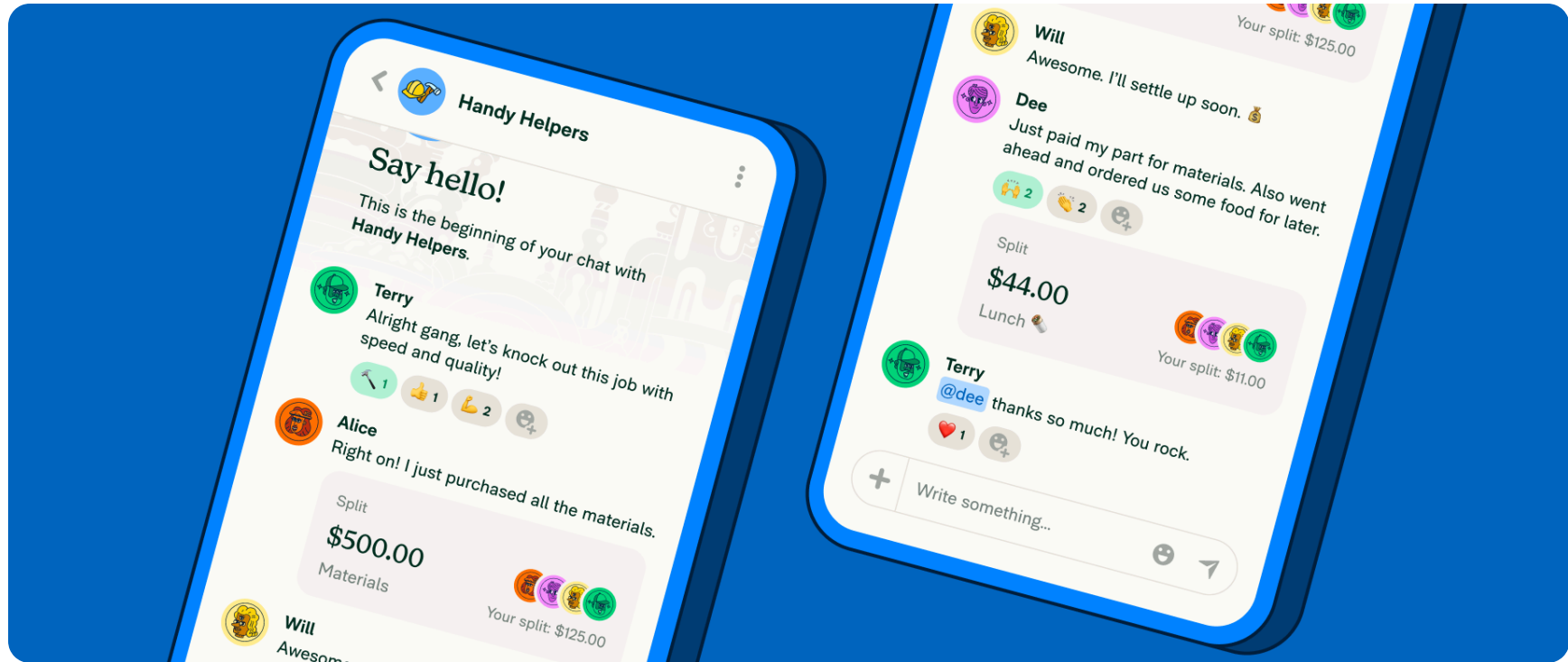


17-695: Design Patterns

Course introduction

Monday, August 24 • Lecture

We need to design an expense-splitting app



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

So, let's get AI to design the app for us



PRODUCT REQUEST

Build an expense-splitting app for a group. Let members record an expense (who paid, how much, and how it splits); see each member's net balance and who owes whom; and record a payment when members settle up.

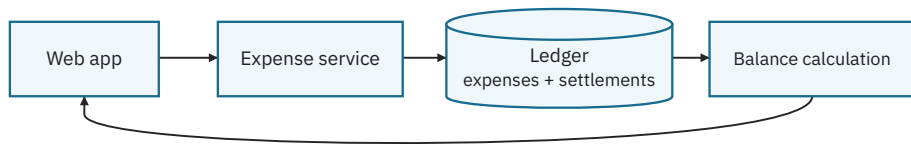
Required behavior: Amounts are exact to the cent. Every expense uses explicit shares that sum to its total.



Add context



Design A: derive current balances



Stored

Expenses and settlements

Computed on read

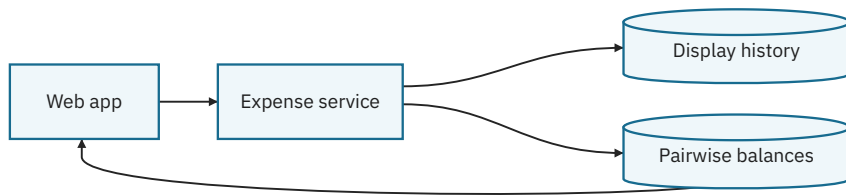
Current balances

```
function addExpense(expense: Expense) {
  ledger.append({ type: "expense", ...expense })
}

function balances(groupId: GroupId) {
  return ledger.forGroup(groupId).reduce(
    applyEntry,
    emptyBalances(),
  )
}

function settle(payment: Payment) {
  ledger.append({ type: "settlement", ...payment })
}
```

Design B: store current balances



Source of truth

Current pairwise balances

Updated on write

Every affected pair

```
function addExpense(expense: Expense) {
  db.transaction(() => {
    history.insert(expense)
    for (const share of expense.shares) {
      if (share.member !== expense.payer)
        balances.netDebt(share.member, expense.payer,
                        share.cents)
    }
  })
}

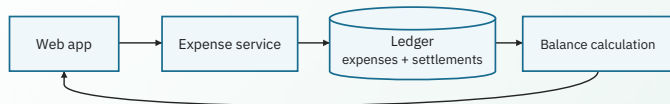
function balancesFor(groupId: GroupId) {
  return balances.forGroup(groupId)
}

function settle(payment: Payment) {
  balances.reduceDebt(payment.from, payment.to,
                    payment.cents)
}
```

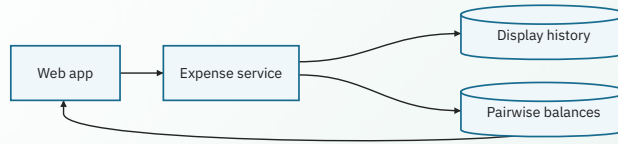
Both implementations behave "as expected"

Action	Expected Result	A	B
Ava pays \$60 for Ava, Ben, Cara	Ben owes Ava \$20; Cara owes Ava \$20	✓	✓
Ben pays \$30 for Ava and Ben	Ben owes Ava \$5; Cara owes Ava \$20	✓	✓
Cara pays Ava \$20	Ben owes Ava \$5; Cara is settled	✓	✓

Which design would you choose?

A

Ledger · store what happened

B

Stored balances · store what is true now

The answer is almost always "it depends"

A

DESIGN A

Keep the ledger

Store each expense and settlement;
derive balances when asked.

Traceable

the full history explains every balance

One source of truth

each event is stored once

Testable rules

balance logic stays in the calculation

B

DESIGN B

Keep current balances

Update each affected balance
whenever activity is recorded.

Fast reads

the common balance view is a direct lookup

Work on write

calculation happens when activity is recorded

Explicit state

debtor-creditor pairs are stored directly

So, which version is "right"?

We didn't specify a design in the prompt

Decision	A • Ledger	B • Balances
Stored representation	expense and settlement events	events + current pairwise balances
Balance work	derive on read	update on write
Main read path	scan + reduce	direct lookup

What are our functionality constraints?



PRODUCT

Valid splits

Shares must sum to the expense total.

Exact money

amounts remain exact to the cent



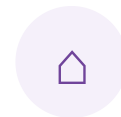
CORRECTNESS

Consistent state

History and balances cannot disagree after a write.

Atomic writes

coupled updates succeed or fail together



ENVIRONMENT

Fixed context

Some choices may already be made for us.

Existing limits

technologies, regulations, budget, and time

"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.

**Design means choosing
trade-offs under real constraints.**

Stepping back: why care about design at all? Can't we just change it later?

AI / AI ENGINEERING / DEVELOPER TOOLS / VIBE CODING

"Code should be regenerated, not maintained": Codeplain makes the case for spec-driven development

The phoenix of Greek myth burns and is reborn from its own ashes -- a metaphor this founder is applying to AI-generated code.

Jun 25th, 2026 1:47pm by [Paul Sawers](#)

Paul Sawers, The New Stack, June 2026 (thenewstack.io)

What has to change if an old expense is edited?

People can now **edit or delete a past expense**, even after settling up.



Imagine: a dinner was \$45, not \$60

What happened

1. Ava records a **\$60 dinner**, split three ways.
2. Ben and Cara each owe Ava **\$20**.
3. Cara pays Ava **\$20**.

Then Ava corrects the expense

The dinner should have cost \$45, not \$60.

The corrected balances

Ben owes Ava \$15, and Ava owes Cara \$5.

Design A appends a correction

```
ledger.append({  
  type: "expense-corrected",  
  expenseId: 42,  
  amount: 45_00,  
})
```

Write

Store one new fact. Leave the original expense and payment intact.

Read

Teach the existing ledger fold to interpret the correction.

Design B reverses and reapplies the old split

```
transaction(() => {  
  const old = history.get(42)  
  const corrected = { ...old, amount: 45_00 }  
  
  balances.remove(split(old))  
  history.replace(42, corrected)  
  balances.add(split(corrected))  
})
```

The edit handler needs the old split

It must reconstruct the exact balance change before it can reverse it.

Cara's \$20 payment still happened

Only the dinner's contribution is reversed and recalculated.

The same edit requires different work

A

DESIGN A

Derive on read

The ledger remains the authoritative state.

Data

Add a correction or reversal record.

Code

Teach the ledger fold to interpret it.

Tests

Exercise edited and deleted histories.

B

DESIGN B

Maintain on write

History and current balances must stay aligned.

Data

Retain enough detail to reverse the old split.

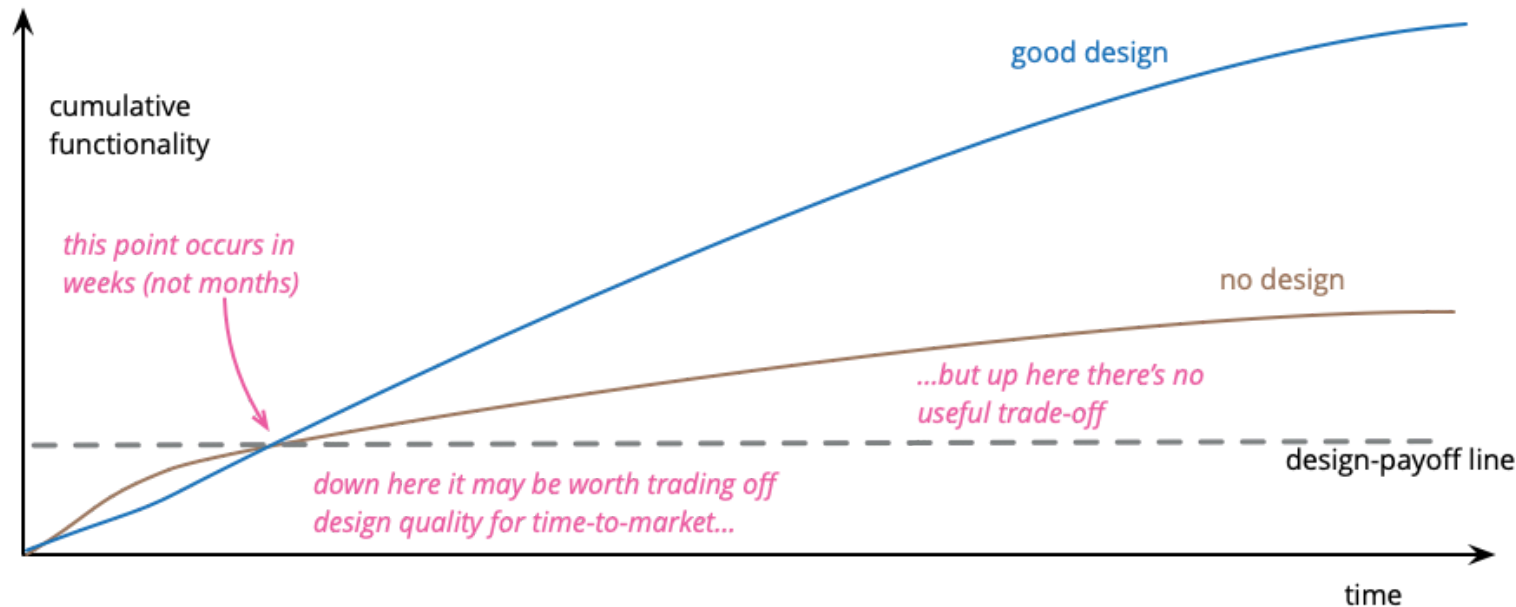
Code

Reverse and reapply deltas atomically.

Tests

Cover netting, ordering, and failed writes.

Back to "can I change it later?"



Martin Fowler, "Design Stamina Hypothesis" (martinfowler.com)

Where do design patterns fit?

A pattern packages a recurring design decision



Context

The problem recurs, and some conditions make it important.



Decision

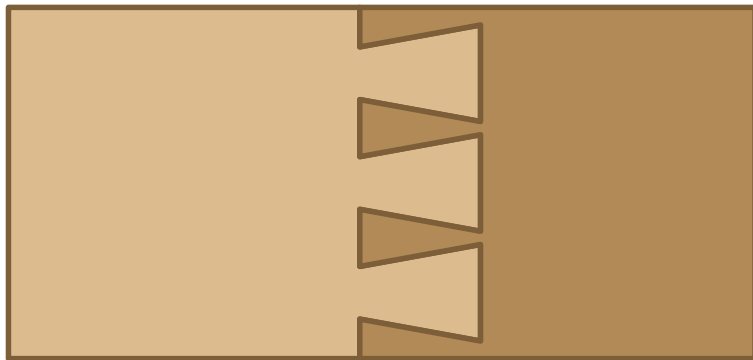
A familiar design move responds to that problem.



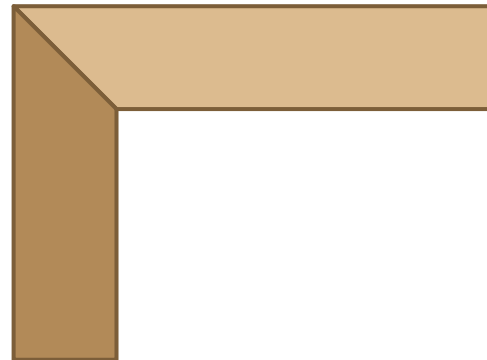
Trade-offs

The move makes some things easier and others harder.

Anyone here familiar with carpentry?



Dovetail joint



Miter joint

“Append-only ledger” works the same way

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.

Patterns are defined by intent, not syntax

Not a code template

The same intent can take different forms in different systems.

Not a vocabulary quiz

Knowing the name does not show that its conditions hold.

Not a quality badge

Extra structure must justify its cost in this context.

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

Whether it belongs here is still your call



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?

So, where do design pattern names fit in?

Start with a concrete problem

What is difficult here, and why does it matter?

Compare designs and trade-offs

Include the direct option, and doing nothing.

Then attach the pattern name

Use the shared vocabulary to discuss whether the move fits.

**Back to our initial example.
Both apps “work.” How do we
know?**

The green checks covered one sequence

The green checks

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

What we can claim

This sequence worked for these inputs, in this order.

What didn't we test?

An awkward split

\$10.00 $\div$ 3 leaves a cent. Who receives it?

Two edits at once

Ava and Ben edit the same expense. Which update survives?

Exact arithmetic

Does every calculation remain exact through storage and serialization?

Turning “it works” into a claim you can check

“Every accepted split allocates **every cent**.”

Test the behavior

Use awkward totals and group sizes.
Assert that the shares sum to the original total.

Inspect the decision

Check the amount type and the rule that assigns the remainder.

What would you check before shipping?



Before you accept generated code

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.

How the course works

Your instructors



INSTRUCTOR

Hammad Ahmad

hammad@andrew.cmu.edu
TCS 341



INSTRUCTOR

Chris Timperley

ctimperley@cmu.edu
TCS 245

Your teaching assistants

TEACHING ASSISTANT

Sanjay Kumar Baskaran

sanjaykb@andrew.cmu.edu

TEACHING ASSISTANT

Riya Elizabeth John

rjohn@andrew.cmu.edu

TEACHING ASSISTANT

Soham Narendra Joshi

sohamnaj@andrew.cmu.edu

The seven weeks ahead

Week	Topic
1	Course introduction (today)
2	Behavioral patterns
3	Creational patterns
4	Structural patterns
5	Concurrency and asynchrony
6	Refactoring and brownfield systems
7	Review, the future of design patterns, and project presentations

Each week typically moves from ideas to practice to independent work

MONDAY



Lecture

Compare designs, name the problems, and build vocabulary.

WEDNESDAY



Studio

Practice design judgment and leave with a concrete artifact.

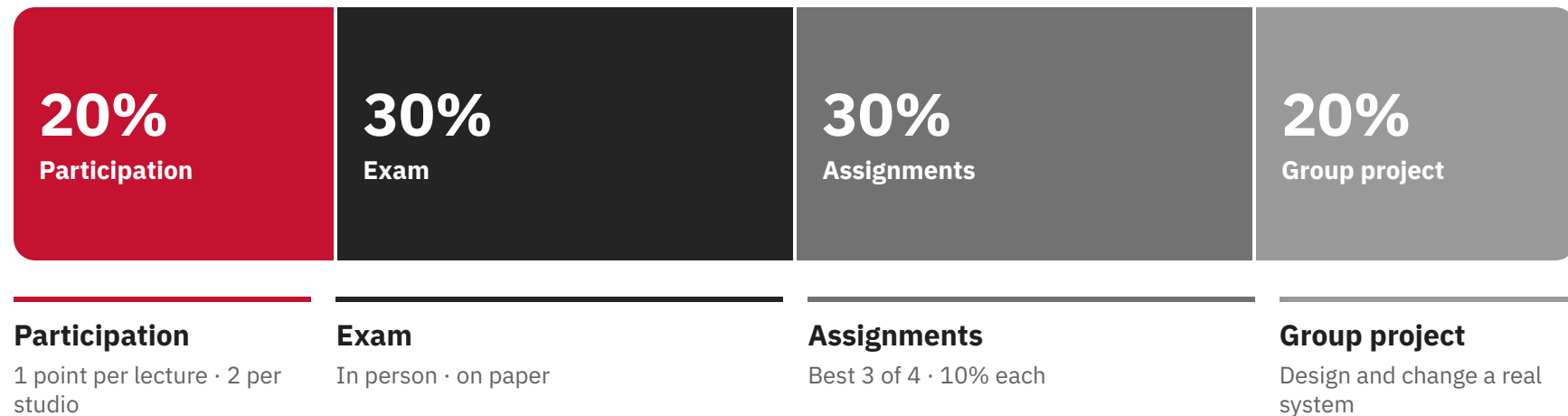
SUNDAY



Assignment

Apply the reasoning independently and explain your evidence.

Grade breakdown



You are responsible for everything you submit

Understand

the design and code you accept

Disclose

AI use through the required logs

Verify

the behavior and the decisions made for you

Defend

your trade-offs with evidence



ASSIGNMENTS

Use AI. Show your work.

Submit the required logs and explain the decisions you accepted or changed.



EXAM

Reason without AI

Demonstrate the same design judgment yourself, in person and on paper.

Course infrastructure

Course website

Published syllabus, schedule, assignment descriptions, and course staff.

cmu-17695.github.io 

Canvas + Gradescope

Submissions, grades, feedback, regrades, and course announcements.

[Canvas · 17-695](#) 

You can access Gradescope through the Canvas course.

Questions about the course? (Or the world, in general?)

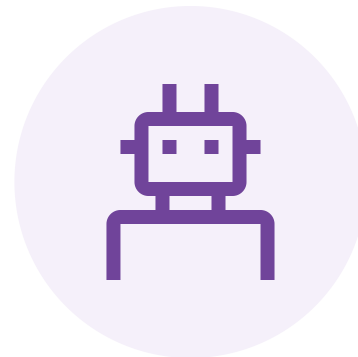


This Wednesday



Your laptop

Charged and ready to use.



Your AI coding agent

Installed, signed in, and able to export a session transcript.



Today, we learned how to ...

- Explain why design quality remains an engineering responsibility when AI generates working code
- Describe a design in terms of change, constraints, quality attributes, and trade-offs
- Defend a preference between working implementations in quality-attribute terms
- Identify evidence needed to accept an AI-assisted implementation