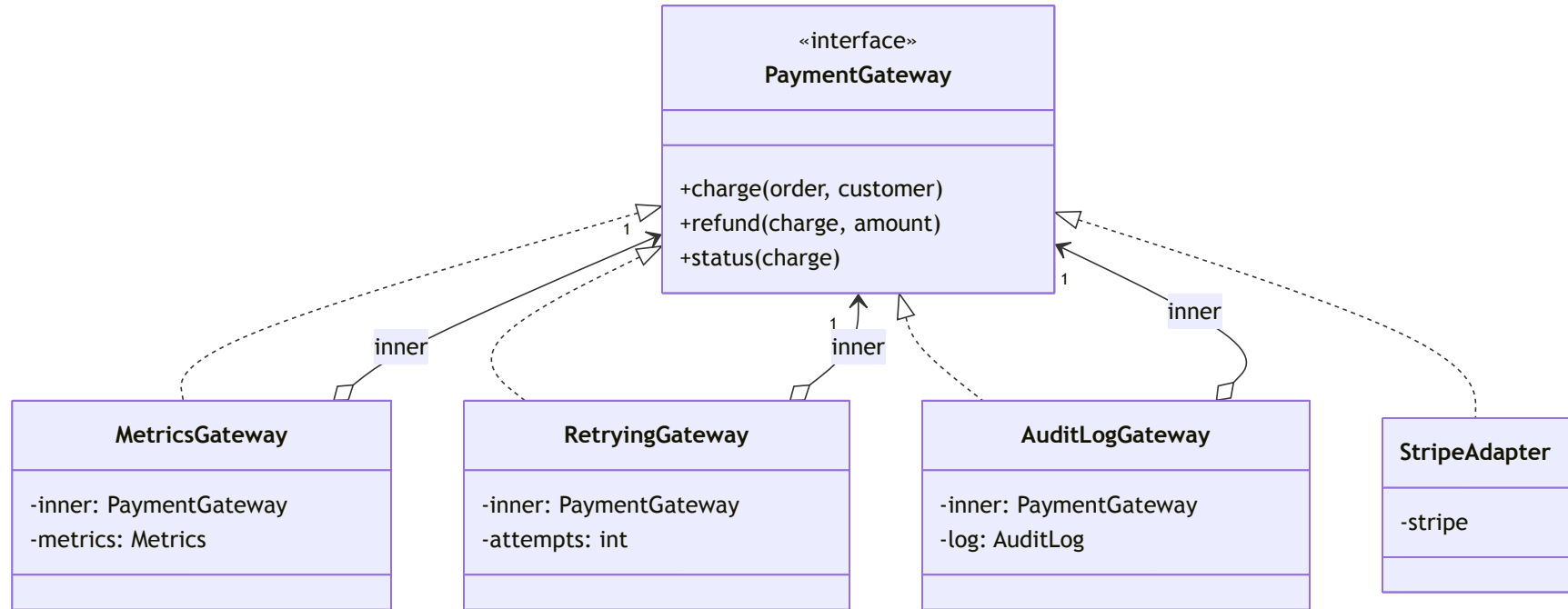
The forwarding tax

Every wrapper must forward the new method. MetricsGateway did not, so `capture()` runs the inherited empty stub and returns `None`. Nothing reaches Stripe, and nothing complains.

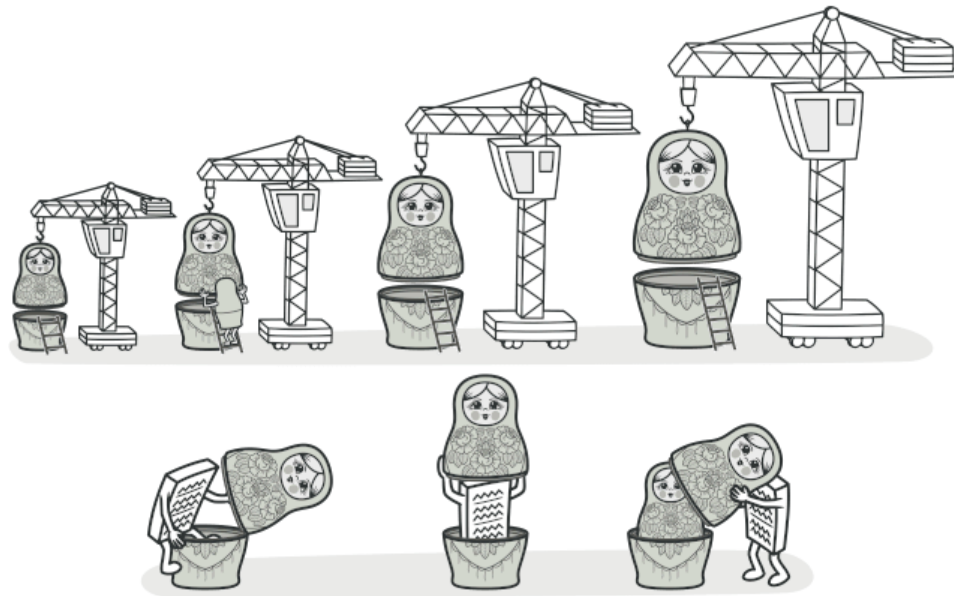
Self-calls skip the wrapper

The adapter's own `status` call is never timed. Only calls from outside are wrapped.

Three wrappers, one shape

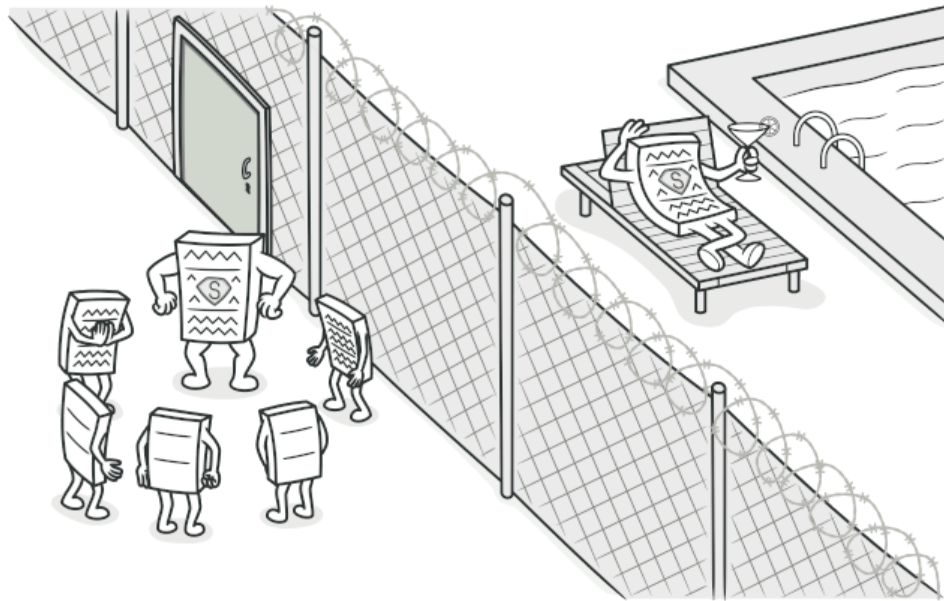


Decorator: layer on behavior the caller knows about and can stack



Decorator. Illustration: [Refactoring.Guru](https://refactoring.guru/) / Dmitry Zhart

Proxy: stand in for the real thing and control access to it



Proxy. Illustration: [Refactoring.Guru](https://refactoring.guru) / Dmitry Zhart

Which is which?

Decorator

Layers on behavior the caller chose, and can stack.

Proxy

Stands in for the real thing. The caller did not choose it and should not notice.

Wrapper

What it does

MetricsGateway	times every call, then forwards
AuditLogGateway	records every call, then forwards
CachingStatusGateway	answers <code>status()</code> from a cache for 30 seconds
RetryingGateway	retries a failed <code>charge()</code> up to three times, same idempotency key

Same diagram, either way

The difference is why the wrapper exists, and who chose it.

Same shape, different decision

	Decorator	Proxy
Who knows the wrapper is there	the caller, or whoever assembles the stack	nobody, ideally
What varies	which behaviors, in which order	how the real thing is reached
In our storefront	metrics, audit	a status cache, if we built one
Design for	concerns that stack	access the caller should not manage

A reminder from two weeks ago

A class diagram shows the shape. The pattern is the decision. (We saw this with Strategy and State.)

You may have already used Decorator and Proxy

```
func withLogging(next http.Handler) http.Handler {  
    return http.HandlerFunc(  
        func(w http.ResponseWriter, r *http.Request) {  
            start := time.Now()  
            next.ServeHTTP(w, r)  
            log.Printf("%s %s", r.URL.Path, time.Since(start))  
        })  
    }  
  
    handler := withLogging(withAuth(mux))  
  
    new BufferedInputStream(  
        new GZIPInputStream(new FileInputStream(f)))
```

Decorators

HTTP middleware
(Go, Express, Django).
The `java.io` stream
stack.

Proxies

ORM lazy loads. Client
stubs for a service on
another machine
(gRPC). Every
generated mock.

Is the stack worth it?

What becomes cheaper

- Cross-cutting concerns compose per configuration. Sandbox without audit, a batch job without metrics.
- Each layer tests alone, against a fake inner.
- The core operation stays one class.

What it adds

- Ordering. `RetryingGateway(MetricsGateway(x))` counts four attempts as four calls.
`MetricsGateway(RetryingGateway(x))` counts one.
Which one did you actually mean?
- A stack trace through three wrappers before the adapter.
`isinstance(gateway, StripeAdapter)` is now false.
- Transparency has limits. A remote proxy reads as a local call, and the network is still there. A cache changes freshness that callers may be relying on.

To wrap or not to wrap

1. One function, one call site, needs a log line.
2. Three of forty endpoints need auth. Middleware, or checks in the three handlers?
3. A loop over 500 orders reads `order.customer`, a query per access. Keep the lazy load?

	Verdict	Why
1	No	Log directly. No second caller, nothing to configure.
2	It depends	The stack is app-wide. The requirement is per route.
3	No	Nothing in the loop says database. That is the proxy working, at 500 queries where one would do. Fetch up front.

The principle

YAGNI for the first. The third is the opposite failure, a proxy so transparent that its cost is invisible.

What is still your decision?

Absorbed, or still yours?

Pattern	Design for	Absorbed by the language or framework	Still your decision
Adapter	a far side you do not control	almost nothing; structural typing helps at the margins	where the boundary sits, who owns the port, what leaks through
Facade	callers that multiply	SDKs, API gateways, backends-for-frontends	the surface: what callers get, what stays hidden
Decorator	concerns that stack	middleware conventions	ordering, and what counts as cross-cutting
Proxy	access the caller should not manage	service meshes, ORMs, generated client stubs (gRPC)	whether callers can afford the behavior hidden behind a local interface

Which design is "correct"?

The payment system for...

A two-person startup, Stripe only, pre-revenue

A regulated fintech, audit trails and vendor redundancy required

The storefront's own SDK, now shipped to other companies

The right amount of structure

The direct SDK calls were fine. The port is an investment in a change that may never arrive.

The full stack. Adapters because dual-vendor is mandated, decorators because audit is cross-cutting, a facade because three frontends already exist.

The facade became someone else's vendor surface. Hyrum's law now applies to **you**.

It depends. On what?

On which of these three rooms you are standing in. And on one more thing. If the change never comes, the indirection was all cost.

Design principles so far

Principle	One sentence summary	Where you saw it
Cohesion	related work lives together	one handler for every event kind (w2)
Coupling	what you must know to change another part	Stripe in the schema, the status, the tests
Information hiding	hide what is likely to change	the vendor behind <code>PaymentGateway</code>
Interface segregation	callers see only the operations they use	<code>complete_purchase</code>
Open-closed	add variants without editing	PayPal as a second adapter
Dependency inversion	depend on the abstraction	checkout depends on the port, not on <code>stripe</code>
Dependency injection	supply dependencies from outside	the gateway stack assembled at startup (w3)
DRY	one place for each piece of knowledge	the choreography, three times
Policy and mechanism	rules apart from the code that runs them	the fee ladder (w2)
Hyrum's law	every observable behavior becomes a dependency	the facade, the cache
YAGNI	no building for variation that is not coming	every "to X or not to X" slide

Reading: the rest of the structural catalog

Pattern	The idea in one line
<u>Bridge</u>	abstraction and implementation vary independently; JDBC is the case the lecture only sketched
<u>Composite</u>	treat a tree and a leaf through one interface; the DOM, React trees, filesystems
<u>Flyweight</u>	share the parts many objects have in common; interning is the everyday case, and the runtime does it for you

Read the intent, structure, and applicability sections. You can skip the code samples.

This week

Wednesday · studio

Finding the boundaries in a food-delivery platform's restaurant integrations. Then reviewing an RFC that proposes where the boundaries should go.

A3 · released today, due Sunday, September 20

Review a proposed change to a real codebase. Handout on the course website.

On your way out

Hand in your page. Name on it, the diagram as far as you got. That is today's attendance.



Today, we learned how to ...

- Recognize when a boundary addresses a real problem and when it only adds a layer
- Argue a boundary decision against a credible alternative, including no boundary at all
- Critique a wrapper against the job it does: convert a shape you do not own, shrink a surface you own, add behavior, or control access