

17-695: Design Patterns

Behavioral Patterns

Monday, August 31 • Lecture

What did we talk about last time?

One prompt, many designs

Two working expense apps made different decisions about what to store and what to compute.

Design means choosing trade-offs under real constraints

A design is better only with respect to the qualities you care about.

The design sets the floor on what a change touches

Editing a past expense cost different amounts in the two designs.

A pattern packages a recurring design decision

Context, decision, trade-offs.



Learning Goals

- Identify the design problem that motivates a behavioral pattern in unfamiliar code
- Argue for or against applying a behavioral pattern using concrete trade-offs
- Explain which parts of a pattern modern tooling has absorbed and which decisions remain yours

Last night's A1 submissions

Quick show of hands if this was true for you:

"The notification system decides what to do with a message inside one long conditional."

Let's see another example.

```
def handle_trip_event(trip: Trip, event: TripEvent) -> None:
    if event.kind == EventKind.DRIVER_ACCEPTED:
        trip.state = TripState.MATCHED
        eta = maps_client.eta(event.driver_location, trip.pickup)
        push.send(trip.rider_id, f"Driver on the way, {eta} min out")
        analytics.track("trip_matched", trip.id, driver=event.driver_id)
    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)
    elif event.kind == EventKind.CANCELED_BY_RIDER:
        trip.state = TripState.CANCELED
        push.send(trip.driver_id, "Rider canceled the trip")
        fee = 0.0
        if minutes_since(trip.matched_at, event.timestamp) > 5:
            fee = 5.00
        if trip.rider_is_new:
            fee = 0.0
        payments.charge(trip.rider_id, fee)
        analytics.track("trip_canceled", trip.id, by="rider")
    # ... DRIVER_ARRIVED, TRIP_STARTED, CANCELED_BY_DRIVER
    trip_repo.save(trip)
```

Some design issues...

Symptom	In the handler
A large conditional tree	One function grows a branch per event kind.
Hard-wired dependencies	The handler calls push, payments, analytics, and maps directly.
Duplicated workflows	Change state, notify, track. Every branch does it a little differently.
An object that knows too much	The handler knows every downstream consumer by name.
Rules stacked inline	Which fee rule wins is a fact about line order.

... and their associated principles

Symptom	Principle	One sentence summary
A large conditional tree	Cohesion	related work lives together; this function does five jobs
Hard-wired dependencies	Coupling	what you must know to change another part; five services by name
Duplicated workflows	DRY ("don't repeat yourself")	one place for each piece of knowledge; notify-and-track is repeated per branch
An object that knows too much	Information hiding	hide the decisions likely to change; pricing, consumers, and fee order are exposed
Rules stacked inline	Policy and mechanism	keep the rules apart from the code that runs them; the fee order is an accident of lines

This does happen in production, by the way

Mattermost, websocket_actions.ts

One function, `handleEvent`, dispatches every WebSocket event type in one switch. About 350 lines, dozens of cases, behind an `eslint-disable max-lines` comment.

[github.com/mattermost/mattermost · websocket_actions.ts](https://github.com/mattermost/mattermost/blob/master/websocket_actions.ts) 

**Each of these problems has a
"standard" answer.**

(We'll name the answers at the end.)

Who chooses the behavior?

```
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)
```

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 * 0.8, 2)

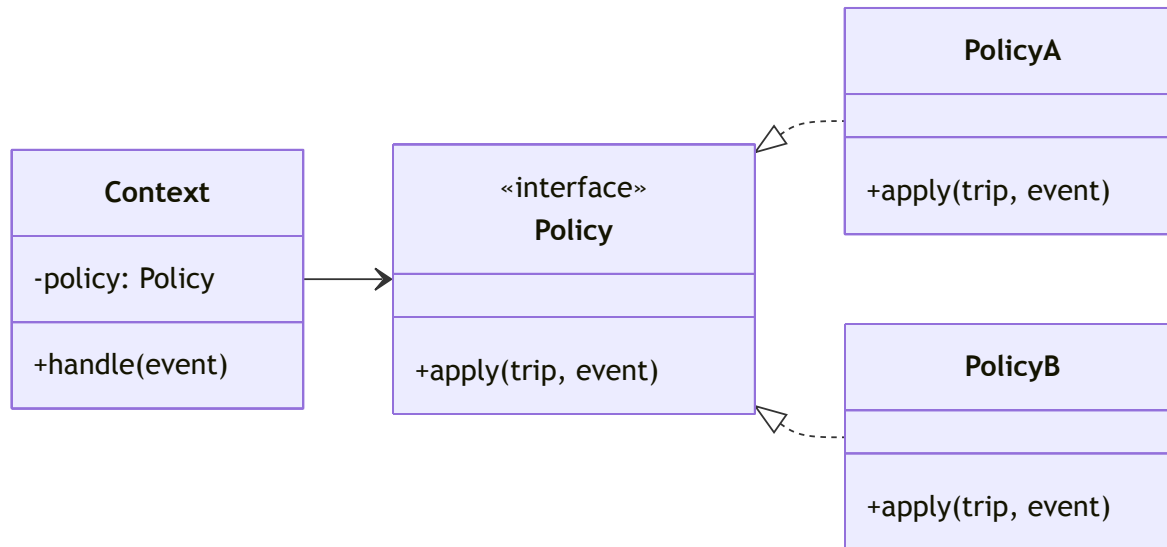
# in the handler...
elif event.kind == EventKind.TRIP_COMPLETED:
    fare = pricing.fare(trip, event)
    payments.charge(trip.rider_id, fare)
```

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.



You may have already used Strategy

```
sorted(trips, key=lambda t: t.requested_at)

sorted(trips, key=lambda t: t.fare, reverse=True)
```

```
trips.sort(Comparator.comparing(Trip::requestedAt));

trips.sort(Comparator.comparing(Trip::fare, Comparator.reverseOrder()));
```

At scale, the policy is often chosen by a string

```
LogisticRegression(solver="lbfgs")  
LogisticRegression(solver="saga")  
LogisticRegression(solver="liblinear")
```

```
pd.read_csv(path, engine="c")  
pd.read_csv(path, engine="pyarrow")
```

```
# training.yaml  
model:  
  solver: saga
```

Why a string

When the choice has to leave the code (a config file, a command line, an experiment table), the policy needs a name (a string) to identify it and pass it in.

A different problem

```
def handle_trip_event(trip, event):  
    if event.kind == EventKind.DRIVER_ACCEPTED:  
        trip.state = TripState.MATCHED  
        ...  
    elif event.kind == EventKind.TRIP_COMPLETED:  
        trip.state = TripState.COMPLETED  
        ...  
    elif event.kind == EventKind.CANCELED_BY_RIDER:  
        trip.state = TripState.CANCELED  
        ...
```

Nobody checks the current state

The rider cancels.
A late
DRIVER_ACCEPTED
arrives anyway.
The trip goes back
to matched.

Each state names what can come next

```
class Matched:
    def on(self, event):
        if event.kind == EventKind.TRIP_STARTED:
            return InProgress()
        if event.kind == EventKind.CANCELED_BY_RIDER:
            return Canceled()
        raise InvalidTransition(self, event)
```

```
class Canceled:
    def on(self, event):
        raise InvalidTransition(self, event)
```

```
# in the handler...
trip.state = trip.state.on(event)
trip_repo.save(trip)
...
```

The current state decides

The handler no longer checks anything. A late acceptance on a canceled trip now raises. Design for a lifecycle.

Same class diagram, BTW

A context holds one object behind an interface. Only the decision differs.

Identical structure, different decision

	Strategy	State
Who picks the implementation	the caller	the current implementation
Do variants know each other	no	yes, each names its successors
What varies	the policy (chosen once)	the object's behavior over time
In our opening code	pricing	the trip lifecycle

A rule of thumb

A class diagram shows the shape. The pattern is the decision (e.g., what varies, and who decides).

Absorbed vs. still yours

	Handled by the language or framework	Still your decision
Strategy	First-class functions. <code>sorted(key=...)</code> is the pattern in one argument.	When a class is still right: it needs a name and a docstring, it is chosen by a string in config (<code>solver: saga</code>), or it is three related operations.
State	The big ones: workflow engines (Temporal, AWS Step Functions) run long-lived state machines. The tiny ones: an enum and a <code>match</code> .	The medium one: an in-process lifecycle with a handful of states and rules about transitions. Most trip lifecycles live here.

To Strategy or not to Strategy

Context: one ETA policy, no more on the horizon.

```
class EtaPolicy(Protocol):  
    def eta(self, origin, destination): ...  
  
class MapsEta:  
    def eta(self, origin, destination):  
        return maps_client.eta(origin, destination)  
  
eta_policy = MapsEta() # in the handler...  
eta = eta_policy.eta(event.driver_location, trip.pickup)
```

What the code needed

`eta = maps_client.eta(...)` . One line.

The principle

YAGNI ("you aren't gonna need it"). One implementation and nothing on the roadmap is not a variation point.

Who reacts to a change?

```
def handle_trip_event(trip: Trip, event: TripEvent) -> None:
    if event.kind == EventKind.DRIVER_ACCEPTED:
        trip.state = TripState.MATCHED
        eta = maps_client.eta(event.driver_location, trip.pickup)
        push.send(trip.rider_id, f"Driver on the way, {eta} min out")
        analytics.track("trip_matched", trip.id, driver=event.driver_id)
    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)
    elif event.kind == EventKind.CANCELED_BY_RIDER:
        trip.state = TripState.CANCELED
        push.send(trip.driver_id, "Rider canceled the trip")
        fee = 0.0
        if minutes_since(trip.matched_at, event.timestamp) > 5:
            fee = 5.00
        if trip.rider_is_new:
            fee = 0.0
        payments.charge(trip.rider_id, fee)
        analytics.track("trip_canceled", trip.id, by="rider")
    # ... DRIVER_ARRIVED, TRIP_STARTED, CANCELED_BY_DRIVER
    trip_repo.save(trip)
```

Record the change once, let consumers subscribe

```
elif event.kind == EventKind.TRIP_COMPLETED:           # trips/events.py
    trip.state = TripState.COMPLETED
    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
```

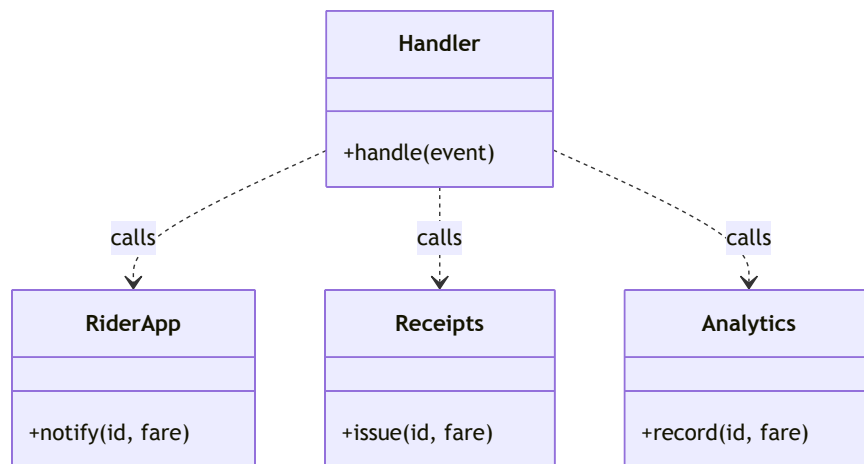
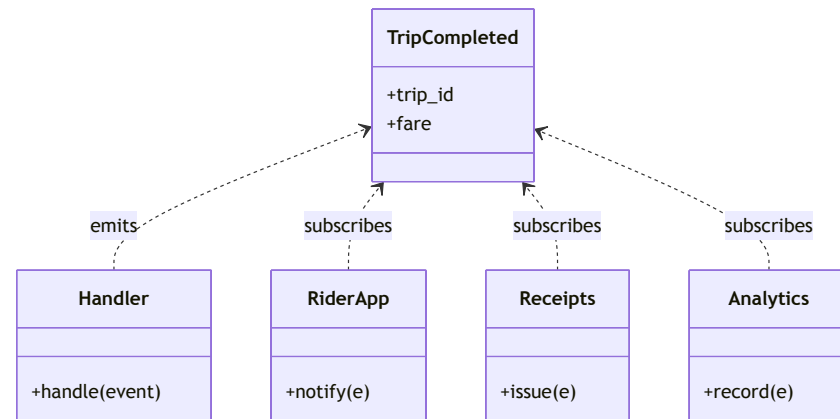
What changed

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 no longer depends on its consumers. Both depend on the event type. **Dependency inversion**. Design for consumers that come and go.

Note: Observer flips the arrows



You may have already used Observer

```
const trips = new EventEmitter()

trips.on('completed', (trip) => receipts.issue(trip))
trips.on('completed', (trip) => analytics.record(trip))

trips.emit('completed', trip)
```

```
button.addEventListener('click', save)
button.addEventListener('click', track)
```

The JDK deprecated its own Observer

“The event model supported by Observer and Observable is quite limited, the order of notifications delivered by Observable is unspecified, and state changes are not in one-for-one correspondence with notifications.

— `JAVA.UUTIL.OBSERVABLE`, `JAVA 9 API DOCUMENTATION`

The shape was easy to ship, but the decisions were not.

The complaints above (ordering, lifecycle, what counts as a change) are the decisions a library cannot make for you.

Eventing is everywhere, mostly as library surface

Where	Mechanism
Language	C# <code>event</code>
Standard surface	Node <code>EventEmitter</code> , DOM <code>addEventListener</code>
Framework	Django signals, Qt signals
Different shape	Go channels
Deprecated	Java <code>Observable</code>

What stays your decision

Decision	In the scenario
Lifecycle	Who unsubscribes, and when. (This is where the leaks can sneak in.)
Ordering	The receipt must not go out before the charge succeeds.
Errors	One observer throws. An analytics failure must not block the payment.
Traceability	Who reacts to <code>TripCompleted</code> ? Can you just <code>grep</code> it?
Backpressure	Trips complete faster than analytics can record them. Who slows down, and who is told? (Plain Observer does not address this.)

To Event or not to Event

Context: only one consumer (receipts) and it is staying that way "for a while".

```
# trips/events.py
events.emit(TripCompleted(trip.id, fare))

# receipts/wiring.py, somewhere else
events.subscribe(TripCompleted, receipts.issue)
```

What the code needed

`receipts.issue(trip, fare)`, in the handler.

What the event cost

Nobody reading the handler can see that a receipt goes out.

Which rule decides?

A new rider who cancels late pays nothing

```
elif event.kind == EventKind.CANCELED_BY_RIDER:  
    fee = 0.0  
    if minutes_since(trip.matched_at, event.timestamp) > 5:  
        fee = 5.00  
    if trip.rider_is_new:  
        fee = 0.0  
    payments.charge(trip.rider_id, fee)
```

Policy or bug?

The waiver wins because it comes last. Nothing says that order was intended.

Each rule answers or passes

```
def new_rider_waiver(trip, event):  
    return 0.0 if trip.rider_is_new else None  
  
def late_cancellation(trip, event):  
    mins = minutes_since(trip.matched_at, event.timestamp)  
    return 5.00 if mins > 5 else None  
  
FEE_RULES = [new_rider_waiver, late_cancellation]  
  
def cancellation_fee(trip, event):  
    for rule in FEE_RULES:  
        fee = rule(trip, event)  
        if fee is not None:  
            return fee  
    return 0.0
```

Who picks? The rules, in order

The waiver comes first in the list, so it wins on purpose. Before, it won because its line came last.

The principle

Rules are policy, the loop is mechanism, and each can change alone. Design for a rule set that grows and gets reordered.

Two flavors for Chain of Responsibility

1

FIRST MATCH WINS

Fee rules

The first rule that answers ends the walk.

```
for rule in FEE_RULES:
    fee = rule(trip, event)
    if fee is not None:
        return fee
```

∀

RUN THEM ALL

HTTP middleware

Every handler acts and passes the request on.

```
app.use(express.json())
app.use(authenticate)
app.use(rateLimit)
app.get('/trips', listTrips)
```

Absorbed vs. still yours

	Handled by the language or framework	Still your decision
First match wins	Rules engines package it for large rule sets.	Termination: no rule answers, then what (a default, an error, or silence). Ordering: swap two fee rules and new riders get charged.
Run them all	Web frameworks made the pipeline a feature. <code>app.use(...)</code> is the ordered chain.	Ordering: swap two handlers and authentication runs after the rate limit. Assembly: where the chain is built, and whether it changes at runtime.

To Chain or not to Chain

Context: two rules, fixed order, both known, none coming.

```
def cancellation_fee(trip, event):  
    for rule in FEE_RULES:  
        fee = rule(trip, event)  
        if fee is not None:  
            return fee  
    return 0.0
```

```
def cancellation_fee(trip, event):  
    if trip.rider_is_new:  
        return 0.0  
    started = trip.matched_at  
    now = event.timestamp  
    if minutes_since(started, now) > 5:  
        return 5.00  
    return 0.0
```

Same precedence, no machinery

The early returns state the order just as clearly. The list and loop are justified when rules are added, reordered, or configured.

What is still your decision?

What was absorbed, and what is still yours

Pattern	Design for	Handled by the tooling	Still your decision
Strategy	a known axis of change	first-class functions	naming, config selection, multi-method variants
State	a lifecycle	the large ones: workflow engines	the medium ones: transition topology (where state lives)
Observer	consumers that come and go	events, signals	lifecycle, ordering, errors, backpressure, traceability
Chain	a growing, reorderable rule set	middleware conventions	ordering, termination, first-match or run-all, assembly

Spoiler for Wednesday

Every row "bets" on a change. For this week's studio: which bet do you place before you know what will change?

Avoid flexibility that nobody uses

Pattern	Skip it when	Do this instead
Strategy	one implementation, none on the roadmap	keep the conditional, or the one-liner
State	two or three states, transitions that never change	an enum and a <code>match</code>
Observer	one consumer, and you know who it is	call the method
Chain	a few rules, fixed order, none coming	write the if/else with early returns

Another rule of thumb

Each pattern adds indirection to make one change cheaper. If that change never comes, it is all cost. When it does come, you still paid (an extra hop to read, a call site that no longer says what happens, etc.). Avoid "pattern soup"!

Summary: design principles from today

Principle	One sentence summary	Where you saw it
Cohesion	related work lives together	one handler for every event kind
Coupling	what you must know to change another part	five services called by name
Information hiding	hide what is likely to change	pricing, consumers, fee order exposed
DRY	one place for each piece of knowledge	notify-and-track, three times
Policy and mechanism	rules apart from the code that runs them	the fee ladder
Open-closed	add variants without editing	<code>pricing.fare(...)</code>
Dependency inversion	depend on the abstraction	the arrows flip toward <code>TripCompleted</code>
YAGNI	no building for variation that is not coming	every "to X or not to X" slide

Reading: the rest of the behavioral catalog

Pattern	The idea in one line
<u>Command</u>	a request becomes an object you can queue, log, or undo
<u>Template Method</u>	the skeleton is fixed; subclasses fill in the steps
<u>Iterator</u>	walk a collection without exposing how it is stored
<u>Mediator</u>	objects talk through a hub instead of to each other
<u>Memento</u>	snapshot an object's state so you can restore it
<u>Visitor</u>	add a new operation without touching the classes it visits

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

This week

Wednesday • studio

Picking a driver for this platform. The theme: does choosing the policy deserve a structure?

A2 • due Sunday, September 13

Design judgment applied to a real codebase. (Handout on the course website.)



Today, we learned how to ...

- Identify the design problem that motivates a behavioral pattern in unfamiliar code
- Argue for or against applying a behavioral pattern using concrete trade-offs
- Explain which parts of a pattern modern tooling has absorbed and which decisions remain yours