# modular & legible software for AI (& human) coders

Daniel Jackson · AI & Coding · Expanding Horizons in Computing · Jan 23, 2026

# problems with AI coding

# a benchmark and its analysis

SWE-bench

Can Language Models Resolve Real-World GitHub Issues?

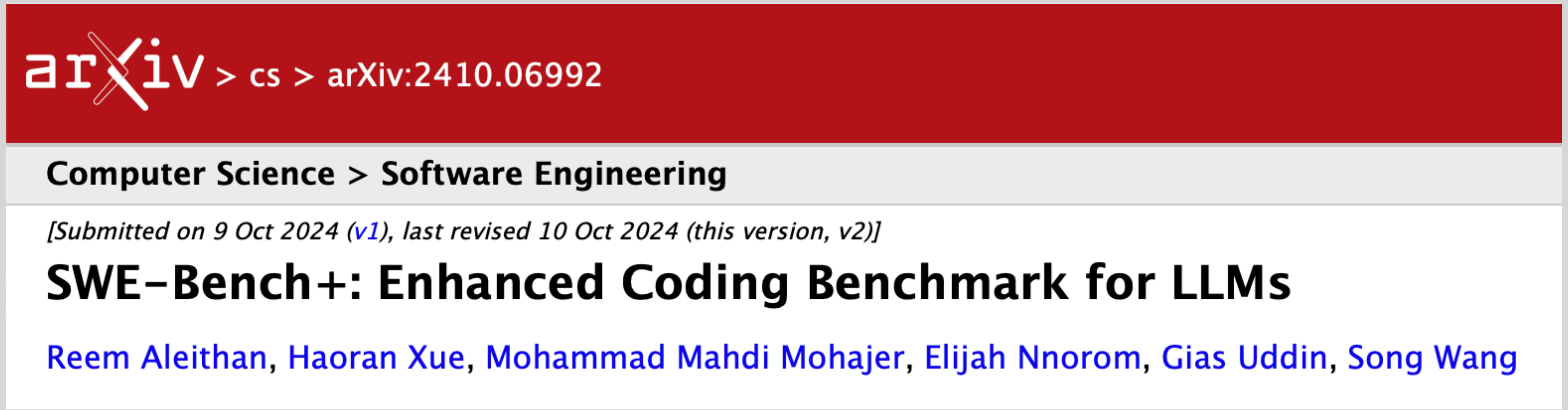
ICLR 2024

Carlos E. Jimenez*, John Yang*,
Alexander Wettig, Shunyu Yao, Kexin Pei,
Ofir Press, Karthik Narasimhan

**a benchmark for realistic coding problems**

2,294 issue/pull request pairs from 12 Python repos

best LLM resolves 65% of issues



arXiv > cs > arXiv:2410.06992

Computer Science > Software Engineering

*[Submitted on 9 Oct 2024 (v1), last revised 10 Oct 2024 (this version, v2)]*

**SWE-Bench+: Enhanced Coding Benchmark for LLMs**

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, Song Wang

**follow-up study at York University**

33% of good patches "cheated": code appears in issue

31% of patches deemed correct by incomplete tests

94% issues were present before training cutoff

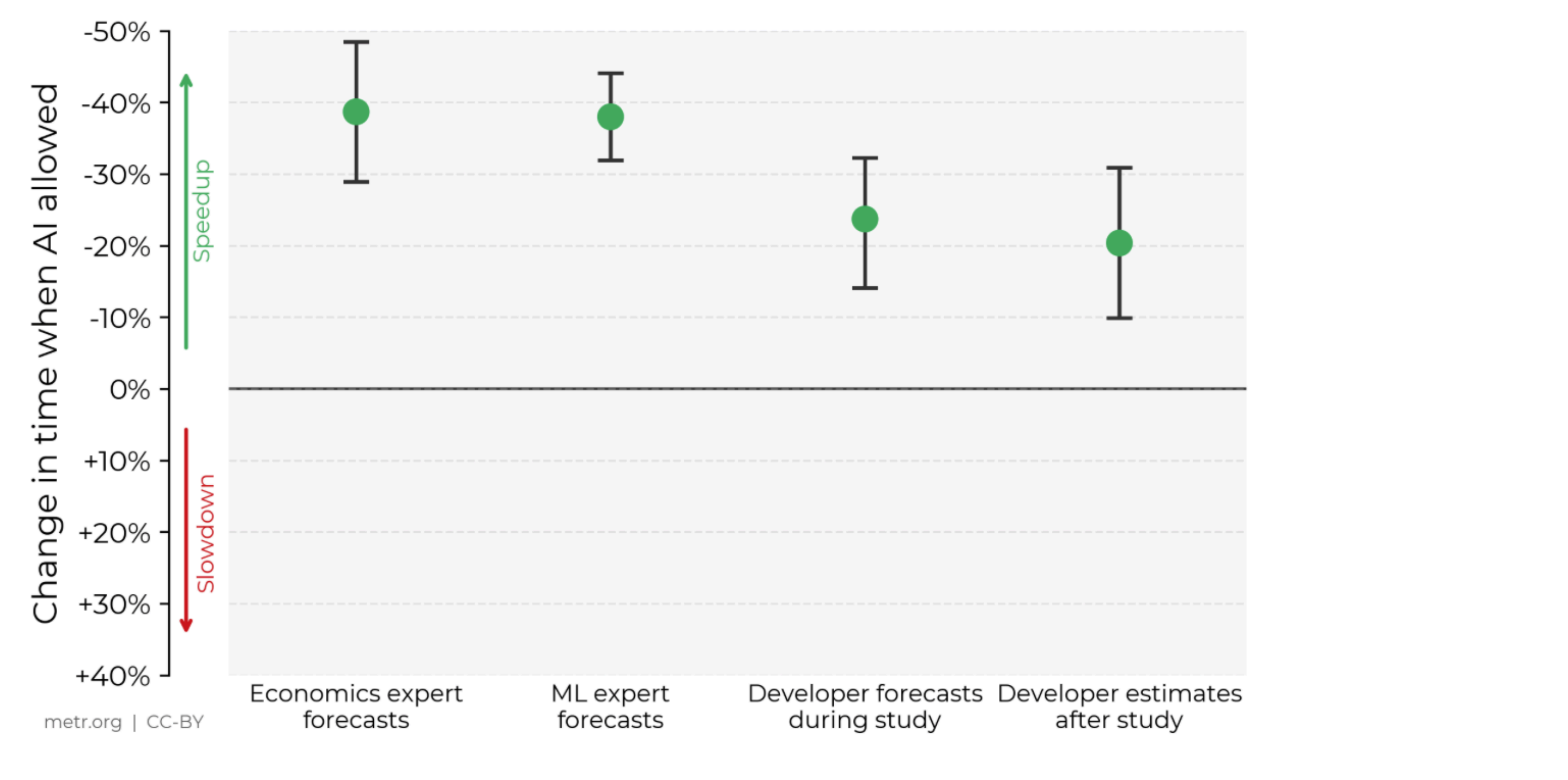
with all this, resolution rate for GPT-4 falls to 0.55%

**in short: LLM-based coding assistants**

often suggest code that doesn't work

and breaks existing functionality

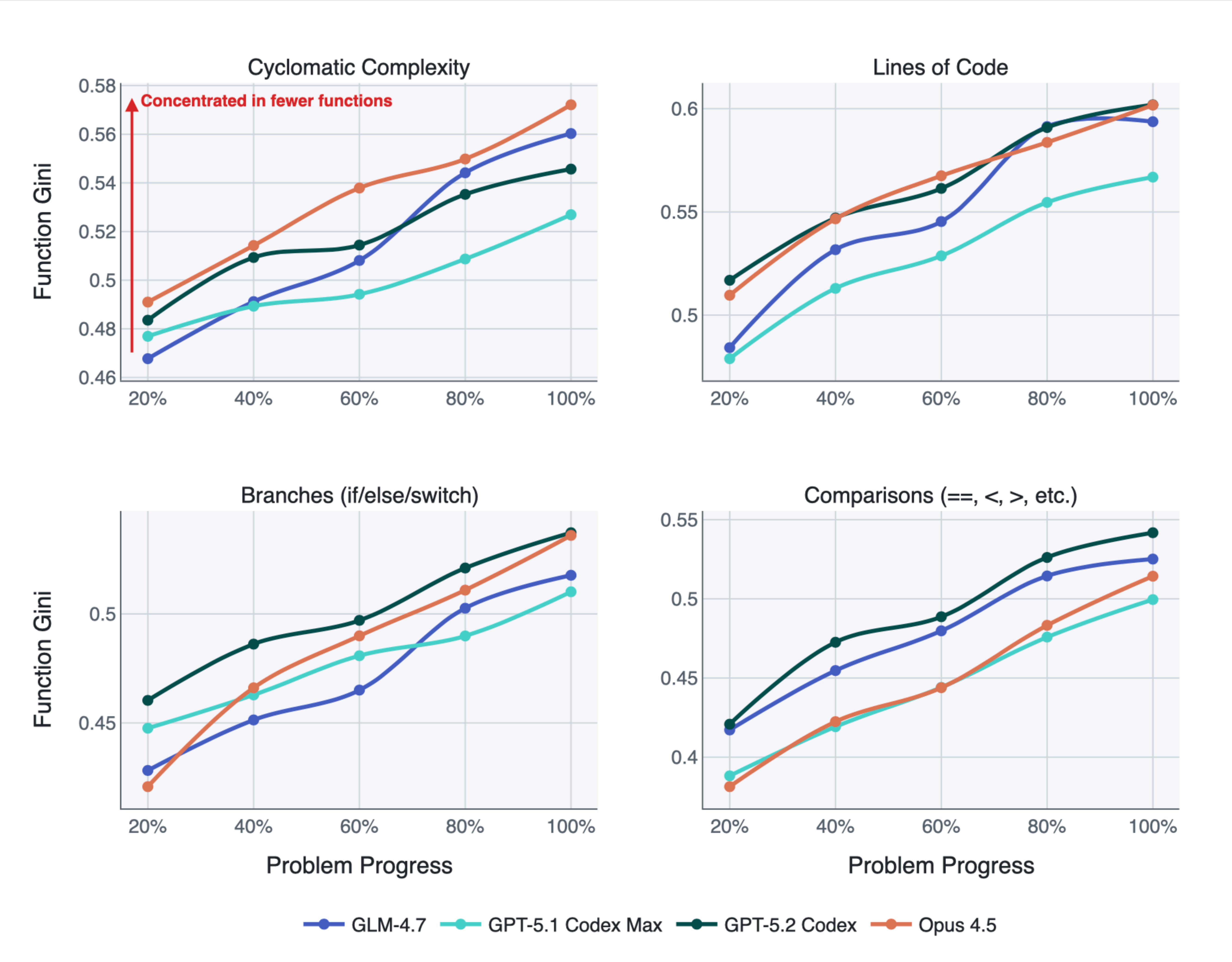
# how much does AI speedup skilled developers on real codebases?



**METR study (10 July 2025)**
Randomized control trial
16 developers on 246 tasks

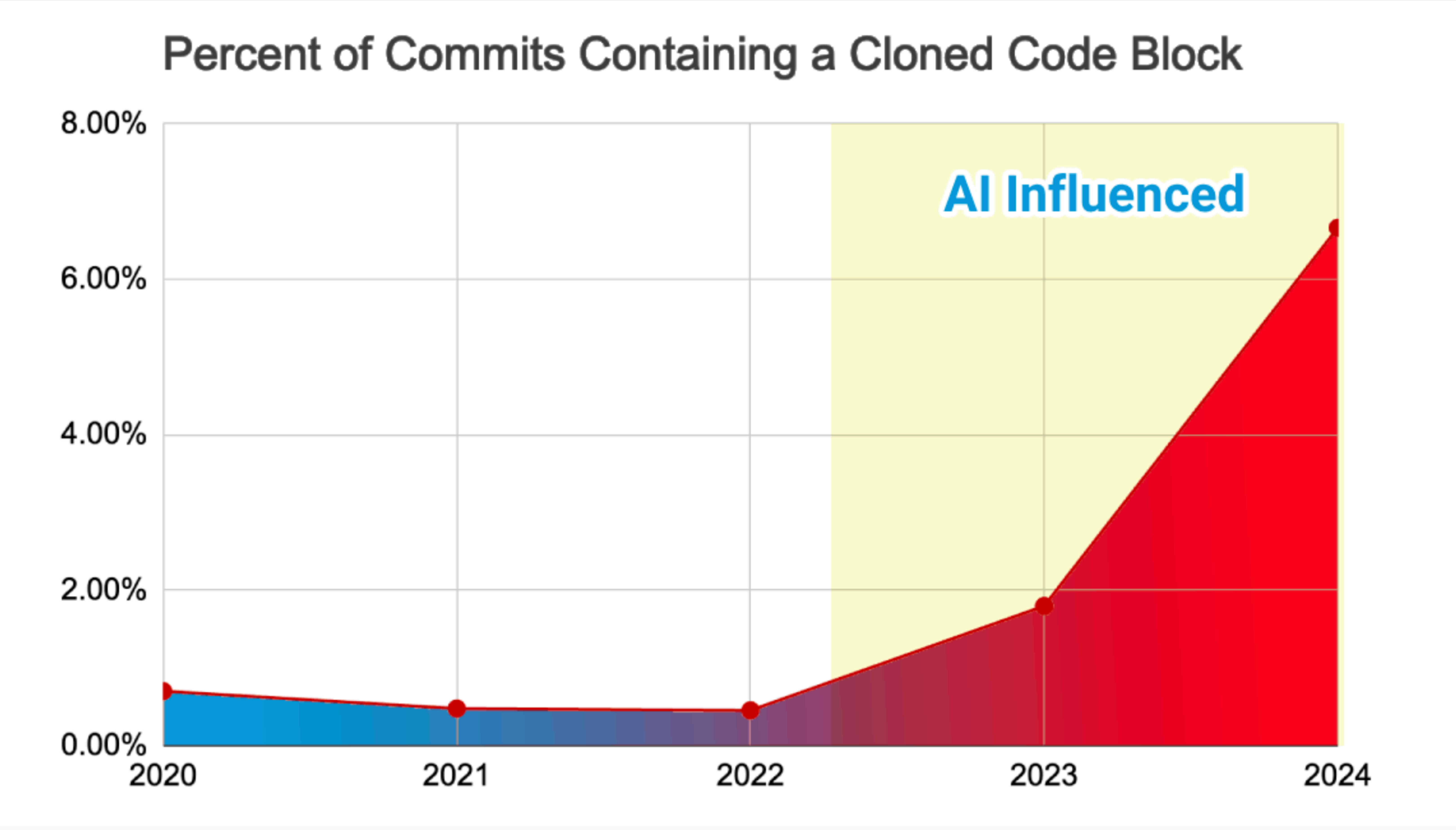
Developers qualitatively note LLM tooling performs worse in more complex environments. One developer says “**it also made some weird changes in other parts of the code** that cost me time to find and remove […] My feeling is the refactoring necessary for this PR was “too big” [and genAI] introduced as many errors as it fixed.” Another developer comments that one prompt “failed to properly apply the edits and **started editing random other parts** of the file,” and that these failures seemed to be heavily related to “the size of a single file it is attempting to perform edits on”.

# what happens when AI coders are faced with iterative changes?



Slopcodebench (Orlanski et al, 2026)

# how are AI agents affecting code?



Gitclear study of >200m lines of code from Google, Microsoft, Meta et al over 5 years (2025)

# two ways
# to work with AI

| automate the mess | design for AI |
|---|---|
| take current practices as given | rethink our practices |
| let AI agents find structure | make expressive structures |
| outsource thinking to AI | outsource subtasks to AI |
| give AI unlimited context & power | focus AI intentionally |

# what’s wrong with current practices

# what AI (and human) coders need

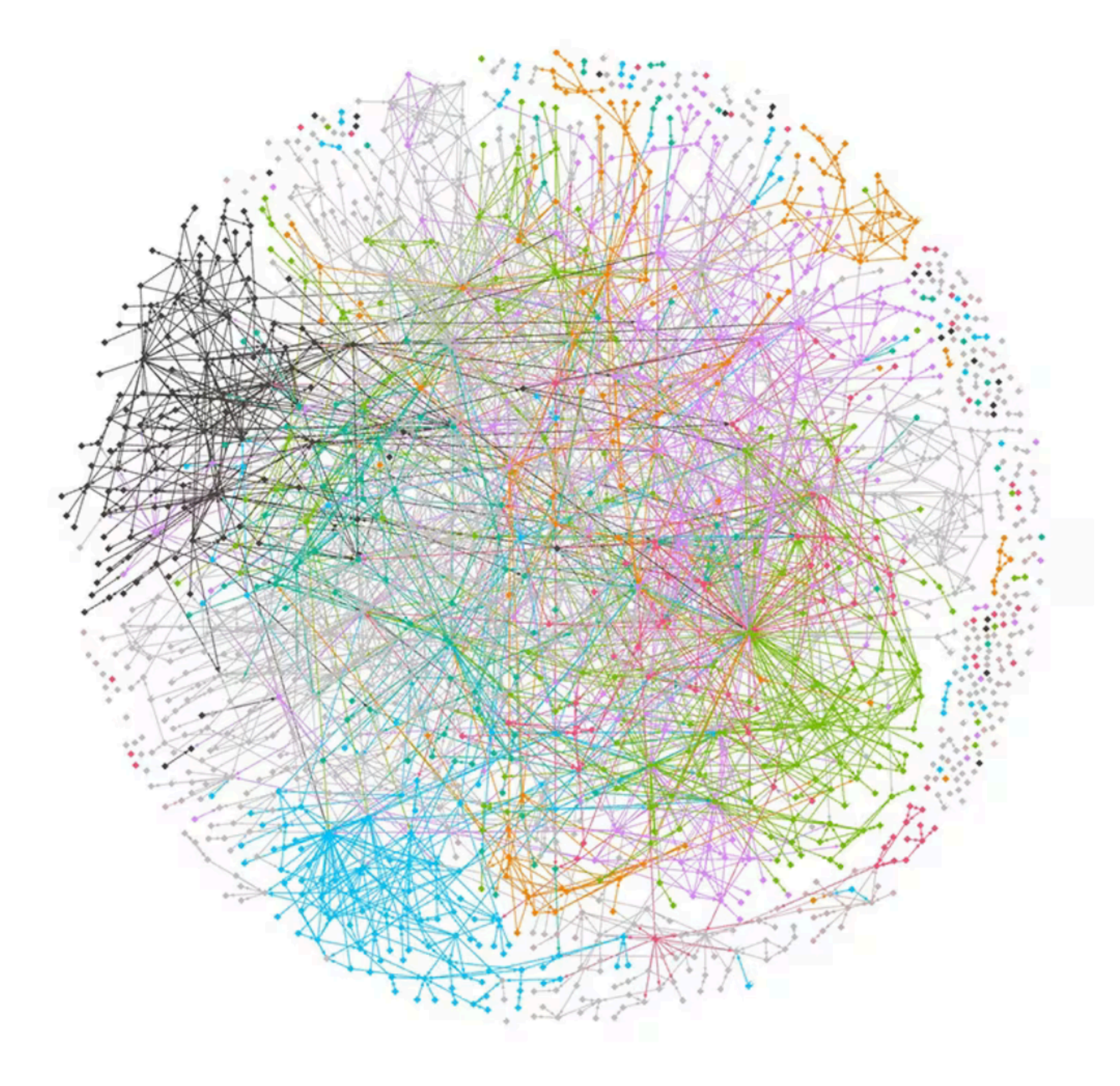
modularity (structuring software with independent units)

**reuse, incremental work, division of labor (in teams), local change**

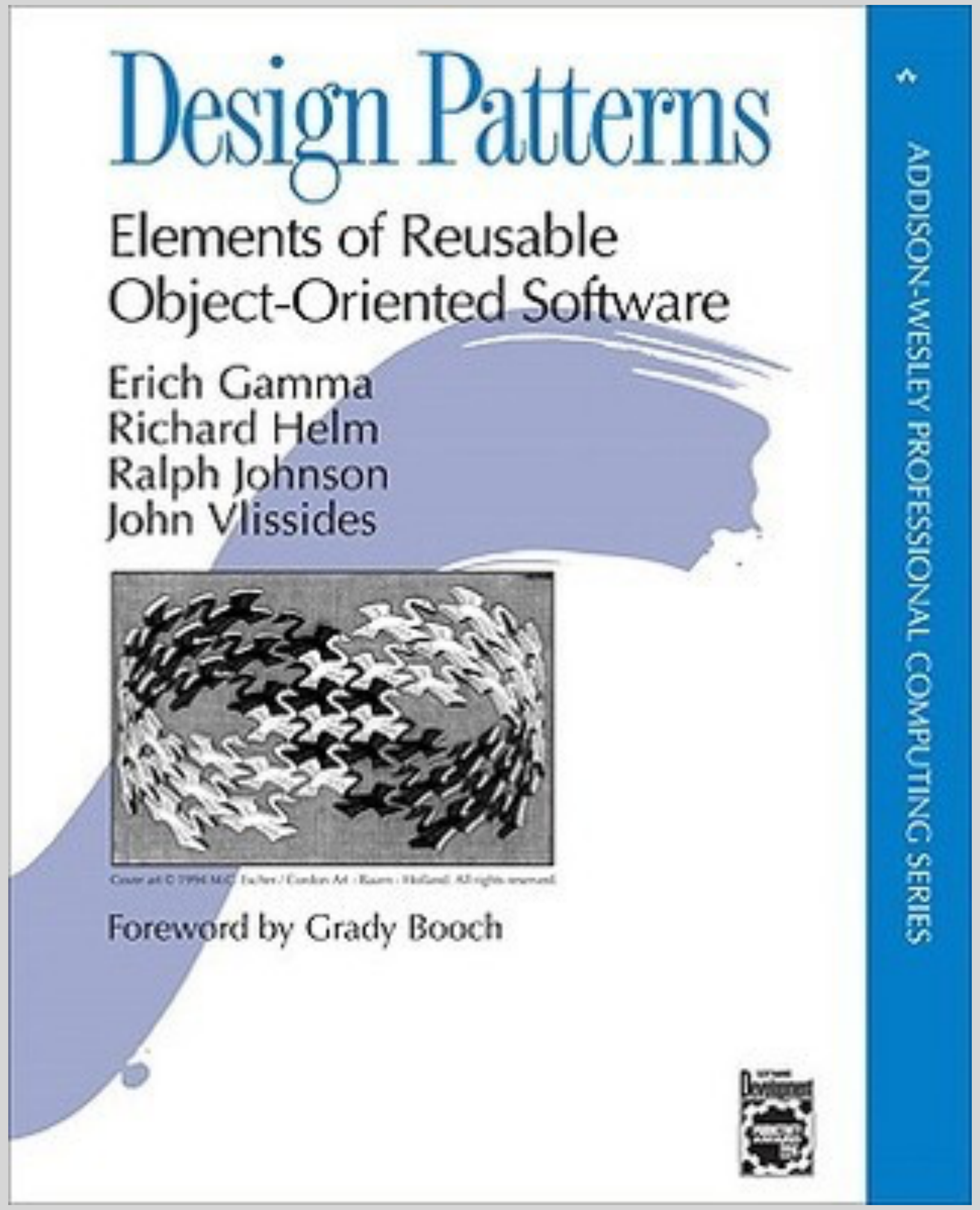
legibility (“what you see is what it does”)

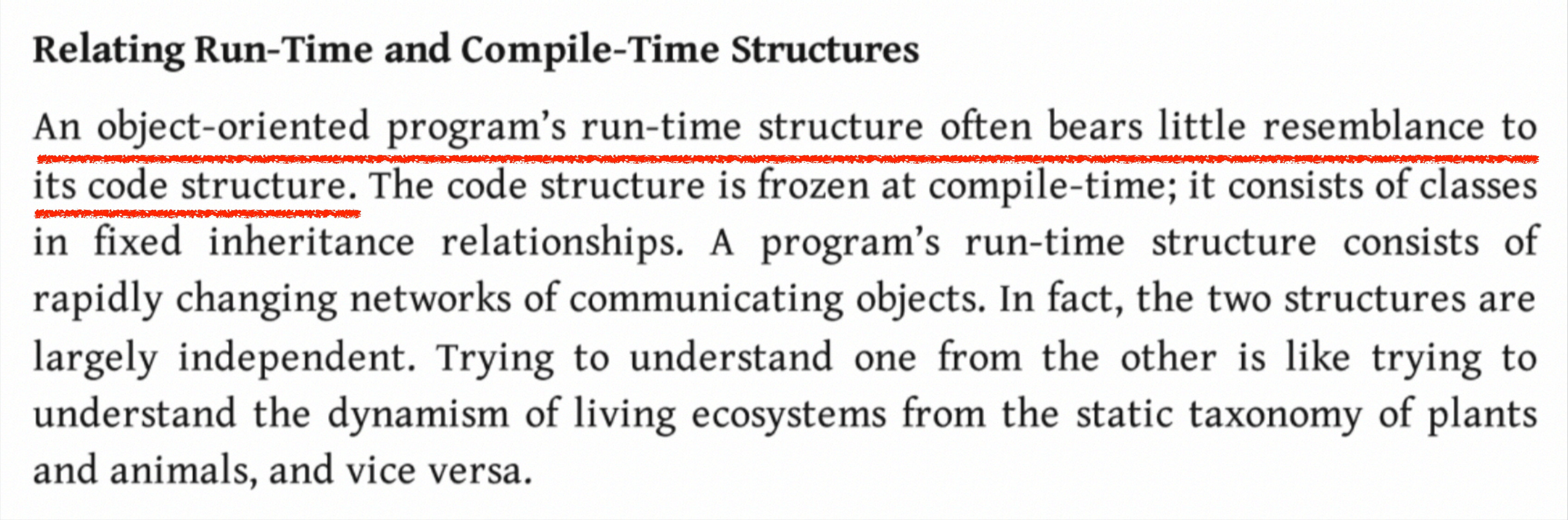
**code is easier to generate, understand, modify**

# “microservice dependency hell”



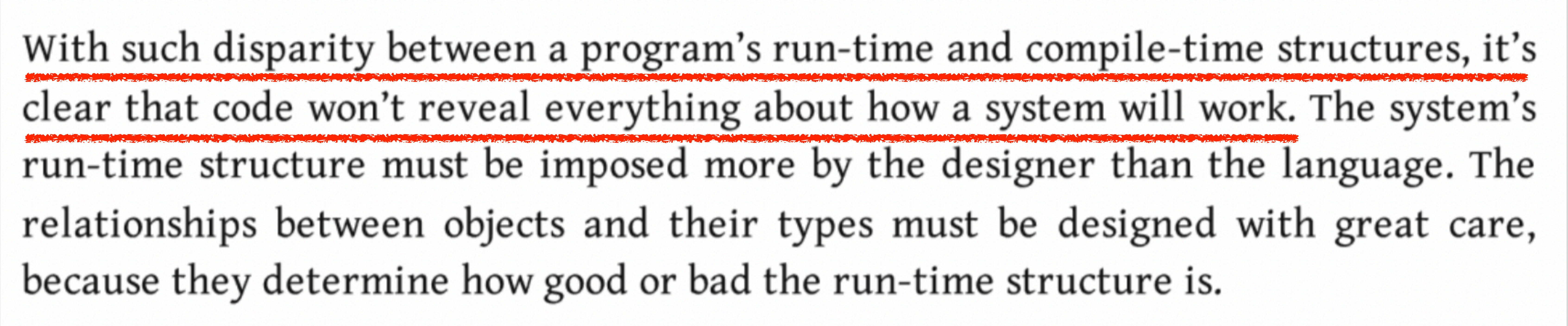
Monzo: > 1,500 services with 9,300 dependencies


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



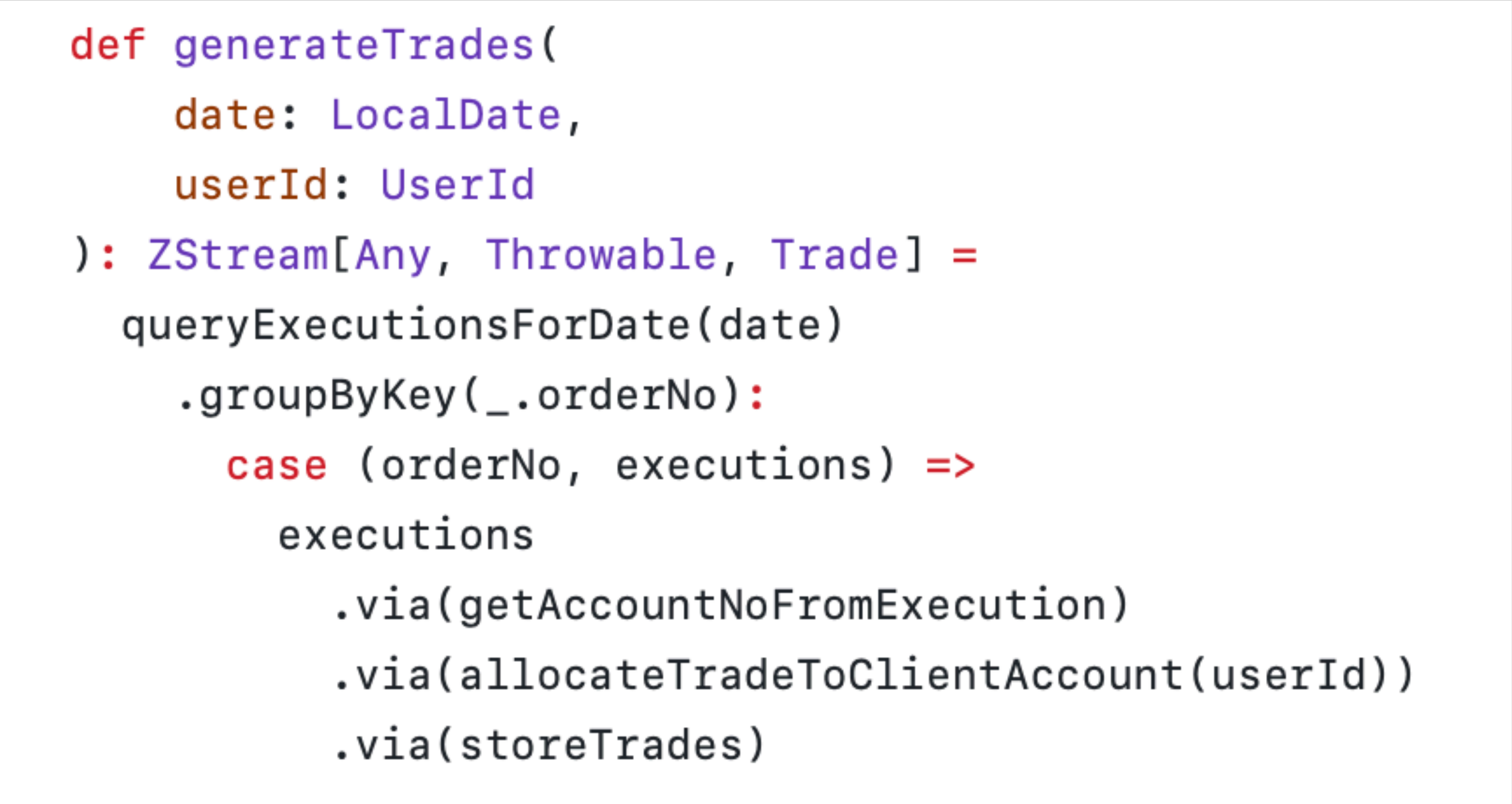
### Relating Run-Time and Compile-Time Structures

An object-oriented program's run-time structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program's run-time structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa.

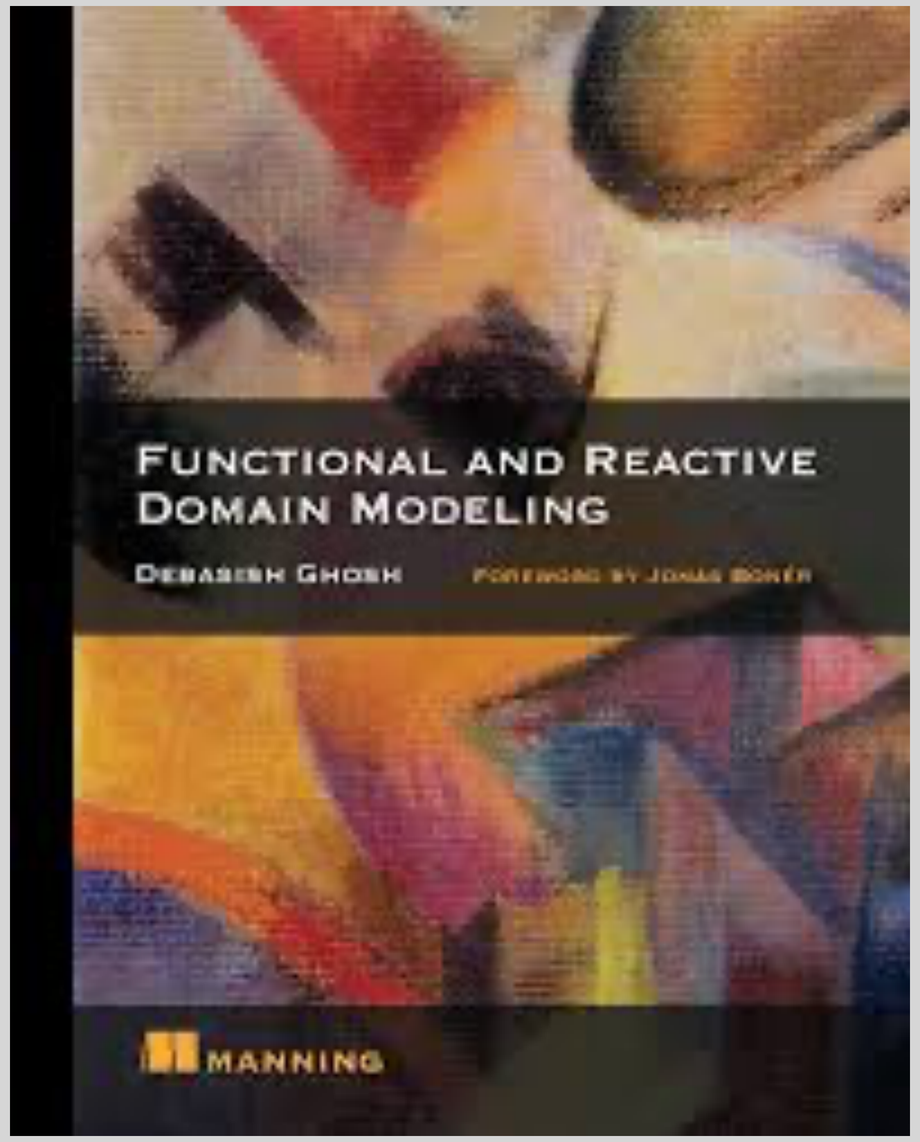


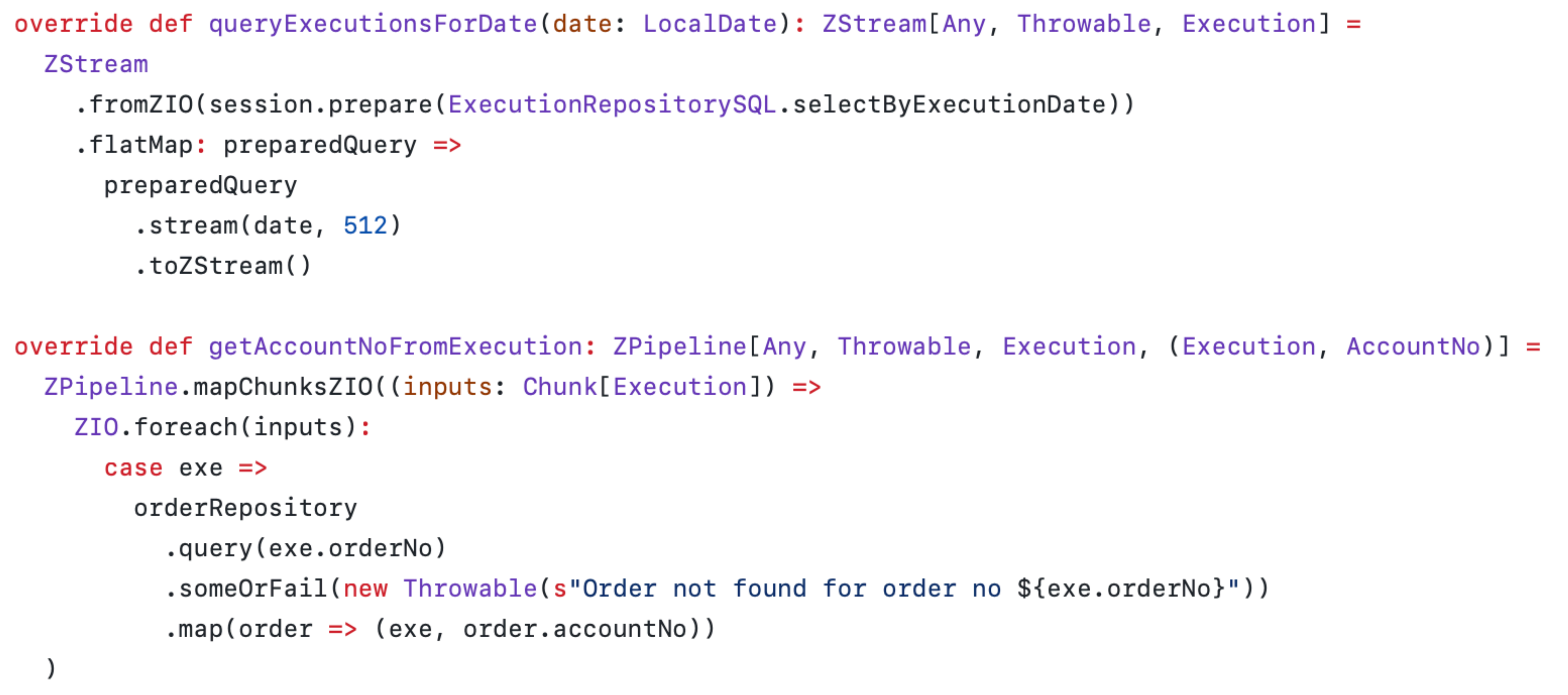
With such disparity between a program's run-time and compile-time structures, it's clear that code won't reveal everything about how a system will work. The system's run-time structure must be imposed more by the designer than the language. The relationships between objects and their types must be designed with great care, because they determine how good or bad the run-time structure is.

**Trygve Reenskaug called this "a frightening observation"**



```scala
def generateTrades(
    date: LocalDate,
    userId: UserId
): ZStream[Any, Throwable, Trade] =
  queryExecutionsForDate(date)
    .groupByKey(_.orderNo):
      case (orderNo, executions) =>
        executions
          .via(getAccountNoFromExecution)
          .via(allocateTradeToClientAccount(userId))
          .via(storeTrades)
```





```scala
override def queryExecutionsForDate(date: LocalDate): ZStream[Any, Throwable, Execution] =
  ZStream
    .fromZIO(session.prepare(ExecutionRepositorySQL.selectByExecutionDate))
    .flatMap: preparedQuery =>
      preparedQuery
        .stream(date, 512)
        .toZStream()

override def getAccountNoFromExecution: ZPipeline[Any, Throwable, Execution, (Execution, AccountNo)] =
  ZPipeline.mapChunksZIO((inputs: Chunk[Execution]) =>
    ZIO.foreach(inputs):
      case exe =>
        orderRepository
          .query(exe.orderNo)
          .someOrFail(new Throwable(s"Order not found for order no ${exe.orderNo}"))
          .map(order => (exe, order.accountNo))
  )
```

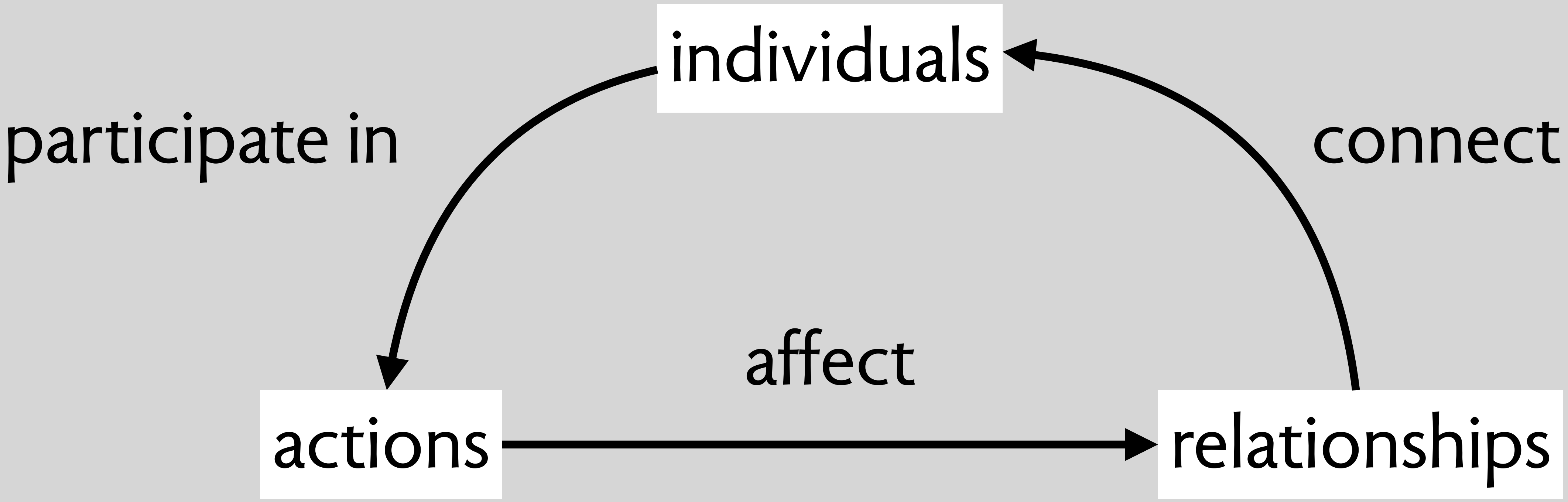
# what is behavior?

# 3 kinds of phenomena

**individuals**
have persistent identity
and limited lifetimes

**relationships**
between individuals
and also to values

**actions**
atomic occurrences
involve individuals



# example: restaurant reservation phenomena

**individuals**

| | | |
|---|---|---|
| *users* | Alice | Bob |
| *restaurants* | Maido | Rosetta |
| *slots* | Slot_1 | Slot_2 |
| *reservations* | Res_1 | |

**relationships**

for (Res_1, Slot_2)

by (Res_1, Alice)

at (Slot_2, Maido)

time (Slot_2, 7:30pm)

date (Slot_2, Jan-20-26)

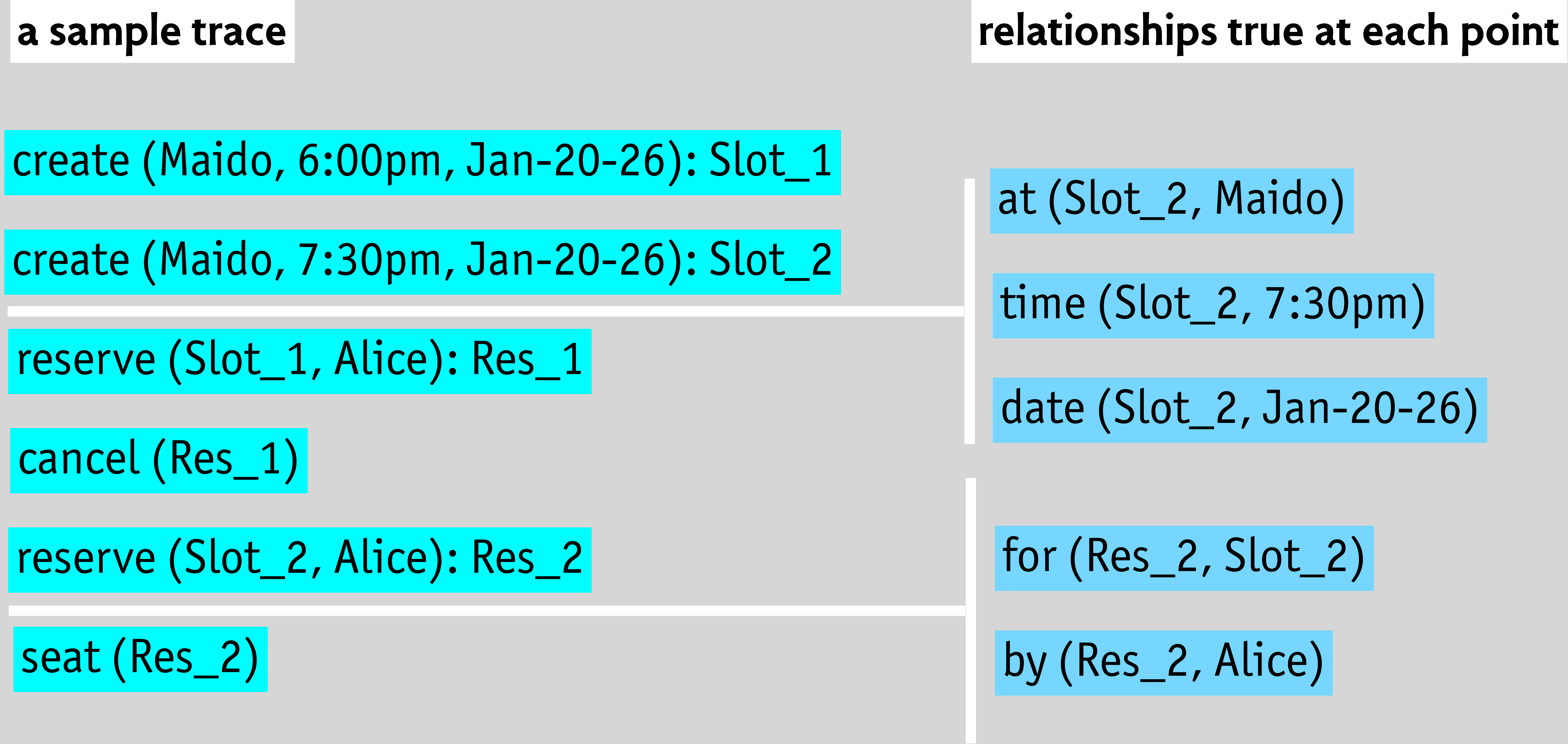
**actions**

reserve (Slot_2, Alice): Res_1

cancel (Res_1)

create (Maido, 7:30pm, Jan-20-26): Slot_2

# traces: histories of actions



**a sample trace**

- create (Maido, 6:00pm, Jan-20-26): Slot_1
- create (Maido, 7:30pm, Jan-20-26): Slot_2
- reserve (Slot_1, Alice): Res_1
- cancel (Res_1)
- reserve (Slot_2, Alice): Res_2
- seat (Res_2)

**relationships true at each point**

- at (Slot_2, Maido)
- time (Slot_2, 7:30pm)
- date (Slot_2, Jan-20-26)

- for (Res_2, Slot_2)
- by (Res_2, Alice)

# finding structure in behavior

**types of individuals**

**User**

**Restaurant**

**Slot**

**Reservation**

**types of relationships (aka relations)**

for (Reservation, Slot)

by (Reservation, User)

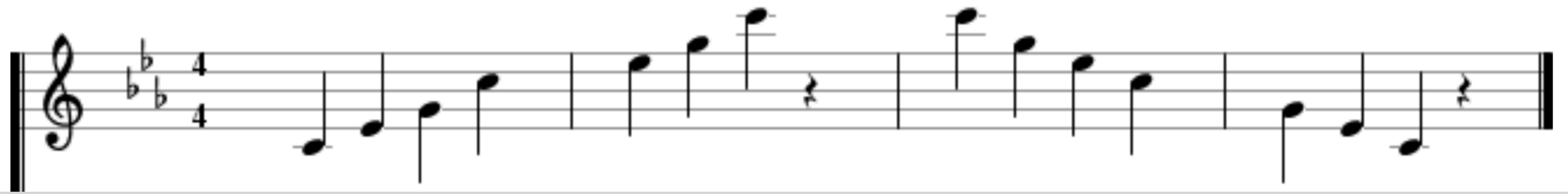
at (Slot, Restaurant)

**types of actions**

reserve (Slot, User): Reservation

cancel (Reservation)

but we need *larger* scale structure too for modularity why object orientation leads to illegible software



16 notes/rests

but easy to recall

just a Cm arpeggio

so what are the chunks of software behavior?

# object-orientation: individual types = chunks/modules

| **individual types**<br>become classes | **relations**<br>become fields | **actions**<br>become methods |
|---|---|---|
| **Reservation** | for (Reservation, Slot) | seat (Reservation) |
| | by (Reservation, User) | |
| | seated (Reservation) | |

```
class Reservation {
  Slot for;
  User by;
  bool seated;
  void seat ()
}
```

# we’ve forgotten how complicated this is!

**which class to assign an action to?**
action may have >1 individual
not clear even when exactly one!

reserve (Slot, User): Reservation

could belong to Slot, User or Reservation

cancel (Reservation)

can’t belong to Reservation if deletes it

**which class to assign a relation to?**
relation may involve >1 individual
may need to access both ways

by (Reservation, User)

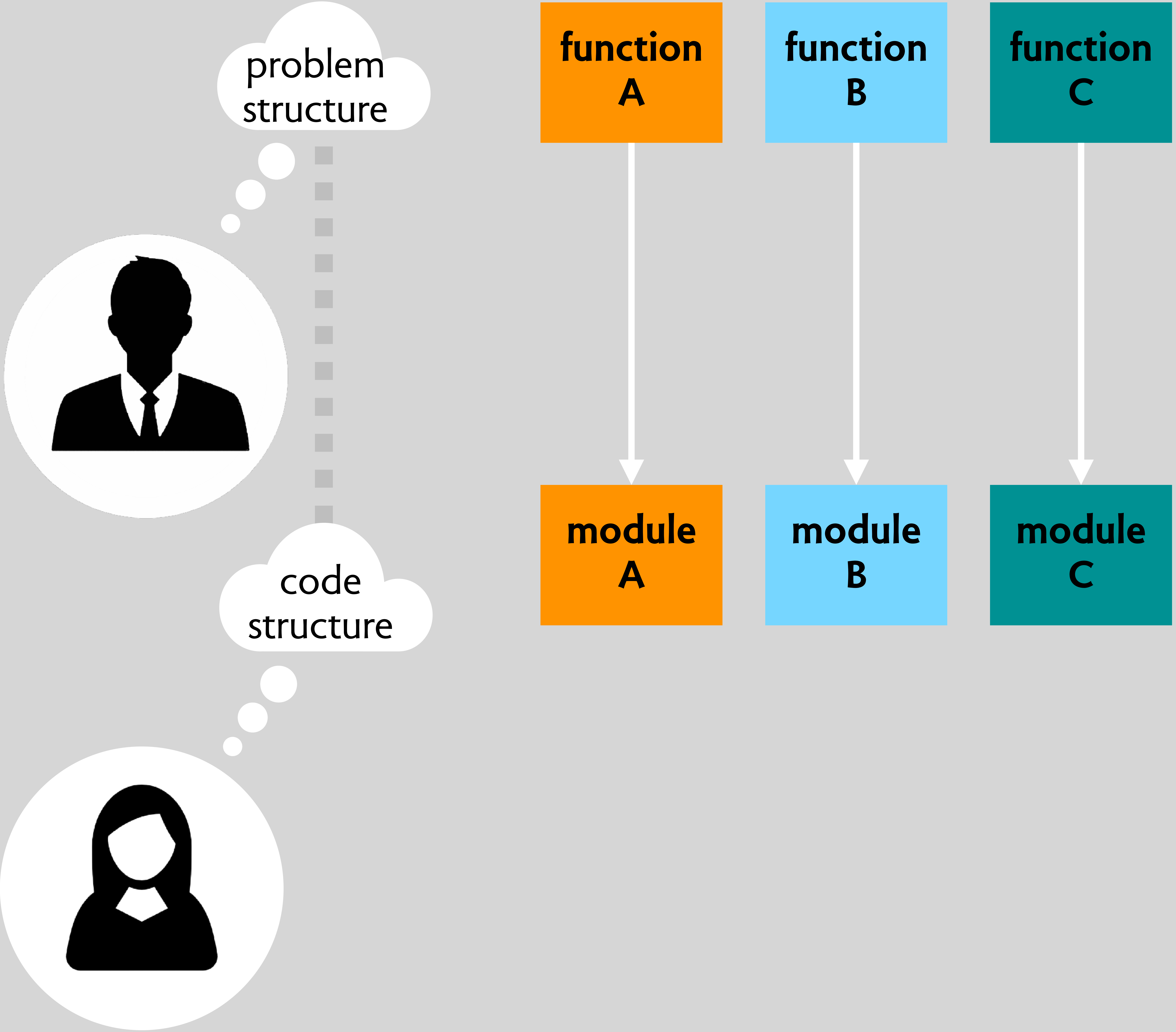
in Reservation so can see who?
or in User so can see which reservations they have?

**may have to invent a class**
not corresponding to individuals

authenticate (username, password): user

can’t belong to User, so create a class Users that holds all users

# what is modularity?



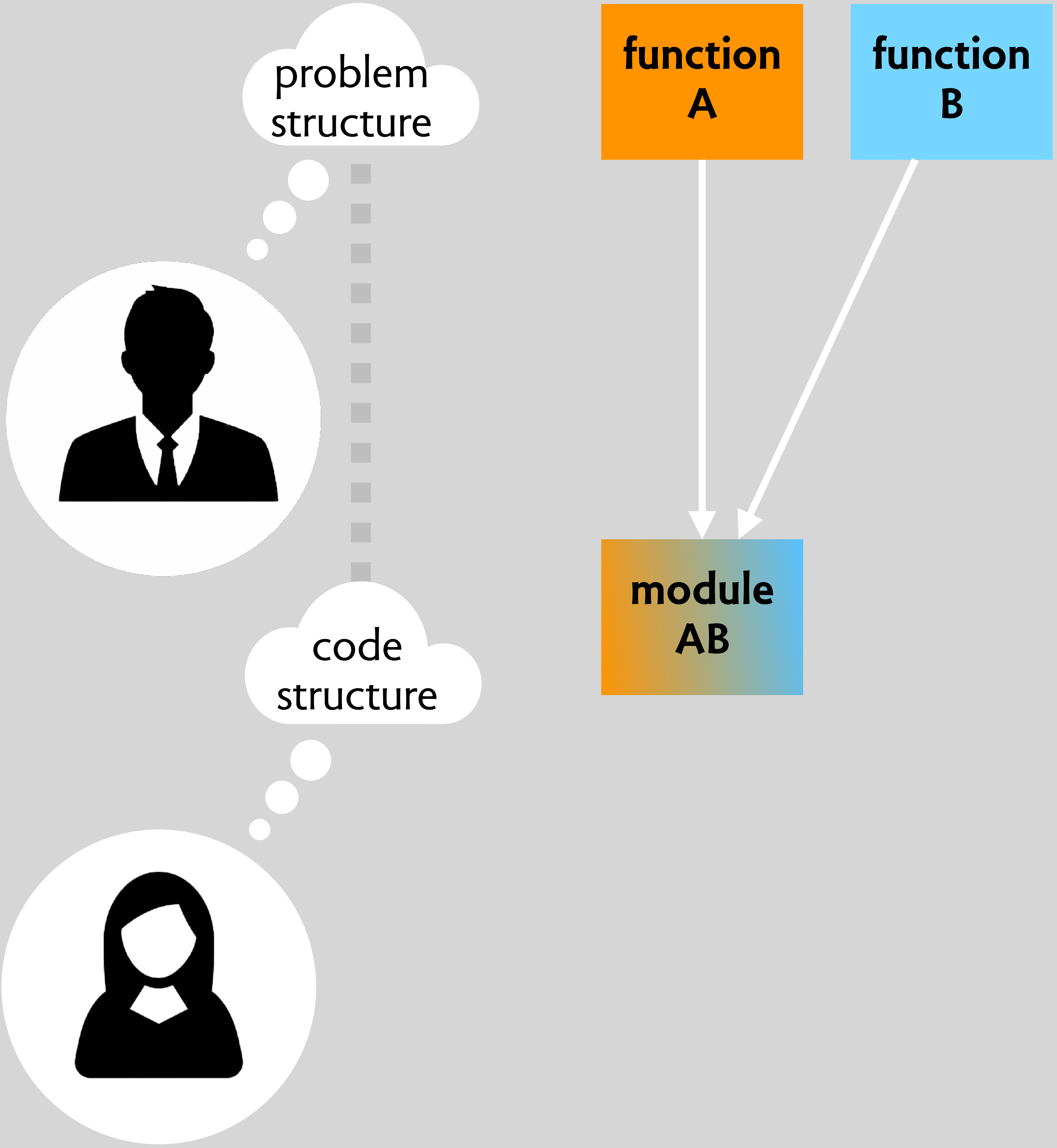
**if we can do this**

modules can be coded independently

familiar functions -> reusable modules

**and modifications easy too**

change function A -> modify module A



**conflation**

>1 function for a module

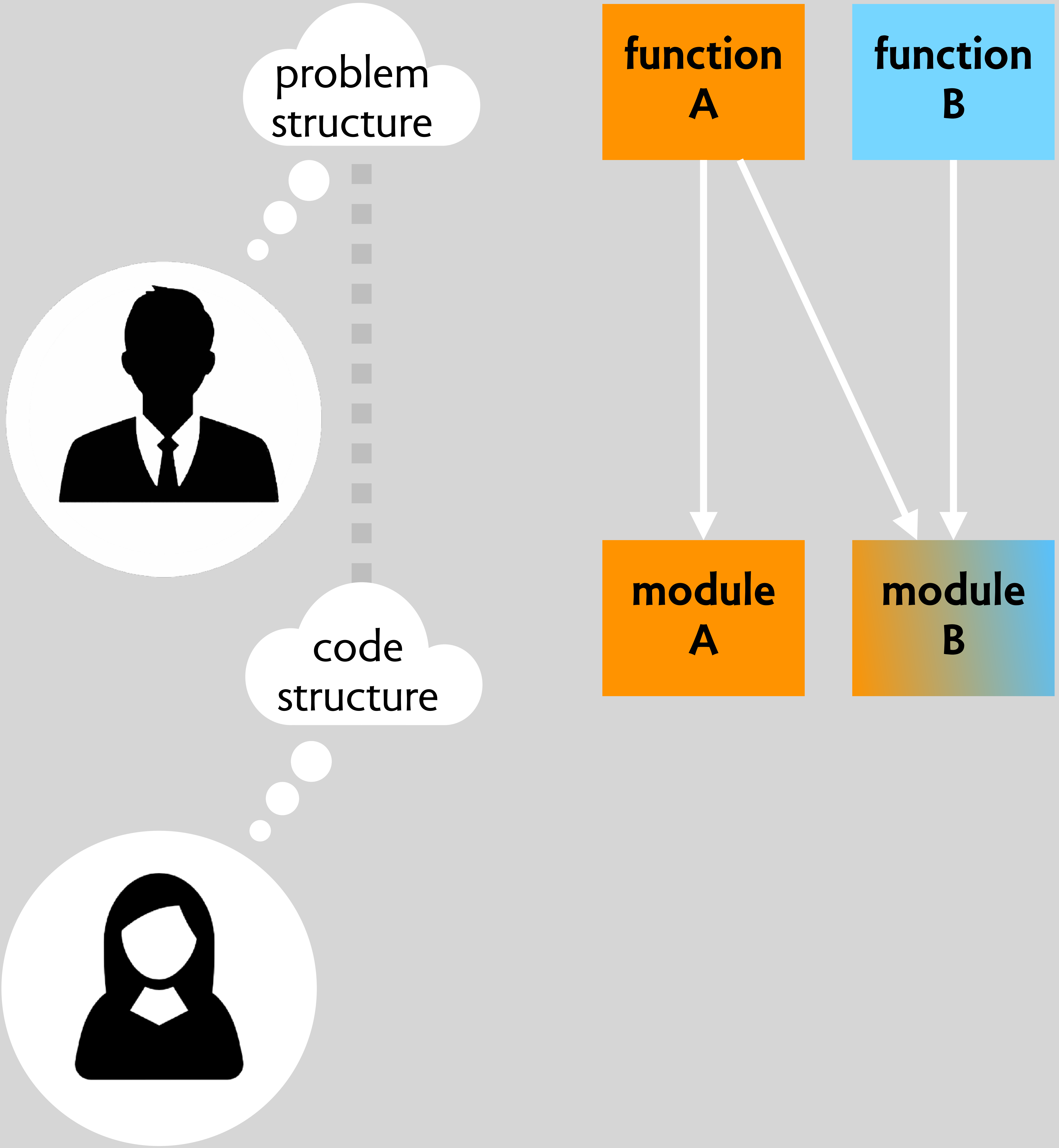
failure to "separate concerns"

**why this is bad**

module is no longer familiar

has become app-specific

harder to code reliably



problem structure

code structure

function A

function B

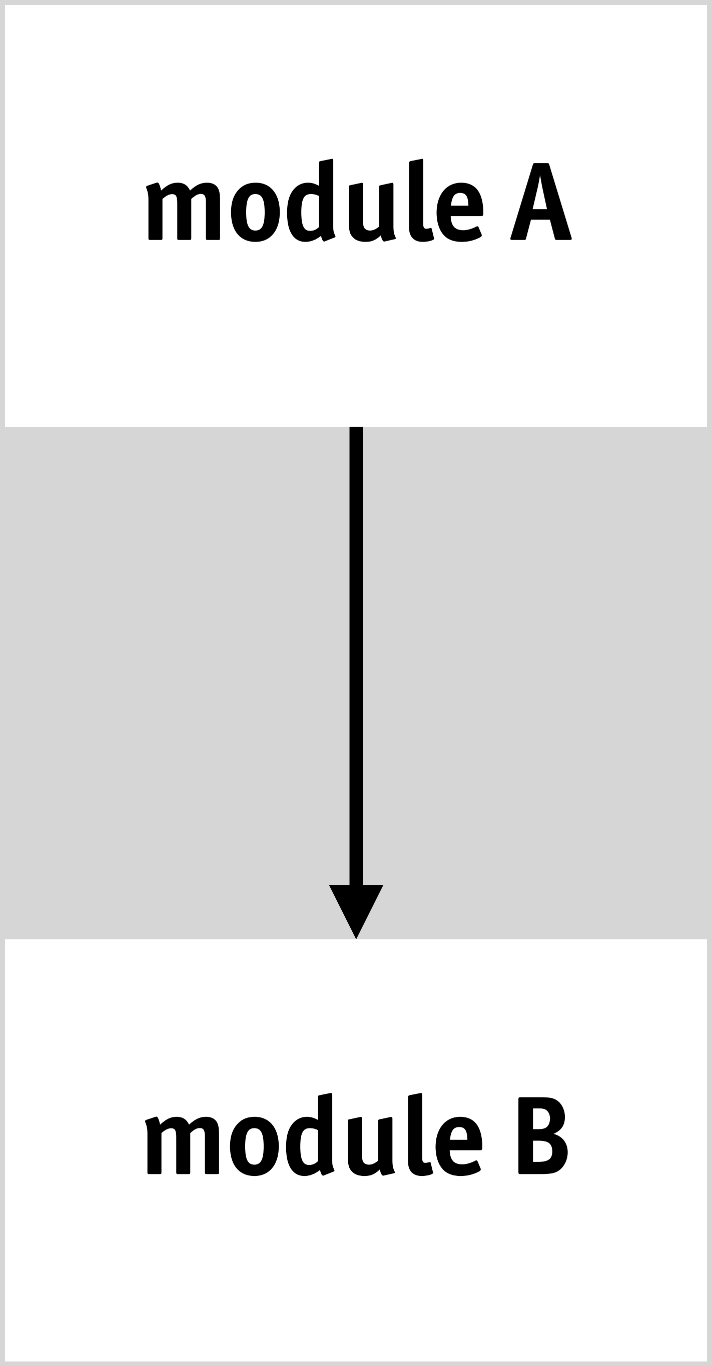
module A

module B

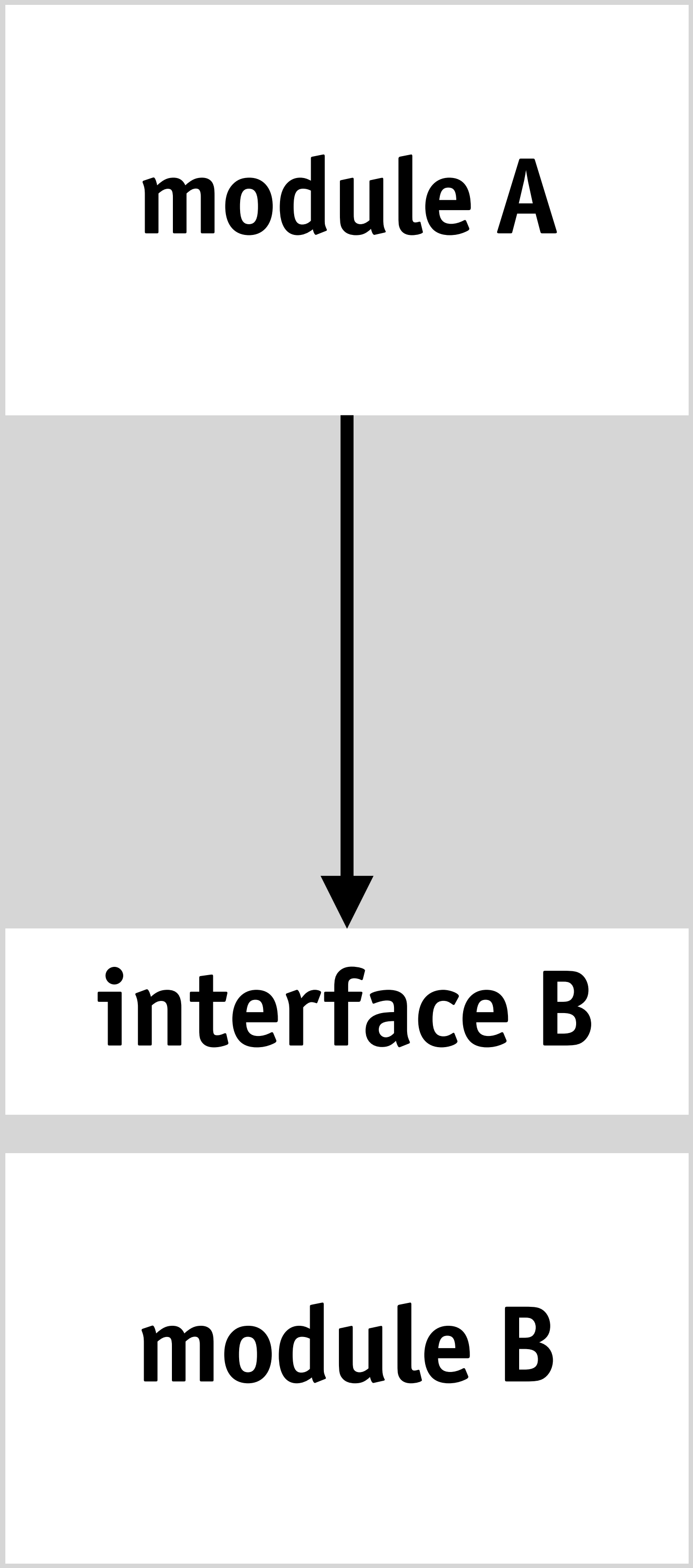
**fragmentation**
>1 module for a function
failure to “encapsulate”

**why this is bad**
modules need coordination
leads to dependencies
changes no longer localized

# modularity in current practice



**undesirable coupling**
B exposes implementation



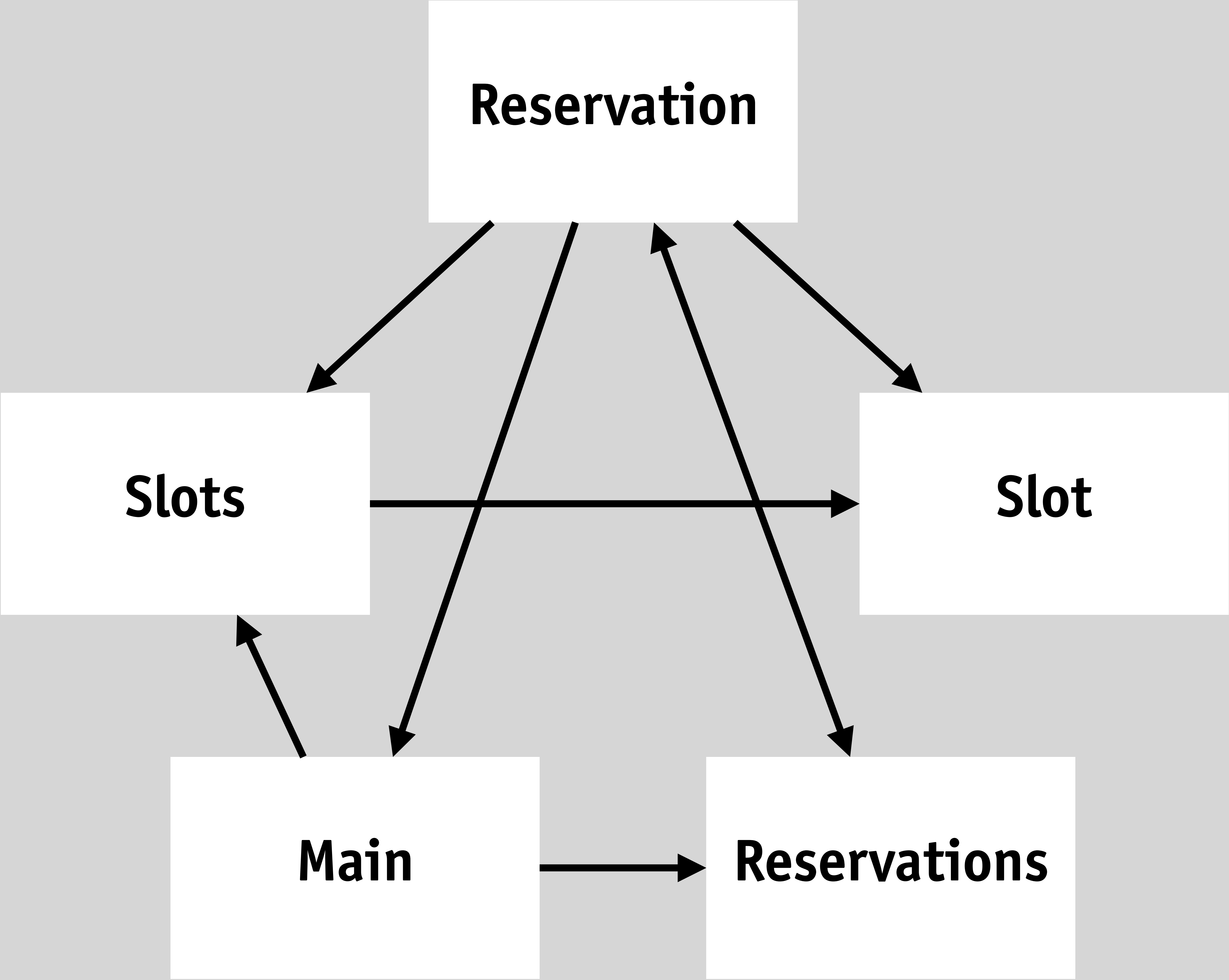
**modular version**
A only knows interface of B

why object orientation leads to **coupled software**

```
class Reservation {
 Slot for;
 User by;
 void reserve (Table table, Time time, User user) {
  if (!Main.reservations.hasReservation (time, user))
    slot = Main.slots.findSlot (table, time);
    slot.setUnavailable ();
  }
 }
```

```
class Slots {
 Set [Slot] slots;
 findSlot  Slot (Table table, Time time)
 }
```

```
class Slot {
  Table table;
  Time time;
  bool available;
  void setUnavailable ()
  }
```

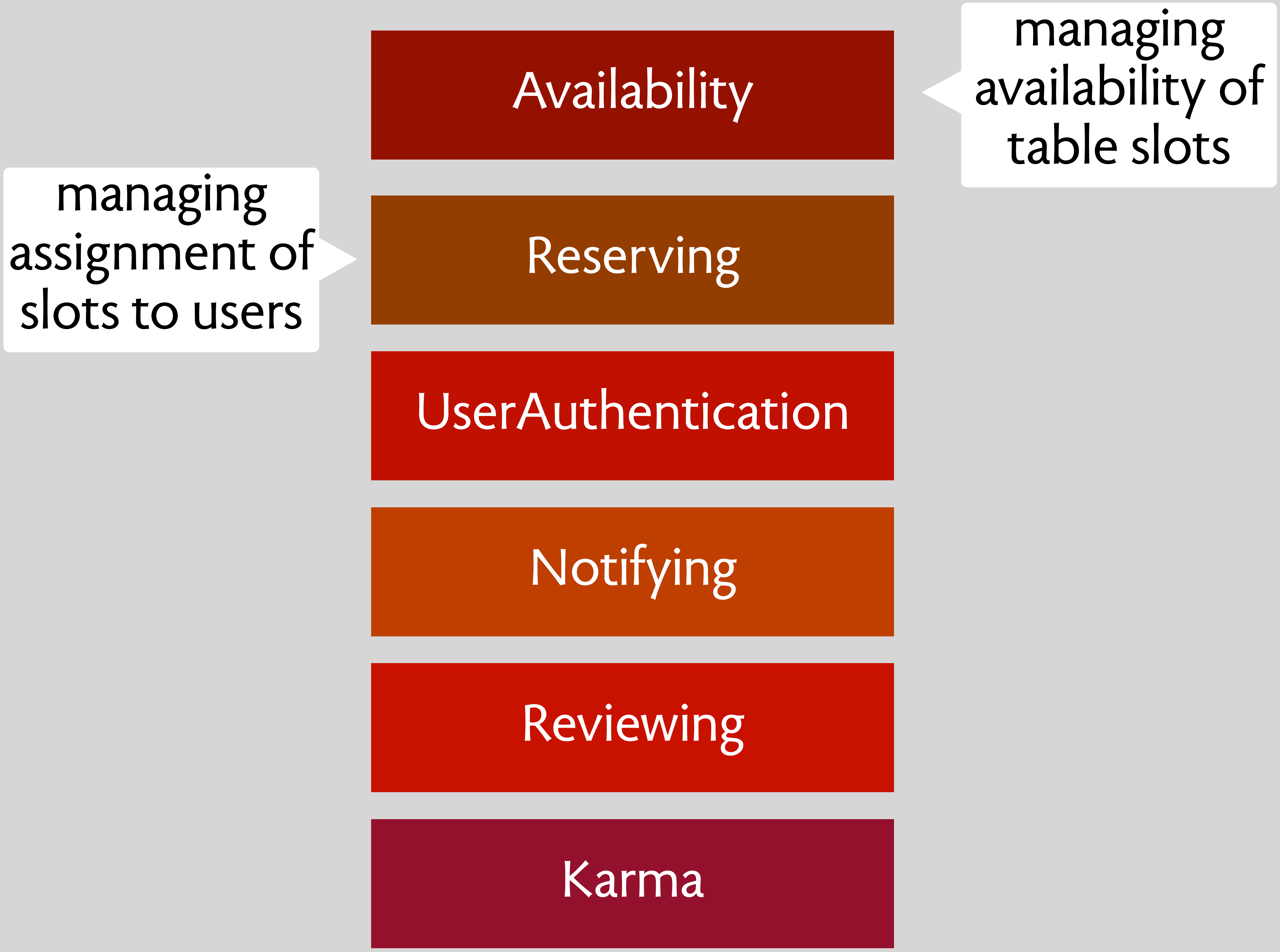


# concept design

## a simpler approach

**concept: an aspect of functionality**
complete: makes sense alone
coherent: doesn’t mingle distinct aspects
purposive: serves a defined purpose
reusable: often application-independent

**a concept is a service**
usually an ongoing activity
maintains persistent state
initiates and responds to actions



Availability

managing availability of table slots

Reserving

managing assignment of slots to users

UserAuthentication

Notifying

Reviewing

Karma

**map relations and actions to concepts**
not by the type of the individuals
but by which aspect of functionality

for (Reservation, Slot)

by (Reservation, User)

at (Slot, Restaurant)

table (Slot, Table)

Availability

Reserving

UserAuthentication

Notifying

Upvoting

Karma

create (Restaurant, ...): Slot

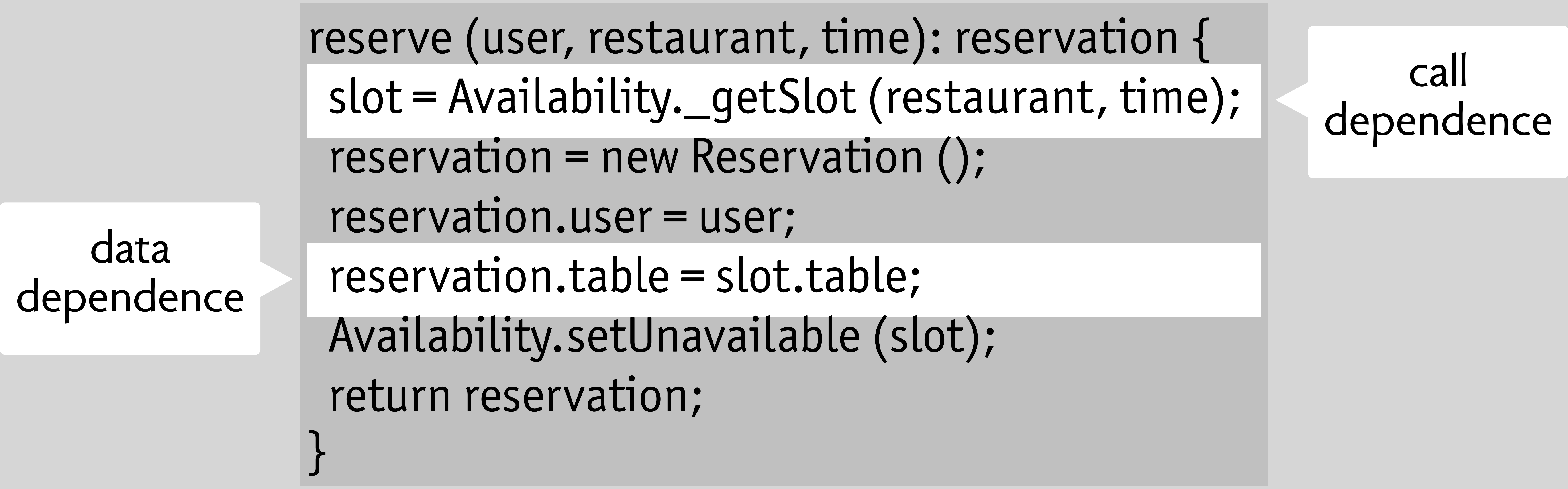
reserve (Slot, User): Reservation

cancel (Reservation)

upvote (User, Restaurant)

# how concepts avoid fragmentation

## Reserving



```
reserve (user, restaurant, time): reservation {
  slot = Availability._getSlot (restaurant, time);
  reservation = new Reservation ();
  reservation.user = user;
  reservation.table = slot.table;
  Availability.setUnavailable (slot);
  return reservation;
}
```

## Availability

```
setUnavailable (slot) {...}
```

```
_getSlot (restaurant, time): slot {...}
```

## Reserving

```
action reserve (user, slot): reservation {
  reservation = new Reservation ();
  reservation.slot = slot;
  reservation.user = user;
  return reservation;
}
```

## Availability

```
action setUnavailable (slot) {...}
```

```
query _getSlot (restaurant, time): slot {...}
```

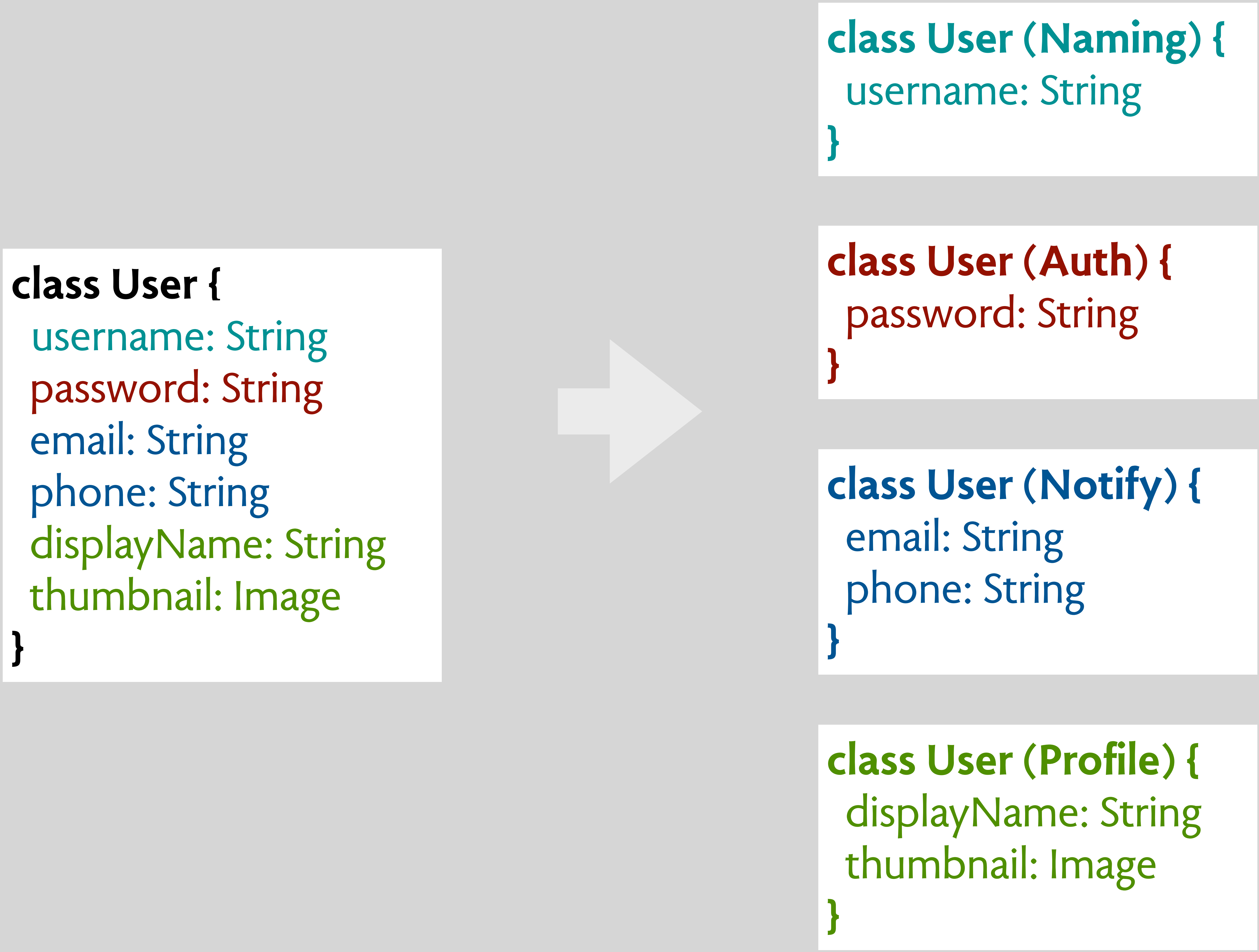
## Synchronization

```
when Request.reserve (user, restaurant, time)
where
  Availability._getSlot (restaurant, time): slot
then
  Reserving.reserve (user, slot)
  Availability.setUnavailable (slot)
```

coordination
externalized

# how concepts
# avoid conflation

# OOP conflates roles



```
class User {
  username: String
  password: String
  email: String
  phone: String
  displayName: String
  thumbnail: Image
}
```

```
class User (Naming) {
  username: String
}
```

```
class User (Auth) {
  password: String
}
```

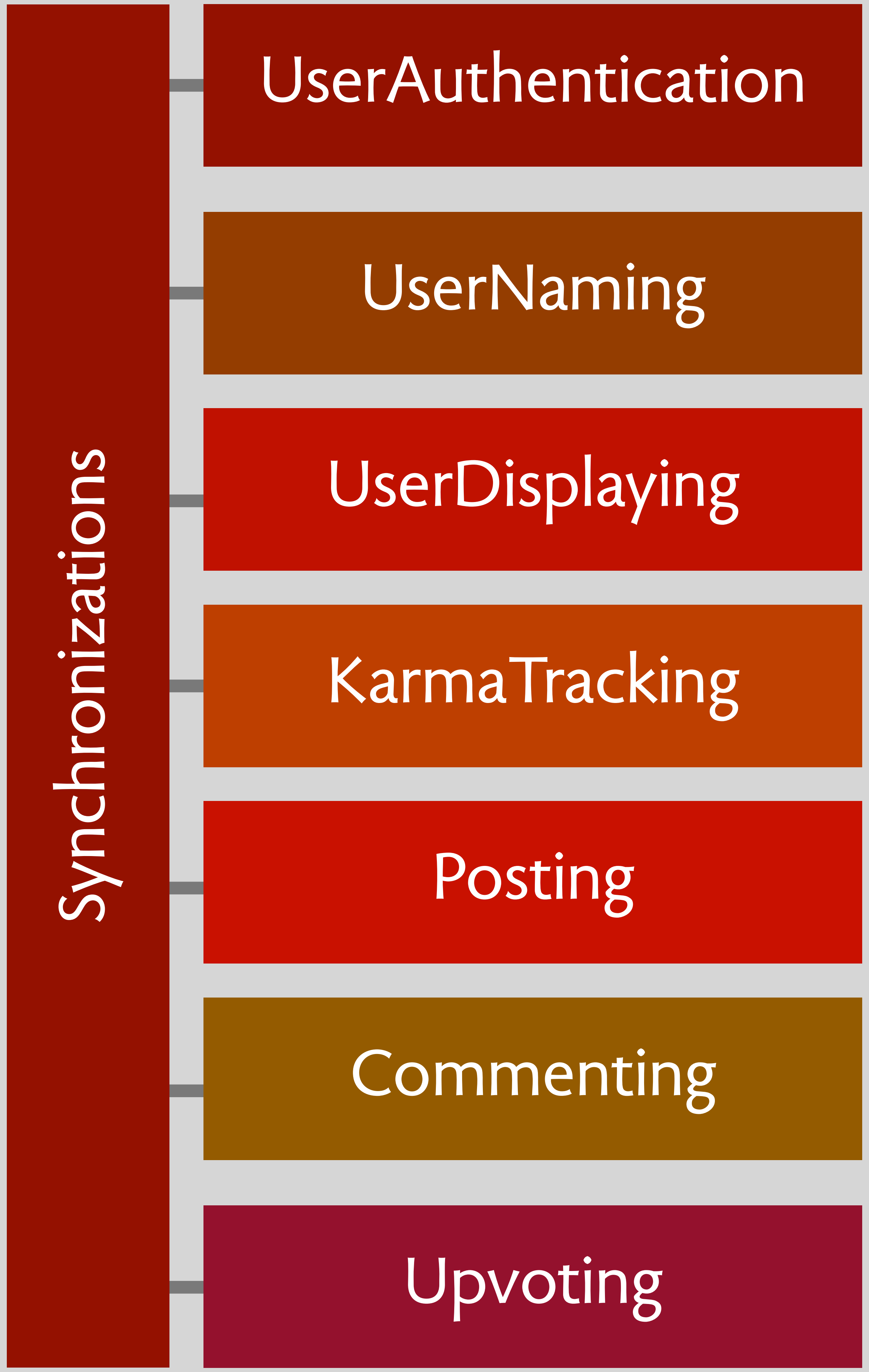
```
class User (Notify) {
  email: String
  phone: String
}
```

```
class User (Profile) {
  displayName: String
  thumbnail: Image
}
```

```
concept UserAuthentication
state
 a set of User with
   a password String
```

```
concept UserNaming
state
 a set of User with
   a unique username String
```

```
concept UserDisplaying
state
 a set of User with
   a displayname String
```

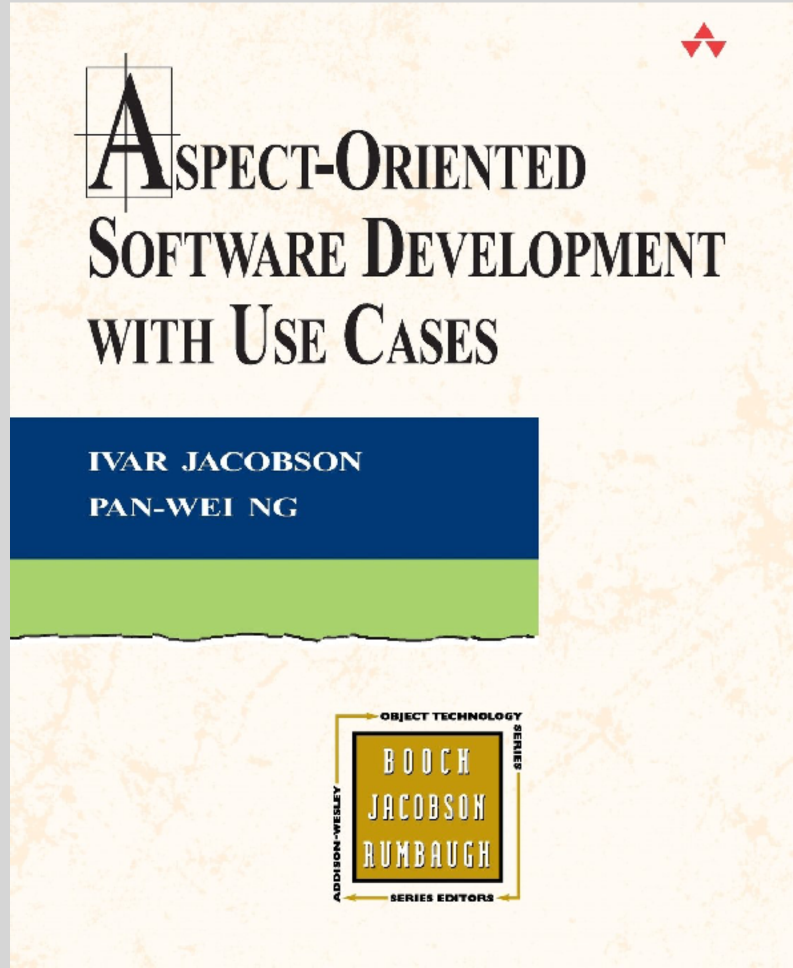


concepts are independent services with state & actions

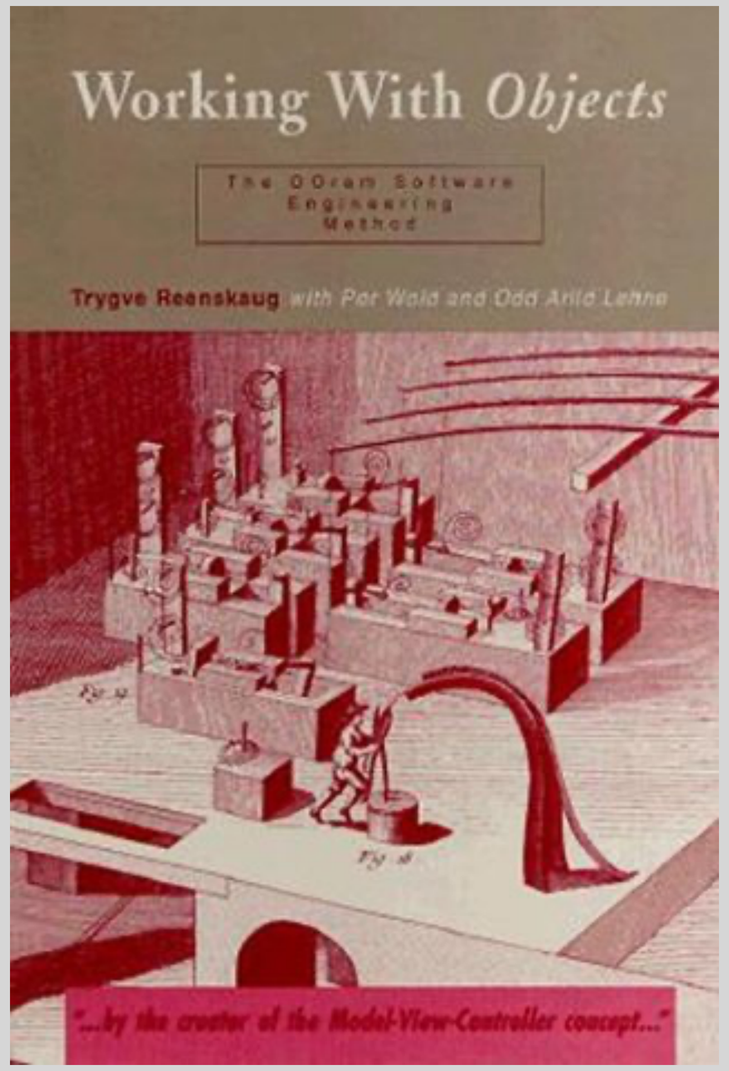
concepts encapsulate coherent aspects of functionality

each concept associates properties with individuals

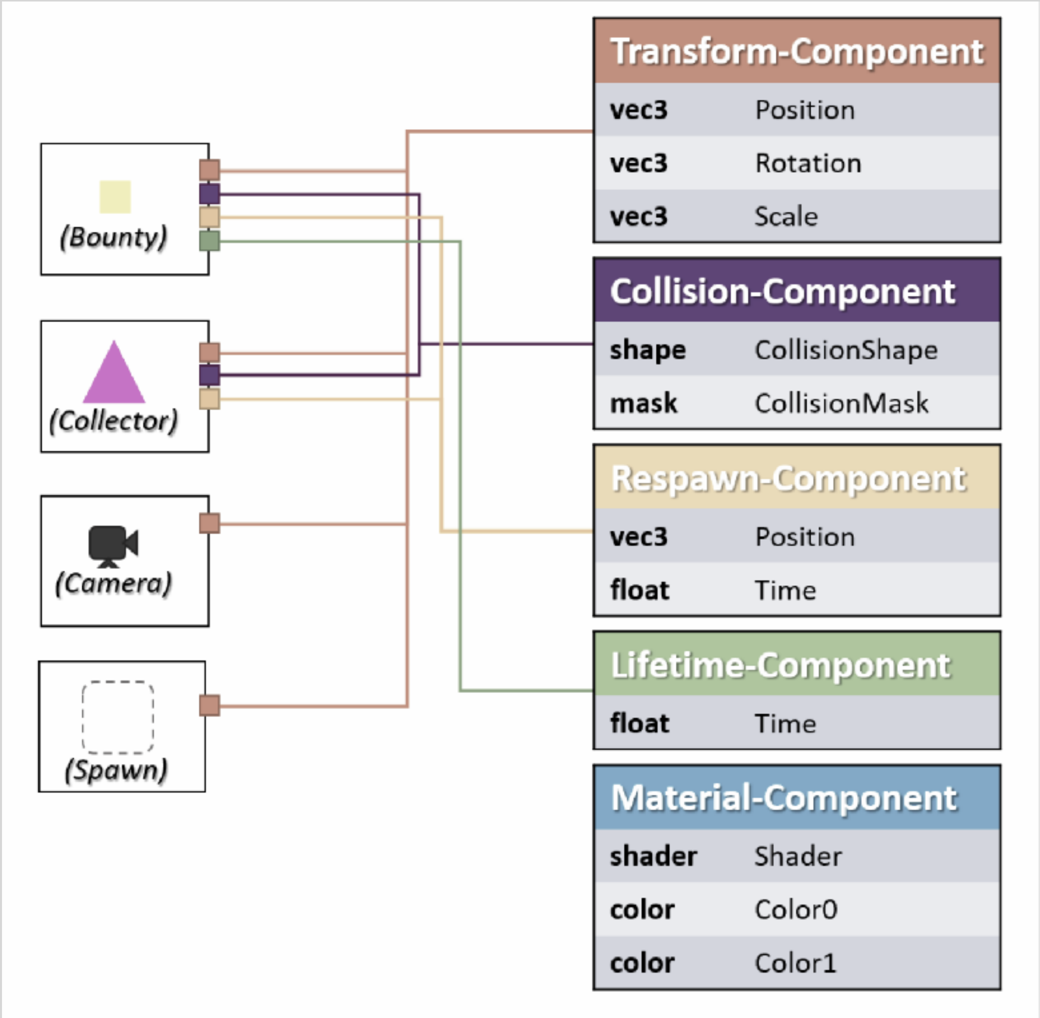
# a long history of fixes for OOP's conflation



Aspect-oriented programming
Kiczales et al (1997)


Working With Objects
The OOram Software Engineering Method
Trygve Reenskaug with Per Wold and Odd Arild Lehne
"...by the creator of the Model-View-Controller concept..."

Role-oriented programming
Reenskaug et al (1983)



| Transform-Component | |
|---|---|
| vec3 | Position |
| vec3 | Rotation |
| vec3 | Scale |

| Collision-Component | |
|---|---|
| shape | CollisionShape |
| mask | CollisionMask |

| Respawn-Component | |
|---|---|
| vec3 | Position |
| float | Time |

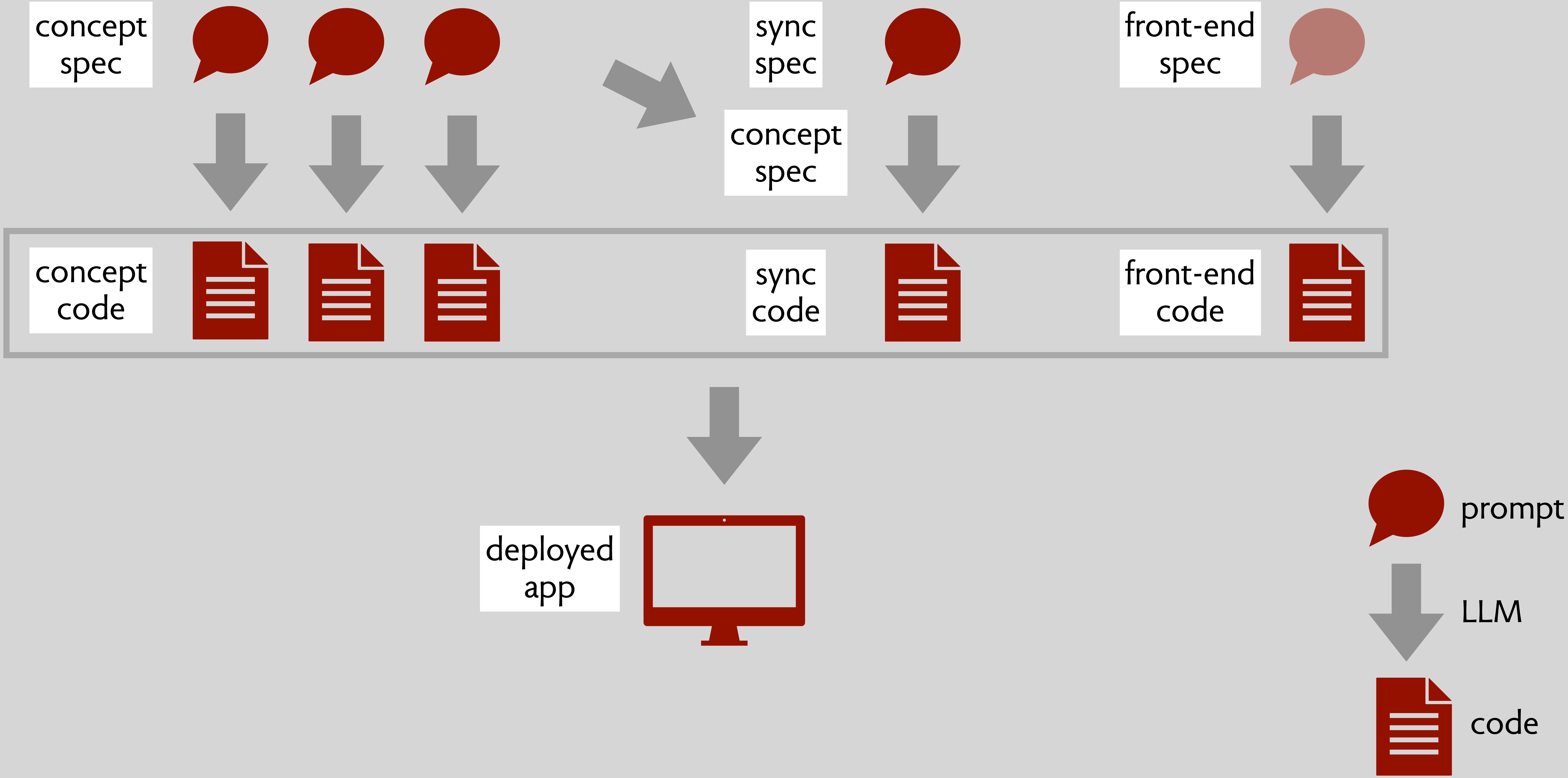
| Lifetime-Component | |
|---|---|
| float | Time |

| Material-Component | |
|---|---|
| shader | Shader |
| color | Color0 |
| color | Color1 |

Entity-component system
Scott Bilas et al (2002)

# generating code
# from concepts

# exploiting concept modularity to generate code



RealWorld example apps
Create
Submit
Sign in
conduit
A place to share your knowledge.
Frontend
Backend
Fullstack

# The mother of all demo apps

See how the exact same application is built using different libraries and frameworks.

**LANGUAGES**

- All
- TypeScript
- JavaScript
- Kotlin
- ClojureScript
- Elm
- PureScript
- Rust
- C#
- Dart
- Mint
- Swift
- ReScript

| | |
|---|---|
| **Android Native** MOBILE<br>coding-blocks-archives/Conduit_Android_Kotlin | Kotlin |
| **Android Native + Retrofit + Jetpack** MOBILE<br>Marvel999/Conduit-Android-kotlin | Kotlin |
| **Angular**<br>khaledosman/angular-realworld-example-app | TypeScript |
| **Angular**<br>AndyT2503/angular-conduit-signals | TypeScript |
| **Angular**<br>iancharlesdouglas/ng-realworld-ssr | TypeScript |
| **Angular + NgRx + Nx** | TypeScript |

# RealWorld benchmark

**LLM generated in one-shot**

all concept specs (authentication, posting, favoriting, tagging)

implementations of concept specs

**LLM generated with iteration**

implementations of synchronizations

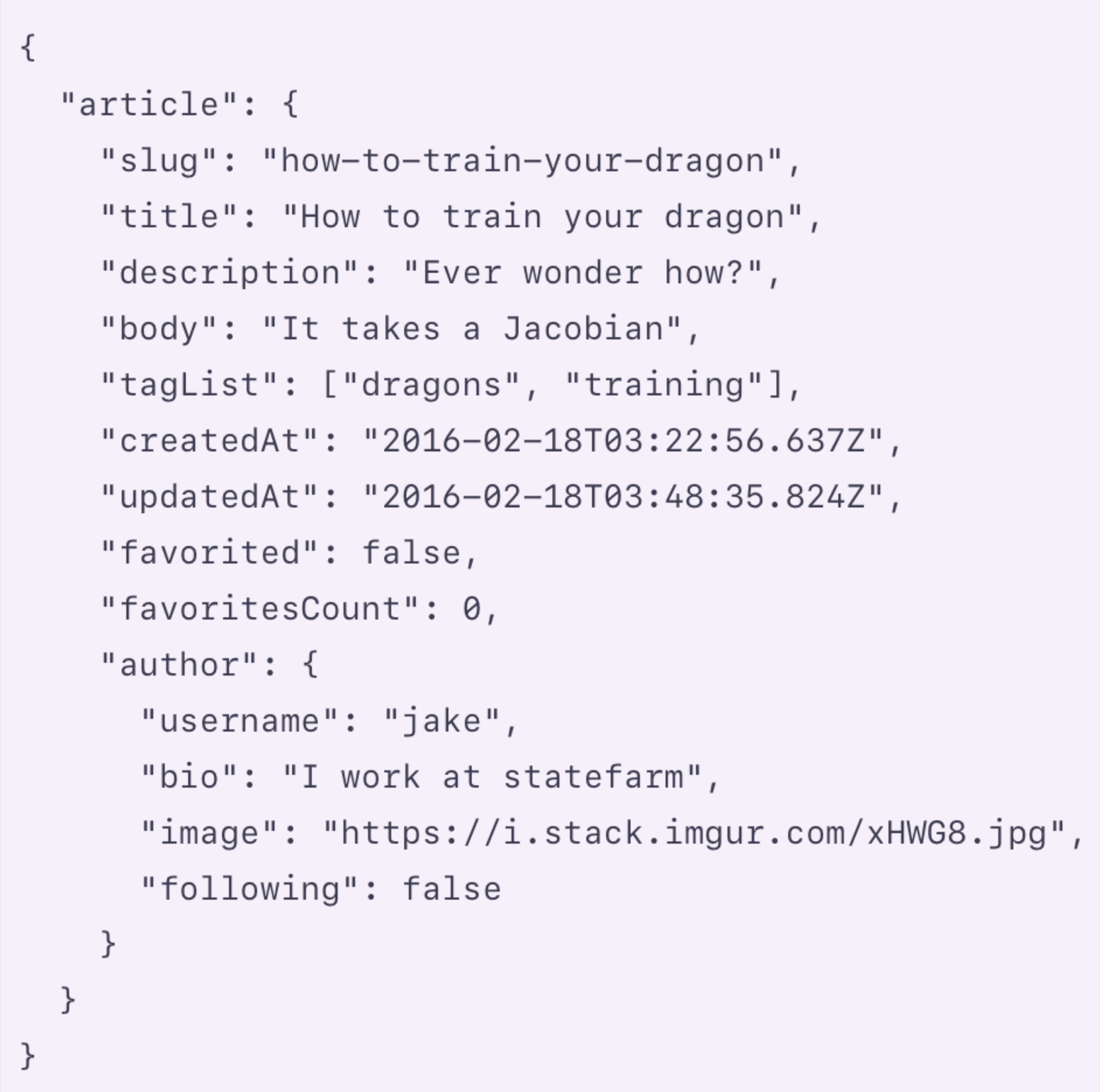
given RealWorld API specification

**why are syncs harder?**

concepts are completely familiar and generic

syncs are application-specific

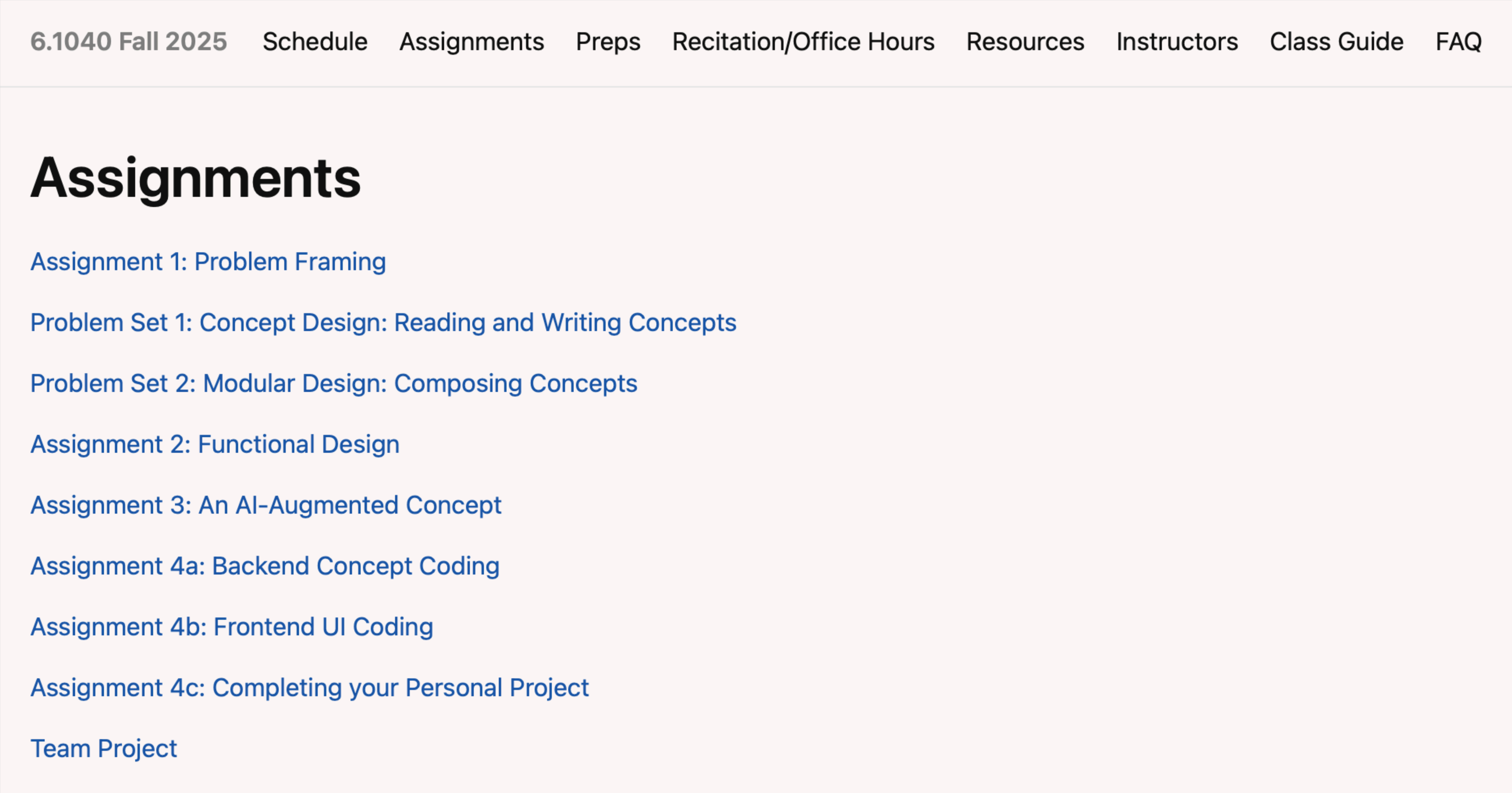
RealWorld API conflates concepts



```
{
  "article": {
    "slug": "how-to-train-your-dragon",
    "title": "How to train your dragon",
    "description": "Ever wonder how?",
    "body": "It takes a Jacobian",
    "tagList": ["dragons", "training"],
    "createdAt": "2016-02-18T03:22:56.637Z",
    "updatedAt": "2016-02-18T03:48:35.824Z",
    "favorited": false,
    "favoritesCount": 0,
    "author": {
      "username": "jake",
      "bio": "I work at statefarm",
      "image": "https://i.stack.imgur.com/xHWG8.jpg",
      "following": false
    }
  }
}
```

*response format for get article*

# 6.104 class



## Assignments

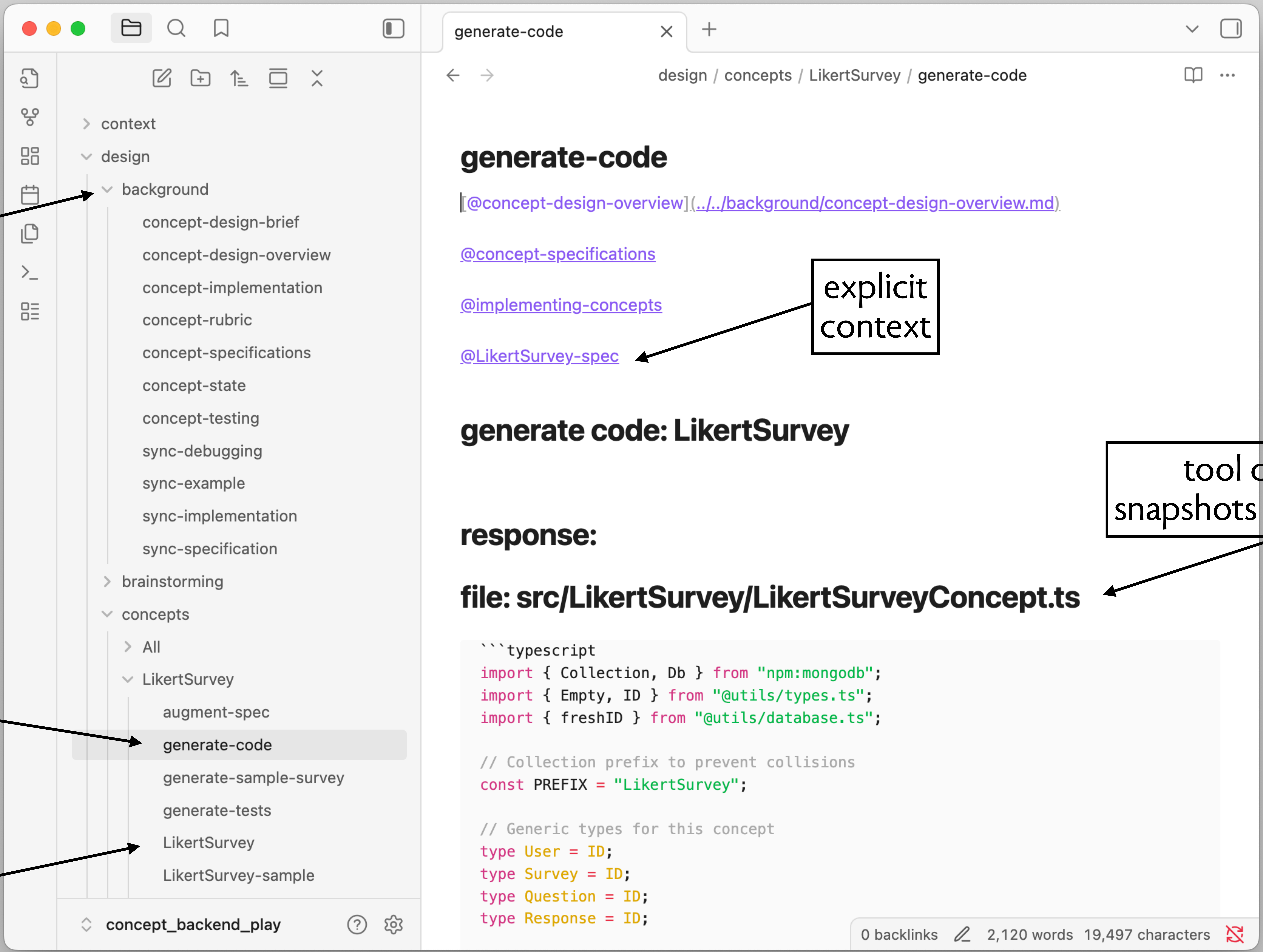
- Assignment 1: Problem Framing
- Problem Set 1: Concept Design: Reading and Writing Concepts
- Problem Set 2: Modular Design: Composing Concepts
- Assignment 2: Functional Design
- Assignment 3: An AI-Augmented Concept
- Assignment 4a: Backend Concept Coding
- Assignment 4b: Frontend UI Coding
- Assignment 4c: Completing your Personal Project
- Team Project

**75 students**
mostly 3rd & 4th year
TypeScript but no web experience

**learned concept design**
in two weeks, two homeworks

**LLM coded backends**
using our Context tool
use Google's Gemini API
custom background docs

**LLM coded frontends**
using agentic tools like Cursor



# generate-code

[@concept-design-overview](../../background/concept-design-overview.md)

@concept-specifications

@implementing-concepts

@LikertSurvey-spec

## generate code: LikertSurvey

## response:

## file: src/LikertSurvey/LikertSurveyConcept.ts

~~~
```typescript
import { Collection, Db } from "npm:mongodb";
import { Empty, ID } from "@utils/types.ts";
import { freshID } from "@utils/database.ts";

// Collection prefix to prevent collisions
const PREFIX = "LikertSurvey";

// Generic types for this concept
type User = ID;
type Survey = ID;
type Question = ID;
type Response = ID;
~~~

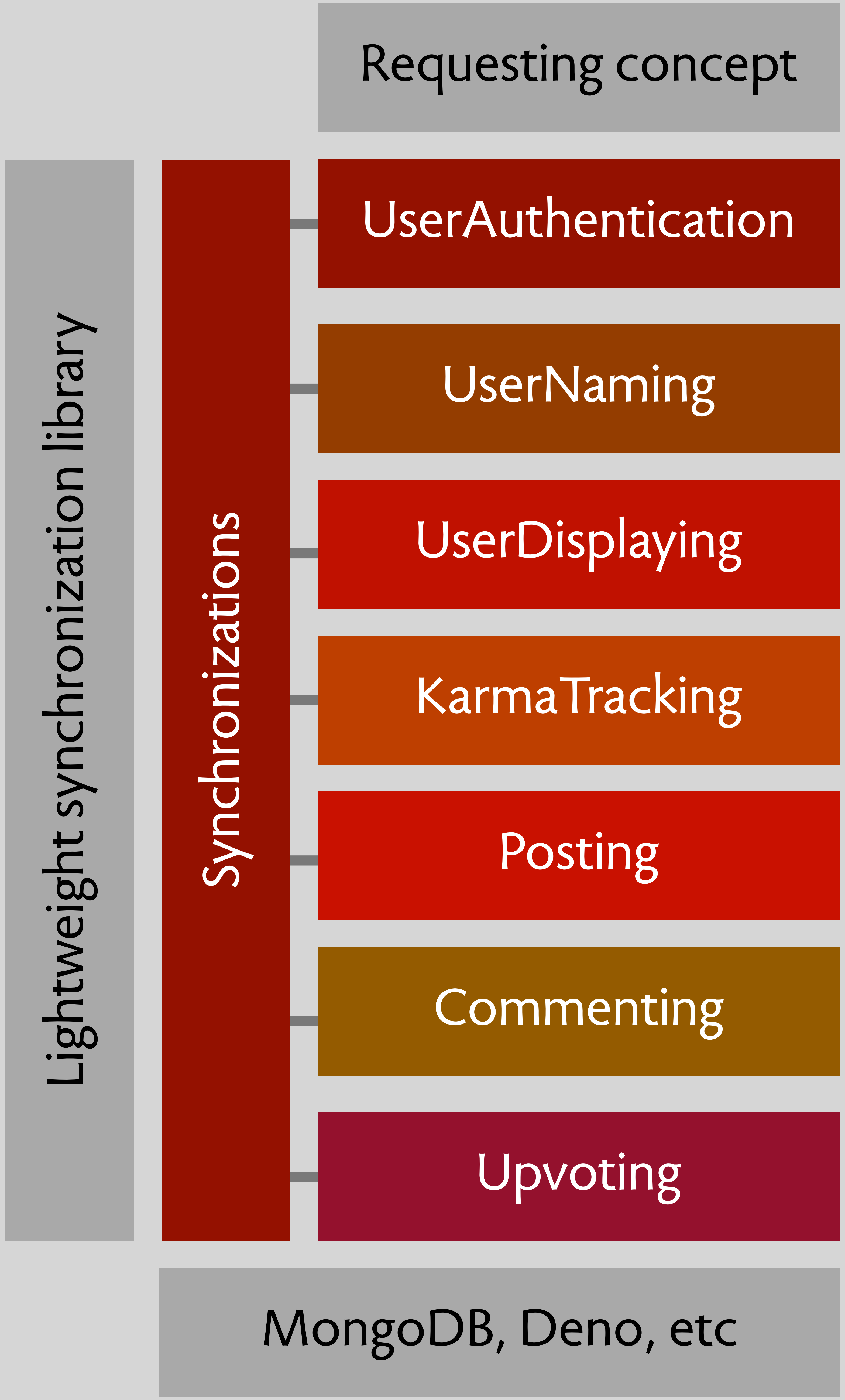
background documents

explicit context

tool calls Gemini snapshots & links prompts

LLM prompt

concept spec



Requesting concept

Lightweight synchronization library

Synchronizations

UserAuthentication

UserNaming

UserDisplaying

KarmaTracking

Posting

Commenting

Upvoting

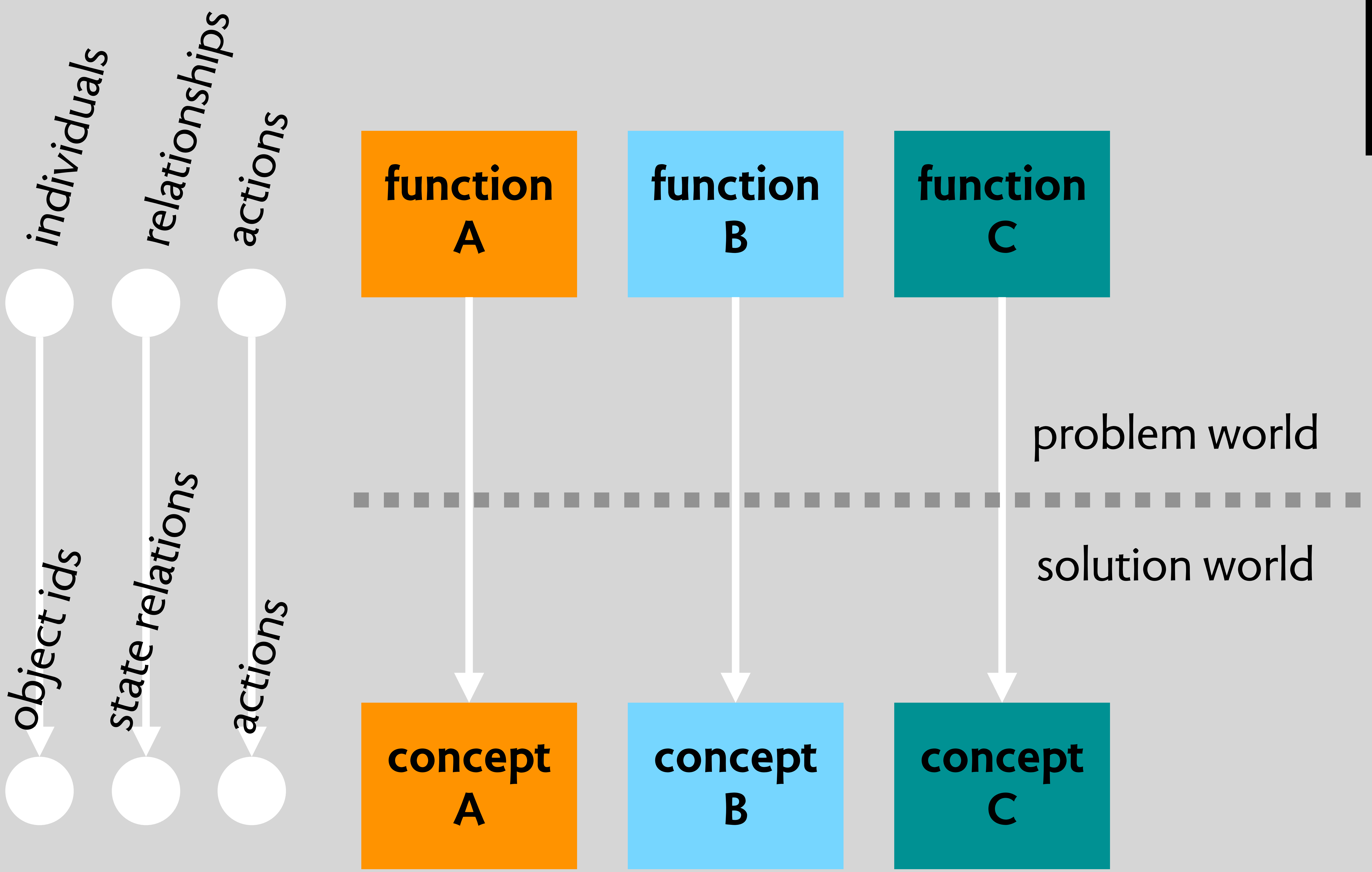
MongoDB, Deno, etc

encapsulates HTTP requests no routes needed!

also supports logging & provenance tracking

# summary

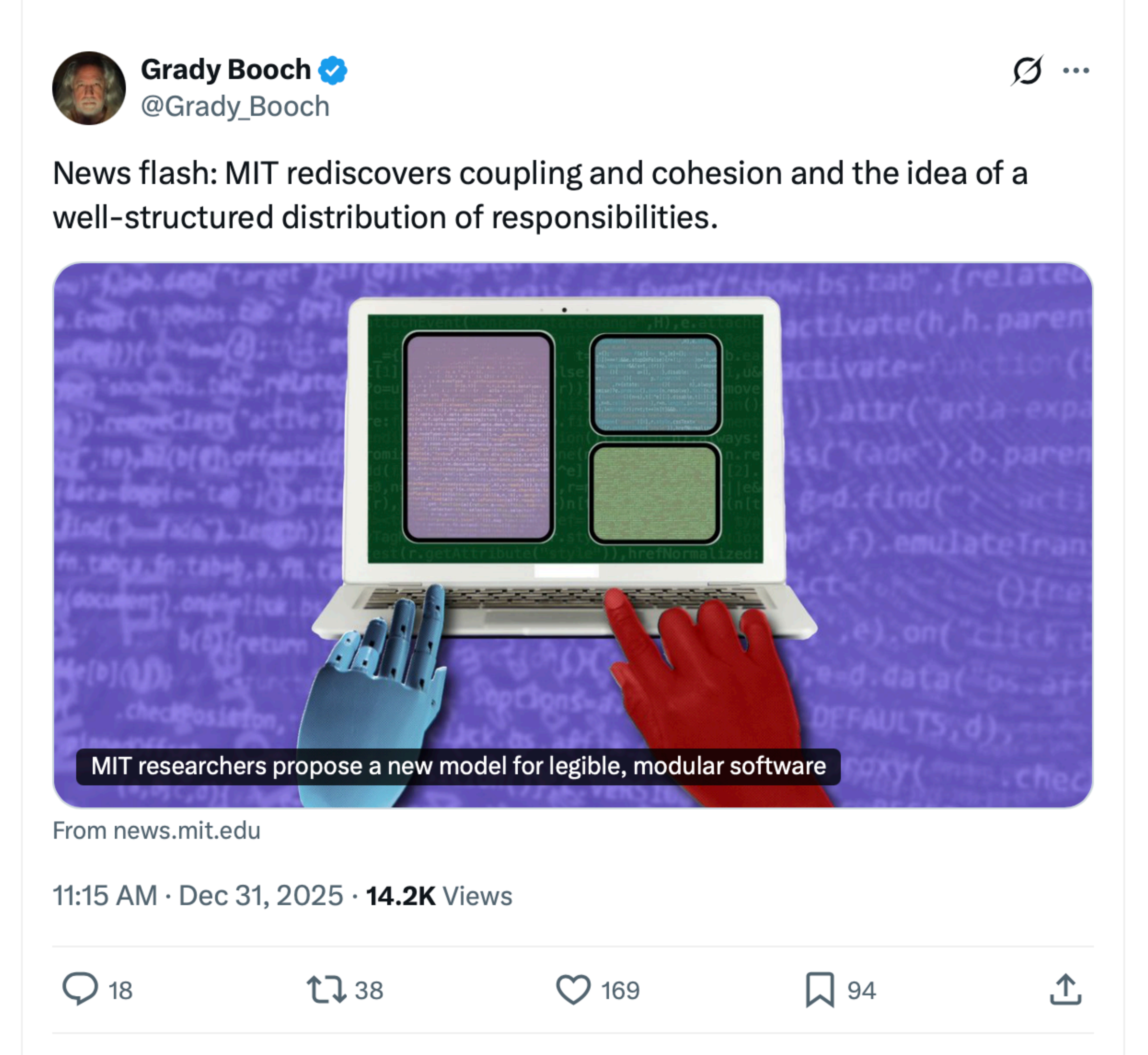
# legibility & modularity: preserving phenomena and structure



let's chat!
**dnj@mit.edu**

# backup
# slides

# coupling & cohesion



Grady Booch
@Grady_Booch

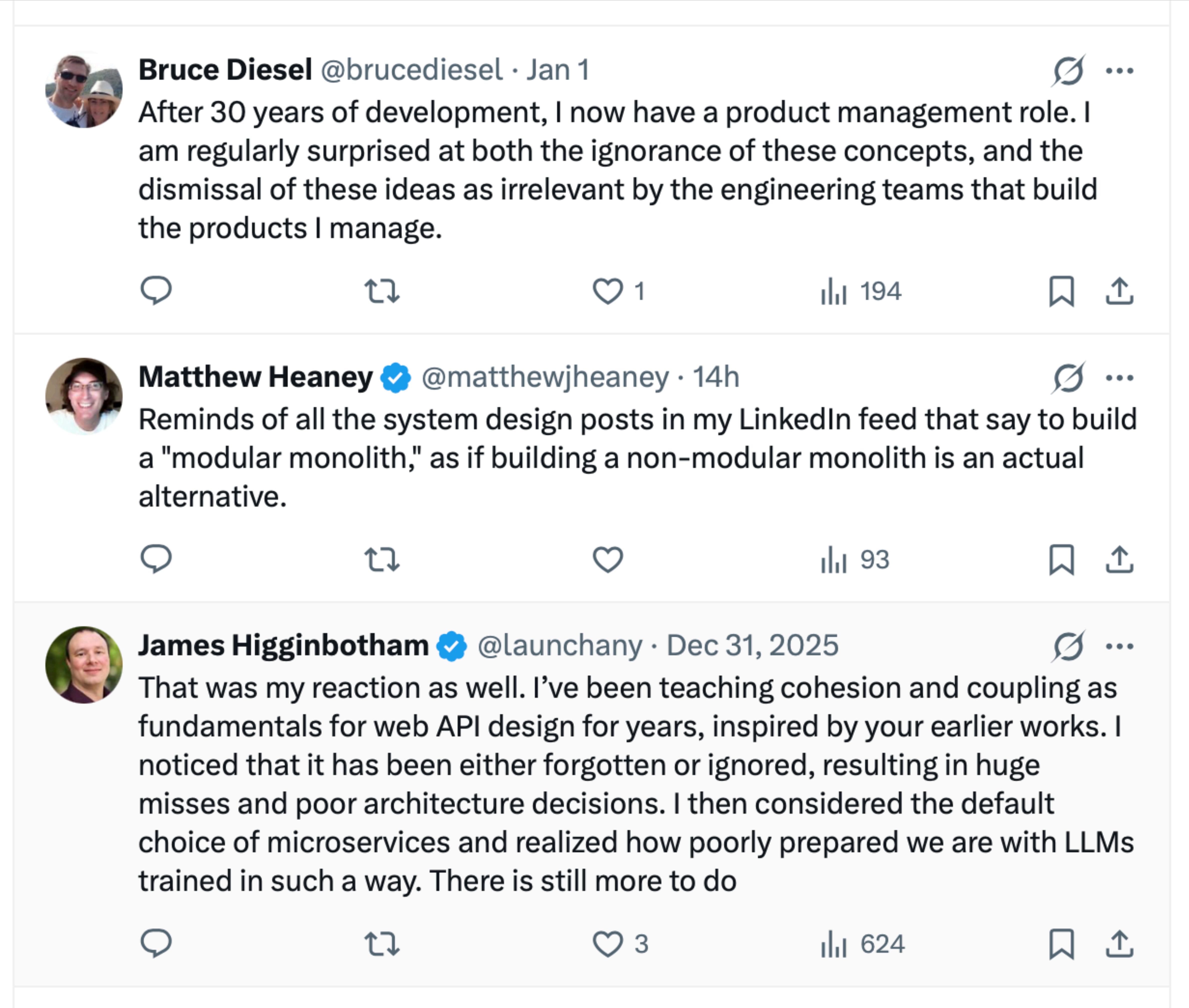
News flash: MIT rediscovers coupling and cohesion and the idea of a well-structured distribution of responsibilities.

MIT researchers propose a new model for legible, modular software

From news.mit.edu

11:15 AM · Dec 31, 2025 · **14.2K** Views

18 38 169 94



**Bruce Diesel** @brucediesel · Jan 1

After 30 years of development, I now have a product management role. I am regularly surprised at both the ignorance of these concepts, and the dismissal of these ideas as irrelevant by the engineering teams that build the products I manage.

♡ 1 · 194

**Matthew Heaney** @matthewjheaney · 14h

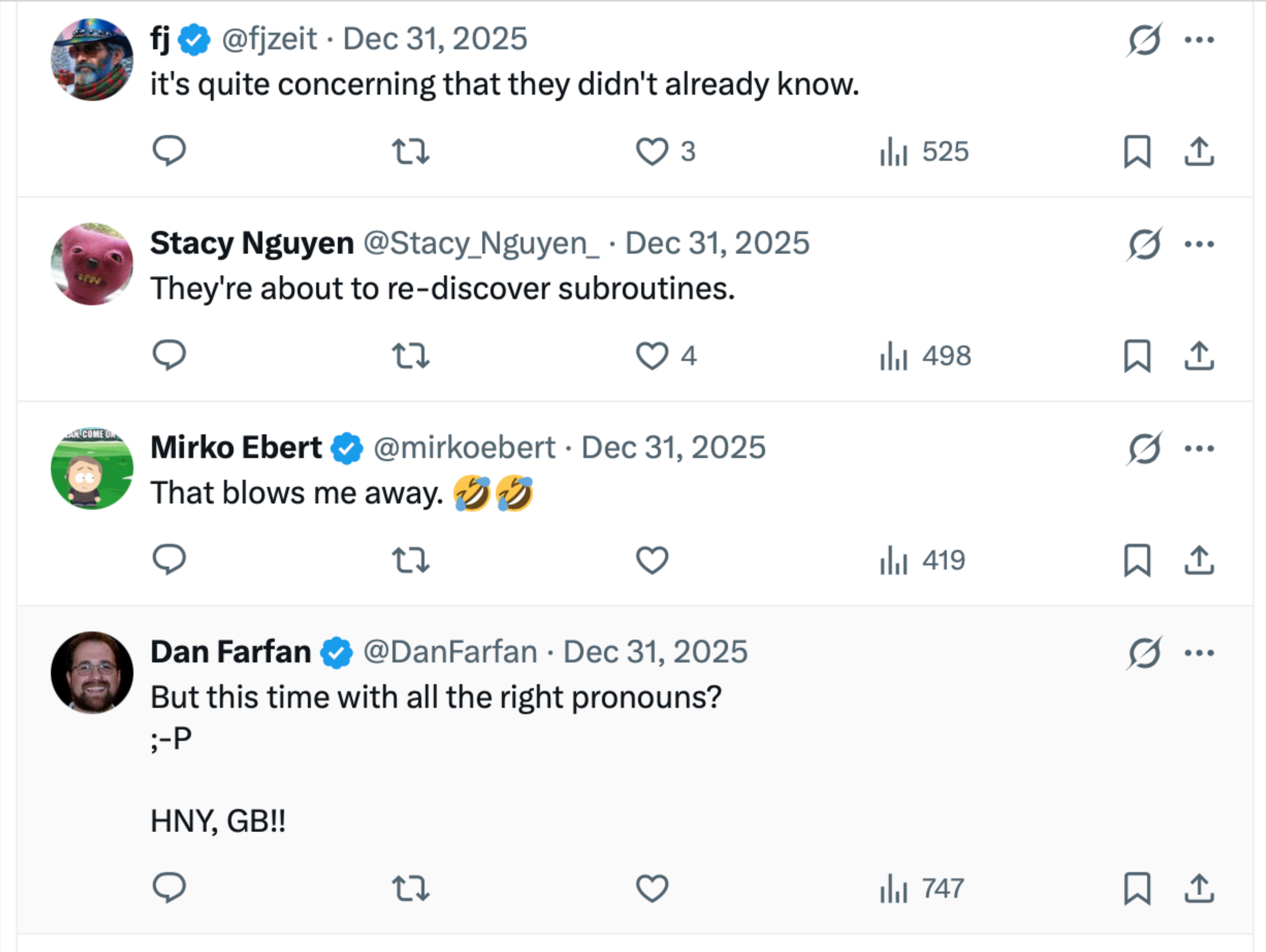
Reminds of all the system design posts in my LinkedIn feed that say to build a "modular monolith," as if building a non-modular monolith is an actual alternative.

93

**James Higginbotham** @launchany · Dec 31, 2025

That was my reaction as well. I've been teaching cohesion and coupling as fundamentals for web API design for years, inspired by your earlier works. I noticed that it has been either forgotten or ignored, resulting in huge misses and poor architecture decisions. I then considered the default choice of microservices and realized how poorly prepared we are with LLMs trained in such a way. There is still more to do

♡ 3 · 624



**fj** @fjzeit · Dec 31, 2025

it's quite concerning that they didn't already know.

♡ 3 · 525

**Stacy Nguyen** @Stacy_Nguyen_ · Dec 31, 2025

They're about to re-discover subroutines.

♡ 4 · 498

**Mirko Ebert** @mirkoebert · Dec 31, 2025

That blows me away. 🤣🤣

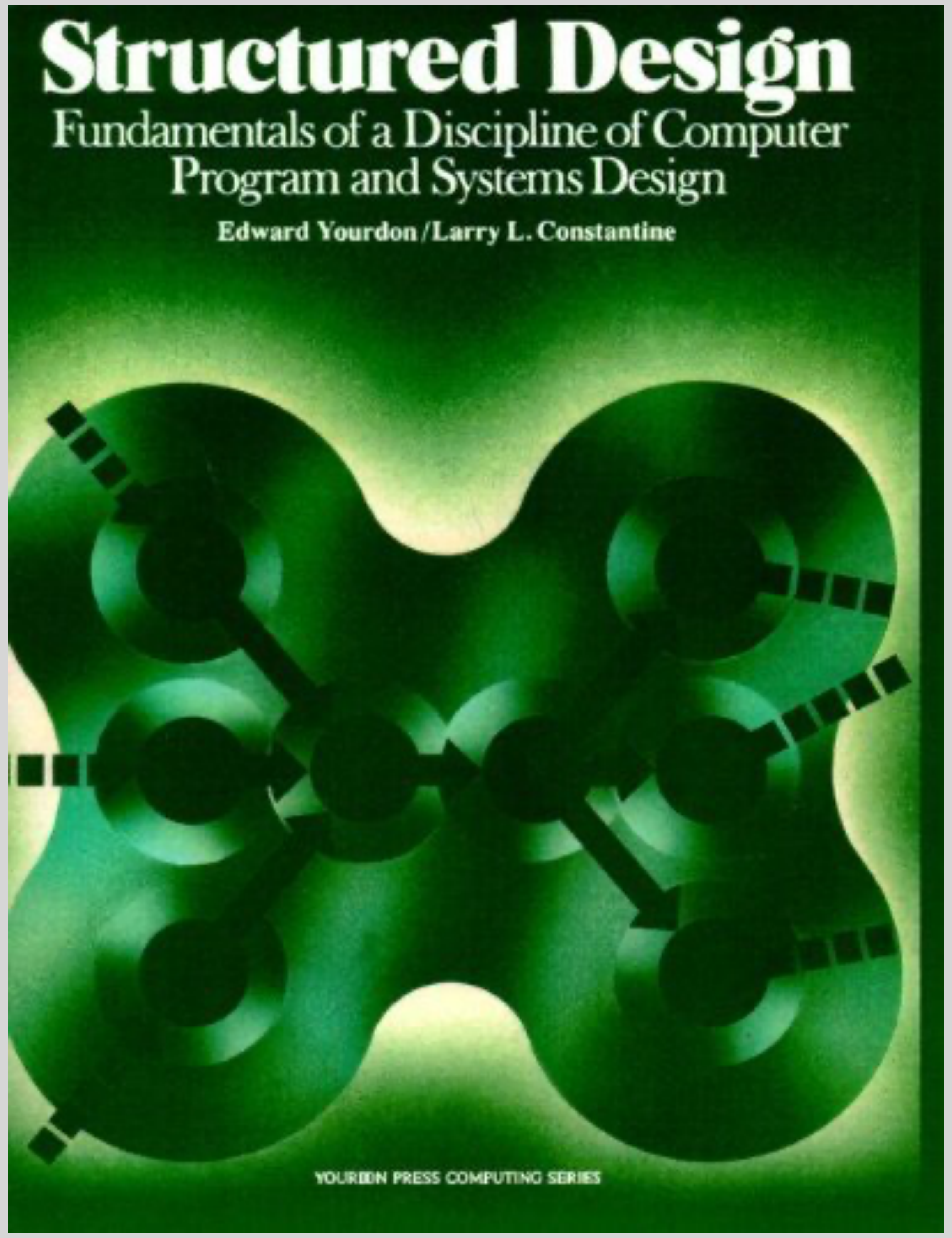
419

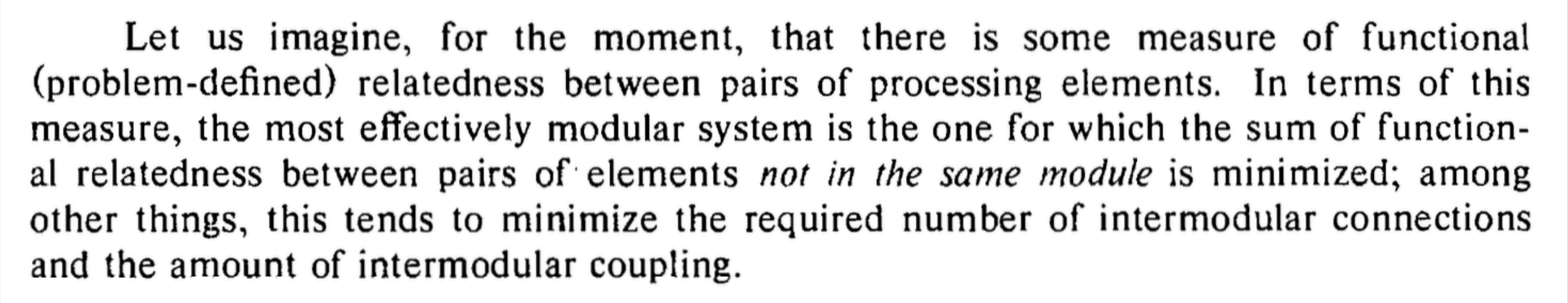
**Dan Farfan** @DanFarfan · Dec 31, 2025

But this time with all the right pronouns?
;-P

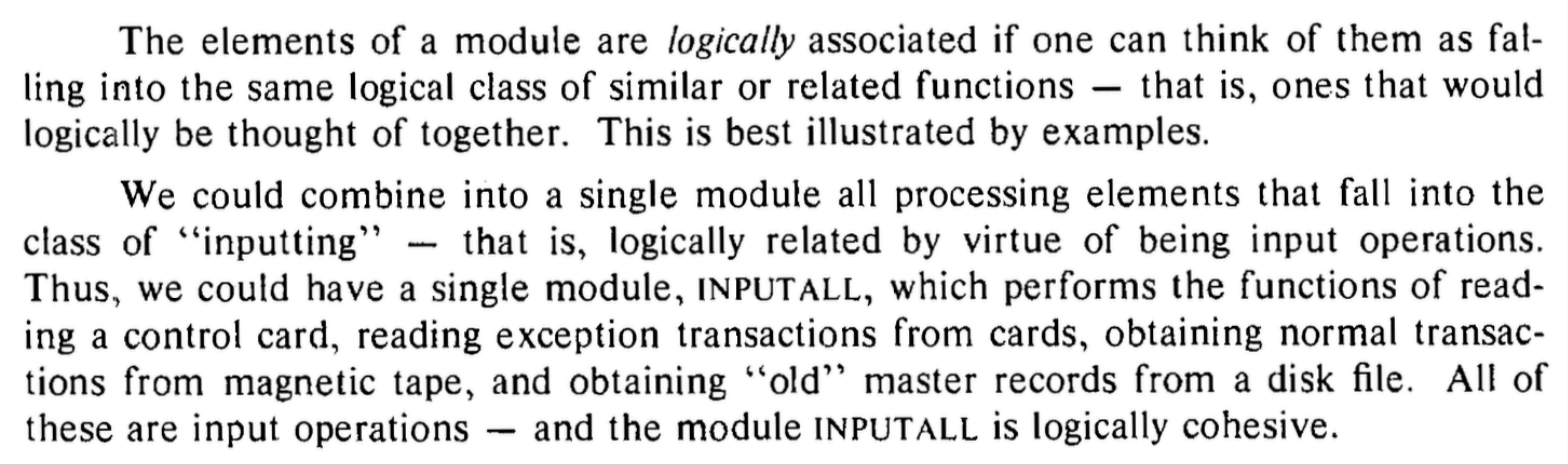
HNY, GB!!

747


Structured Design
Fundamentals of a Discipline of Computer Program and Systems Design
Edward Yourdon/Larry L. Constantine
YOURDON PRESS COMPUTING SERIES



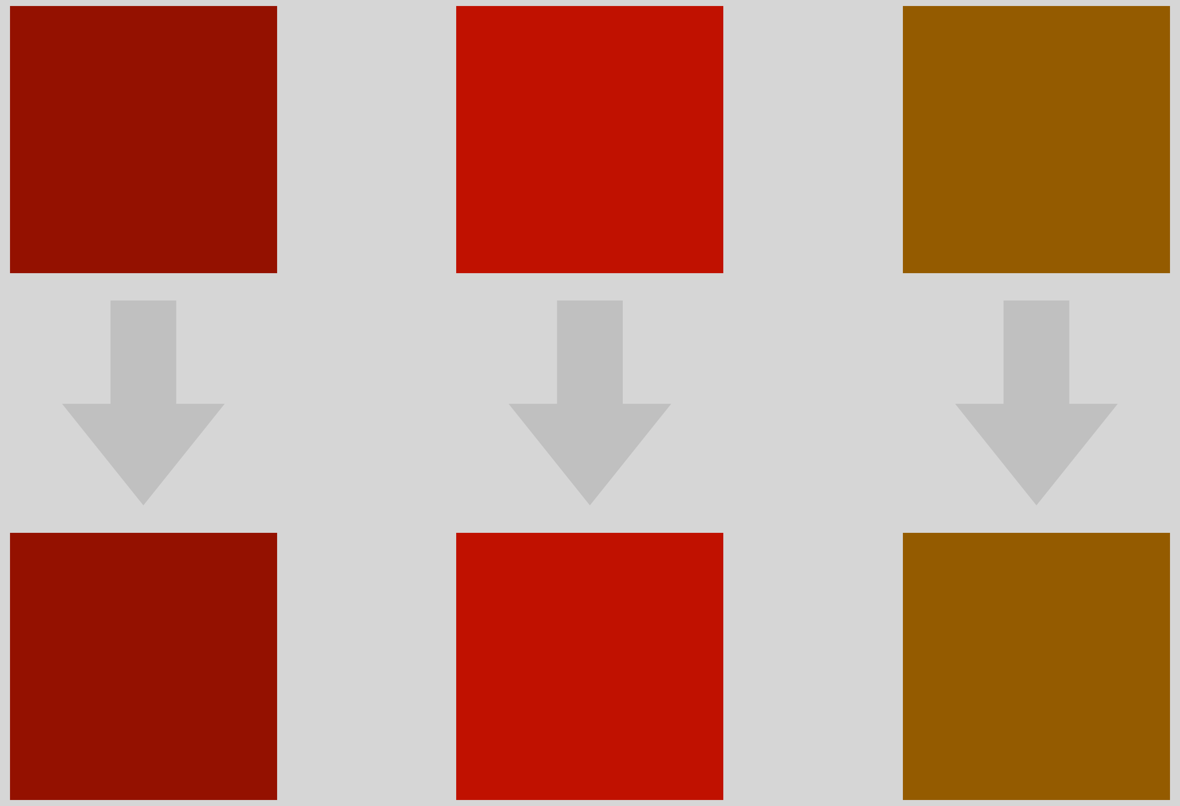
Let us imagine, for the moment, that there is some measure of functional (problem-defined) relatedness between pairs of processing elements. In terms of this measure, the most effectively modular system is the one for which the sum of functional relatedness between pairs of elements *not in the same module* is minimized; among other things, this tends to minimize the required number of intermodular connections and the amount of intermodular coupling.



The elements of a module are *logically* associated if one can think of them as falling into the same logical class of similar or related functions — that is, ones that would logically be thought of together. This is best illustrated by examples.

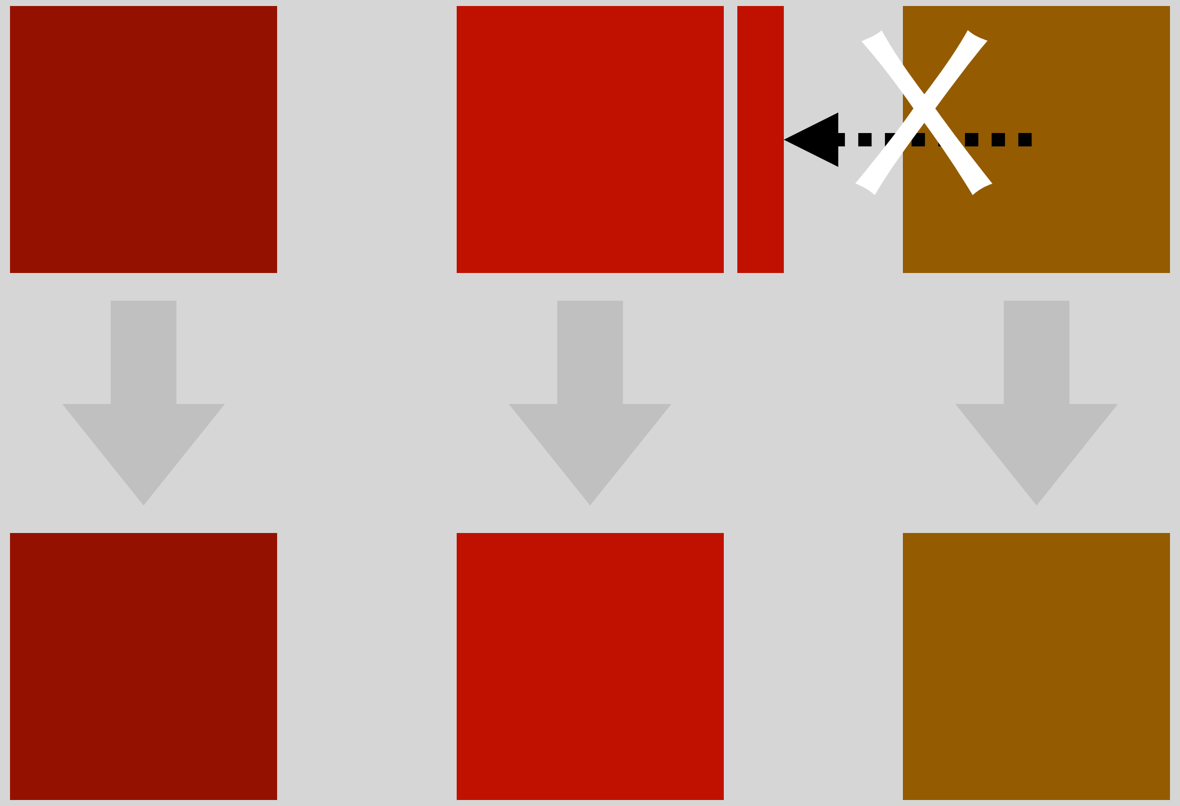
We could combine into a single module all processing elements that fall into the class of ‘‘inputting’’ — that is, logically related by virtue of being input operations. Thus, we could have a single module, INPUTALL, which performs the functions of reading a control card, reading exception transactions from cards, obtaining normal transactions from magnetic tape, and obtaining ‘‘old’’ master records from a disk file. All of these are input operations — and the module INPUTALL is logically cohesive.

# structuring goals & their implications

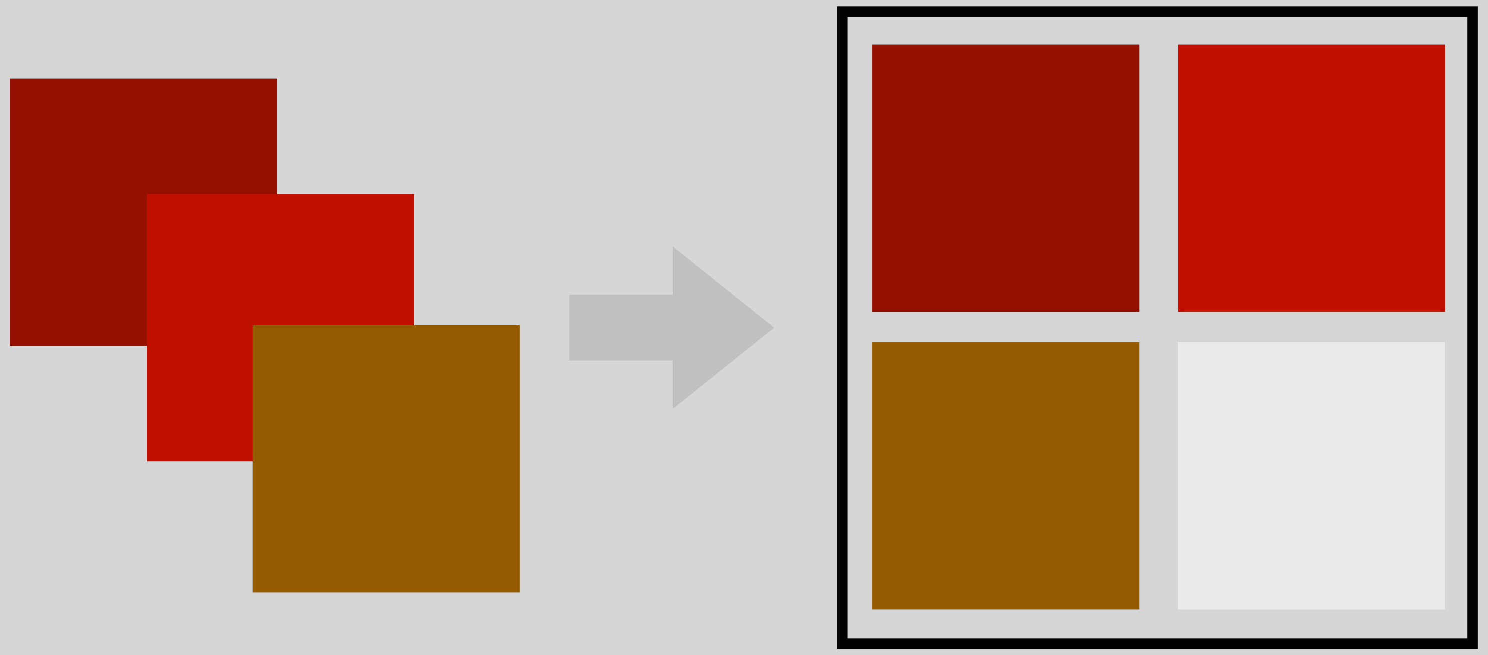


**independent work**
design, build & test independently
minimal context or ordering constraints

implies

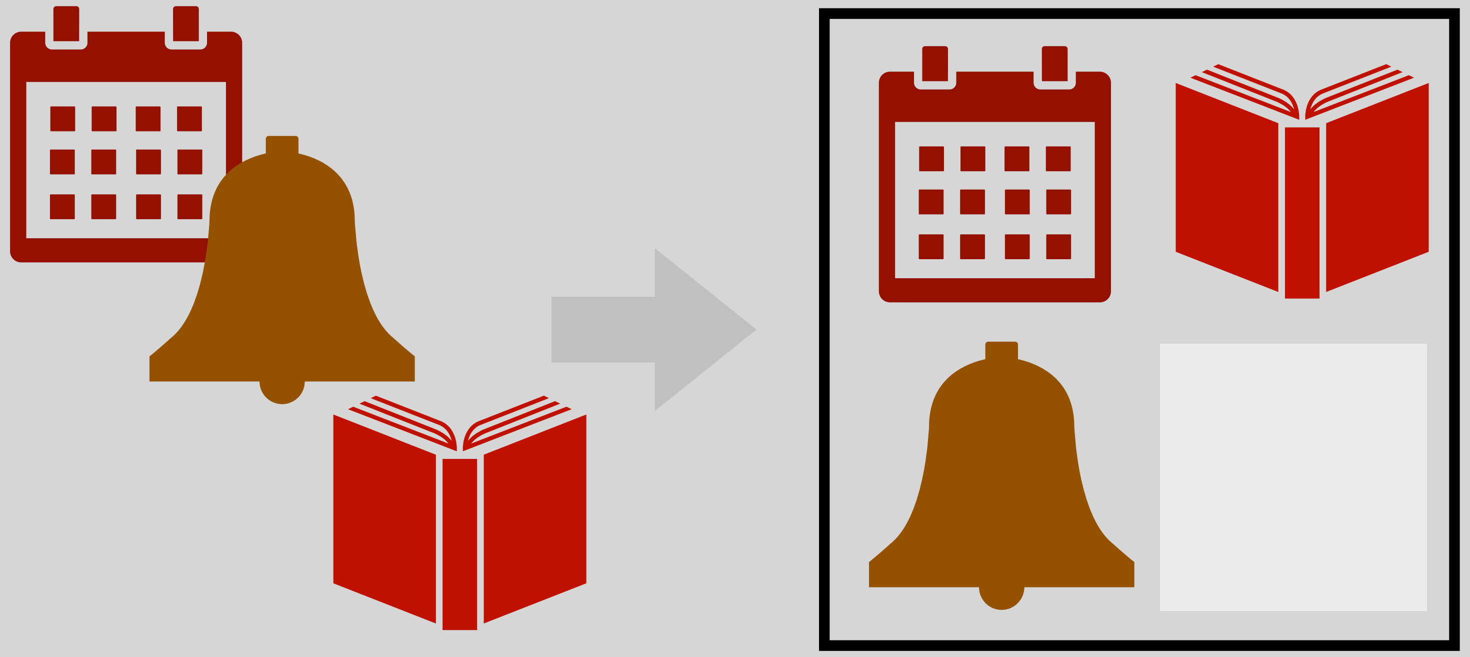


**perfect modularity**
no dependencies, even interfaces
no fragmentation of functionality

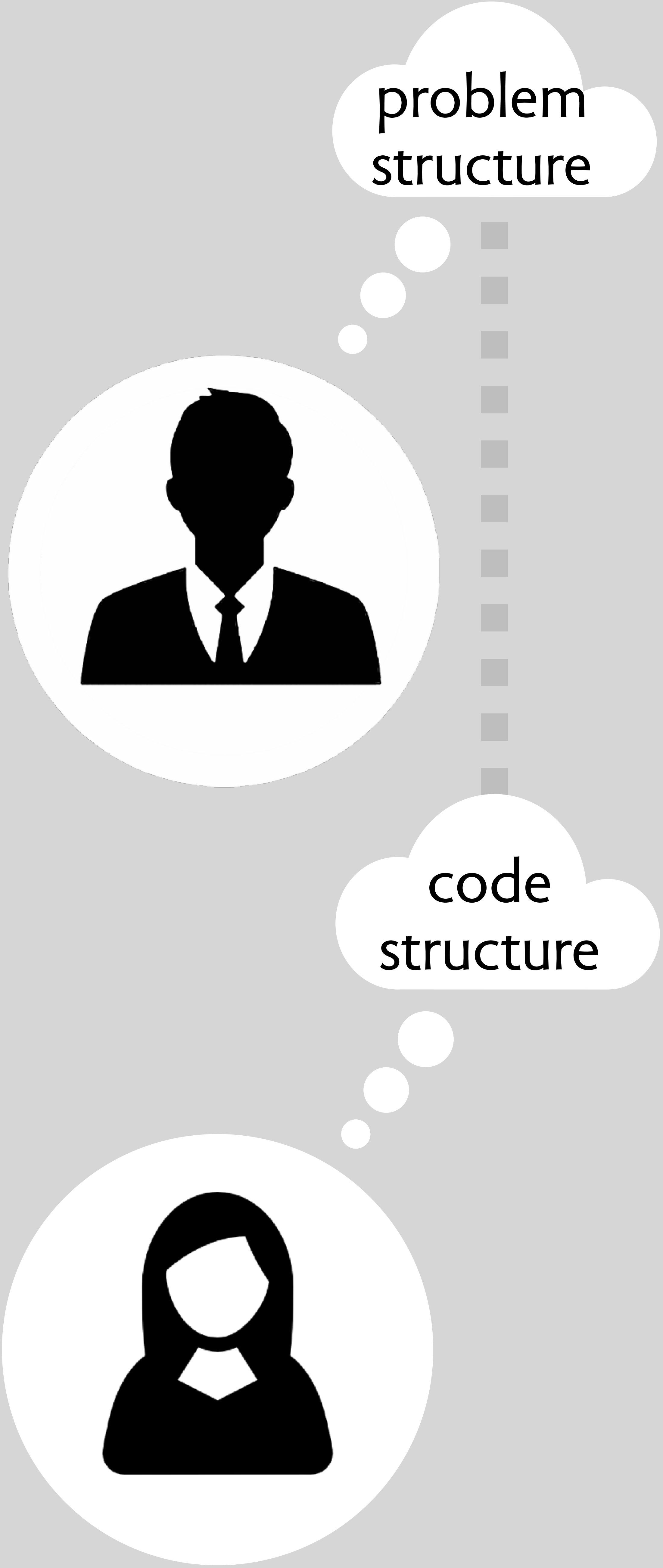


**reuse of designs and code**
development becomes assembly
seal audited components

**implies**



**factor the familiar**
don’t taint with app-specific features
no conflation of functionality



problem structure

code structure

**what you see is what it does**

code directly describes behavior

behavior in code matches behavior in world

**implies**

**base code on behavioral phenomena**
individuals, relationships, actions

**eliminate artifacts**
collection classes, internal events

**avoid behavior-obscuring patterns**
callbacks, factories, proxies

**make coordination explicit**
action rules, not hidden calls

object-oriented thinking shapes many systems

RealWorld example apps
Create
Submit
Sign in
conduit
A place to share your knowledge.
Global feed
Popular tags
Lorem Ipsum

# The mother of all demo apps

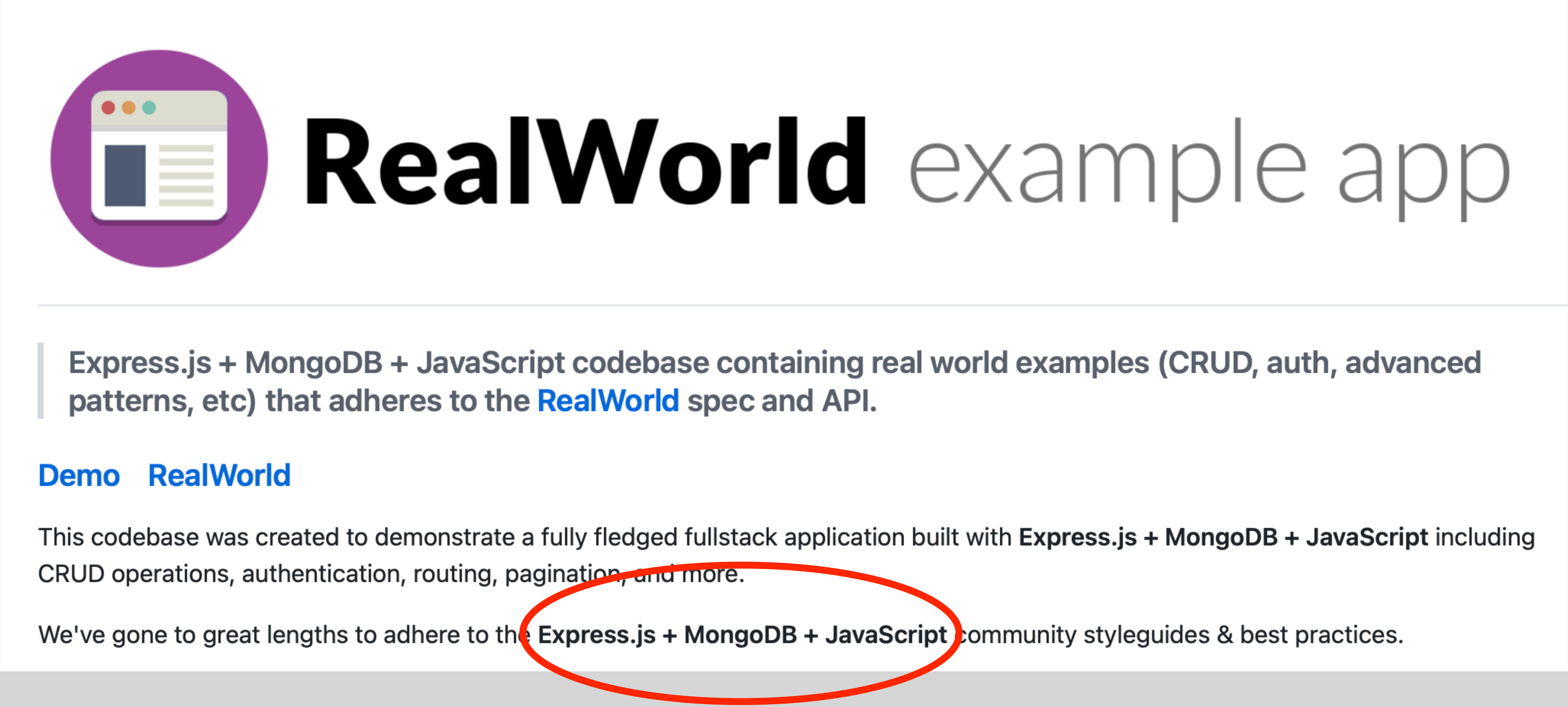
See how the exact same application is built using different libraries and frameworks.

Frontend
Backend
Fullstack

**LANGUAGES**

- All
- TypeScript
- JavaScript
- Kotlin
- ClojureScript
- Elm
- PureScript
- Rust
- C#
- Dart
- Mint
- Swift
- ReScript

| | |
|---|---|
| **Android Native** MOBILE<br>coding-blocks-archives/Conduit_Android_Kotlin | Kotlin |
| **Android Native + Retrofit + Jetpack** MOBILE<br>Marvel999/Conduit-Android-kotlin | Kotlin |
| **Angular**<br>khaledosman/angular-realworld-example-app | TypeScript |
| **Angular**<br>AndyT2503/angular-conduit-signals | TypeScript |
| **Angular**<br>iancharlesdouglas/ng-realworld-ssr | TypeScript |
| **Angular + NgRx + Nx** | TypeScript |



# RealWorld example app

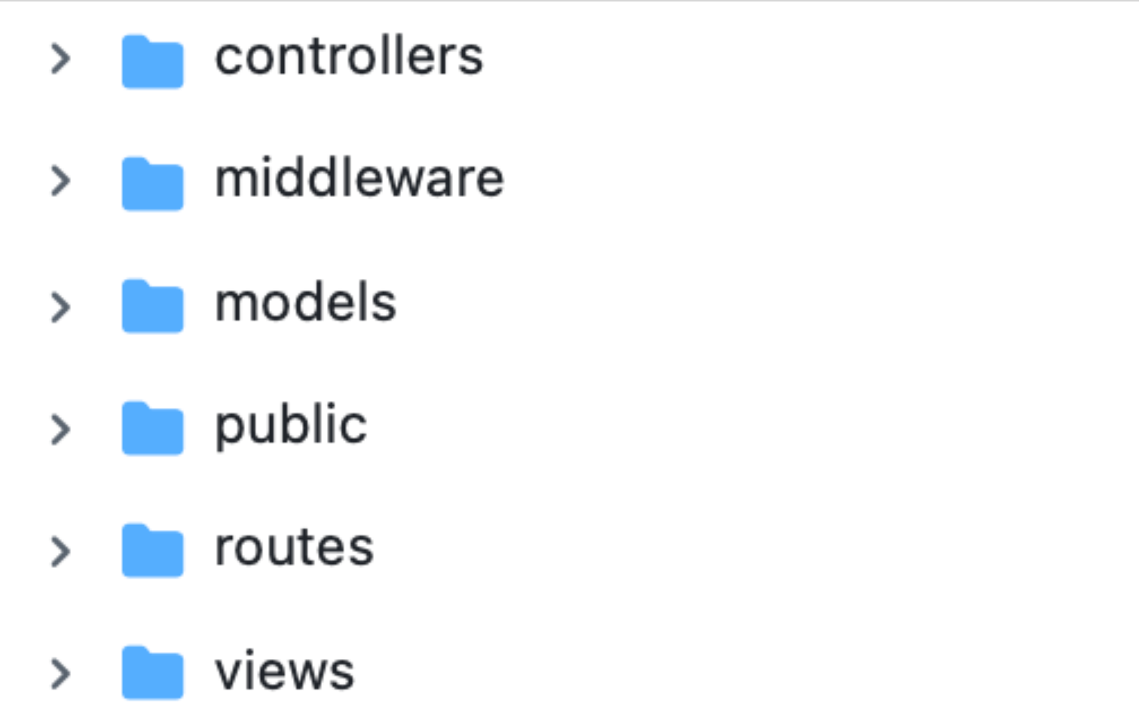
> **Express.js + MongoDB + JavaScript codebase containing real world examples (CRUD, auth, advanced patterns, etc) that adheres to the RealWorld spec and API.**

**Demo** **RealWorld**

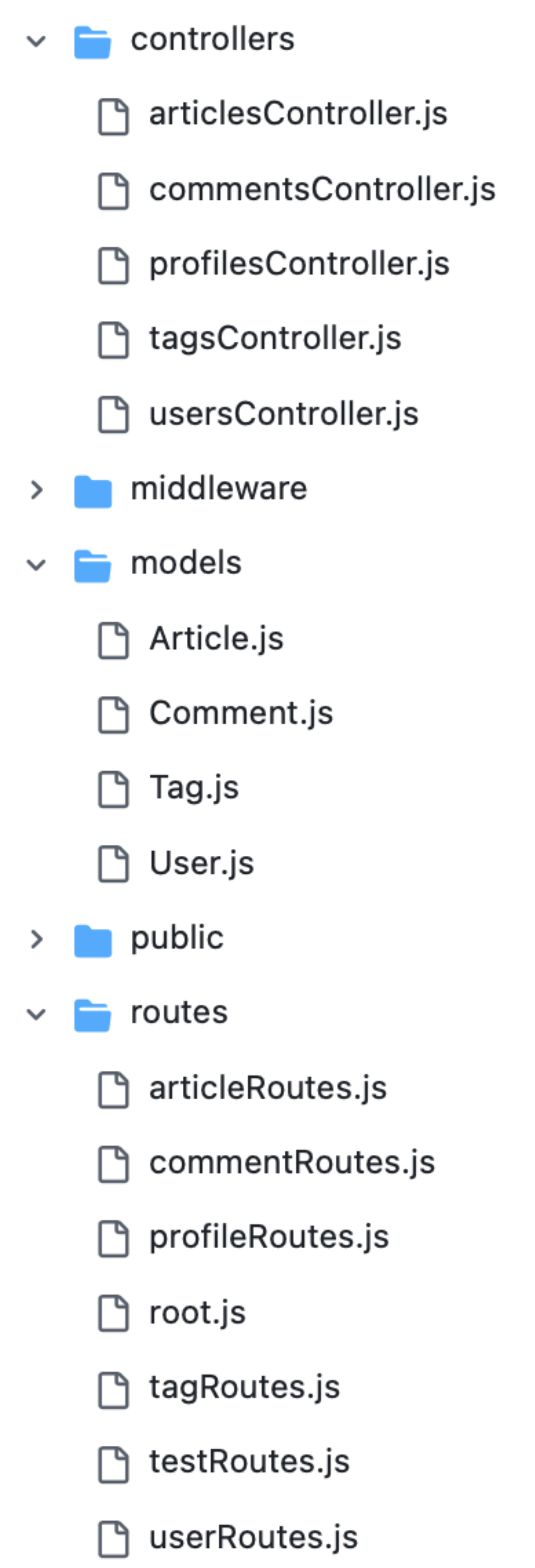
This codebase was created to demonstrate a fully fledged fullstack application built with **Express.js + MongoDB + JavaScript** including CRUD operations, authentication, routing, pagination, and more.

We've gone to great lengths to adhere to the **Express.js + MongoDB + JavaScript** community styleguides & best practices.

https://github.com/winterrrrrff/realWorld-server

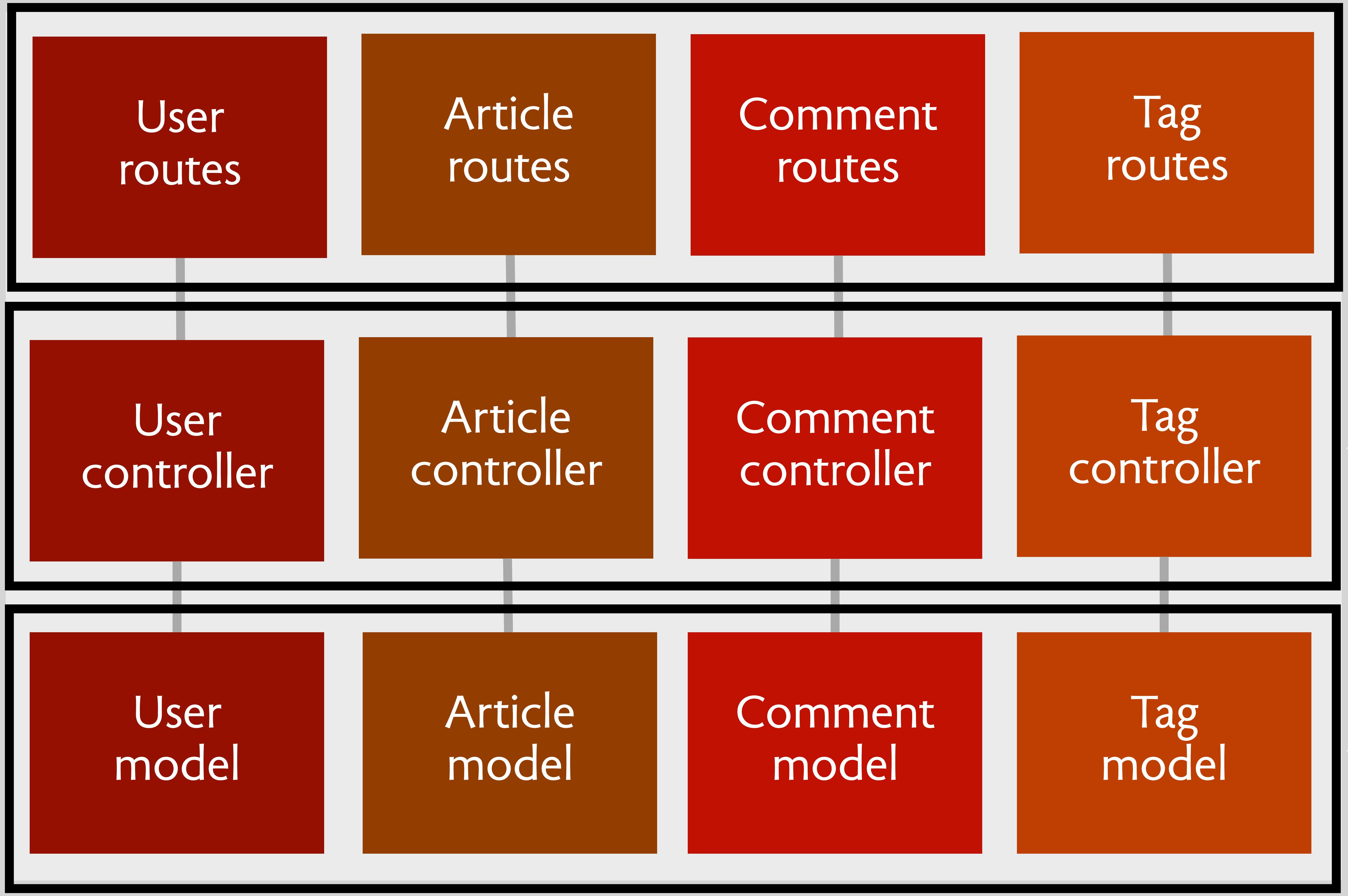


- controllers
- middleware
- models
- public
- routes
- views



- controllers
  - articlesController.js
  - commentsController.js
  - profilesController.js
  - tagsController.js
  - usersController.js
- middleware
- models
  - Article.js
  - Comment.js
  - Tag.js
  - User.js
- public
- routes
  - articleRoutes.js
  - commentRoutes.js
  - profileRoutes.js
  - root.js
  - tagRoutes.js
  - testRoutes.js
  - userRoutes.js

# an object-oriented architecture



User routes

Article routes

Comment routes

Tag routes

User controller

Article controller

Comment controller

Tag controller

User model

Article model

Comment model

Tag model

**routing layer**
encapsulates
HTTP

**controller layer**
encapsulates
business logic

**model layer**
encapsulates
database storage

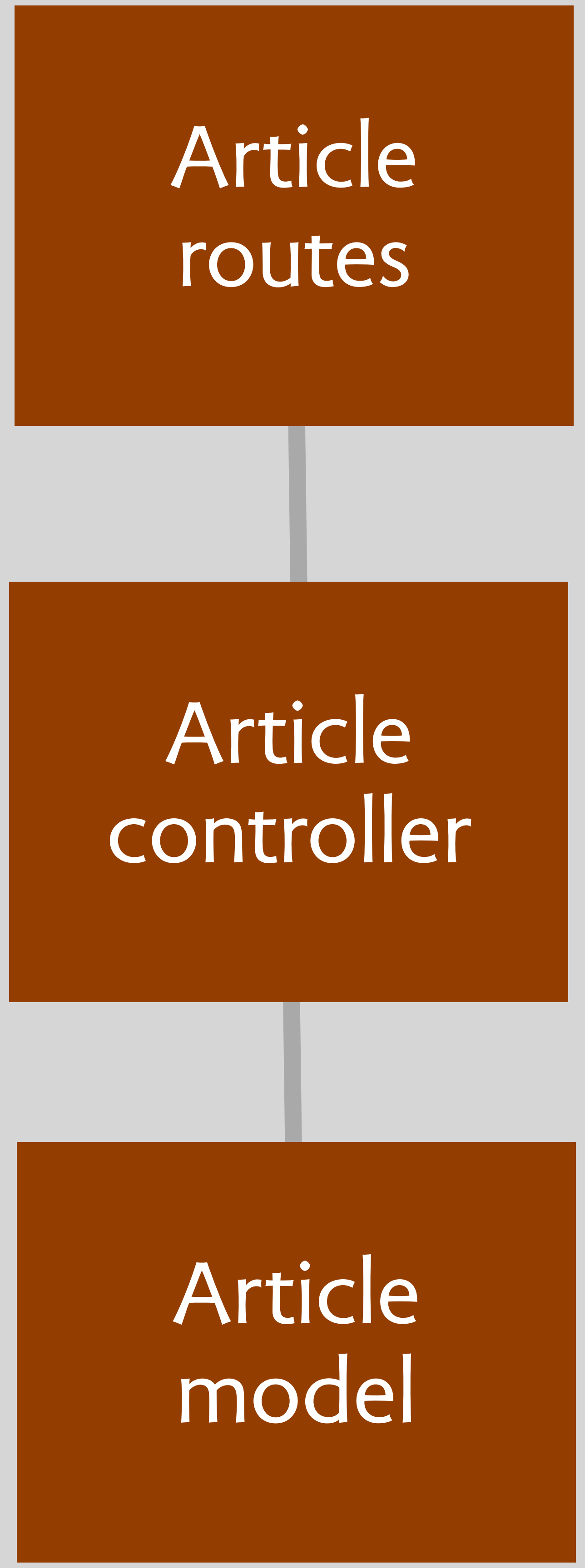
# favoriting an article (part 1)

```
router.post('/:slug/favorite', verifyJWT, articleController.favoriteArticle);
router.delete('/:slug/favorite', verifyJWT, articleController.unfavoriteArticle);
```

```
favoriteArticle = asyncHandler((req, res) => {
   loginUser = User.findById(id).exec();
   article = Article.findOne({slug}).exec();
   loginUser.favorite(article._id);
   updatedArticle = article.updateFavoriteCount();
   ... });
```

```
Article = new mongoose.Schema({
   favouritesCount: {type: Number, default: 0}, ... });

Article.methods.updateFavoriteCount = function () {
   favoriteCount = User.count({favouriteArticles: {$in: [this._id]}});
   this.favouritesCount = favoriteCount;
   return this.save(); }
```



Article routes

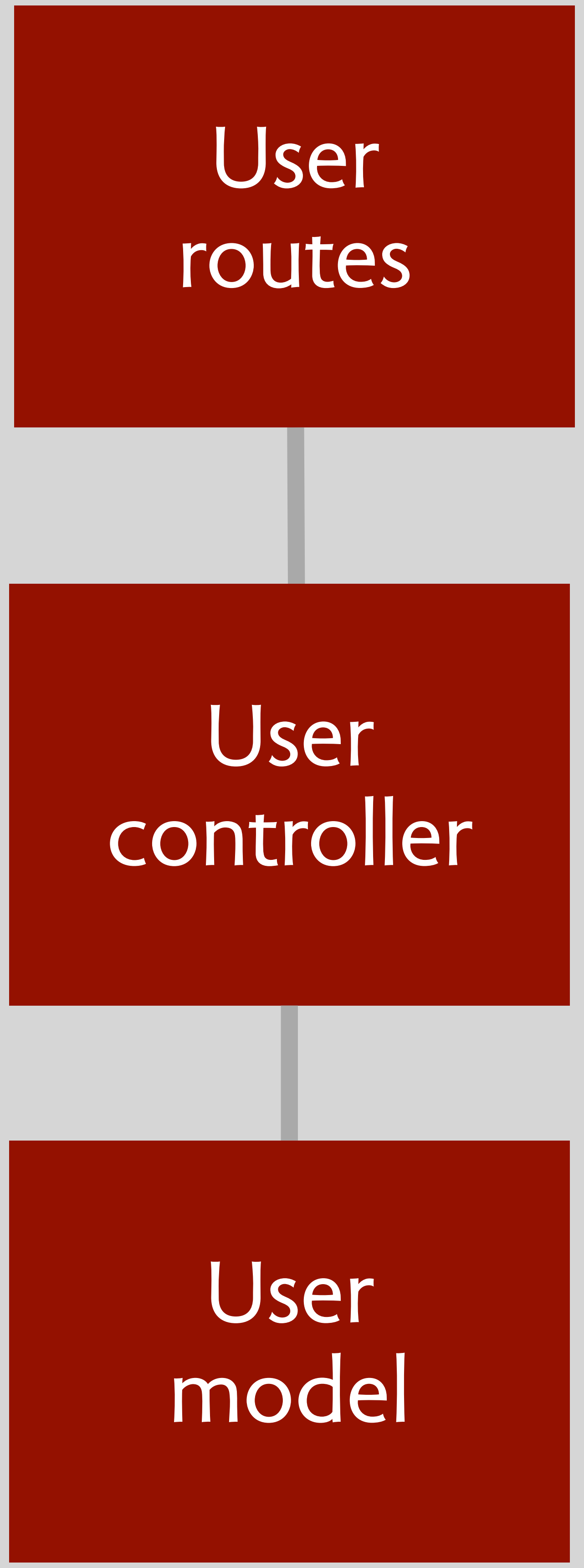
Article controller

Article model

# an favoriting an article (part 2)

```
User = new mongoose.Schema({
  favouriteArticles: [{
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Article'}],...});

User.methods.favorite = function (id) {
  if(this.favouriteArticles.indexOf(id) === -1)
    this.favouriteArticles.push(id);
  // const article = Article.findById(id).exec();
  // article.favouritesCount += 1;
  // article.save();
  return this.save(); }
```



User routes

User controller

User model