

17-695: Design Patterns

Review and the Future of Design Patterns

Wednesday, September 30 • Lecture

What did we talk about last time?



Learning Goals

- Answer each week's question on the scenario it came from
- Know the exam format and how to prepare for it
- Separate what an agent can produce from what is still your call

(Recall from Week 1)

**"Software engineering is dead. AI
killed it."**

The course as five questions

Week 1: which design is better?

One prompt, two working apps



PRODUCT REQUEST

Build an expense-splitting app for a group. Record expenses, see who owes whom, and record payments when members settle up.

+ Add context



Design A

Append-only ledger. Balances are derived on read.

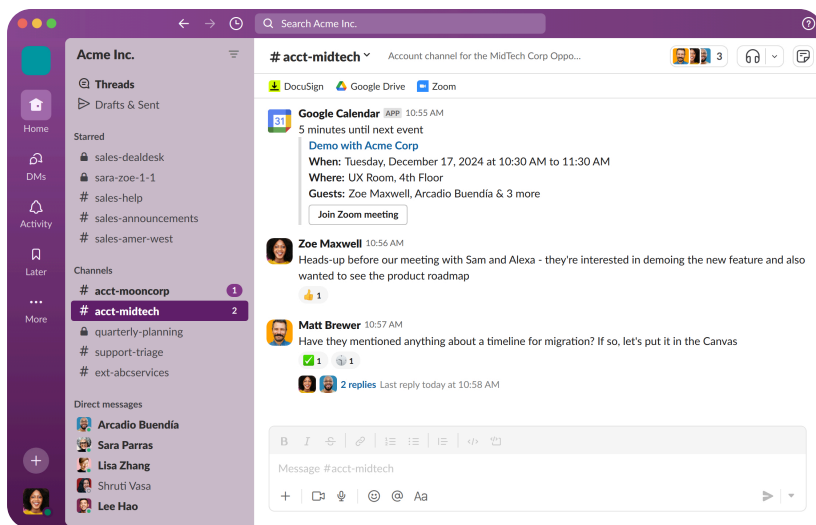
Design B

Stored pairwise balances. Updated on every write.

Both passed the same checks

The prompt never named a design. The agent picked one anyway, without saying so.

Then the whole room did it



One prompt

Every student built the same mini-Slack.

Many designs

Different languages, structures, and places for the notification rule.

Which one fits?

Getting it to run told us nothing about that.

So, which design is "right"?

"Better design" depends on our quality attributes of interest



Every design involves trade-offs

A

DESIGN A

Derive on read

Keep the ledger; calculate balances when requested.

Pro • traceable balances

Recorded expenses and payments explain the result.

Con • repeated computation

Each uncached read reduces the ledger again.

B

DESIGN B

Maintain on write

Store current balances; update them whenever activity is recorded.

Pro • direct reads

The balance view reads the state it needs.

Con • consistency risk

Every write path must keep history and balances aligned.

Good for today can be bad for tomorrow

A new requirement arrives: people can **edit or delete a past expense**, even after settling up.

Design A

Add a correction record. The history already explains every balance.

Design B

The old split is gone. Every stored balance it touched has to be unpicked.

Design for change

Both designs met today's requirements. The design decides what tomorrow's change costs.

Before you accept generated code

- ✓ add one ordinary expense
- ✓ show the expected balances
- ✓ record one settlement

What that shows

One sequence worked, for these inputs, in this order. Not "the app works."

What did it decide?

Find the choices the prompt left open.

What does the evidence cover?

State the inputs, paths, and conditions actually exercised.

Why are we accepting this?

Record what you kept, what you changed, and why.

Week 2: who chooses the behavior?

Who chooses the pricing?

```
elif event.kind == EventKind.TRIP_COMPLETED:
    trip.state = TripState.COMPLETED
    fare = base_fare(trip) * surge_mult(trip.region, event.timestamp)
    if trip.rider_is_new:
        fare = round(fare * 0.8, 2)
    payments.charge(trip.rider_id, fare)
```

What varies, and who picks it?

The pricing rule varies. Put it behind one interface, and the caller picks which one the handler gets.

Move the varying part behind one interface

```
class Pricing(Protocol):
    def fare(self, trip, event) -> Decimal: ...

class StandardPricing:
    def fare(self, trip, event):
        surge = surge_mult(trip.region, event.timestamp)
        return base_fare(trip) * surge

class NewRiderPricing:
    def fare(self, trip, event):
        base = StandardPricing().fare(trip, event)
        return round(base * Decimal("0.8"), 2)

# in the handler...
elif event.kind == EventKind.TRIP_COMPLETED:
    fare = pricing.fare(trip, event)
    payments.charge(trip.rider_id, fare)
```

What changed

The handler shrinks to one call. New policies stop touching it. Somebody still has to choose pricing.

The principle

Open to new policies, closed to edits of the handler. The **open-closed principle**. Design for a known axis of change.

Strategy: the caller picks the behavior



Strategy. Illustration: [Refactoring.Guru](#) / Dmitry Zhart

The handler knows every consumer

```
elif event.kind == EventKind.TRIP_COMPLETED:
    trip.state = TripState.COMPLETED
    fare = base_fare(trip) * surge_mult(trip.region, event.timestamp)
    if trip.rider_is_new:
        fare = round(fare * 0.8, 2)
    payments.charge(trip.rider_id, fare)
    push.send(trip.rider_id, f"Trip complete: ${fare:.2f}")
    analytics.track("trip_completed", trip.id, fare=fare)
```

What happens when receipts wants an email?

Another line in this branch. Every new consumer is an edit to the handler.

Who reacts when a trip completes?

```
elif event.kind == EventKind.TRIP_COMPLETED:           # trips/events.py
    trip.state = TripState.COMPLETED
    fare = pricing.fare(trip, event)
    payments.charge(trip.rider_id, fare)
    events.emit(TripCompleted(trip.id, fare))
```

```
events.subscribe(TripCompleted, rider_app.notify)      # rider_app/wiring.py
events.subscribe(TripCompleted, receipts.issue)        # billing/wiring.py
events.subscribe(TripCompleted, analytics.record)      # analytics/wiring.py
```

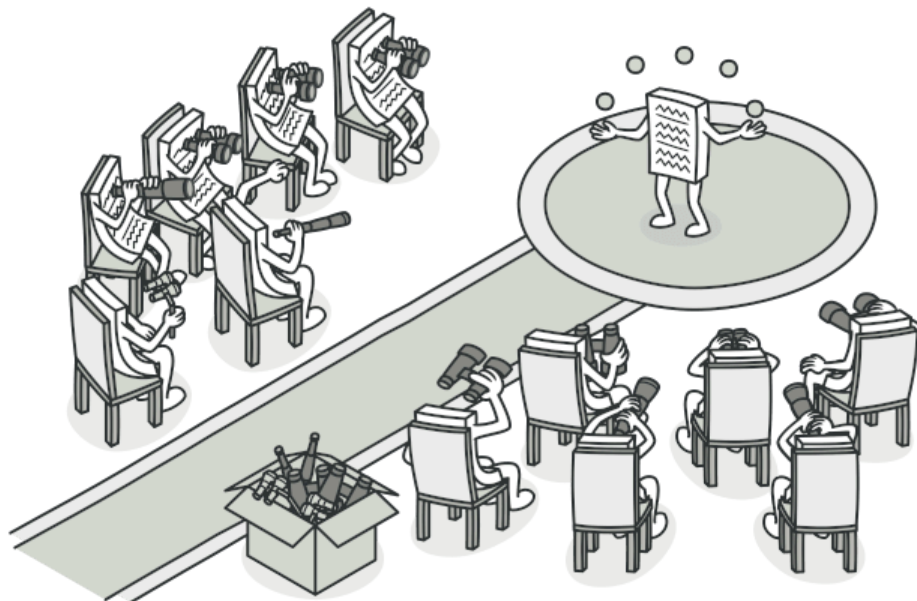
What changed

The handler no longer knows who listens. Adding a consumer (e.g., an email notifier) is one subscribe call somewhere else.

The principle

The handler and its consumers both depend on the event type, not on each other (**dependency inversion**).

Observer: state the change once, and let consumers subscribe



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

Week 3: who chooses what gets created, and when?

What's wrong with `publish` now?

```
class ReleaseService:
    def publish(
        self, repo_name: str, tag: str, files: list[str],
    ) -> None:
        token = os.environ["GITHUB_TOKEN"]
        client = GitHubClient(
            token=token,
            base_url="https://api.github.com",
        )
        repository = GitHubRepository(client, repo_name)
        release = GitHubRelease(client, repository.id)

        release.create(tag=tag, draft=False)
        for path in files:
            release.upload(path)

        GitHubChecks(client, repository.id).mark_released(tag)
```

The method creates its own client

It can only publish to GitHub, and no test can run it without a network. Nothing lets a caller pass in something else.

Why does creation matter?

Carnegie Mellon University

17-695 / DESIGN PATTERNS

What would need to change for GitLab?

```
class ReleaseService:
    def publish(
        self, repo_name: str, tag: str, files: list[str],
    ) -> None:
        token = os.environ["GITHUB_TOKEN"]
        client = GitHubClient(
            token=token,
            base_url="https://api.github.com",
        )
        repository = GitHubRepository(client, repo_name)
        release = GitHubRelease(client, repository.id)

        release.create(tag=tag, draft=False)
        for path in files:
            release.upload(path)

        GitHubChecks(client, repository.id).mark_released(tag)
```

This class is coupled to GitHub: credentials, URL, classes, and API calls.

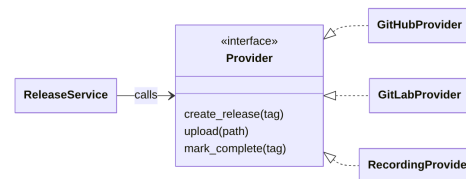
Software and Societal
Systems Department

7

Carnegie Mellon University

17-695 / DESIGN PATTERNS

Decouple the service with a provider interface



The service knows provider operations; each provider knows its vendor API.

Software and Societal
Systems Department

9

Dependency injection

The service stops creating its provider and receives one instead. GitLab becomes a second provider, and a test can pass a fake.

Who chooses the implementation?

Provider is chosen and configured at startup



```
with HttpClient() as http:
    provider = provider_for(settings.provider, http)
    service = ReleaseService(provider)
    service.publish(tag, files)
```

Python's with block manages resource lifetime

Entering the context binds the client to http.
Leaving the block closes it, even if publishing raises an exception.

Match the mechanism to the requirement

Requirement

Start here

The caller already knows the dependency	Supply an object
Configuration chooses one implementation	Factory function
Independent extensions add implementations	Registry of factories
A workflow needs a fresh object at a particular step	Supplied factory or Factory Method
Callers need separate, compatible products	Abstract Factory

Choose the simplest option that meets the requirement.

Choose once, at startup

A factory function reads the settings and builds the provider. Move up the table only when a requirement forces you to.

When does it make sense to use a Builder?

Collect inputs as they become available

```
from dataclasses import dataclass, field

@dataclass
class ReleaseSpecBuilder:
    tag: str = ""
    notes: str = ""
    draft: bool = False
    prerelease: bool = False
    assets: list[str] = field(
        default_factory=list,
    )
```

```
builder = ReleaseSpecBuilder()
builder.from_config(repo_config)
builder.override_with(cli_args)
builder.add_assets(["dist/app.zip"])
spec = builder.build()
```

Only build() validates and returns the completed specification.

Sometimes a constructor is enough

```
spec = ReleaseSpec(
    tag=tag,
    notes=notes,
    draft=True,
)
```

Tip: Use named parameters

They explain each value. The constructor can also validate inputs and return an immutable object.

All values are available in one place.

Use a Builder when collecting inputs in stages needs a separate object.

When the unfinished object needs somewhere to live

Inputs arrive in stages, so the builder holds them and validates once. If every value is already at hand, use named arguments.

Do you need a Singleton to share one object?

Carnegie Mellon University

17-695 / DESIGN PATTERNS

One request changes another's credential

```
# Both requests received the same mutable client.
client_a = shared_client
client_b = shared_client

client_a.token = "customer-A-token" # A starts
client_b.token = "customer-B-token" # B runs before A publishes

# A resumes, but the shared token now belongs to B.
client_a.create_release(spec_a) # uses customer B's token
```

Passing the same client to both requests shares its token too.



58

Carnegie Mellon University

17-695 / DESIGN PATTERNS

Separate creation, lifetime, and access



CREATION

How many?

Create one HTTP client to reuse connections.



LIFETIME

Owned by whom?

An application owns it and closes it at shutdown.



ACCESS

How do callers get it?

Pass the client to each consumer explicitly.

Reusing one object does not require global access.



59

One instance is three decisions

How many, owned by whom, and how callers get it. Singleton answers all three at once, with global access. Reuse only needed the first.

Who owns what you create?

Carnegie Mellon University

17-695 / DESIGN PATTERNS

Startup creates and connects the objects

```
def main(settings, spec):  
    load_provider_plugins()  
    with HttpClient() as http:  
        provider = provider_for(  
            settings.provider, http,  
        )  
        service = ReleaseService(provider)  
        service.publish(spec)
```

Composition root
The startup code that creates and connects the application's objects.

- 1 Load provider types and select one.
- 2 Pass dependencies to the provider and service.
- 3 Own the HTTP client's lifetime.

63

The creator closes it

The composition root builds and connects the objects. `main()` opened the HTTP client, so it closes it. The provider only borrows it.

Week 4: after the boundary, what still knows the vendor exists?

```
# 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)
```

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: ...  
  
@dataclass  
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

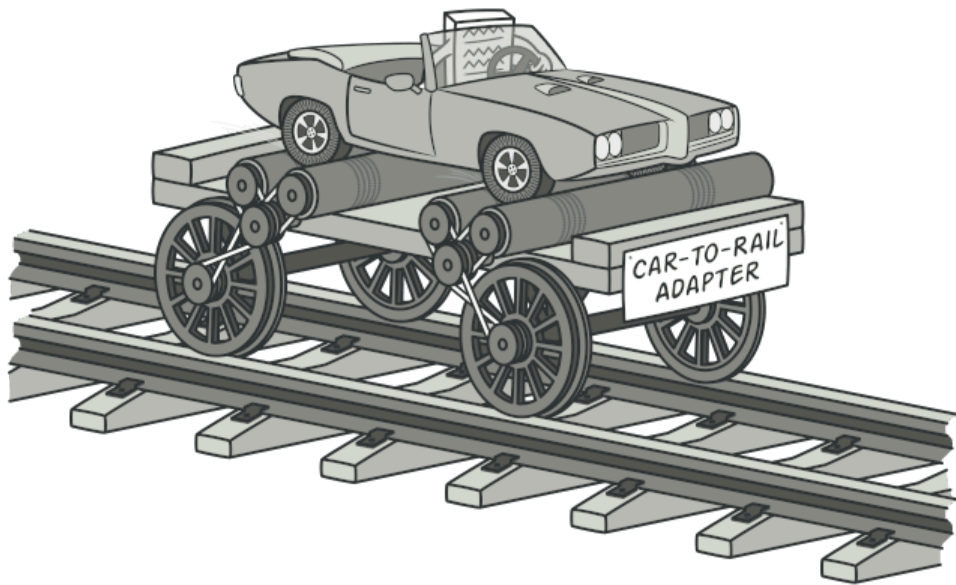
No `stripe`. No vendor status strings. The order's columns are ours. Somebody still has to choose gateway.

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`.

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



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

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

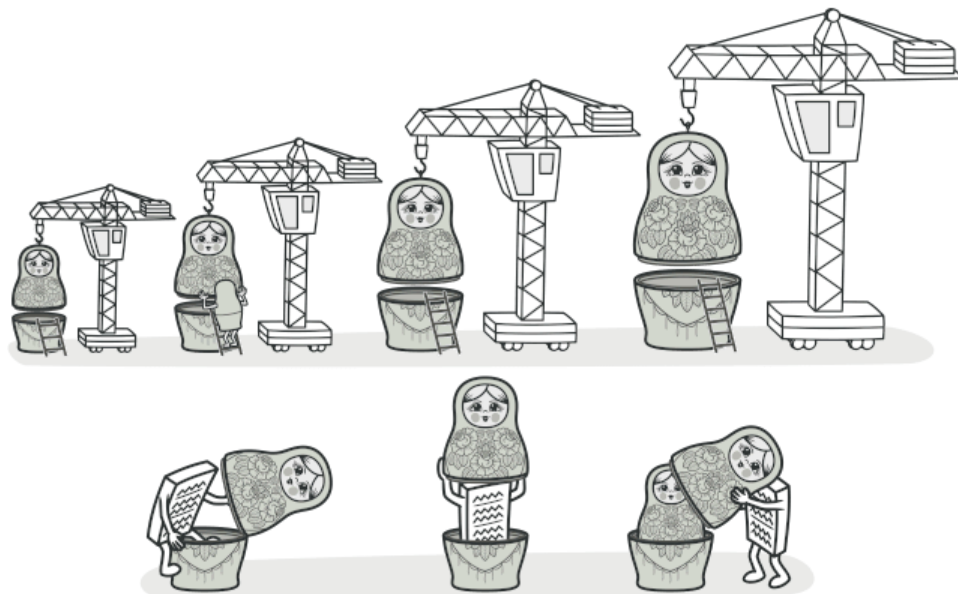
One class per concern. Each implements the port and holds one. Callers still use the PaymentGateway interface.

From week 3

Whoever constructs the gateway decides what wraps it.

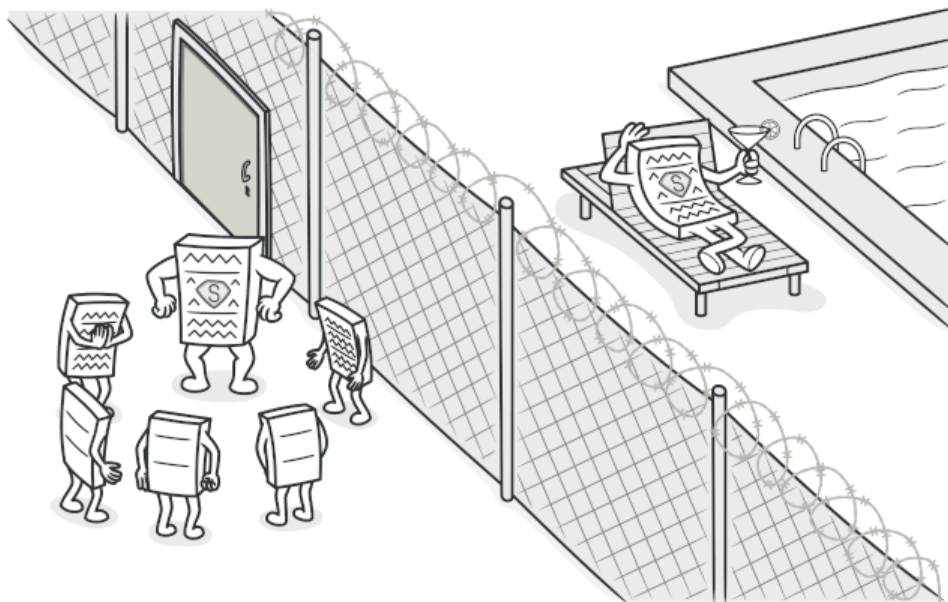
Retry and audit have the same shape.

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



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

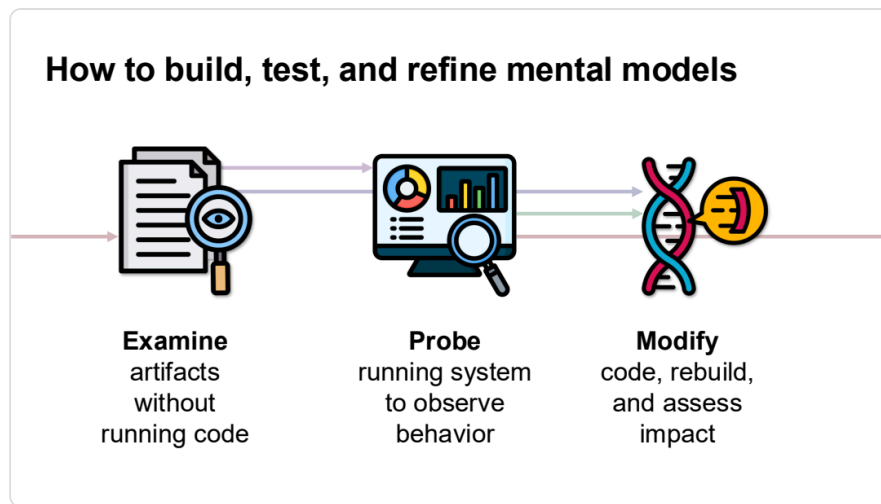
Proxy: stand in for the real thing and control access to it



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

Week 5: how do you change code you did not write?

How do you understand new code?



You will never understand the whole system

Build a working model of the part you need, then test it. An AI explanation is a hypothesis until you probe it.

How do you refactor?

Why are you refactoring?

Performance

Where are the bottlenecks? Is there low-hanging fruit?



Technical Debt

Where is the bulk of complexity? Where are the smells?



Modernization and Feature Extensions

What is your goal? What are the least changes required?



Refactoring

Change the structure, keep the external behavior.

Preparatory refactoring

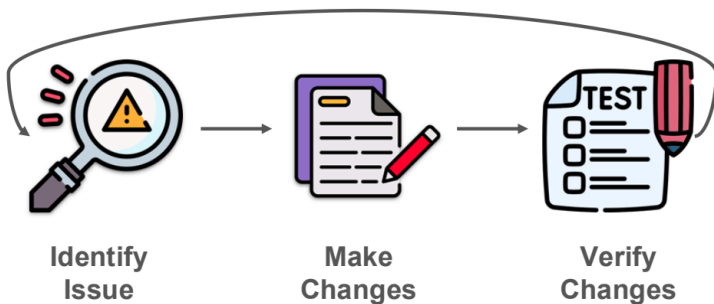
Make the change easy. Then make the easy change.

Beware the allure of clean code

Scope the refactoring to the change you need.

How do you avoid breaking things?

Safe Refactoring: Change > Test > Commit
how can you verify that you didn't break things



Characterization Testing (Snapshot Testing)
use existing code to create E2E test oracles

- Test system under different scenarios. Use the concrete output of existing system as oracle.
- A good fit for systems that have easily observable, synchronous input-output behavior
- Assumes a **fixed, unchanging environment** is used during both characterization and test runs.
- Assumes deterministic test outcomes!



For your project

Your RFC asks "How will you demonstrate that your changes did no harm?" For each behavior, name an existing test, a new test, or a manual check.

How do you write a design doc?

Your proposal: answer three questions

A reviewable draft: one to two pages plus a small sketch

What is the problem?

Problem and scope

Who needs what? What is out of scope?

Context

What exists today? What must stay true?

What is the proposal?

Design and alternatives

Sketch a flow. Compare choices and their costs.

Validation

How will you check success and preserve behavior?

What feedback do we need?

Open questions

What is still unknown? State assumptions.

Questions for reviewers

Ask two specific questions about your design.

Include one success check and one check of behavior that must still work.

29

How will we know the change helped?

Did we solve the customer's problem?

Can customers view and cancel their subscription without the mobile app?

Did we improve the design?

If the cancellation policy changes, how many places must we update across the app and website?

What must still work?

Cancelling stops renewal while preserving access through the paid period.



Code review before the code exists

State the problem, compare alternatives and their costs, and say how you will check it. "Did we improve the design?" is the check teams forget.

The principles behind the patterns

Principle	In one sentence	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 (w4)
Information hiding	hide what is likely to change	the vendor behind <code>PaymentGateway</code> (w4)
Open-closed	add variants without editing	<code>pricing.fare(...)</code> (w2), a second adapter (w4)
Dependency inversion	depend on the abstraction	checkout depends on the port, not on <code>stripe</code> (w4)
Dependency injection	supply dependencies from outside	the provider handed to the service at startup (w3)
Resource ownership	the creator closes it, unless it hands it over	<code>main()</code> owns the HTTP client (w3)
Hyrum's law	every observable behavior becomes a dependency	the cache (w4)
Preserve behavior first	pin what the code does before you restructure it	characterization tests (w5)
YAGNI	do not build for variation that is not coming	every "to X or not to X" slide

Let's practice

```
class Pricing(Protocol):
    def fare(self, trip, event) -> Decimal: ...

class StandardPricing:
    def fare(self, trip, event):
        surge = surge_mult(trip.region, event.timestamp)
        return base_fare(trip) * surge

class NewRiderPricing:
    def fare(self, trip, event):
        base = StandardPricing().fare(trip, event)
        return round(base * Decimal("0.8"), 2)

# trips/events.py
elif event.kind == EventKind.TRIP_COMPLETED:
    trip.state = TripState.COMPLETED
    fare = pricing.fare(trip, event)
    payments.charge(trip.rider_id, fare)
    events.emit(TripCompleted(trip.id, fare))

# one line in each consumer's wiring.py
events.subscribe(TripCompleted, rider_app.notify)
events.subscribe(TripCompleted, receipts.issue)
events.subscribe(TripCompleted, analytics.record)
```

The change

The new-rider discount now depends on the city: 20% in most, 10% where rides are subsidized.

Write

What has to change, and where? Does the handler stay untouched? Why/why not? Three lines at most.

Let's practice, part b

```
class Pricing(Protocol):
    def fare(self, trip, event) -> Decimal: ...

class StandardPricing:
    def fare(self, trip, event):
        surge = surge_mult(trip.region, event.timestamp)
        return base_fare(trip) * surge

class NewRiderPricing:
    def fare(self, trip, event):
        base = StandardPricing().fare(trip, event)
        return round(base * Decimal("0.8"), 2)

# trips/events.py
elif event.kind == EventKind.TRIP_COMPLETED:
    trip.state = TripState.COMPLETED
    fare = pricing.fare(trip, event)
    payments.charge(trip.rider_id, fare)
    events.emit(TripCompleted(trip.id, fare))

# one line in each consumer's wiring.py
events.subscribe(TripCompleted, rider_app.notify)
events.subscribe(TripCompleted, receipts.issue)
events.subscribe(TripCompleted, analytics.record)
```

A proposal says

"Since the handler emits `TripCompleted`, adding a consumer is one subscribe call. Nothing else changes."

Write

One review comment on that sentence: a tag (blocking, question, or agree) and the consequence you traced in the code.

The exam

Logistics

In person, on paper, in this room

Saturday, October 10, 8:30 to 11:30 am. We will aim for a 2-hour exam.

Closed book, with one cheat sheet

One US letter page, front and back, handwritten by you.

Four types of questions

Understand a design. Compare two designs. Critique a proposal. Change a design.

Focus on design reasoning over pattern definitions

Basically what we have been practicing the whole course.

How to prepare

Attempt the practice exam

Should be out in the coming days. Similar structure, but for a different product.

Review the lecture slides and studio packets

Our in-class design reasoning activities will be helpful.

Go over your own RFC for G1 and any feedback on it

Monday's critique asked your design the kinds of questions the exam will ask.

So, is it dead?



AI can reproduce the mechanism



GENERATION

The agent can propose

Produce a familiar implementation quickly.

Structure

Interfaces, classes, and wiring

Variants

An idiomatic form for the requested language

Checks

Tests for the behavior stated in the prompt

Does it belong here?



JUDGMENT

The engineer must decide

Decide whether that mechanism belongs here.

Conditions

Does the recurring problem actually exist?

Priorities

Which quality matters in this system?

Cost

Is the added machinery worth paying for?

What this class showed

Studio 1: one prompt, many designs

One spec, many diverging designs.

A1: decisions you did not make

The code had parts you didn't ask for, and they likely influenced the cost of the three changes.

A3: the review

The provided review checked the code carefully. It endorsed the RFC's central argument without testing it.

What can we use AI for?

AI can	Still yours
Build the pattern you ask for	Deciding what varies in this product
Estimate what a change touches in the repository	Knowing the roadmap, and who depends on you downstream
Write and run the tests you describe	Deciding which behavior has to stay the same
Argue for alternatives, and review a proposal	Making the decision, and answering for it

This is, of course, not an exhaustive list. But you get the idea.

Are the pattern names still important?

DESCRIBE IT FROM SCRATCH



Explain the mechanism

Keep every expense and settlement as the authoritative history. Recalculate balances from those records instead of updating a second representation on every write.

USE SHARED SHORTHAND

A

Append-only ledger

Events are authoritative; current balances are derived when requested.

Where the design work went

Into the request

One prompt, many designs. What you leave out of the request, the agent decides for you.

Into reviewing a proposal

Before anything is built. The RFC and the anchored comment are how you did it in this course.

Into reviewing the result

Check that it works. Then look for the decisions you did not ask for. (You started doing this with A1.)

From the syllabus, week 1

"Modern software engineers spend much of their time steering, reviewing, and changing code that someone or something else wrote."

What we do not know

Where the line sits next year

Some of what is still yours will likely move.

Whether today's defaults last

What an agent builds when the request does not say. That may change.

So, what do you think?
Is software engineering dead?

Between now and the exam

G2 · Sunday, Oct 4, 11:59 pm

The combined version in your fork. The final RFC with Response to Review. Your zip uploaded to Gradescope.

Presentations · Monday and Wednesday

Every team comes prepared on Monday. A wheel picks who is next. Attend both sessions.

Exam · Saturday, October 10, 8:30 to 11:30 am

In person, on paper, in this room. Closed book, but you can bring one handwritten cheat sheet.

FCEs are open

Please fill in the course evaluations. We read every bit of feedback you have for us.



Today, we learned how to ...

- Answer each week's question on the scenario it came from
- Know the exam format and how to prepare for it
- Separate what an agent can produce from what is still your call