

# concept design

## part 3: modularity

Daniel Jackson · Autodesk Online Workshop · June 2025

de/composition:  
design is breaking up  
& putting together

# decomposing into parts with purposes



teapot



body

keeping hot

lid

handle

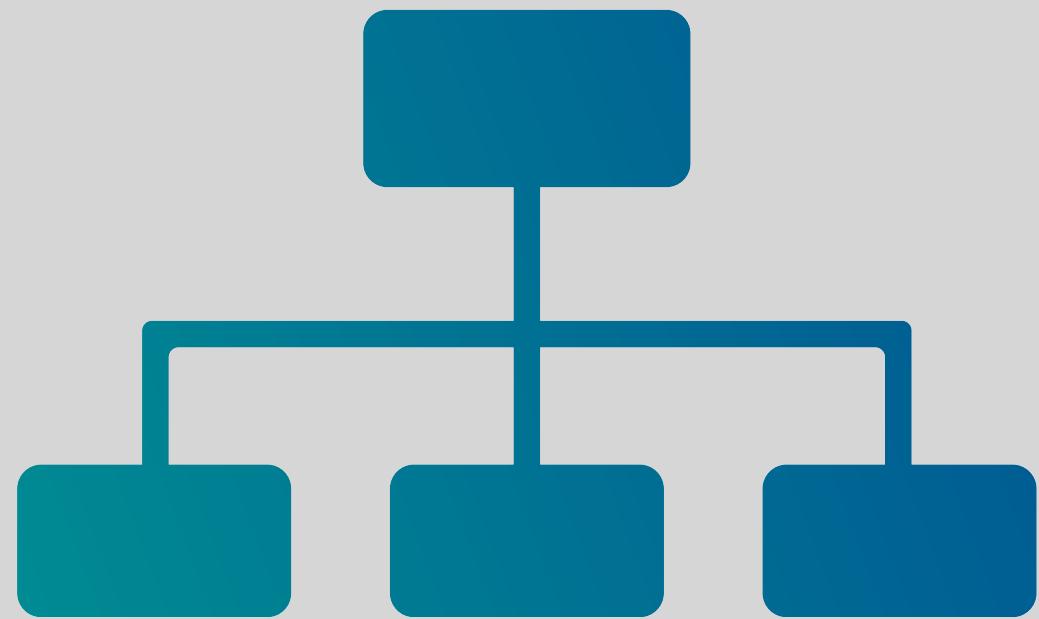
holding

spout

pouring

brewing

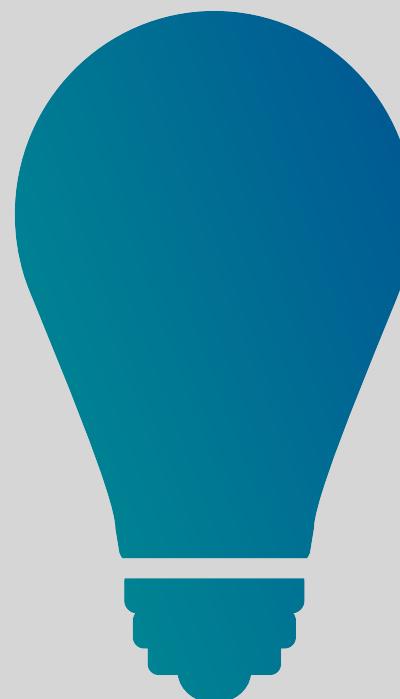
# how does decomposition help?



**incremental work**  
division of labor  
steady progress



**reuse**  
build on experience  
reuse across suite too

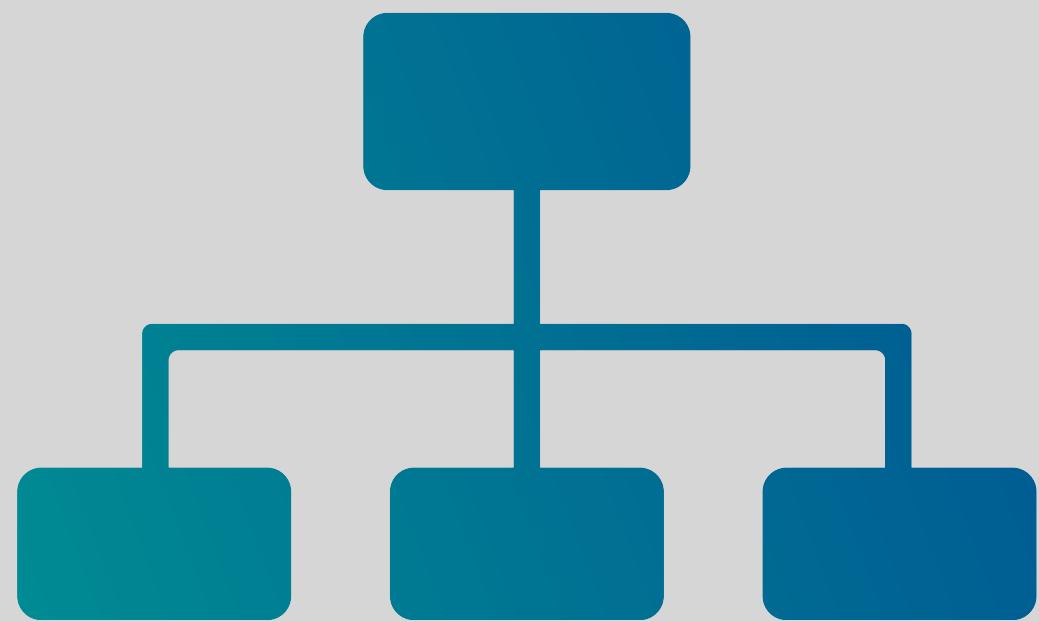


**easier for users**  
identify familiar parts  
learn what you need

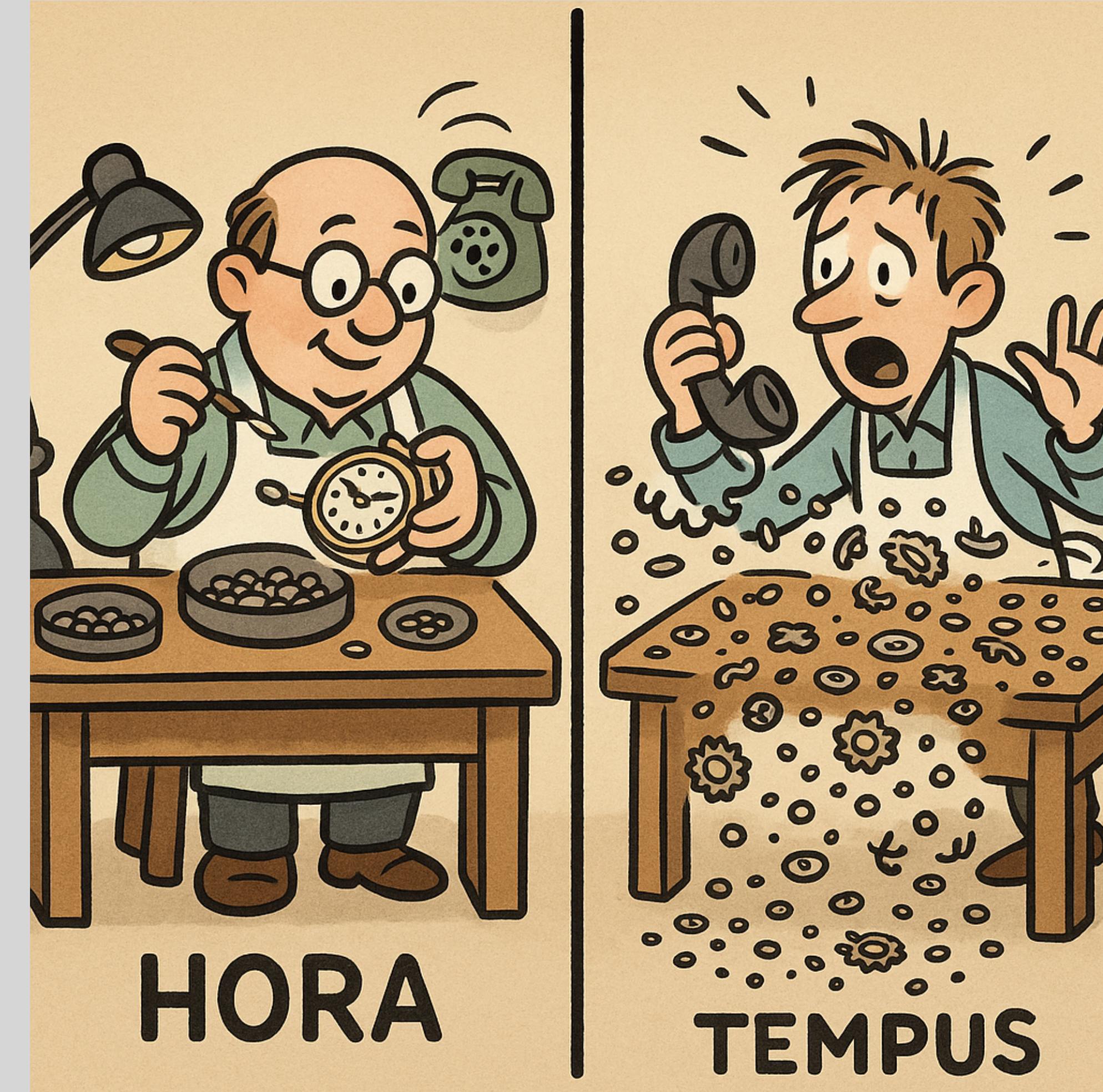


**design focus**  
separate concerns  
drive by purposes

# Simon: the two watchmakers



incremental work  
division of labor  
steady progress



*Herb Simon, The Architecture of Complexity (1962)*

cartoon by ChatGPT

# how unique is it?



reuse

build on experience  
reuse across suite too

**Hacker News** new | past | comments | ask | show | jobs | submit [login](#)

▲ Jackson structured programming ([wikipedia.org](https://en.wikipedia.org))  
106 points by haakonhr 63 days ago | hide | past | favorite | 69 comments

▲ danielnicholas 63 days ago [-]  
If you want an intro to JSP, you might find helpful an annotated version [0] of Hoare's explanation of JSP that I edited for a Michael Jackson festschrift in 2009.  
For those who don't know JSP, I'd point to these ideas as worth knowing:  
- There's a class of programming problem that involves traversing context-free structures can be solved very systematically. HTDP addresses this class, but bases code structure only on input structure; JSP synthesized input and output.  
- There are some archetypal problems that, however you code, can't be pushed under the rug—most notably structure clashes—and just recognizing them helps.  
- Coroutines (or code transformation) let you structure code more cleanly when you need to read or write more than one structure. It's why real iterators (with yield), which offer a limited form of this, are (in my view) better than Java-style iterators with a next method.  
- The idea of viewing a system as a collection of asynchronous processes (Ch. 11 in the JSP book, which later became JSD) with a long-running process for each real-world entity. This was a notable contrast to OOP, and led to a strategy (seeing a resurgence with event storming for DDD) that began with events rather than objects.  
[0] <https://groups.csail.mit.edu/sdg/pubs/2009/hoare-jsp-3-29-09...>

▲ ob-nix 63 days ago [-]  
... this brings back memories! In the late eighties I, as a teenager, found a Jackson Struct. Pr. book at the town library. I remember I was amazed at the text and wondered why I hadn't heard about the method before.  
If I remember correctly did the book clearly point out backtracking as a standard method, while mentioning that most languages lacked that, so it had to be implemented manually.

▲ CraigJPerry 63 days ago [-]  
This is referenced(1) as a core inspiration in the preface to "How to Design Programs" but i never researched it further because i've found the "design recipes" approach in htdp to be pretty solid in real life problems.

no other app is the same as HackerNews

HackerNews = Post + Comment + Upvote + Karma + ...

but its concepts are mostly identical to the concepts in other apps

# Dijkstra: separation of concerns



**design focus**  
separate concerns  
drive by purposes

"Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects.

It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focussing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.

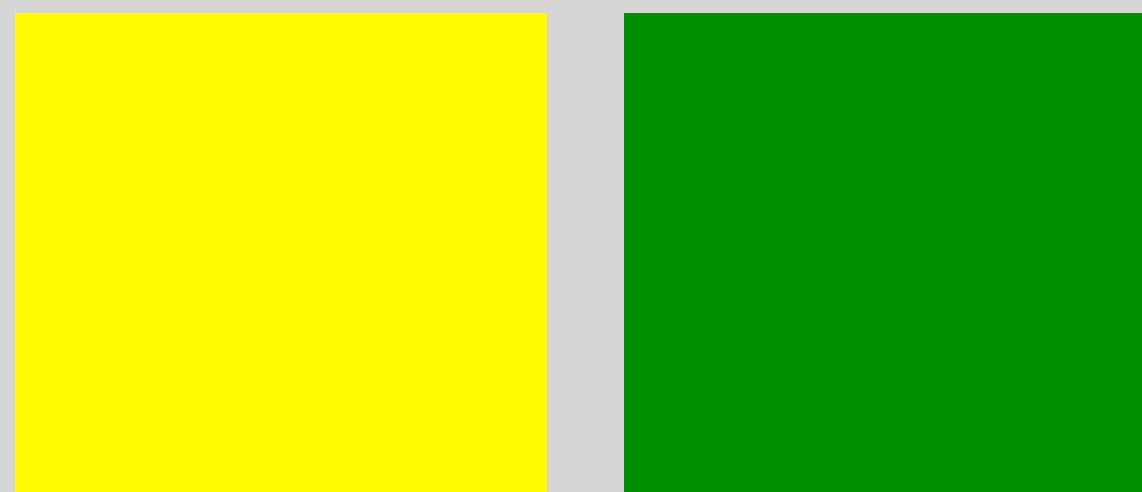
*Edsger Dijkstra, On the role of scientific thought (EWD447, 1974)*

modularity  
3 criteria

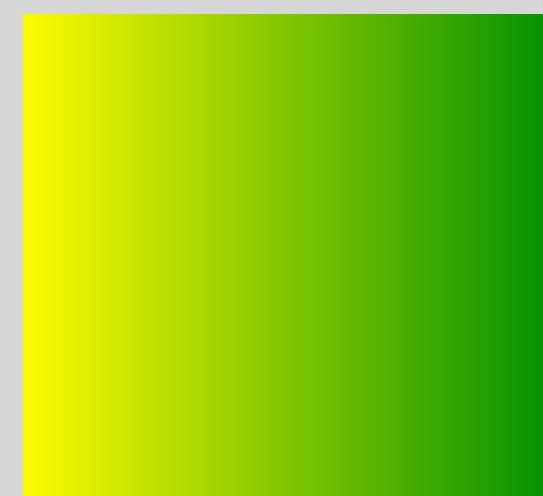
# defining modularity

## non-conflation

a single module doesn't conflate unrelated functions



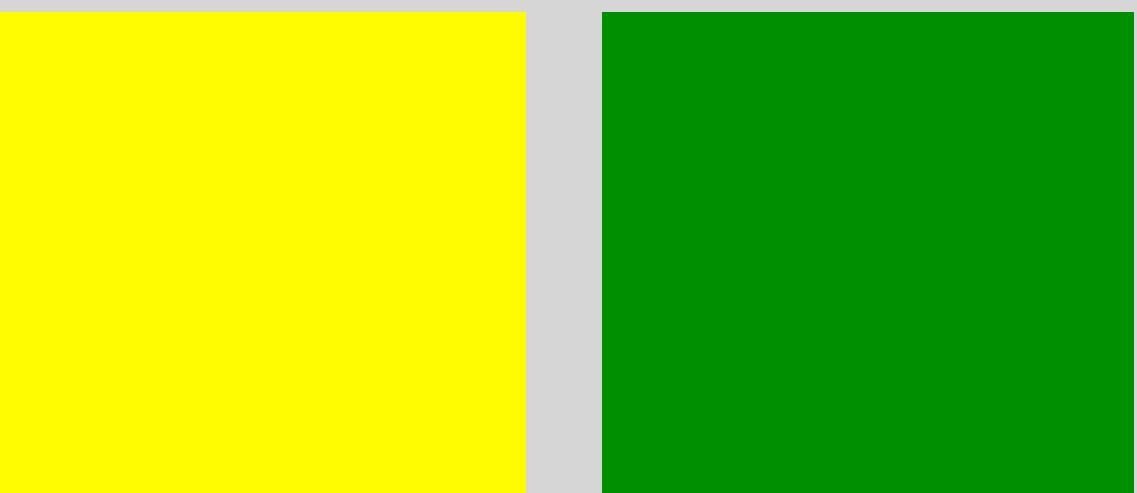
*separated: not conflated*



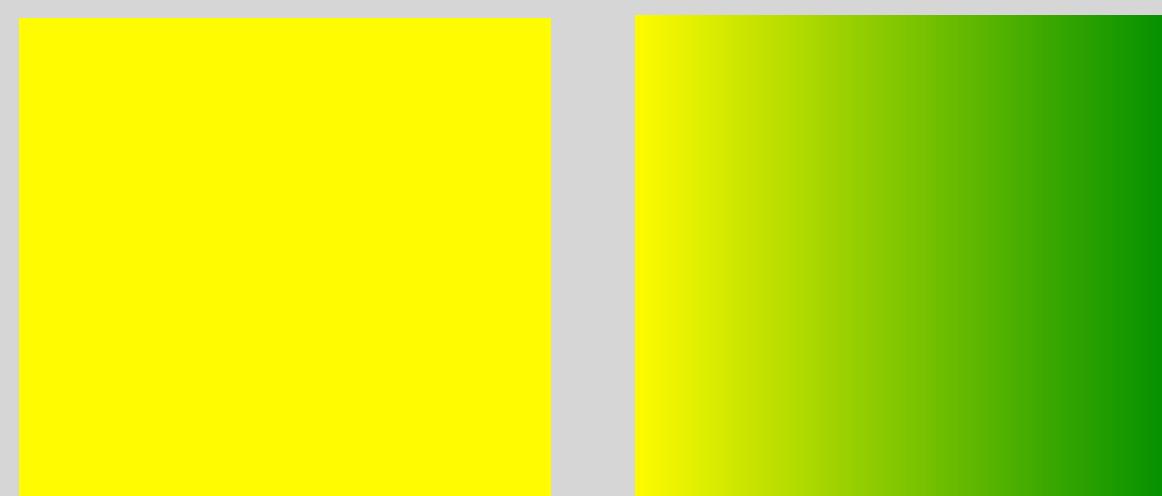
*conflated*

## non-fragmentation

a single module contains all of a function's behavior



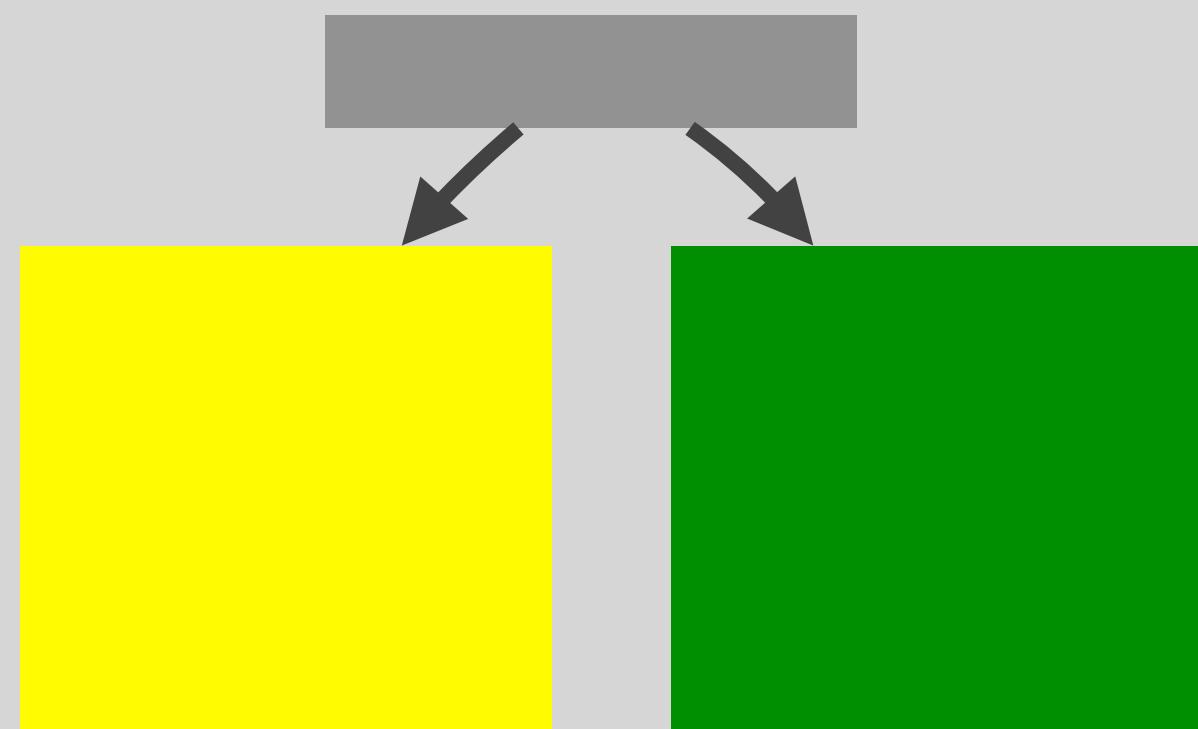
*complete: not fragmented*



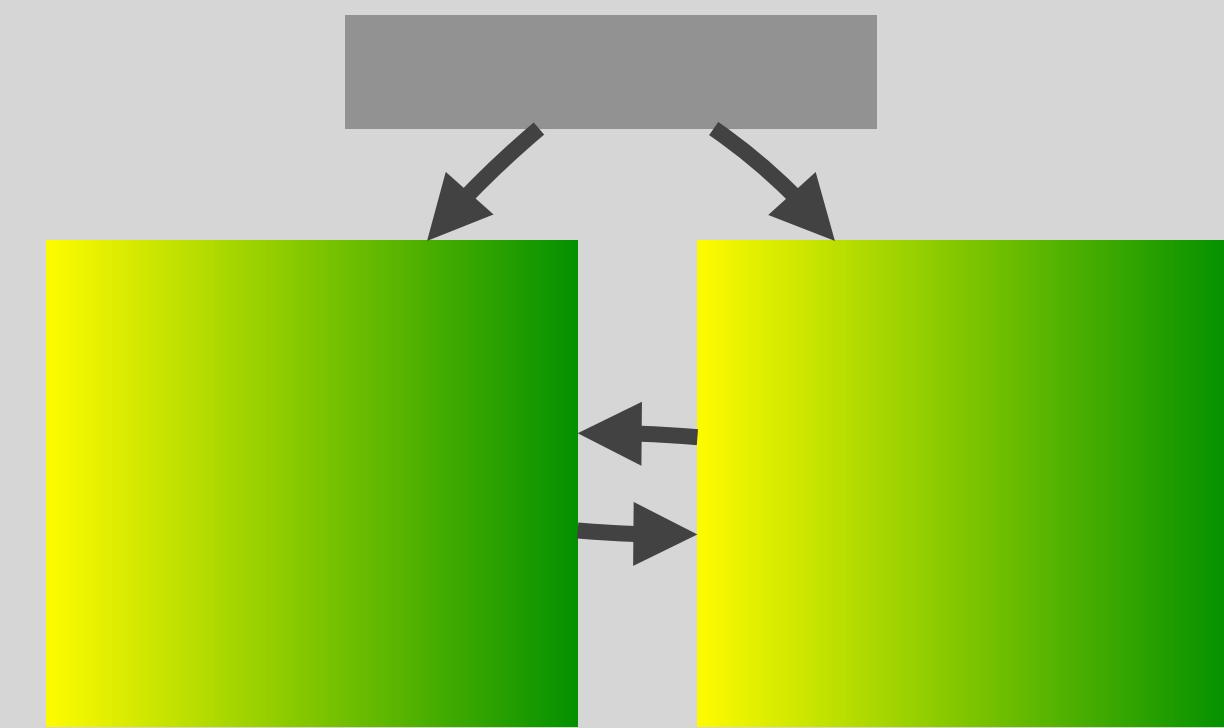
*fragmented*

## independence

one module doesn't rely on another



*independent*



*dependent*

## check your understanding

A concept in a design app lets users create projects that assemble models and track their changes over time.

Which modularity criterion is this likely to violate? (pick one)

- (a) Completeness, because it should also include the ability to edit models
- (b) Independence, because modifications of models in other concepts will affect this one
- (c) Separation, because it mixes purposes related to versioning and aggregation

synchronization  
how to decouple



## ▲ Jackson structured programming (wikipedia.org)

106 points by haakonhr 63 days ago | hide | past | favorite | 69 comments

## ▲ danielnicholas 63 days ago [-]

If you want an intro to JSP, you might find helpful an annotated version [0] of Hoare's explanation of JSP that I edited for a Michael Jackson festschrift in 2009.

For those who don't know JSP, I'd point to these ideas as worth knowing:

- There's a class of programming problem that involves traversing context-free structures can be solved very systematically. HTDP addresses this class, but bases code structure only on input structure; JSP synthesized input and output.
- There are some archetypal problems that, however you code, can't be pushed under the rug—most notably structure clashes—and just recognizing them helps.
- Coroutines (or code transformation) let you structure code more cleanly when you need to read or write more than one structure. It's why real iterators (with `yield`), which offer a limited form of this, are (in my view) better than Java-style iterators with a `next` method.
- The idea of viewing a system as a collection of asynchronous processes (Ch. 11 in the JSP book, which later became JSD) with a long-running process for each real-world entity. This was a notable contrast to OOP, and led to a strategy (seeing a resurgence with event storming for DDD) that began with events rather than objects.

[0] <https://groups.csail.mit.edu/sdg/pubs/2009/hoare-jsp-3-29-09...>

## ▲ ob-nix 63 days ago [-]

... this brings back memories! In the late eighties I, as a teenager, found a Jackson Struct. Pr. book at the town library. I remember I was amazed at the text and wondered why I hadn't heard about the method before.

If I remember correctly did the book clearly point out backtracking as a standard method, while mentioning that most languages lacked that, so it had to be implemented manually.

# adding application-specific functionality

**concept** Upvote

**purpose** rank items by popularity

**actions**

upvote (u: User, i: Item)

downvote (u: User, i: Item)

unvote (u: User, i: Item)

**suppose I want this behavior:**

you can't downvote an item

until you've received

an upvote on your own post

**could just modify Upvote**

why is this bad?

**define a new concept!**

a hint: not just used by Upvote

**concept** Karma

**purpose** privilege good users

**state**

set of users each with  
karma points (an integer)

**actions**

reward (u: User, r: Int)

**concept** Posting

**purpose** share content

**state**

a set of posts each with  
a body (text)  
an author (user)

**actions**

create (u: User, t: Text): Post  
delete (p: Post)  
edit (p: Post, t: Text)

# composing concepts with synchronizations

## concept Upvote

### actions

upvote (u: User, i: Item)

downvote (u: User, i: Item)

unvote (u: User, i: Item)

## concept Web

### actions

request (...)

**when** Upvote.upvote (u, i)  
**where** author of i is u' in Posting  
**then** Karma.reward (u', 10)

**when** User.request (downvote, u, i)  
**where** u has  $\geq 20$  points in Karma  
**then** Upvote.downvote (u, i)

## concept Karma

### state

set of users each with  
a number of karma points

### actions

reward (u: User, r: Int)

## concept Posting

### state

a set of posts each with an author

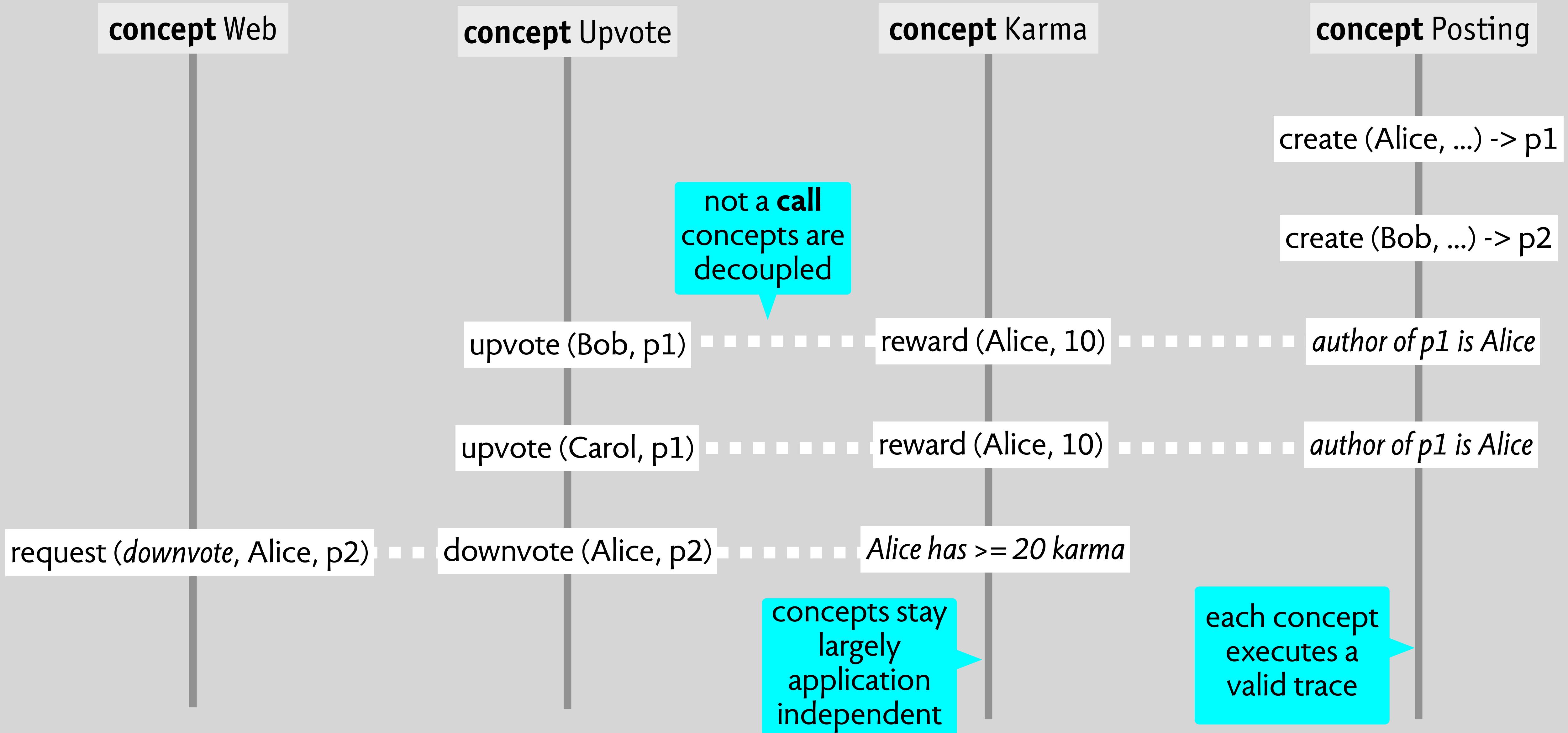
### actions

create (u: User, t: Text): Post

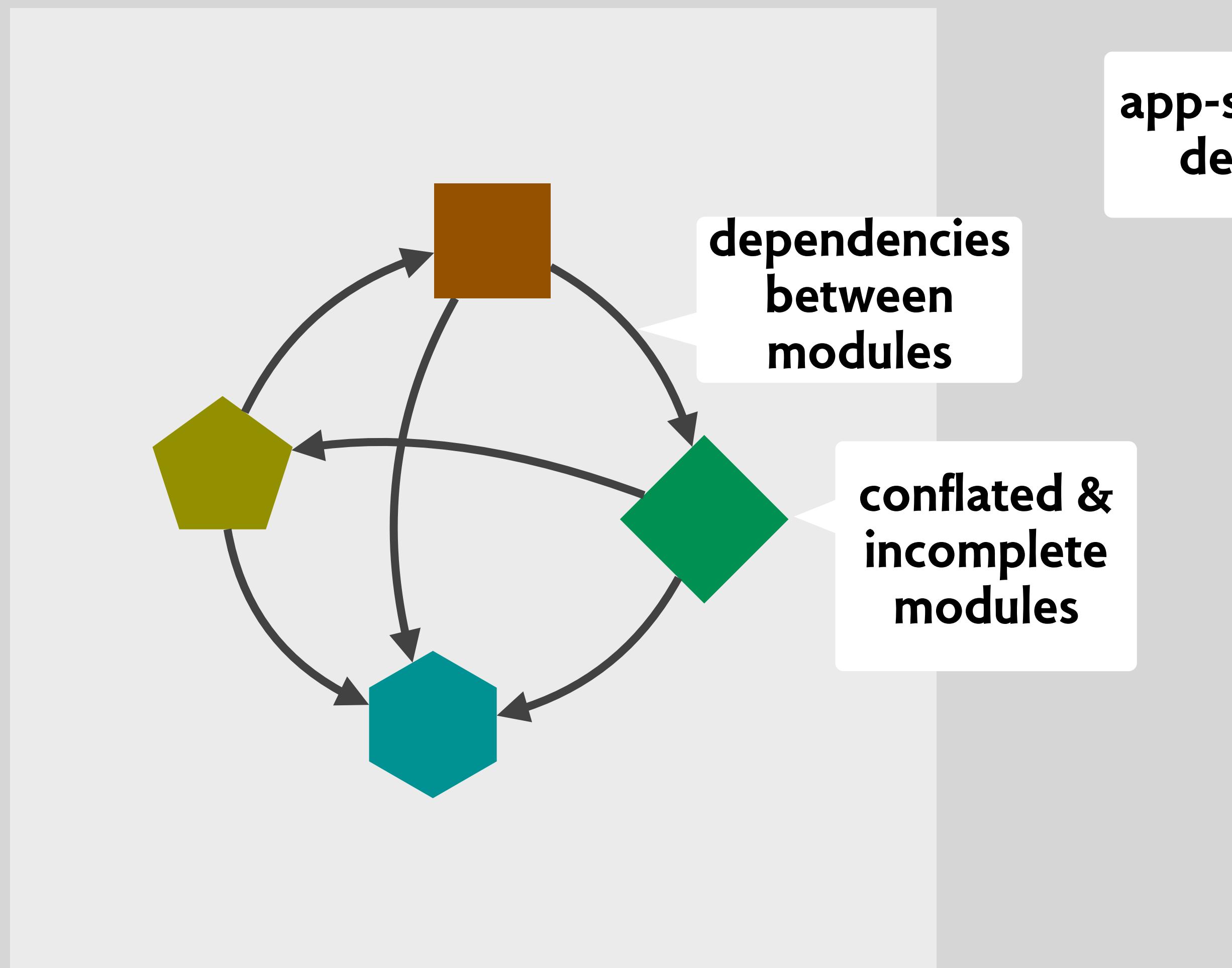
delete (p: Post)

edit (p: Post, t: Text)

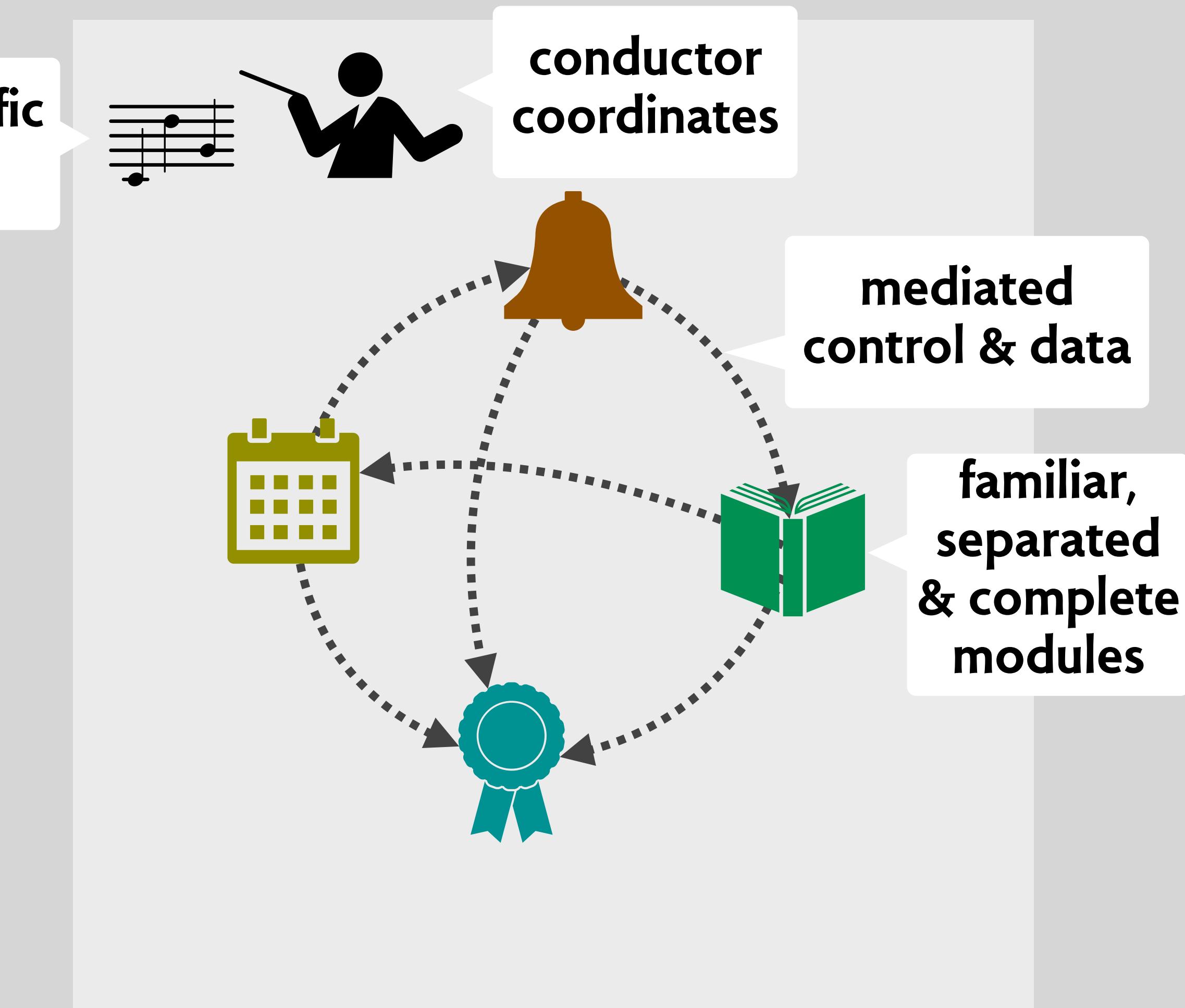
# synchronization viewed over traces



# an architectural view of concept composition



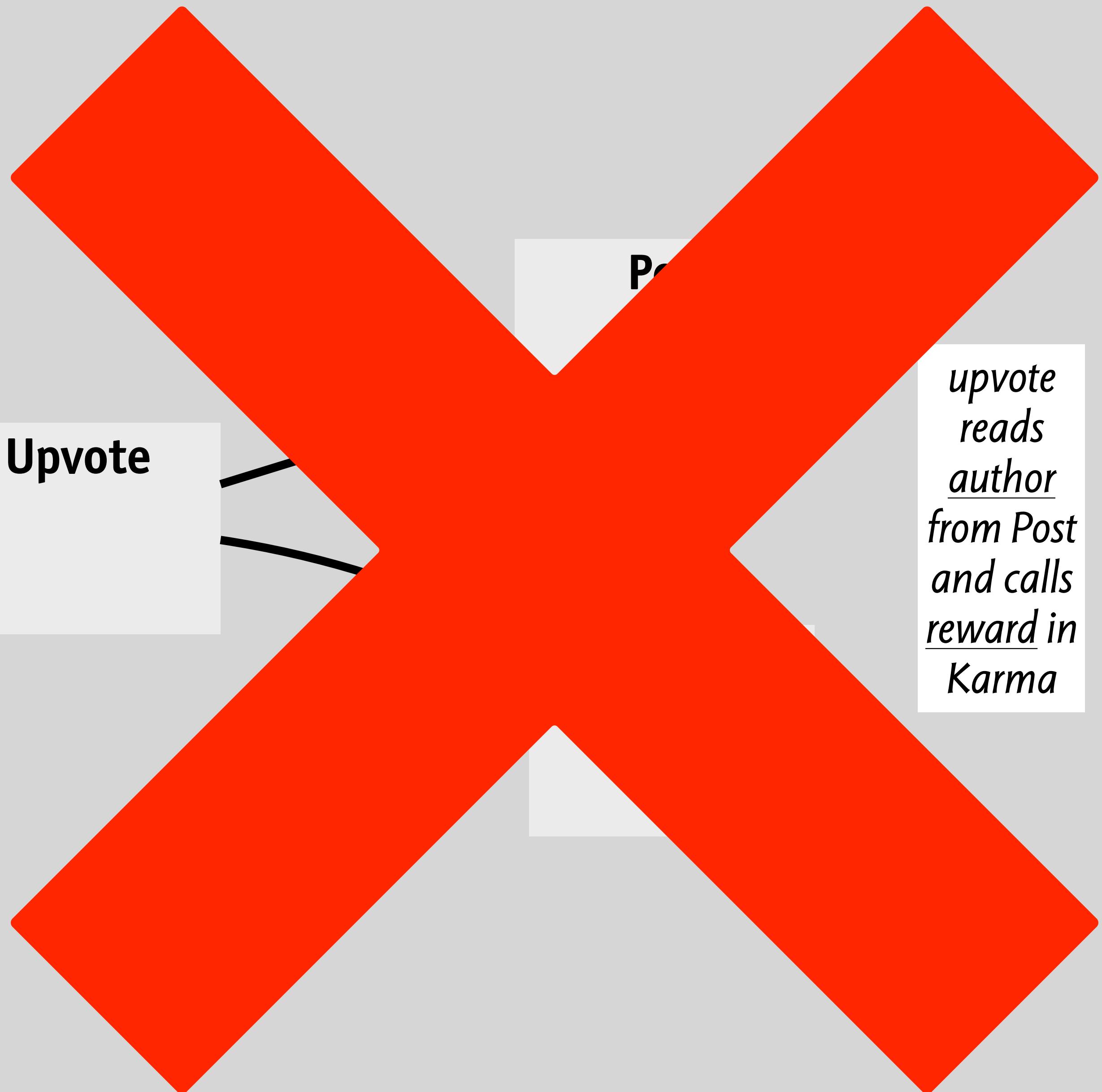
*standard software development*



*concept-based software development*

enforcing  
independence

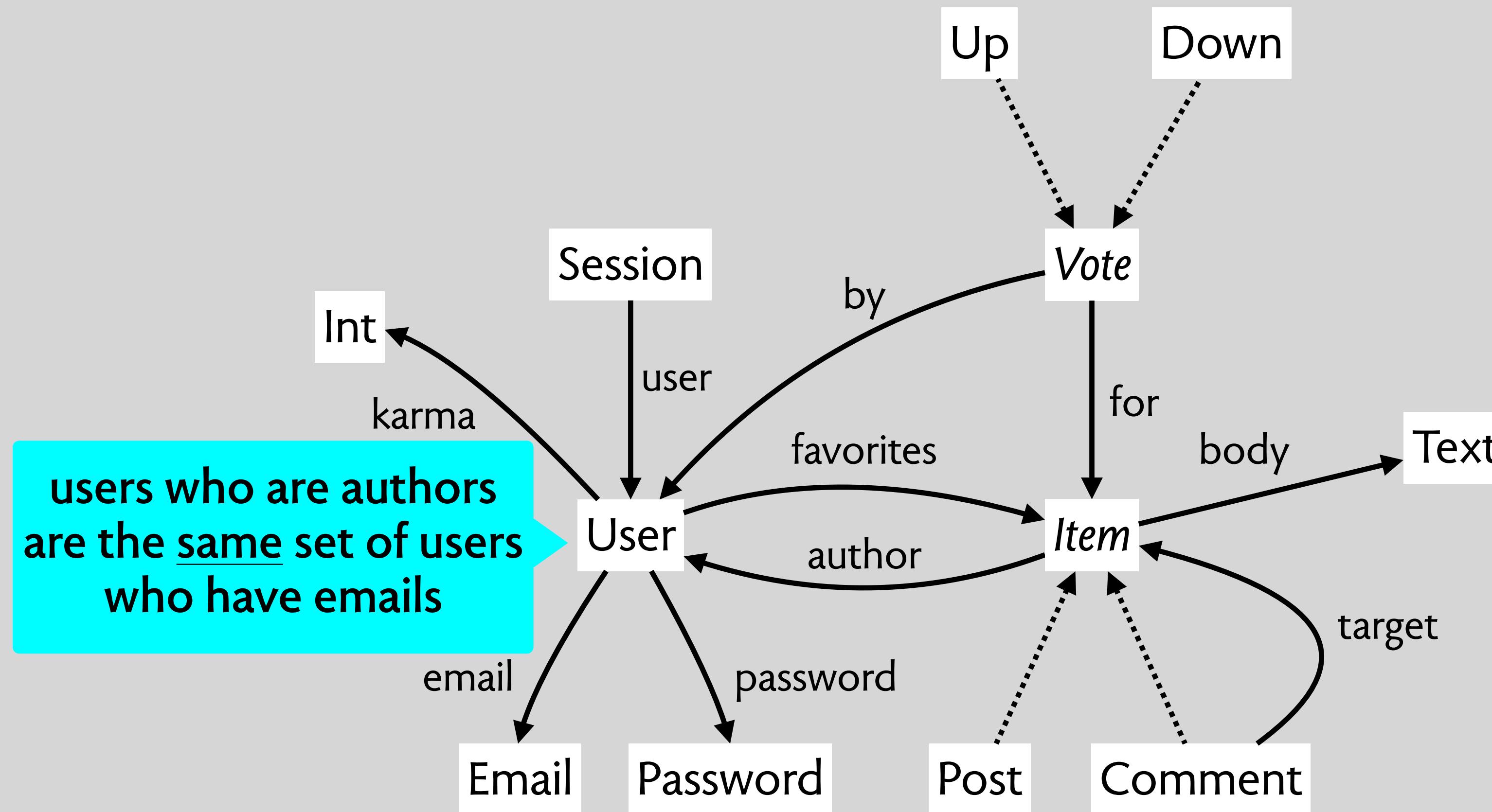
# how concepts do **not** interact



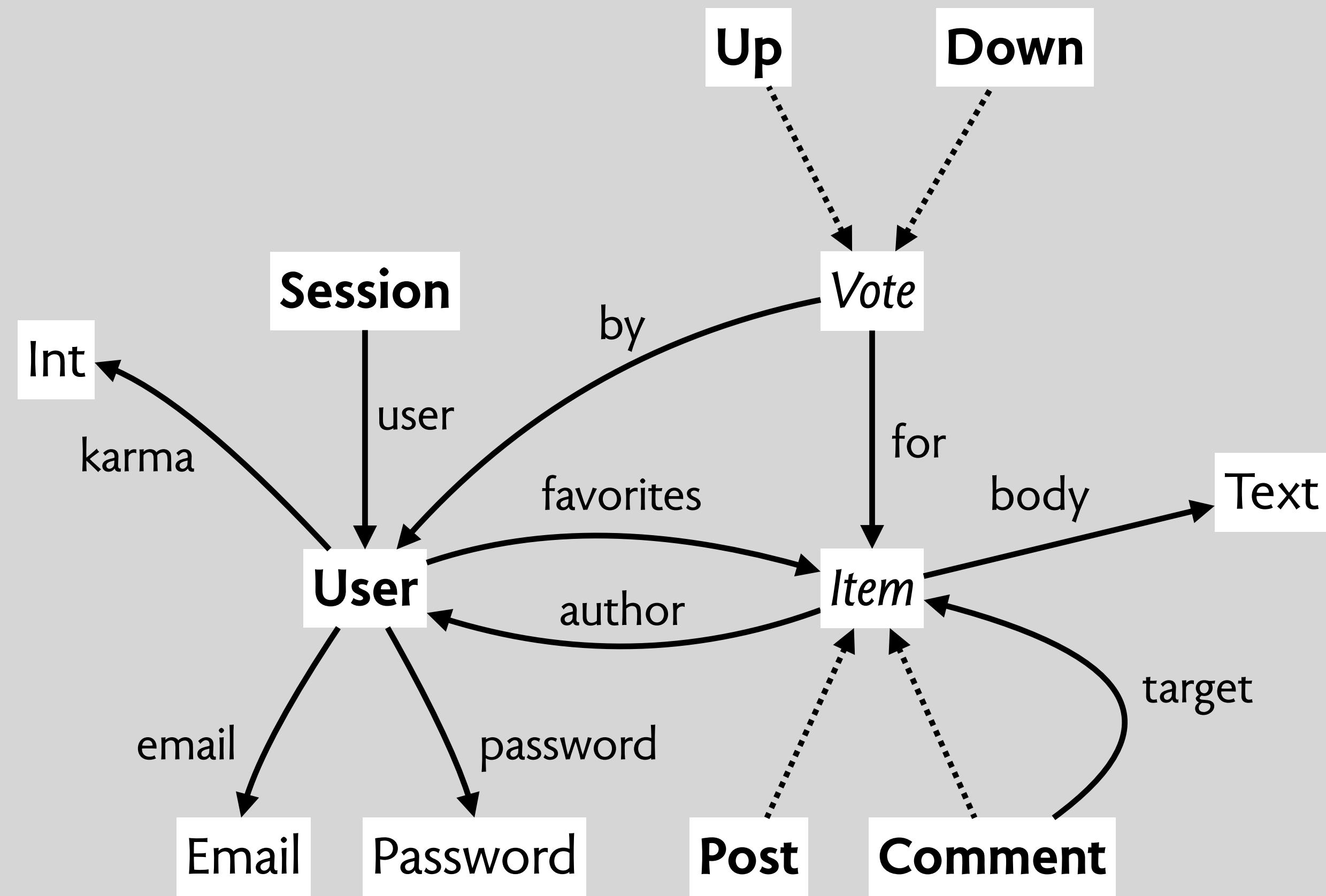
**concept never**  
call each other's actions  
read or write each other's state  
share mutable composite objects

a data model  
perspective

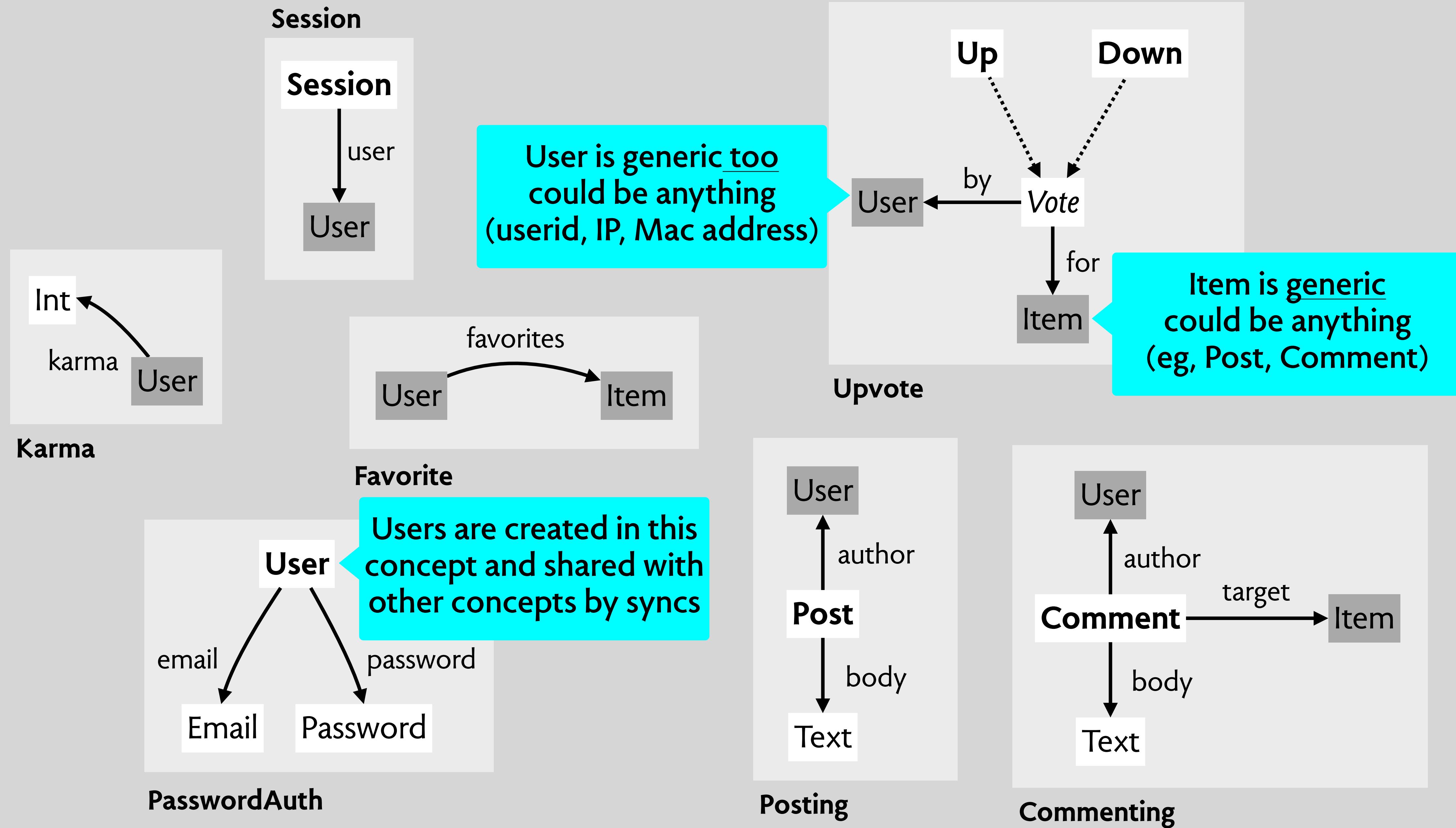
# a data model for hacker news



# highlighting the entities



# data models for concepts



# listing the generic types as parameters of the concept

this means Upvote is generic with respect to the User and Item types

## **concept** Upvote [User, Item]

**purpose** rank items by popularity

### **state**

set of votes each with a user the vote is by an item the votes is for whether Up or Down

### **actions**

upvote (u: User, i: Item)  
downvote (u: User, i: Item)  
unvote (u: User, i: Item)

## **concept** Karma [User]

**purpose** privilege good users

### **state**

set of users each with karma points (an integer)

### **actions**

reward (u: User, r: Int)

## **concept** Posting [User]

**purpose** share content

### **state**

a set of posts each with a body (text)  
an author (user)

### **actions**

create (u: User, t: Text): Post  
delete (p: Post)  
edit (p: Post, t: Text)

## check your understanding

In what key respect is data modeling in concept design different? (select all that apply)

- (a) Concept design decomposes the data model by functionality
- (b) Concept design introduces the idea of generic types from programming into data models
- (c) Concept design distinguishes entities from values

modularity example  
restaurant reservations

## concept RestaurantReservation [User]

**purpose** reducing wait time for tables

**principle** the restaurant makes slots available at various times; a diner reserves for a particular time, and then can be assured of being seated at that time

### state

a set of slots each with the start time (includes date)  
a set of reservations each with the user who made it  
the slot being reserved  
whether seated

### actions

createSlot (t: Time)

**ensures** creates a fresh slot & associates with time t

reserve (u: User, t: Time): Reservation

**requires** some slot at time t not yet reserved

**ensures** creates & returns a fresh reservation

associates it with user u and the slot

seat (r: Reservation)

**requires** r is a reservation for about now

**ensures** mark r as seated

### last time, one module

focused on one aspect: reservations

### this time, whole system

how to organize variety of functions

### main areas of function

identifying users

sending confirmations & reminders

punishing repeat no-shows

laying out tables in dining room

reserving based on party size

defining shifts with different layouts

...

# some easy-ish design issues

## **identifying users**

support standard password access & just email/phone

UserAccount concept to track users through password creation

UserPassword concept to manage password access

Capability concept to generate obscure reservation references?

## **sending confirmations & reminders**

Confirmation concept holds contact preferences, tightly sync'd

Reminder concept, because reminders are different

## **punishing repeat no-shows**

Karma concept debit action, sync'd with noShow action

## **laying out tables in dining room**

FloorPlan concept backing a nice graphical UI

# reserving based on party size

## concept RestaurantReservation [User]

### state

slots with start times  
reservations with user, slot

### actions

createSlot (start: Time)  
reserve (u: User, t: Time): Reservation  
cancel (r: Reservation)  
noShow (r: Reservation)

## concept RestaurantReservation [User, **Table**]

### state

slots with **tables** and start times  
reservations with user, slot, **party size**

### actions

createSlot (start: Time, **t: Table**)  
reserve (u: User, **s: Slot, party: int**): Reservation

## concept FloorPlan

### state

tables with  
position and min/max party sizes

### actions

configureTables (...)

**when** Web.request (*reserve*, user, time, party)  
**where**

slot for table at time (in RestaurantReservation)  
party in range for table (in FloorPlan)

**then**

RestaurantReservation.reserve (user, slot, party)

**why does reserve action now take slot?**  
because need to pick based on floor plan

**why is Table generic for RestaurantReservation?**  
because it doesn't know anything about tables

# turn control in Open Table

## Turn Controls

Hide 

Specify the minimum number of turns by party size or tables.

2 guests

-  +



4 guests

-  +

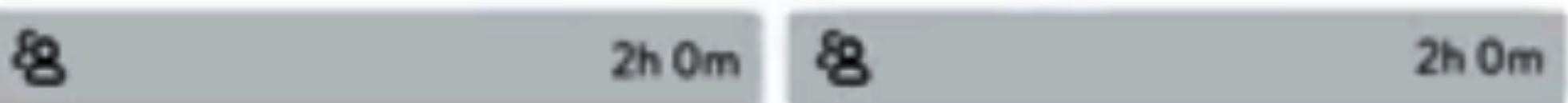


Table 32

-  +



+ Add party size

+ Add table

Select when to release turn control restrictions for the shift.

Don't release



# shifts with different layouts

**concept** RestaurantReservation [User, **Slot**]

**state**

reservations with user, slot, party size

**actions**

~~-createSlot (start: Time, t: Table)~~

reserve (u: User, s: Slot, party: int): Reservation

cancel (r: Reservation)

noShow (r: Reservation)

**concept** FloorPlan

**state**

**floor plans** with tables, tables with position and number of seats

**actions**

configureTables (...): FloorPlan

**concept** Shift [FloorPlan, Table]

**state**

shifts with times, floor plan, slots  
slots with times, min/max party, table

**actions**

setupShift (...)

...

**when** Web.request (*reserve*, user, time, party)

**where**

slot for table at time with party (in Shift)

**then**

    RestaurantReservation.reserve (user, slot, party)

**why does Shift now manage slots?**

because of shift-specific functions

(eg, “turn time by party size”)

# what concept design is and isn't



**not a magic potion**  
helps control complexity  
not eliminate completely

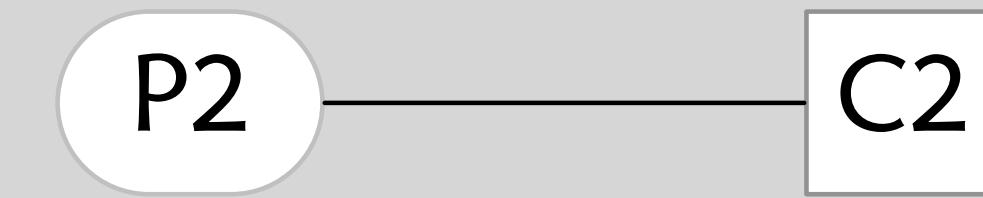
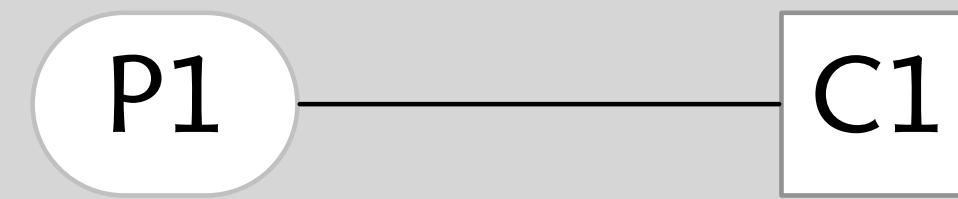


**a framework/language**  
for structuring designs  
exploring collaboratively

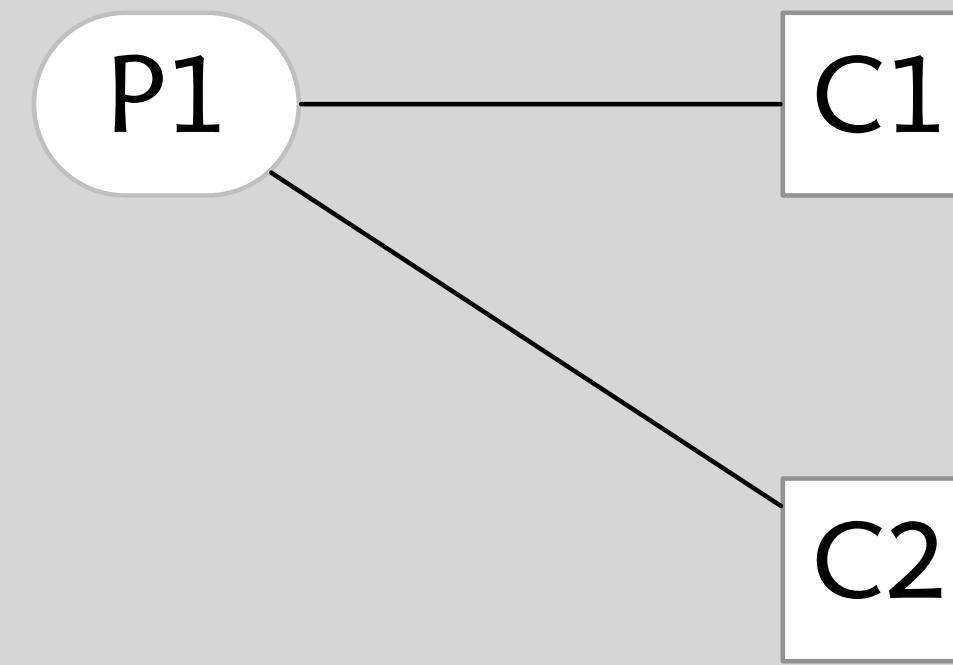
purposes  
& conflation

# a concept design principle

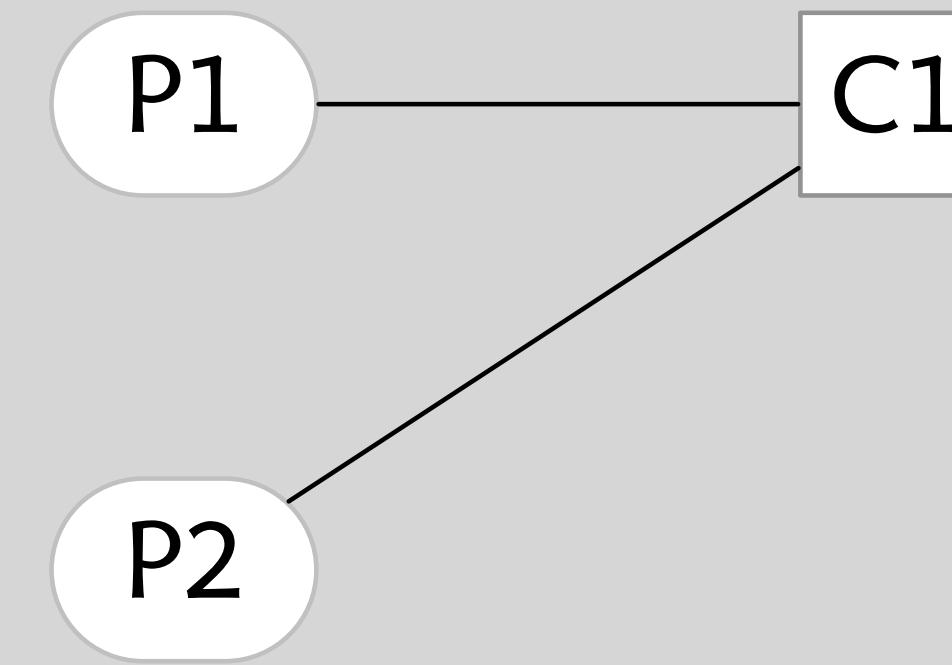
**specificity**  
purposes:concepts are 1:1



**redundancy**  
>1 concept per purpose



**overloading**  
>1 purpose per concept



# overloading leads to conflation



*purpose 1*



*purpose 2*



*conflated:  
two purposes*

**concept** UserAccount

**purpose** ????

**state**

a set of users each with  
**a username**  
**a password**  
**an email address**  
**a phone number**  
**first and last names**  
**profile picture**

**concept** Password

**purpose** authenticate users

**state**

a set of users each with  
a username  
a password

**concept** Notification [User]

**purpose** notify users

**state**

a set of users each with  
an email address  
a phone number

**concept** Profile [User]

**purpose** share user info

**state**

a set of users each with  
first and last names  
profile picture

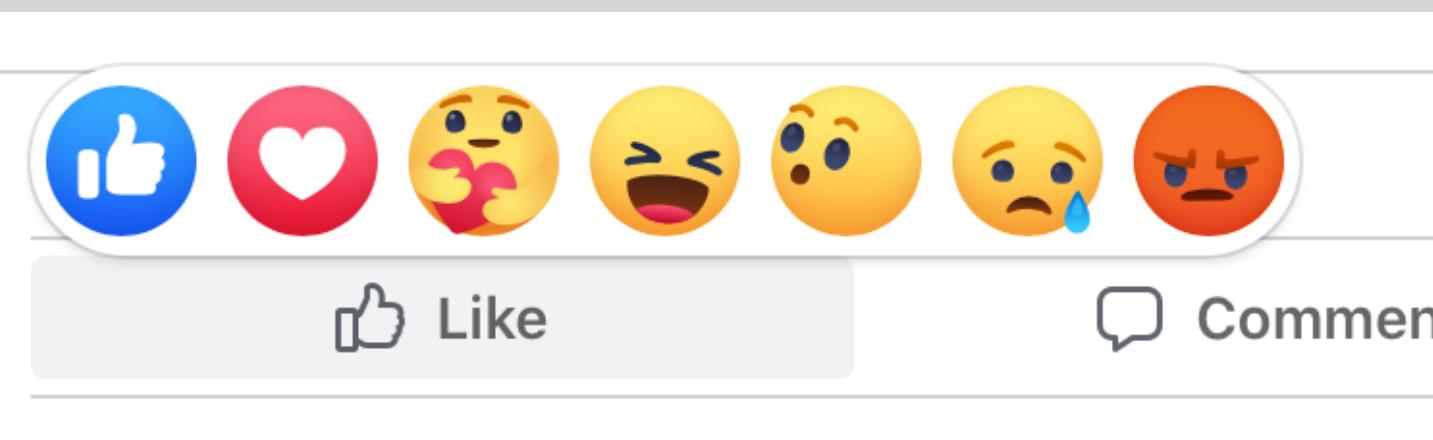
# overloading examples from my book

- US Letter (Manual - Front)
- ✓ US Letter
- US Letter (Manual - Roll)
- US Letter (Sheet Feeder - Borderless)
- US Letter (Manual - Roll (Borderless))

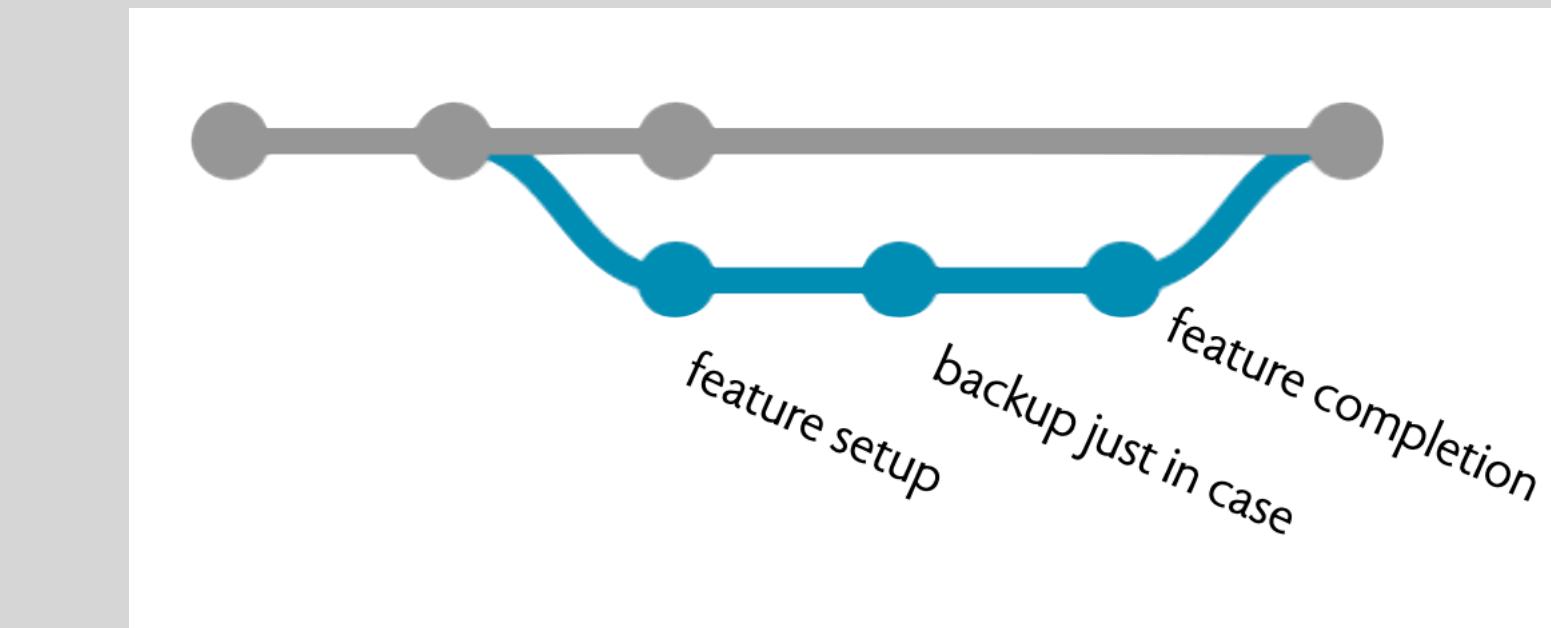
Epson's PaperSize concept



Fujifilm's ImageSize concept



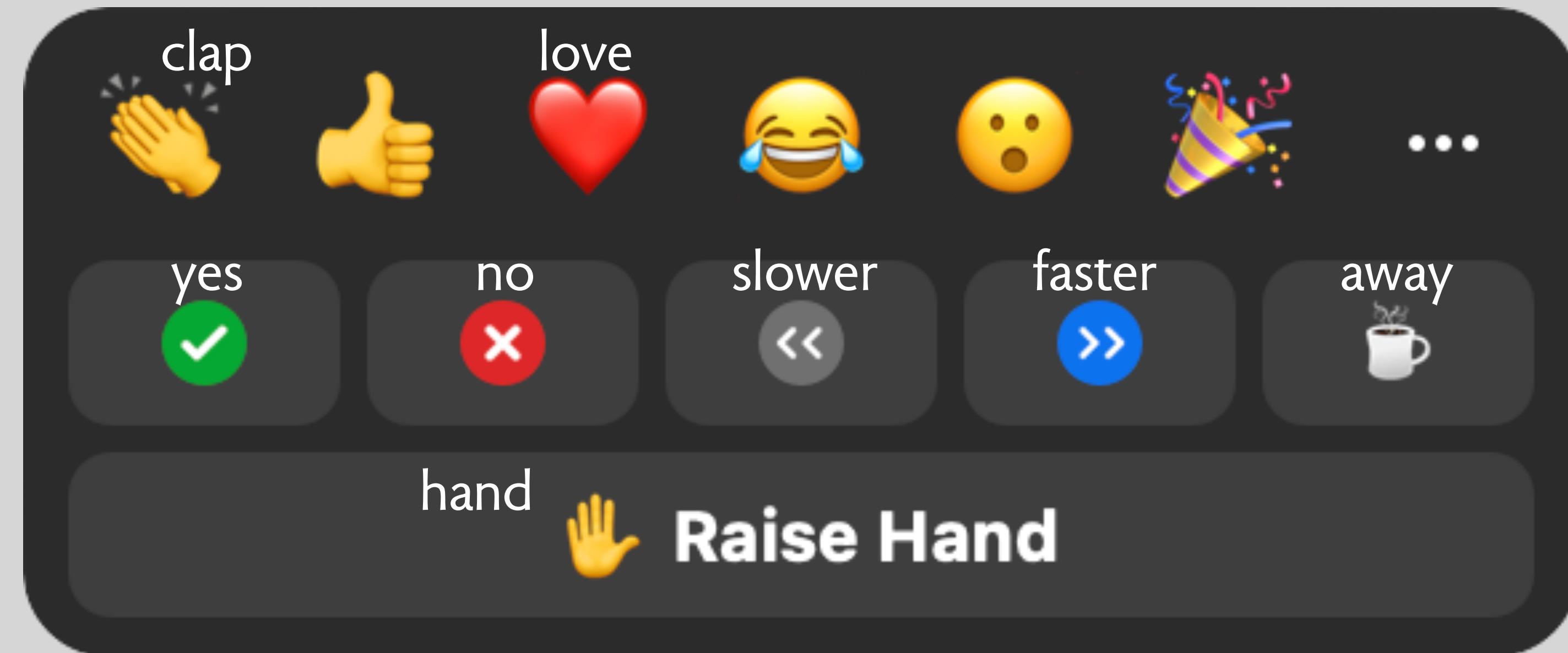
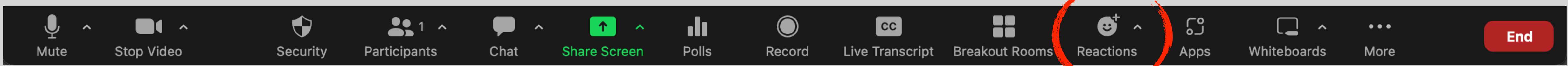
Facebook's Reaction concept



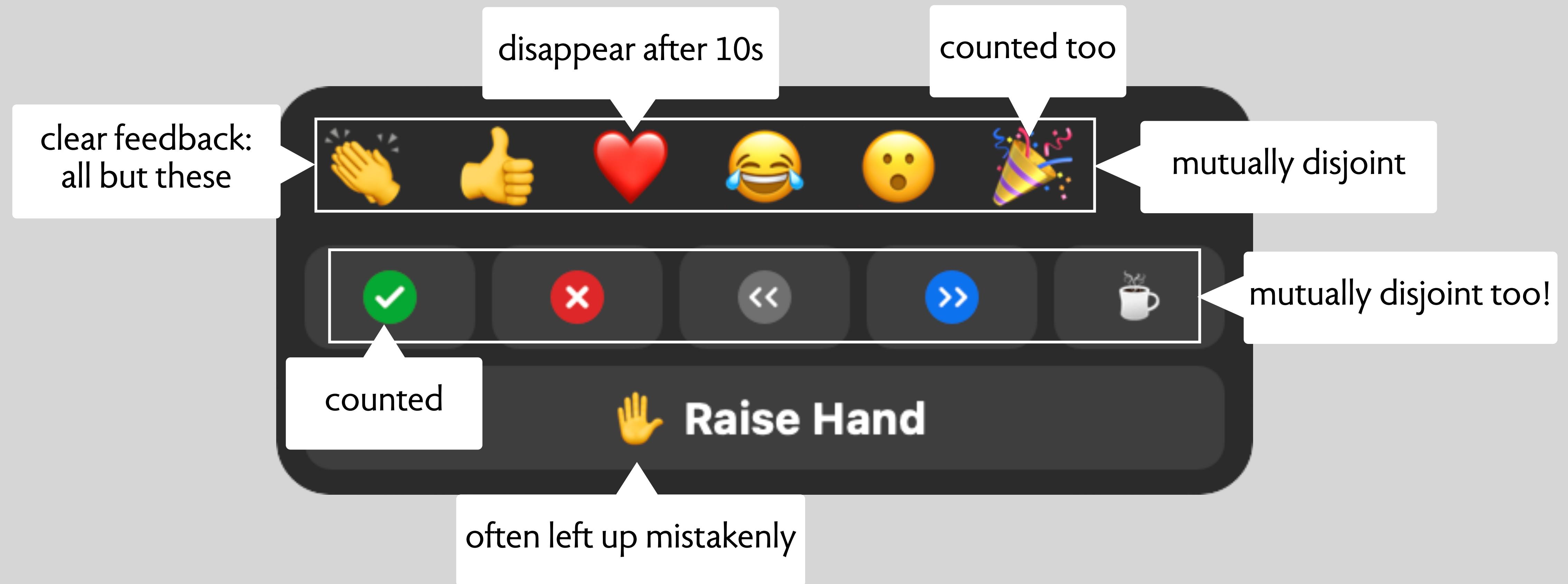
Git's Commit concept

conflation example:  
reactions in Zoom

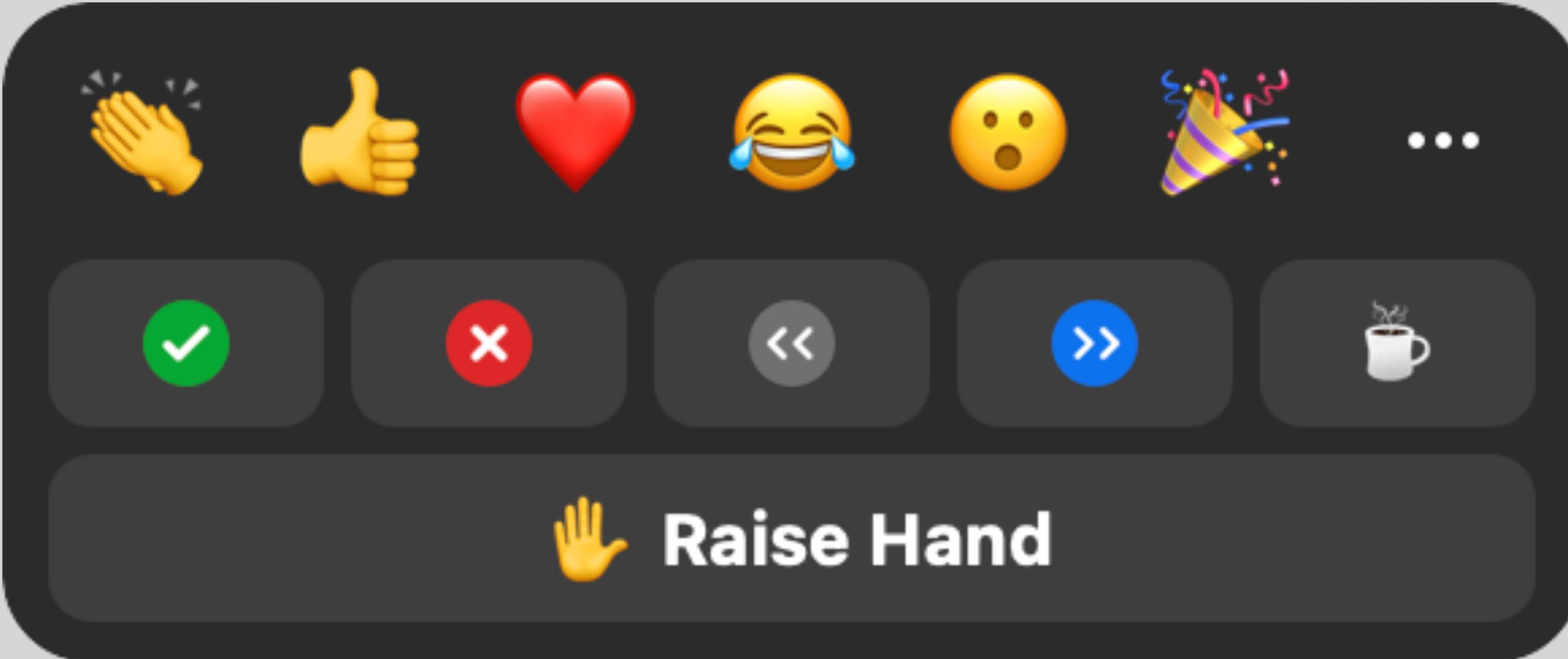
# Zoom's reactions



# anomalous behaviors



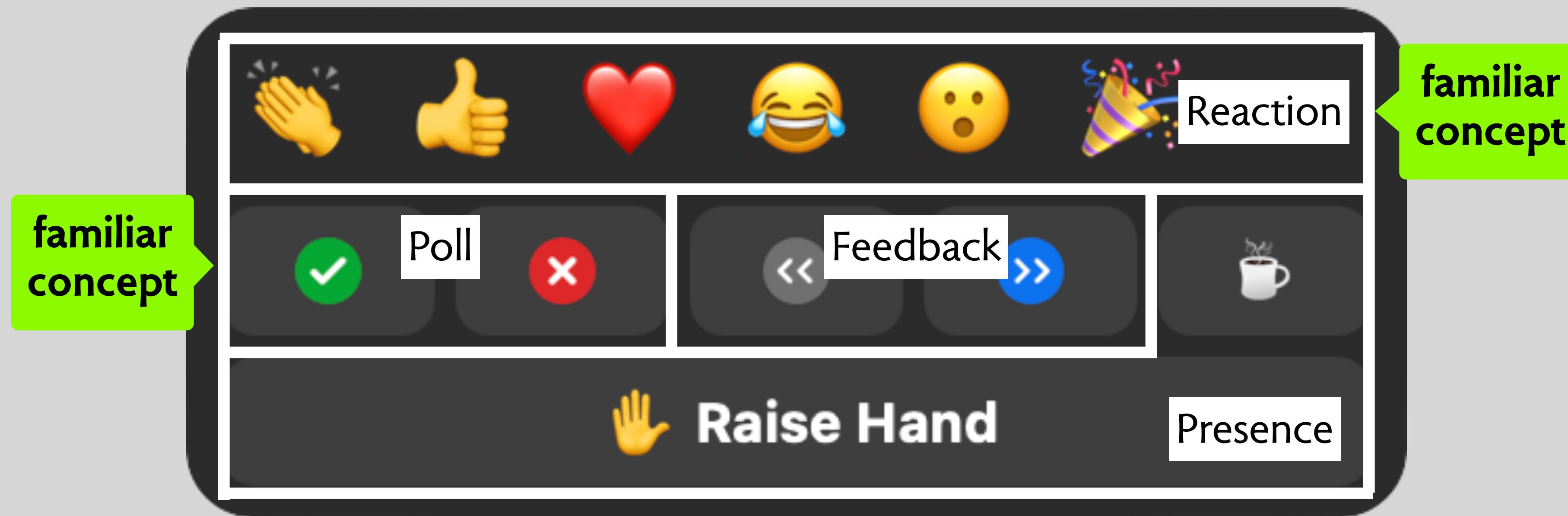
# can we do better?

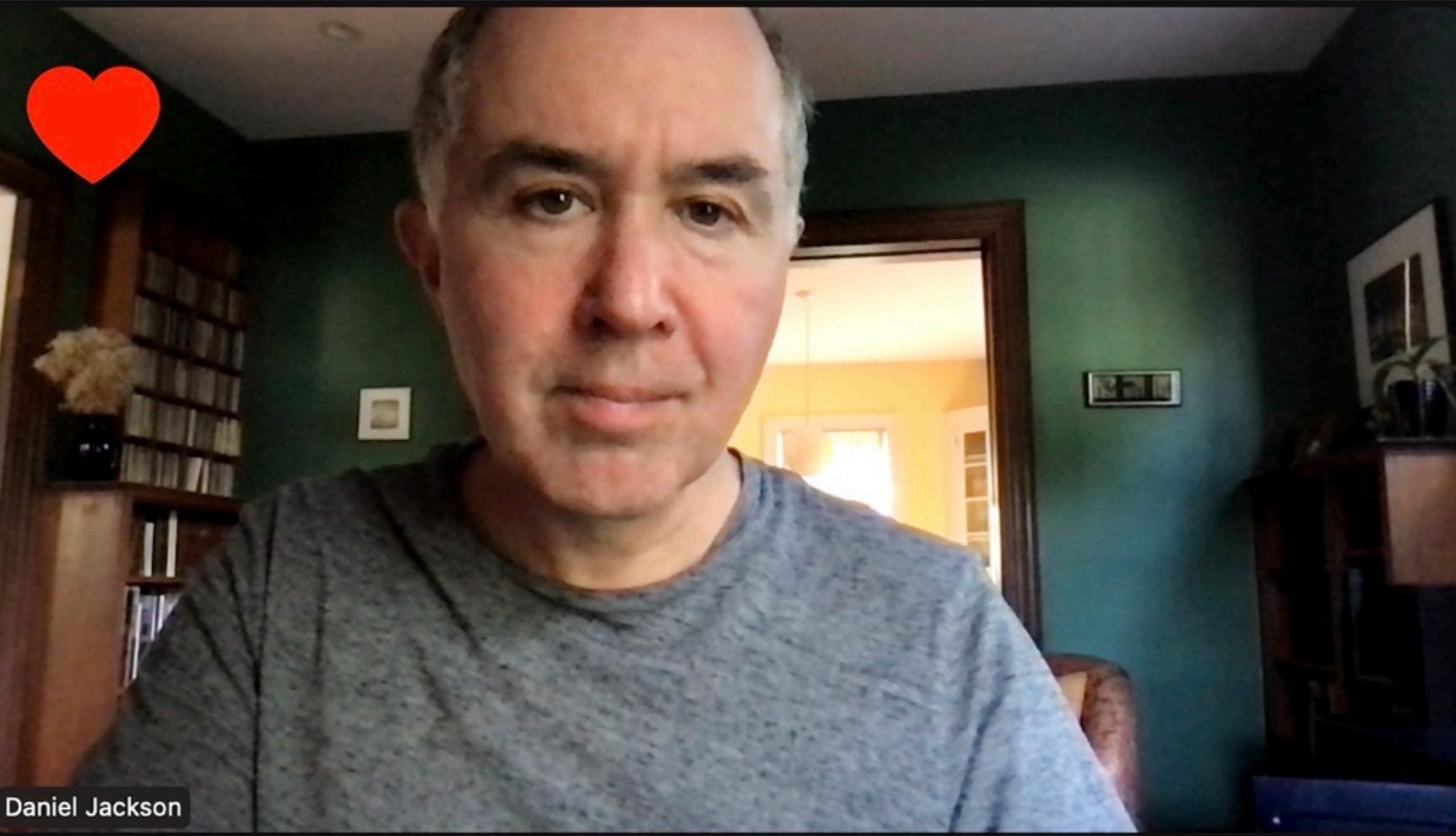


## goals

- break the behavior into a small set of concepts
- use familiar concepts whenever possible
- make each concept simple, robust & understandable
- leave some flexibility to synchronizations

# my take: splitting into coherent concepts





## Presence

## Chat

## Reaction

## Feedback

Request to speak      Watching/listening

Speaking

I'm away

Audience

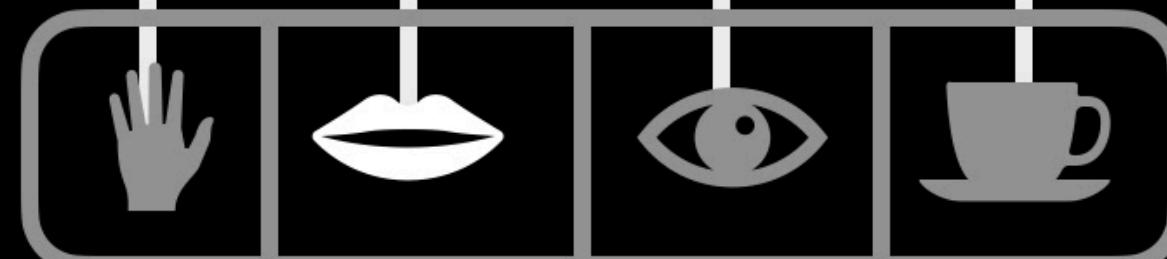
Other emoji

Recent emoji

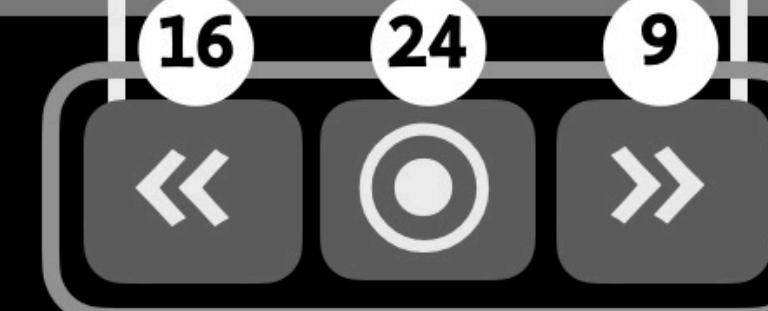
Just right

Slow down

Speed up



Everyone ▾      Type here...

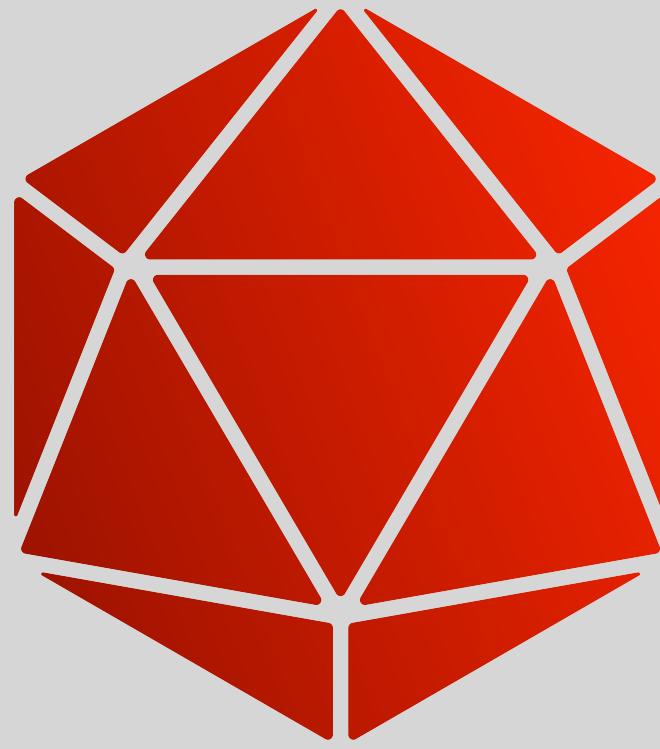


takeaways

# 3 key aspects of concept design



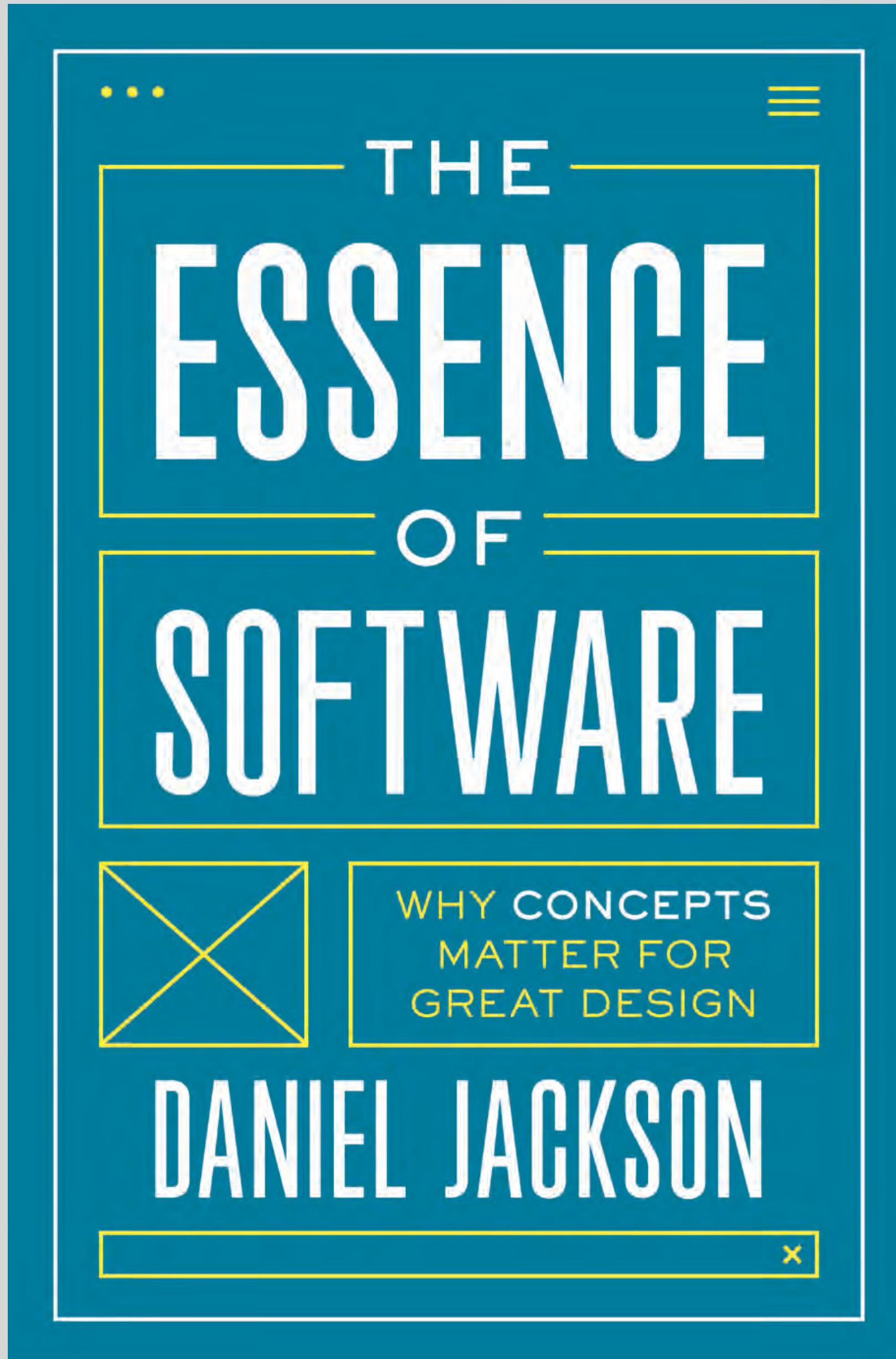
**purpose**  
asking why



**abstraction**  
focus on behavior



**separation**  
independent parts



/tutorials

20 tutorials on concept design

/studies

case studies

/subscribe

sign up to stay in touch

/ask

concept design forum

a code-level  
explanation

# let's look at an example: hacker news

**Hacker News** new | past | comments | ask | show | jobs | submit login

▲ Jackson structured programming (wikipedia.org) Post

106 points by haakonhr 63 days ago | hide | past | favorite | 69 comments

**Upvote**

▲ danielnicholas 63 days ago [-]

user: danielnicholas

created: 63 days ago

karma: 11

You might find helpful an annotated version [0] of Hoare's explanation of JSP that I edited for a Michael Jackson festschrift

; I'd point to these ideas as worth knowing:

ing problem that involves traversing structures can be solved very systematically. HTDP addresses this class, but bases its structure only on input structure; JSP synthesized it.

- The **Karma**ne archetypal problems that, however you code, can't be pushed under the rug—most notably structure clashes—and just recognizing them

- Coroutines (or code transformation) let you structure code more cleanly when you need to read or write more than one structure. It's why real iterators (with yield), which offer a limited form of this, are (in my view) better than Java-style iterators with a next method.

- The idea of viewing a system as a collection of asynchronous processes (Ch. 11 in the JSP book, which later became JSD) with a long-running process for each real-world entity. This was a notable contrast to OOP, and led to a strategy (seeing a resurgence with event storming for DDD) that began with events rather than objects.

[0] <https://groups.csail.mit.edu/sdg/pubs/2009/hoare-jsp-3-29-09...>

▲ ob-nix 63 days ago [-]

... this brings back memories! In the late eighties I, as a teenager, found a Jackson Struct. Pr. book at the town library. I remember I was amazed at the text and wondered why I hadn't heard about the method before.

If I remember correctly did the book clearly point out backtracking as a standard method, while mentioning that most languages lacked that, so it had to be implemented manually.

**Session**

# let's build it!

```
class User {  
    String name;  
    String password;  
    User register (n, p) { ... }  
    User authenticate (n, p) { ... }  
}
```

```
class Post {  
    User author;  
    String body;  
    Post new (a, b) { ... }  
}
```

# adding upvoting

```
class User {  
    String name;  
    String password;  
    User register (n, p) { ... }  
    User authenticate (n, p) { ... }  
}
```

```
class Post {  
    User author;  
    String body;  
Set [User] ups, downs;  
    Post new (a, b) { ... }  
upvote (u) { ... }  
downvote (u) { ... }  
}
```

# adding karma

```
class User {  
    String name;  
    String password;  
int karma;  
    User register (n, p) { ... }  
    User authenticate (n, p) { ... }  
incKarma (i) { ... }  
bool hasKarma (i) { ... }  
}
```

```
class Post {  
    User author;  
    String body;  
    Set [User] ups, downs;  
    Post new (a, b) { ... }  
    upvote (u) { ... }  
downvote (u) {  
        if u.hasKarma (10) ... }  
    }
```

# adding commenting

```
class User {  
    String name;  
    String password;  
    int karma;  
    User register (n, p) { ... }  
    User authenticate (n, p) { ... }  
    incKarma (i) { ... }  
    bool hasKarma (i) { ... }  
}
```

```
class Post {  
    User author;  
    String body;  
    Set [User] ups, downs;  
Seq [Post] comments;  
    Post new (a, b) { ... }  
    upvote (u) { ... }  
    downvote (u) {  
        if u.hasKarma (10) ... }  
addComment (c) { ... }  
}
```

# what's wrong with this code?

```
class User {  
    String name;  
    String password;  
    int karma;  
  
    User register (n, p) { ... }  
    User authenticate (n, p) { ... }  
    incKarma (i) { ... }  
    bool hasKarma (i) { ... }  
}
```

```
class Post {  
    User author;  
    String body;  
    Set [User] ups, downs;  
    Seq [Post] comments;  
    Post new (a, b) { ... }  
    upvote (u) { ... }  
    downvote (u) {  
        if u.hasKarma (10) ... }  
    addComment (c) { ... }  
}
```

User authentication

Posting

Upvoting

Commenting

Karma

## lack of separation

Post class contains posting,  
commenting, upvoting, karma

## dependence

Post class calls *User* class  
to get karma points

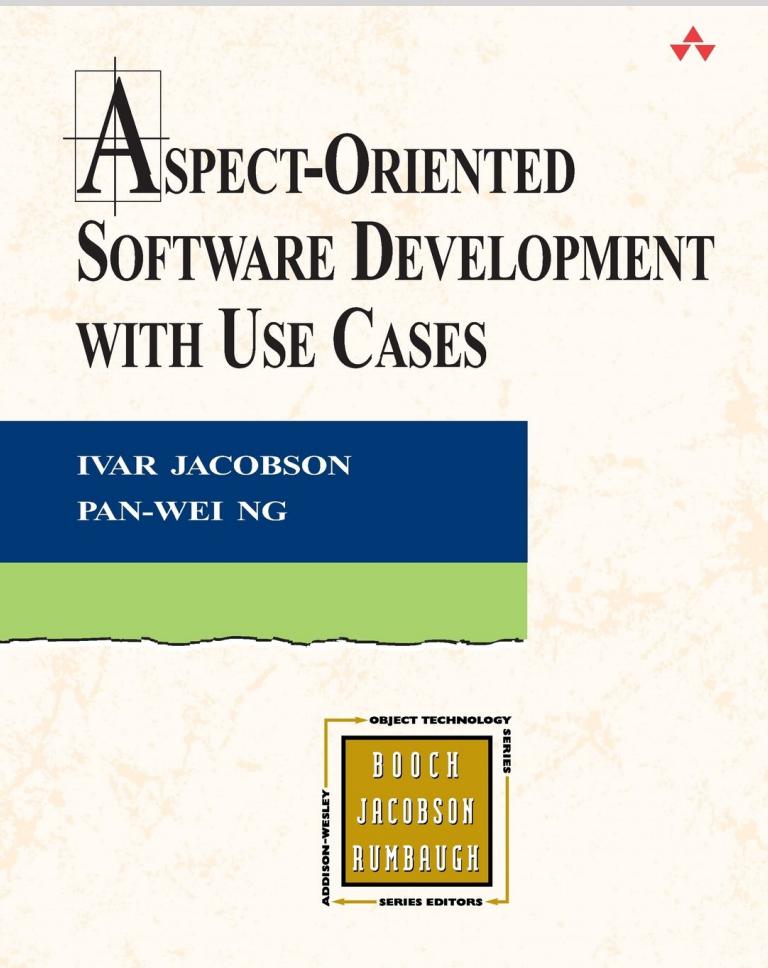
## classes are not reusable

Post class won't work in an app  
that doesn't have karma points

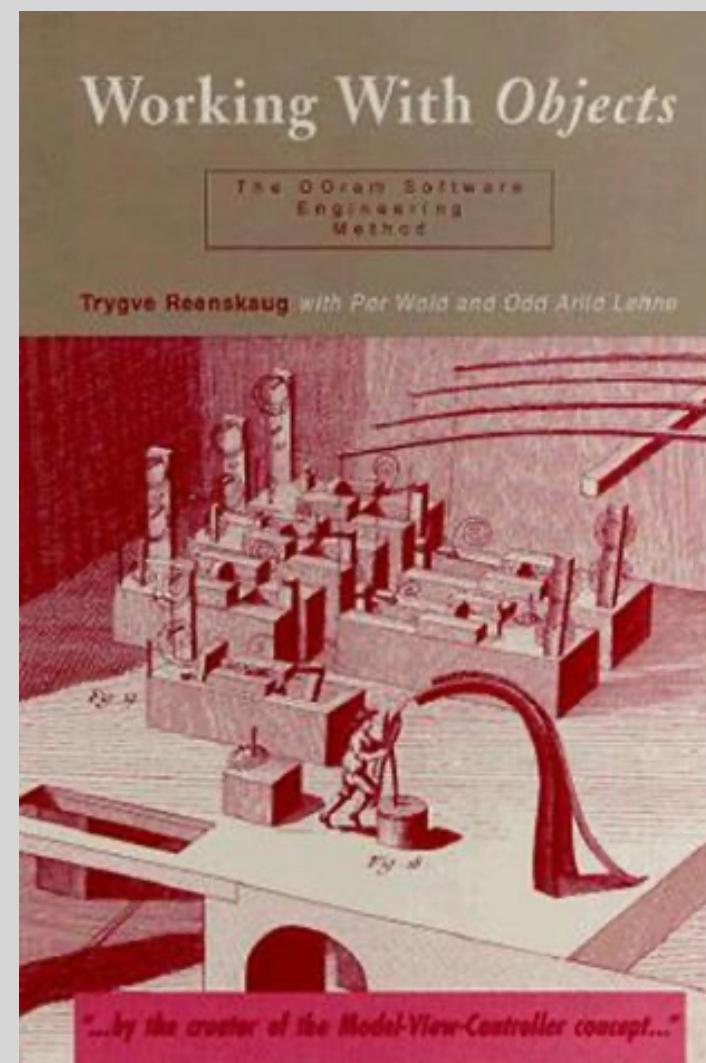
## can't be built independently

to build Post class, need *User* class  
to have been built already

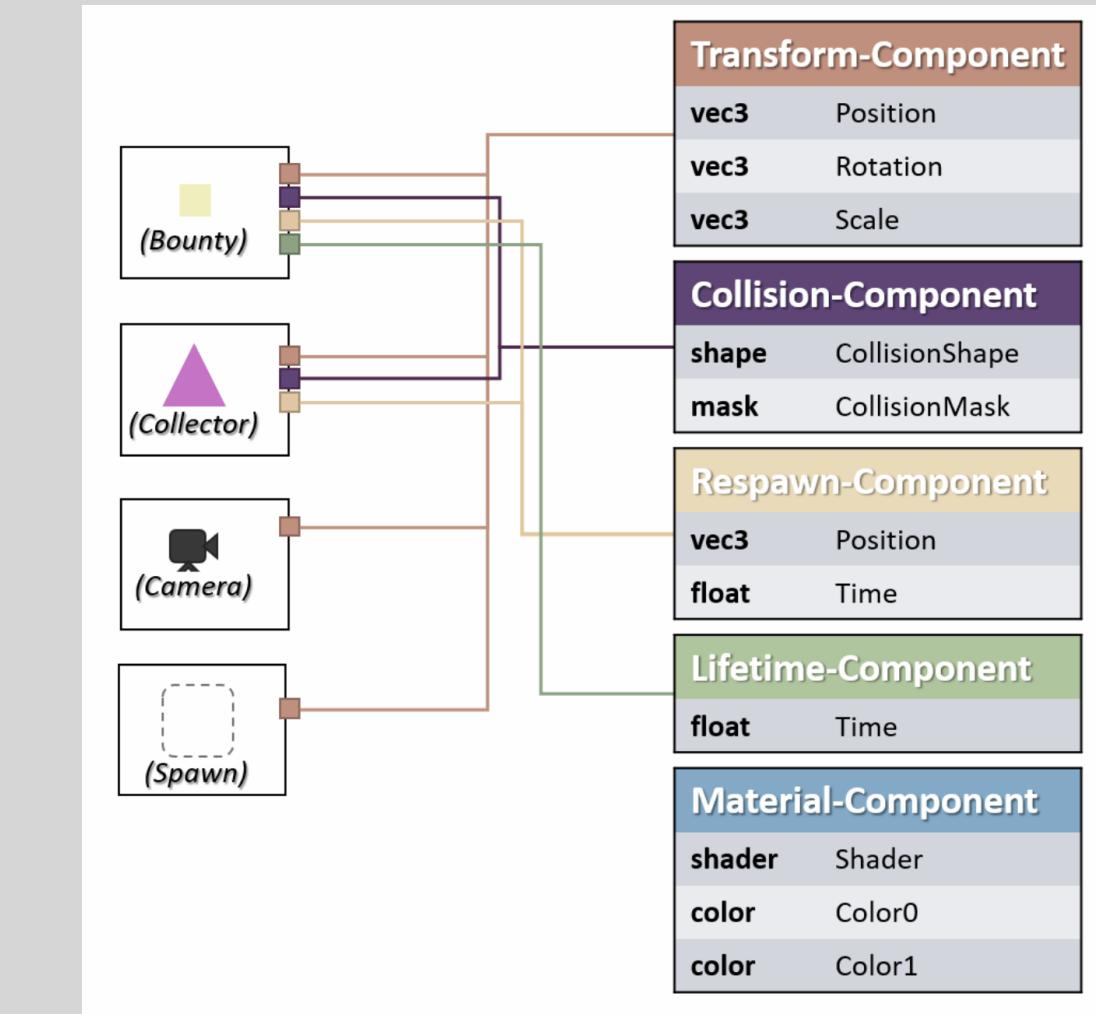
# a long history of fixes for OOP's conflation



Aspect-oriented programming  
Kiczales et al (1997)



Role-oriented programming  
Reenskaug et al (1983)



Entity-component system  
Scott Bilas et al (2002)

# concepts: modularizing user-facing functions

```
concept User {  
    Map [User, String] name;  
    Map [User, String] password;  
    User register (n, p) { ... }  
    User authenticate (n, p) { ... }  
}
```

```
concept Karma [U] {  
    Map [U, Int] karma;  
    incKarma (u, i) { ... }  
    hasKarma (u, i) { ... }  
}
```

↑  
concerns  
now cleanly  
separated

coupling is  
gone: refs are  
polymorphic

```
concept Post [U] {  
    Map [Post, U] author;  
    Map [Post, URL] url;  
    Post new (a, u) { ... }  
}
```

```
concept Upvote [U, I] {  
    Map [U, I] ups, downs;  
    upvote (u, i) { ... }  
    downvote (u, i) { ... }  
}
```

```
concept Comment [U, T] {  
    Map [Comment, U] author;  
    Map [Comment, T] target;  
    Map [Comment, String] body;  
    Comment new (a, t, b) { ... }  
}
```

syncs hold  
cross concept  
functionality

when  
Web.request (downvote, u, i)  
where  
Karma.hasKarma (u, 20)  
then  
Upvote.downvote (u, i)

# a new architectural style

