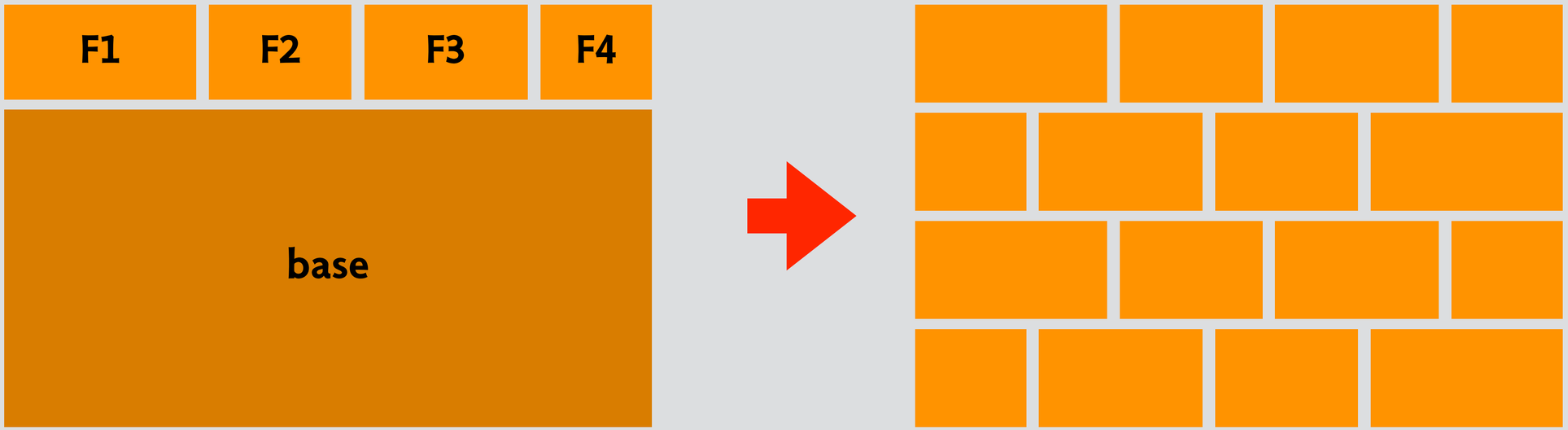
6.1040 · software studio · fall 2022

designing concept state

Daniel Jackson & Arvind Satyanarayan

concept boundaries (review)

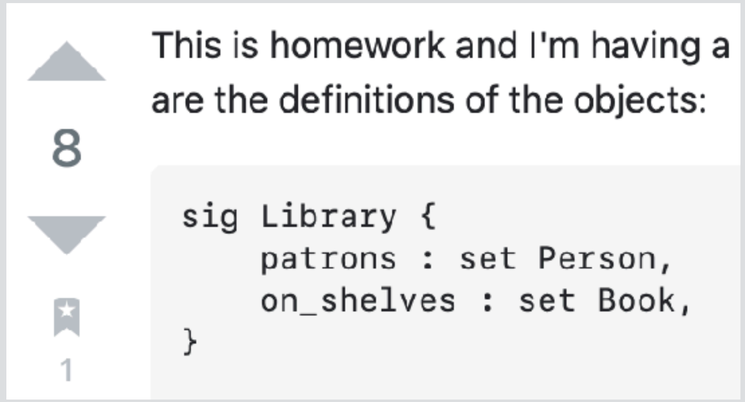
from features to concepts



purposes define reusable functionality

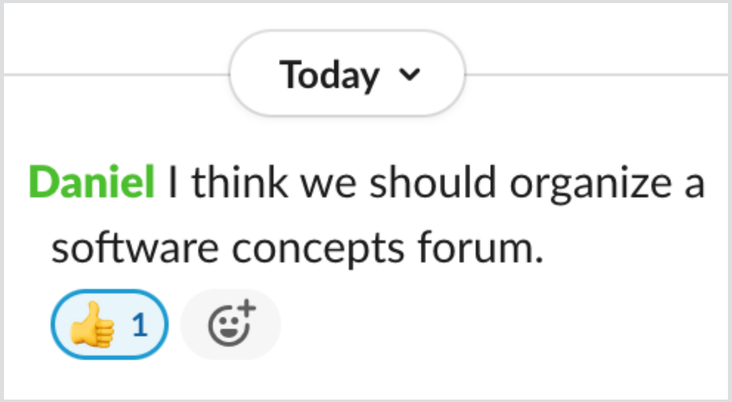
concept Upvote

purpose rank items by popularity



concept Reaction

purpose send reactions to author



concept Recommendation

purpose use prior likes to recommend

actions & local state define concept behavior completely

concept Upvote

purpose rank items by popularity

principle after series of upvotes of items, the items are ranked by their number of upvotes

state
votes for each item, up or down
which user issued each vote
ranked sequence of items

actions
user upvotes an item
user downvotes an item
user unvotes an item

concept Reaction

purpose send reactions to author

principle when user selects reaction, it's shown to the author (often in aggregated form)

state
reactions for each item
which user issued each reaction

actions
user changes reaction to item

concept Recommendation

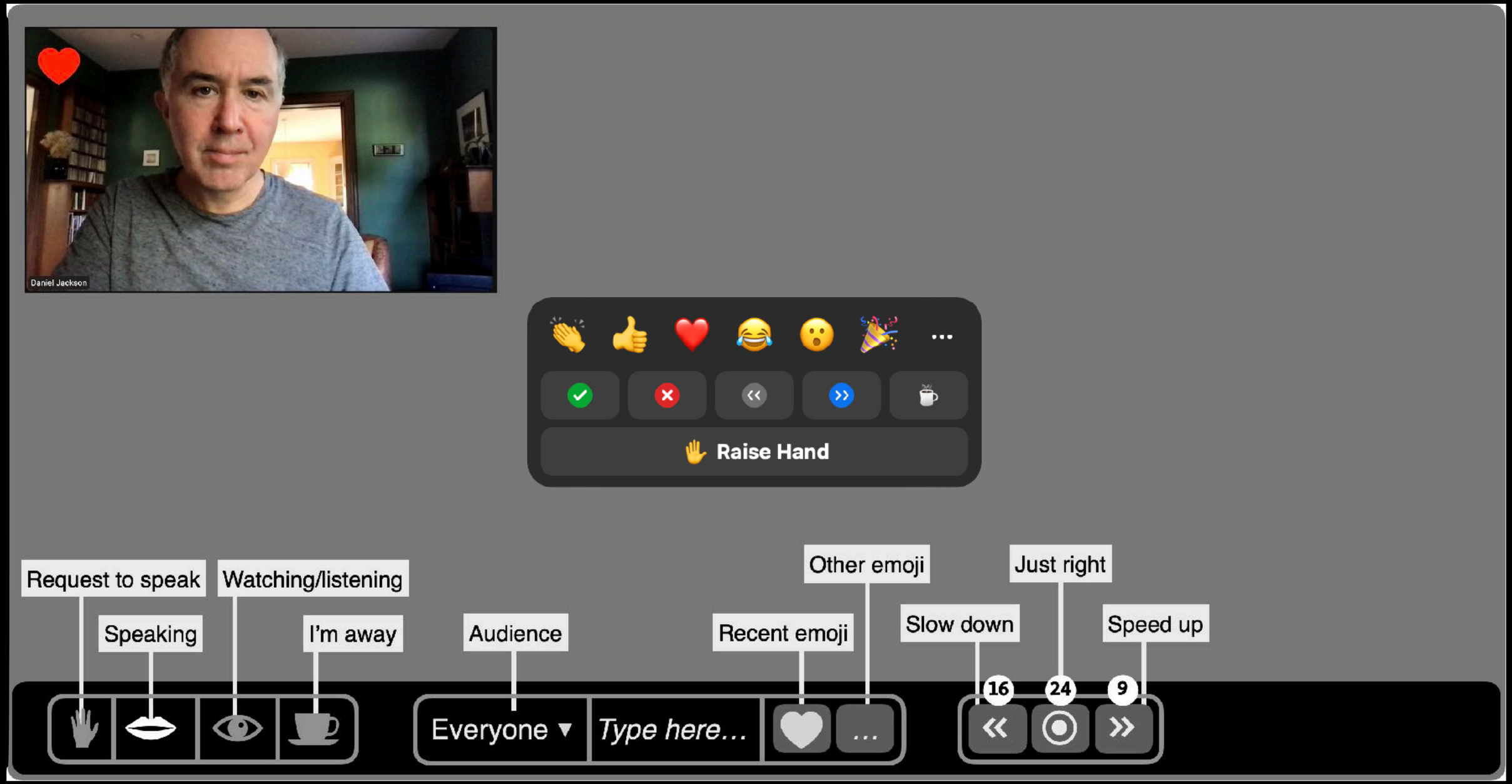
purpose use prior likes to recommend

principle user's likes lead to ranking of kinds of items, determining which items are recommended

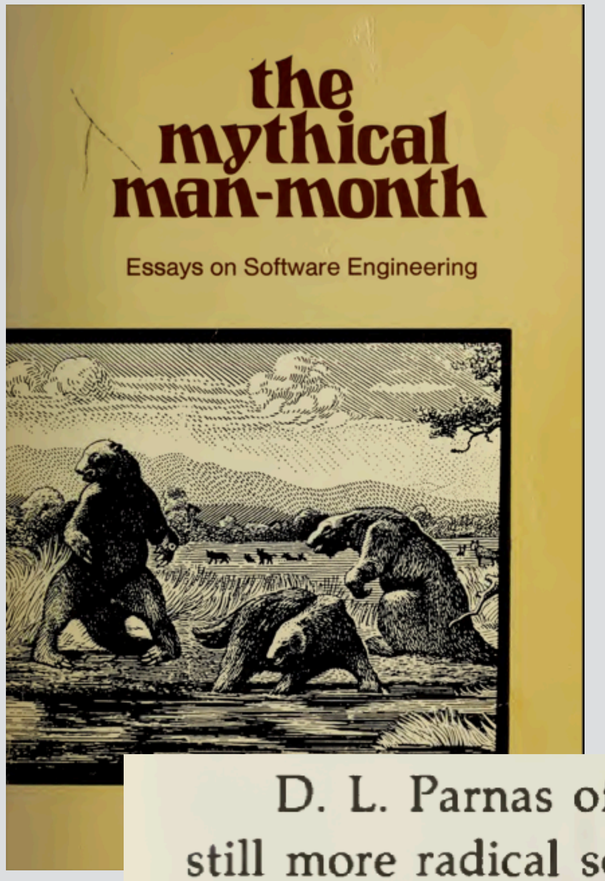
state
likes/dislikes by user for items
classification of items into kinds
recommendations for each user

actions
user likes an item
user dislikes an item

non-uniformities & distinct purposes suggest concepts should be smaller



modularity is essential and hard



David Parnas was right, and I was wrong about information hiding. I am now convinced that information hiding, today often embodied in object programming, is the only way of raising the level of software design.

Fred Brooks, Anniversary edition of MMM, 1995

D. L. Parnas of Carnegie-Mellon University has proposed a still more radical solution.[1] His thesis is that the programmer is most effective if shielded from, rather than exposed to the details of construction of system parts other than his own. This presupposes that all interfaces are completely and precisely defined. While that is definitely sound design, dependence upon its perfect accomplishment is a recipe for disaster. A good information system both exposes interface errors and stimulates their correction.

learning design modularity in stages

A2: Divergent Design
the idea of breaking design ideas into separable concepts: purpose & OP to focus on independence

A3: Convergent Design
the idea of breaking design ideas into separable concepts: state and actions to focus on encapsulation

A4: Refinement Design
revisiting concepts to clarify boundaries and interactions: principles to focus on modularity weaknesses etc

today's plan

moving from design to code
making the state precise
also important for clarifying design

big ideas
relational view of state & rep independence
state invariants, preserved by action
joining concepts = instantiating generics

a relational view of states

revisiting a simple concept

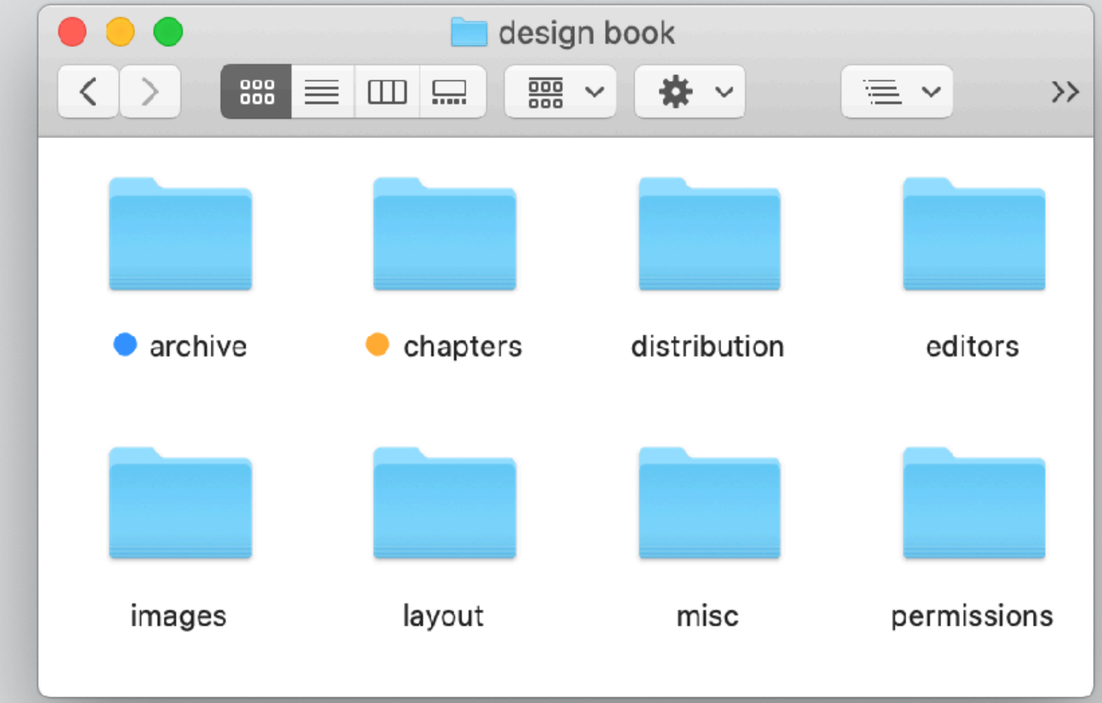
concept Label

purpose organize items

principle if label is added to an item and not removed, then filtering on that label will display that item

state
labels for each item

actions
add label to an item
remove label from an item
filter on a set of labels



let's focus on behavior: state and actions

concept Label

state
labels for each item

actions
add label to an item
remove label from an item
filter on a set of labels

formalizing state

a type variable

concept Label [Item]

concept is “polymorphic” or “generic”

state
labels: Item -> **set** Label

maps each item to zero or more labels

actions
add label to an item
remove label from an item
filter on a set of labels

formalizing actions

concept Label [Item]

state
labels: Item -> **set** Label

actions
add (l: Label, i: Item)
  l.labels += i

remove (l: Label, i: Item)
  l.labels -= i

filter (ls: **set** Label): **set** Item
  return {i: Item | ls **in** i.labels}

add i to the set of labels associated with l

a relational view of state

concept Label [Item]

state
labels: Item -> **set** Label

actions
add (l: Label, i: Item)
 i.labels += l

remove (l: Label, i: Item)
 i.labels -= l

filter (ls: **set** Label): **set** Item
 return {i: Item | ls **in** i.labels}

filter (ls: **set** Label): **set** Item
 return ls. **~labels**

what does this do?

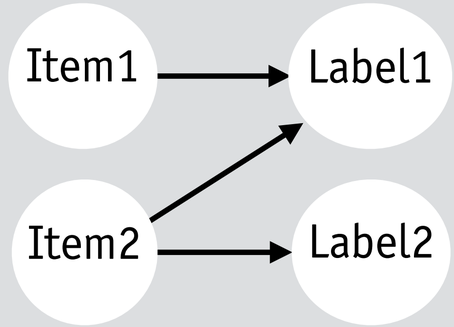
structuring state with objects
familiar from OO languages like Java
 labels: Map [Item, Set[Label]]

our approach: just a relation
mathematically, a set of tuples
show as a table (or graph, if pairs)

relational view is more abstract
no collection objects; flatter & simpler

relational view is more flexible
s.labels is set of labels for all items in set s
 {Item1, Item2}.labels = {Label1, Label2}
~labels is labels transposed (backwards)
 {Label1}.~labels = {Item1, Item2}

Item1	Label1
Item2	Label1
Item2	Label2



representation independence

state
labels: Item -> **set** Label

in standard backend architectures, have to transform state between reps, hence "object relational mappers"

fewer rep options in databases because with query optimization and indexes, choice of rep doesn't determine traversal

how might you represent this state in code?
many ways, depending on patterns of access, performance needs, ...

in an object-oriented language (eg, Java)
hash table mapping items to linked lists of labels
linked list of labels as a property of each item object
tree with path (label, label, ...) leading to collection of item pointers

in a relational database (eg, Oracle)
item/label table with indexes on items, or on labels, or both

in a document database (eg, MongoDB)
collection of items with lists of labels

in an XML file
etc...

mini summary

making state precise
helps work out details of concept design
eases transition to implementation

rep independence
talk about abstract state without getting mired in rep details
even RDB not abstract enough (no union or subtypes, eg)

relational view
what's associated with what
the simplest way to think about state?

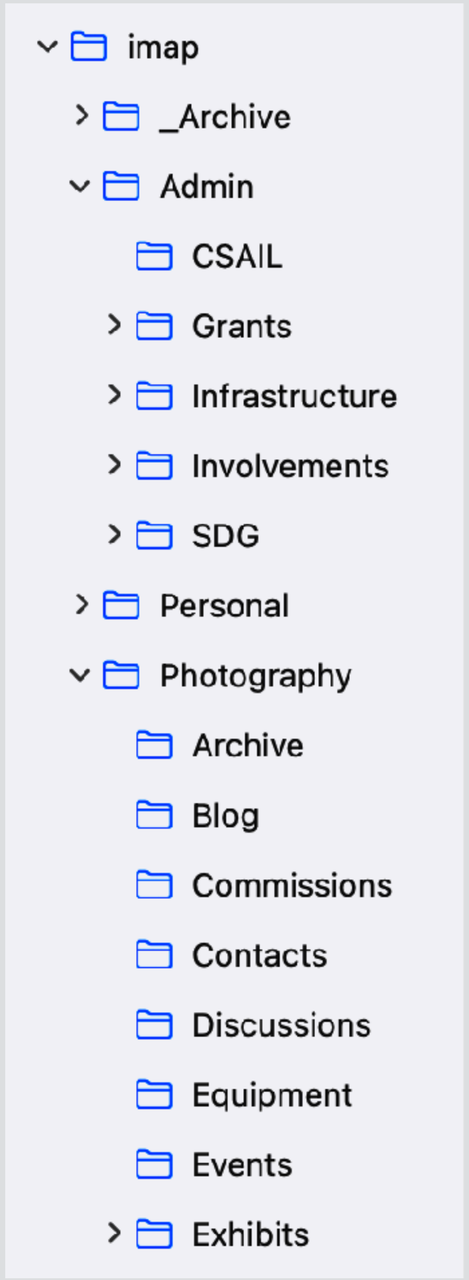
designing
richer states

a simple folder concept

concept Folder

state
a fixed root folder
for each folder, the objects
(folders or items) it contains
a name for each folder

actions
create a folder
delete a folder
move an object to a folder
rename a folder

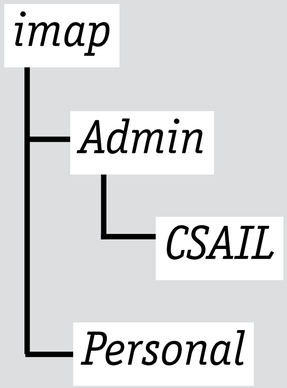


formalizing the state

concept Folder [Item]

state
root: Folder
contents: Folder -> **set** (Item + Folder)
name: Folder -> **one** Name

union



can you draw the folder hierarchy for this state?

contents

Folder0	Folder1
Folder0	Folder2
Folder1	Folder3

name

Folder0	*imap*
Folder1	*Admin*
Folder2	*Personal*
Folder3	*CSAIL*

thinking about actions

concept Folder [Item]

state
root: Folder
contents: Folder -> **set** (Item + Folder)
name: Folder -> **one** Name

actions
rename (f: Folder, n: Name)
delete (f: Folder)
move (o: Folder + Item, to: Folder)

when is each action allowed?

how does each action affect state?

```
rename (f: Folder, n: Name)
 when n not in f.~contents.name
 f.name := n
```

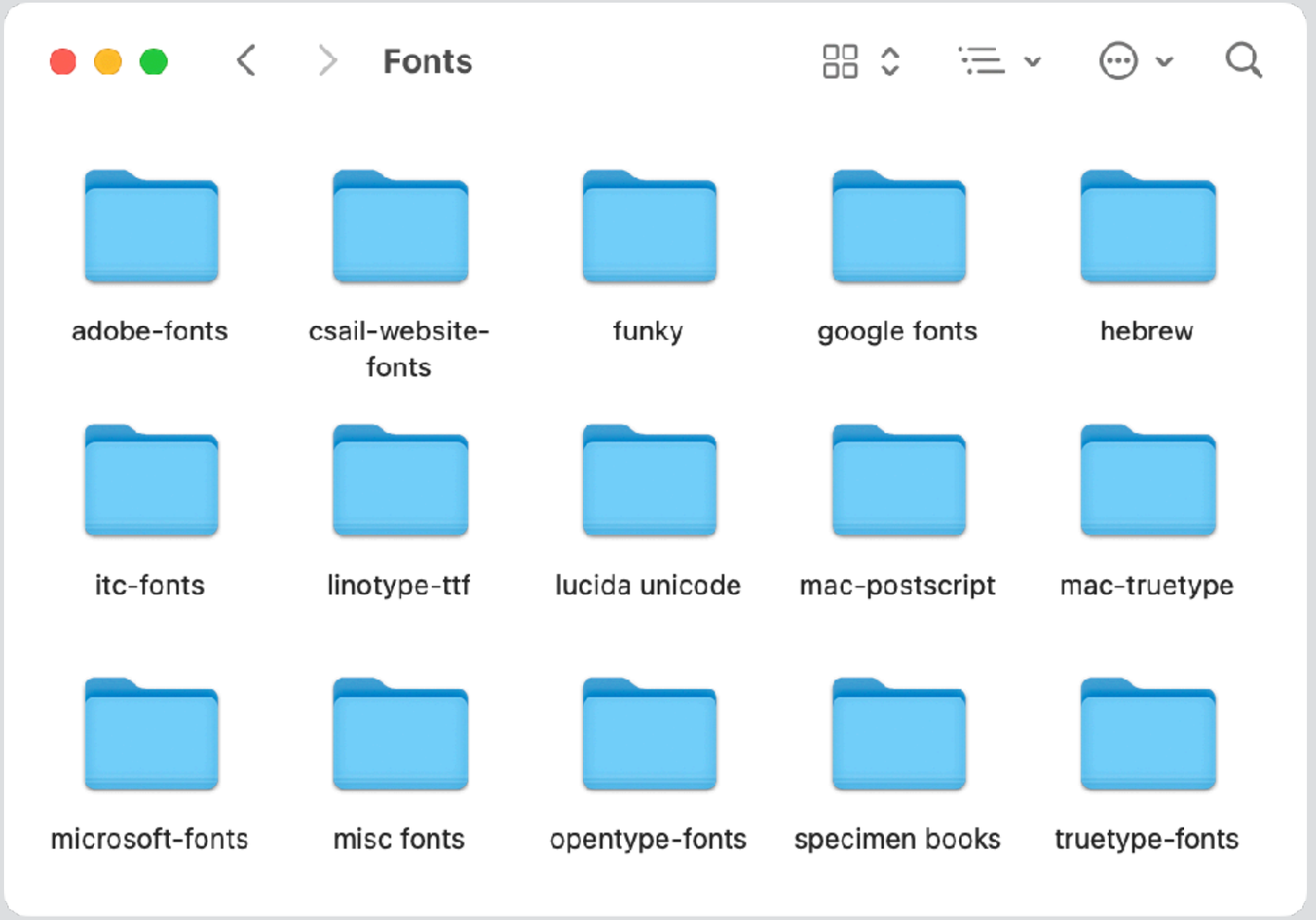
```
delete (f: Folder)
 when f != root
 contents -= Folder -> f.*contents
```

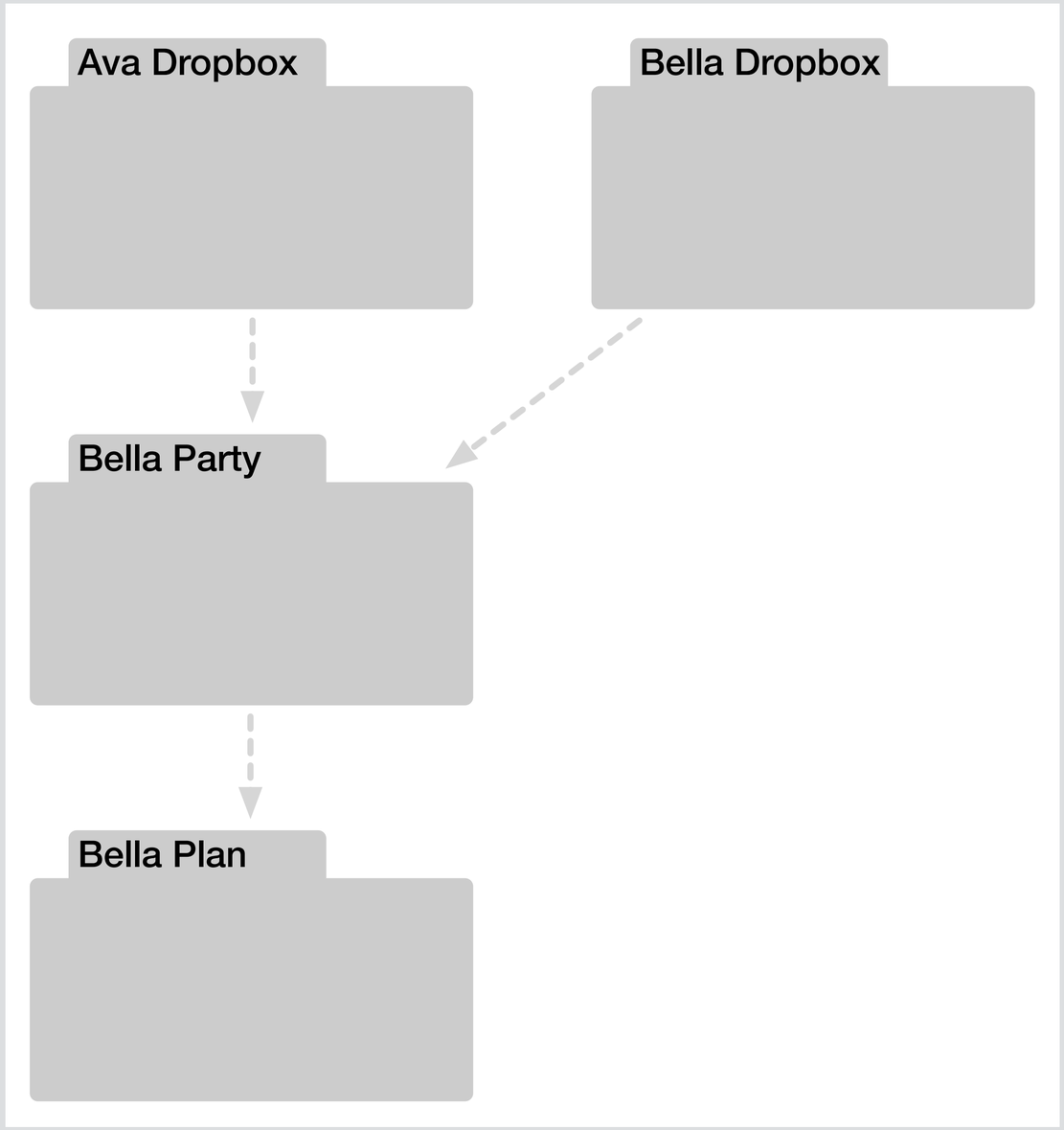
```
move (o: Folder + Item, to: Folder)
 when to not in o.*contents
 to.contents += o
 let from = o.~contents |
  from.contents -= o
```

expression notation of Alloy,
a software modeling language
you don't need to use it
info at alloy tools.org

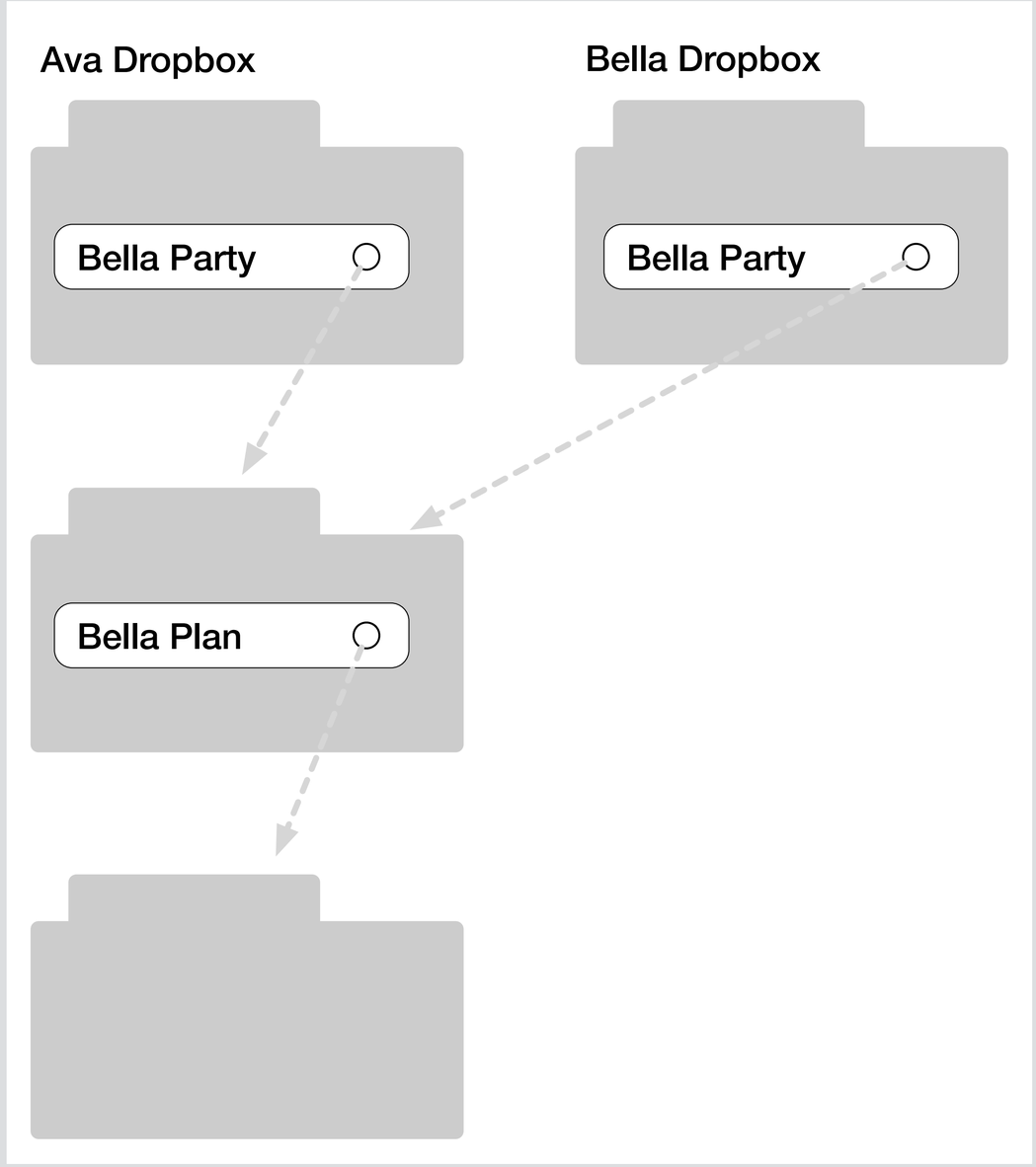
unix's folder concept

concept UnixFolder





state of the simple Folder concept



state of the UnixFolder concept

viewing the state relationally

concept UnixFolder

state
root: Folder
entries: Folder -> **set** Entry
name: Entry -> **one** Name
content: Entry -> **one** (File + Folder)

entries

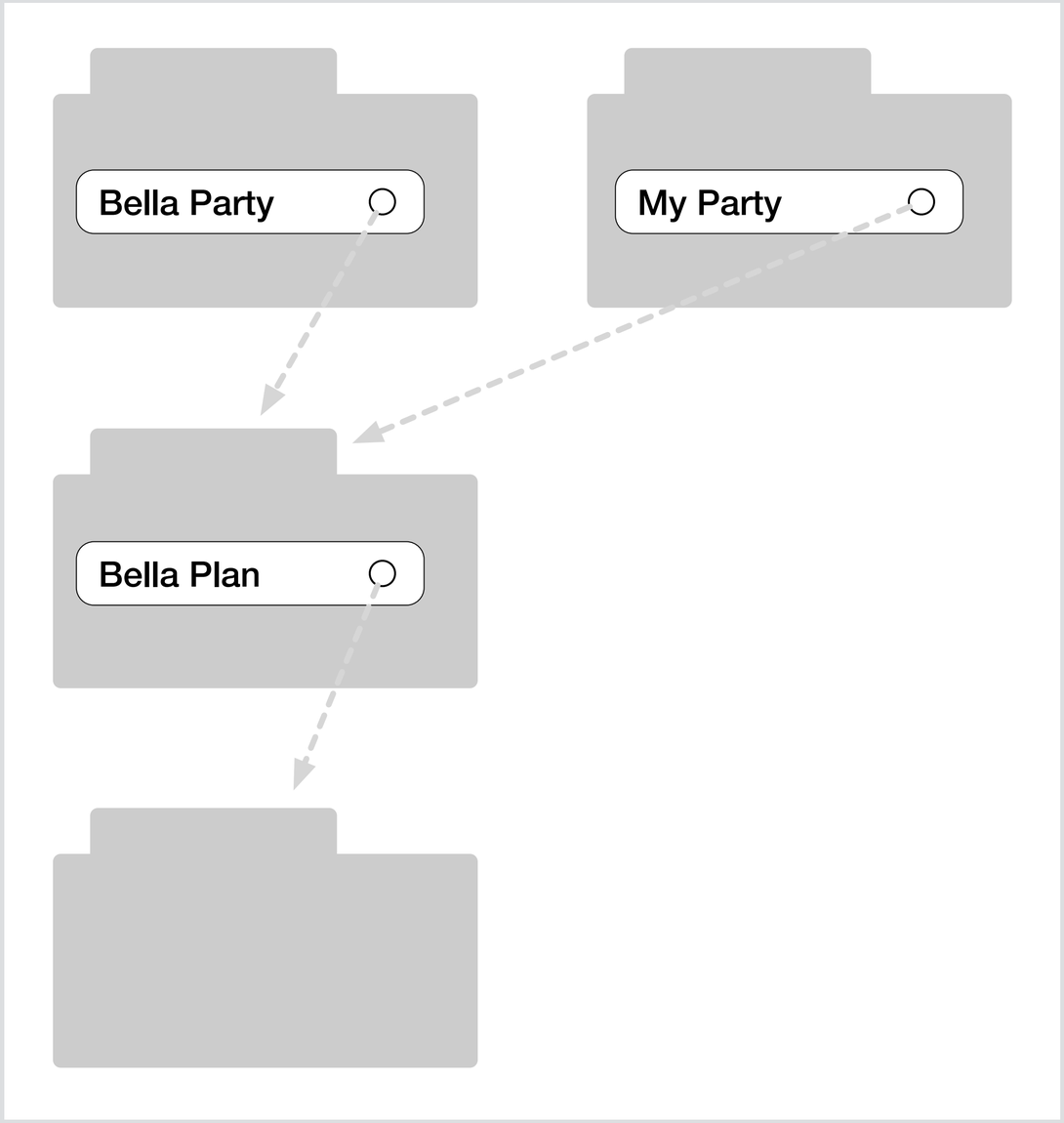
Folder1	Entry1
Folder2	Entry2
Folder3	Entry3

name

Entry1	*Bella Party*
Entry2	*My Party*
Entry3	*Bella Plan*

content

Entry1	Folder3
Entry2	Folder3
Entry3	Folder4



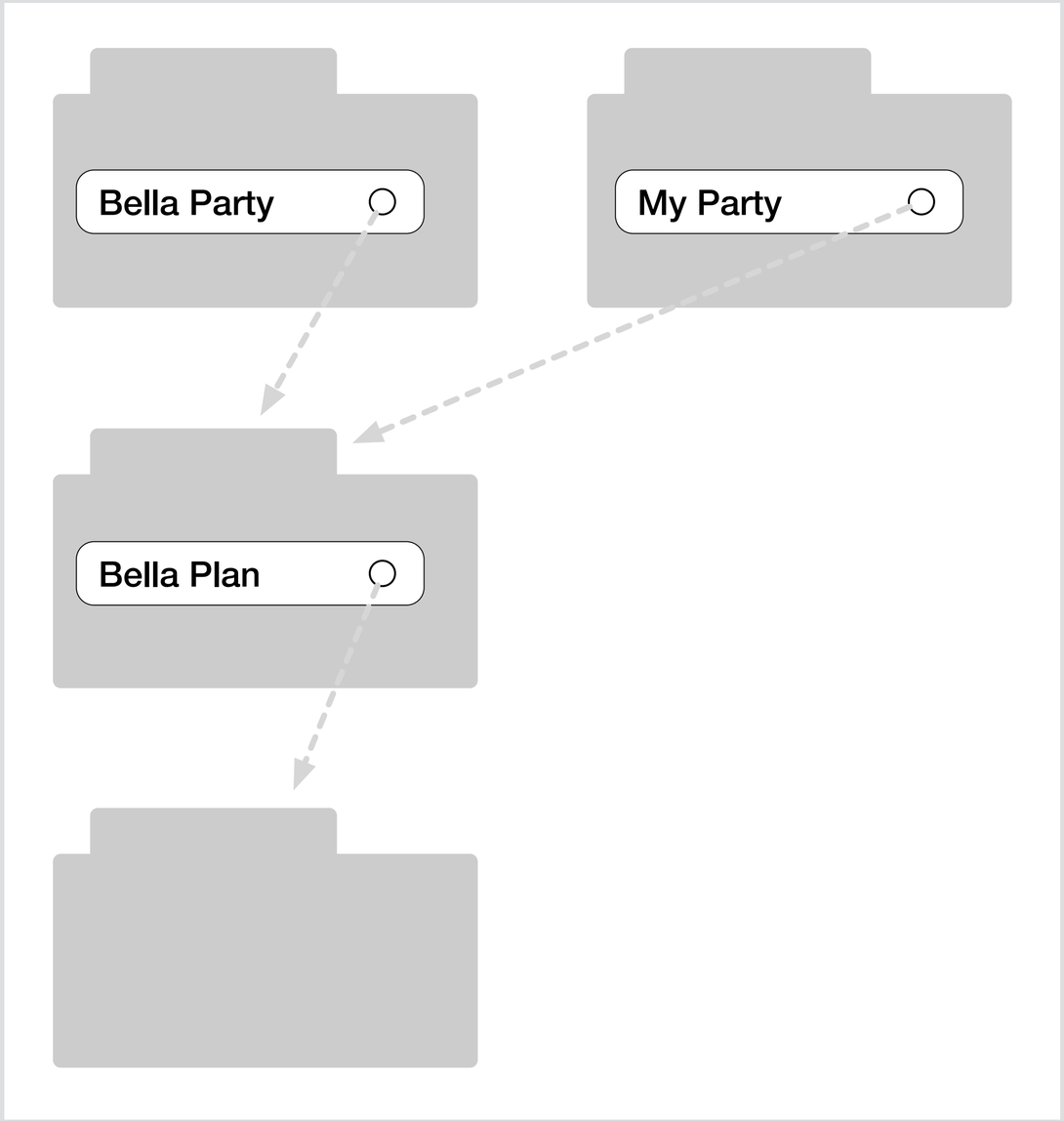
why not just use relational-database-like tables?

concept UnixFolder

state
root: Folder
entries: Folder -> **set** Entry
name: Entry -> **one** Name
content: Entry -> **one** (File + Folder)

	entries	**name**	**content**
Folder1	Entry1	*Bella Party*	Folder3
Folder2	Entry2	*My Party*	Folder3
Folder3	Entry3	*Bella Plan*	Folder4

not “normalized”; leads to redundancies



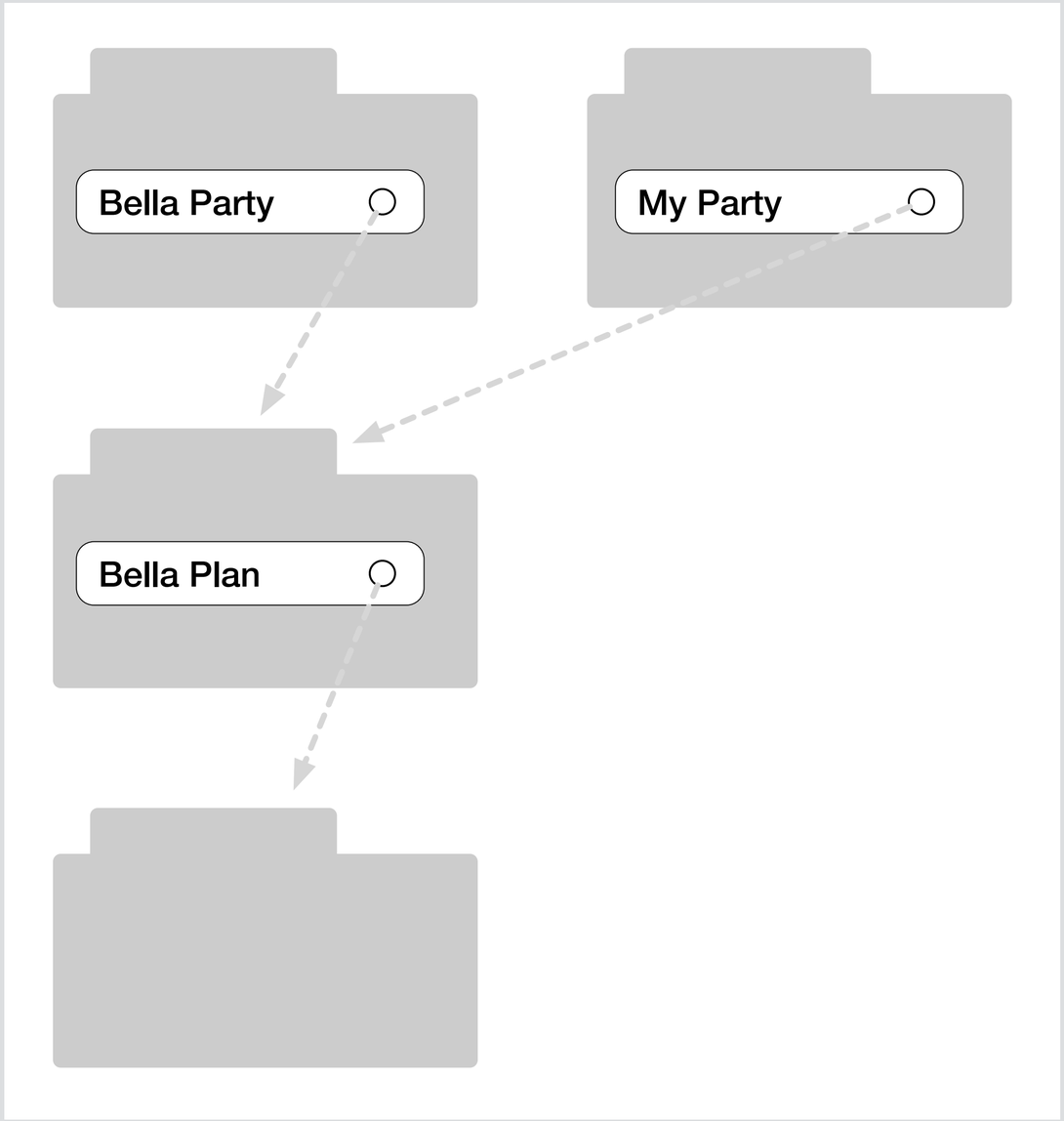
a real use for a higher-parity relation (a table with >2 columns)

concept UnixFolder

state
root: Folder
lookup: Folder -> Name -> **lone** (File + Folder)

lookup

Folder1	*Bella Party*	Folder3
Folder2	*My Party*	Folder3
Folder3	*Bella Plan*	Folder4



tuple objects

concept UnixFolder

state
root: Folder
entries: Folder -> **set** Entry
name: Entry -> **one** Name
content: Entry -> **one** (File + Folder)

in this state, Entry is a "tuple object"
each entry represents a tuple (name, content)

why tuple objects help
avoid need for higher-arity relations
give place for other properties, eg owner

another example of a tuple object
registered: Student -> **set** Offering
class: Offering -> Class
semester: Offering -> Semester

sometimes need a set object
prereqs: Class -> **set** Prereq
requires: Prereq -> **set** Class

mini summary

simple relational view
distinct relations, not tables that combine them
avoid multi-arity relations with tuple and set objects

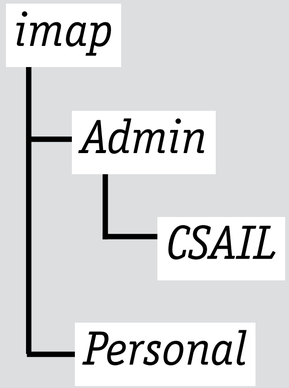
preconditions & postconditions
get tricker for complicated structures
eg, moving a folder into a descendant

state invariants

revisiting the simple folder state

concept Folder [Item]

state
root: Folder
contents: Folder -> **set** (Item + Folder)
name: Folder -> **one** Name



what **invariants** hold for this state?

unique names within a folder

contents is a tree

no cycles

root has no parent

all other objects have one parent

all objects reachable from root

contents

Folder0	Folder1
Folder0	Folder2
Folder1	Folder3

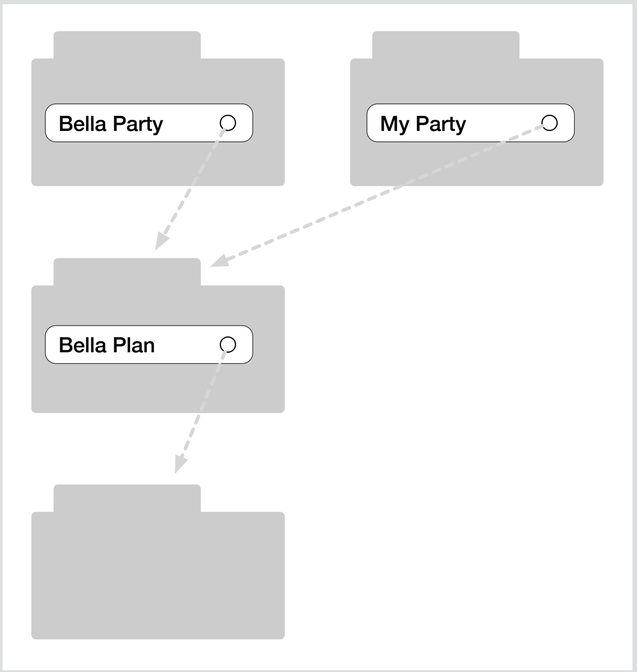
name

Folder0	*imap*
Folder1	*Admin*
Folder2	*Personal*
Folder3	*CSAIL*

invariants for unix folders

concept UnixFolder [Item]

state
root: Folder
entries: Folder -> **set** Entry
name: Entry -> **one** Name
item: Entry -> **one** (Item + Folder)



which **invariants** hold for this state?

unique names within a folder

contents is a tree

- no cycles
- root has no parent
- all other objects have one parent
- all objects reachable from root

when is f a parent of g?
when g is in f.entries.item

our old friend Upvote

concept Upvote

state
votes for each item, up or down
which user issued each vote
ranked sequence of items

can you formalize the state and provide invariants?

state
upvote, downvote: Item -> **set** User
rank: Item -> Int

mini summary

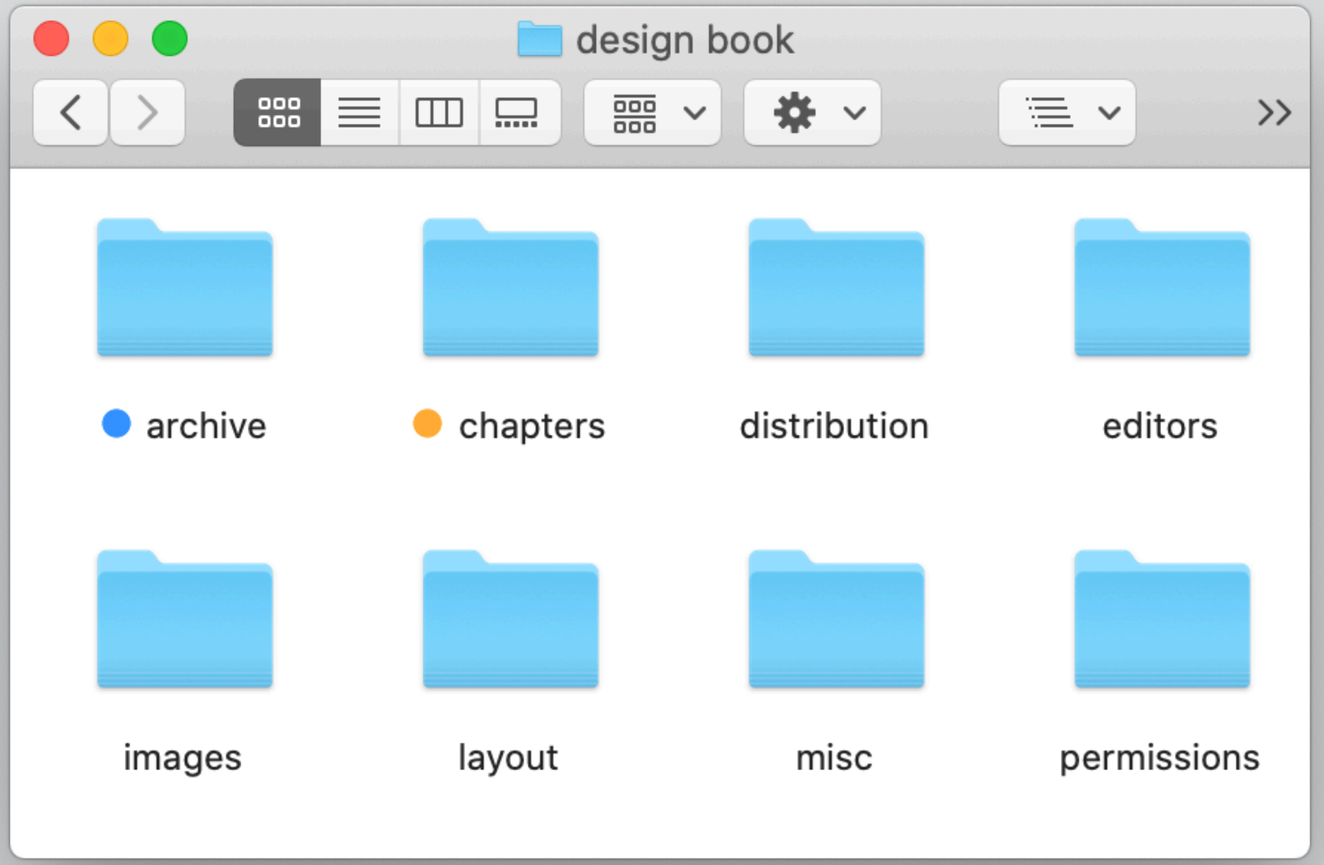
invariants arise when
not all state configurations are legal
just like rep invariants in 6.031

design implications
must make sure that all actions preserve invariants

code implications
become integrity constraints in databases
code must maintain them
may also check dynamically and fix when broken

from
concept state
to app state

app state is combination of concept states



folders and labels

let's draw the state

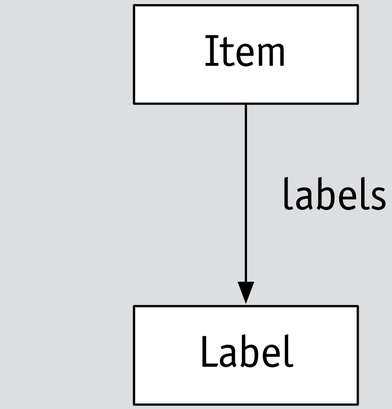
concept Label [Item]

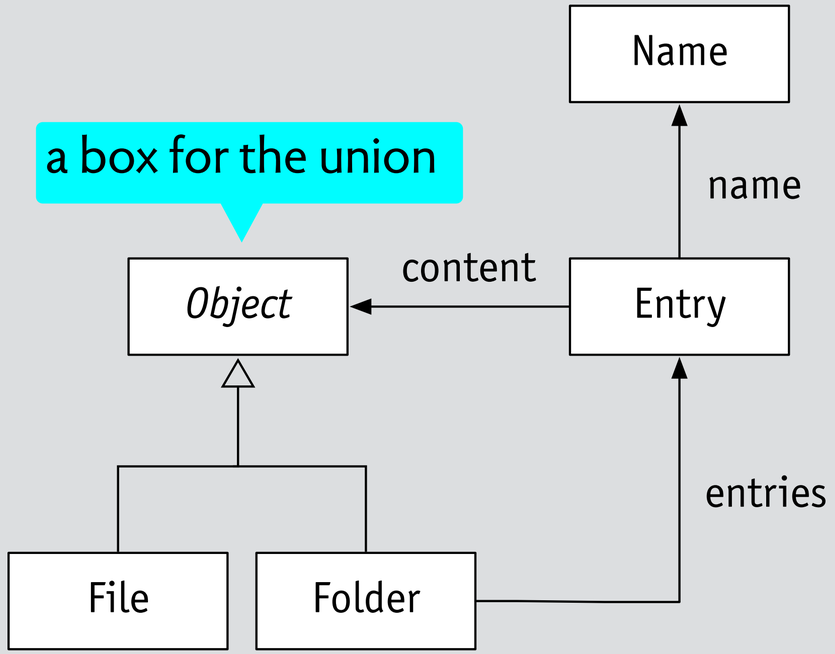
state
labels: Item -> **set** Label

concept UnixFolder

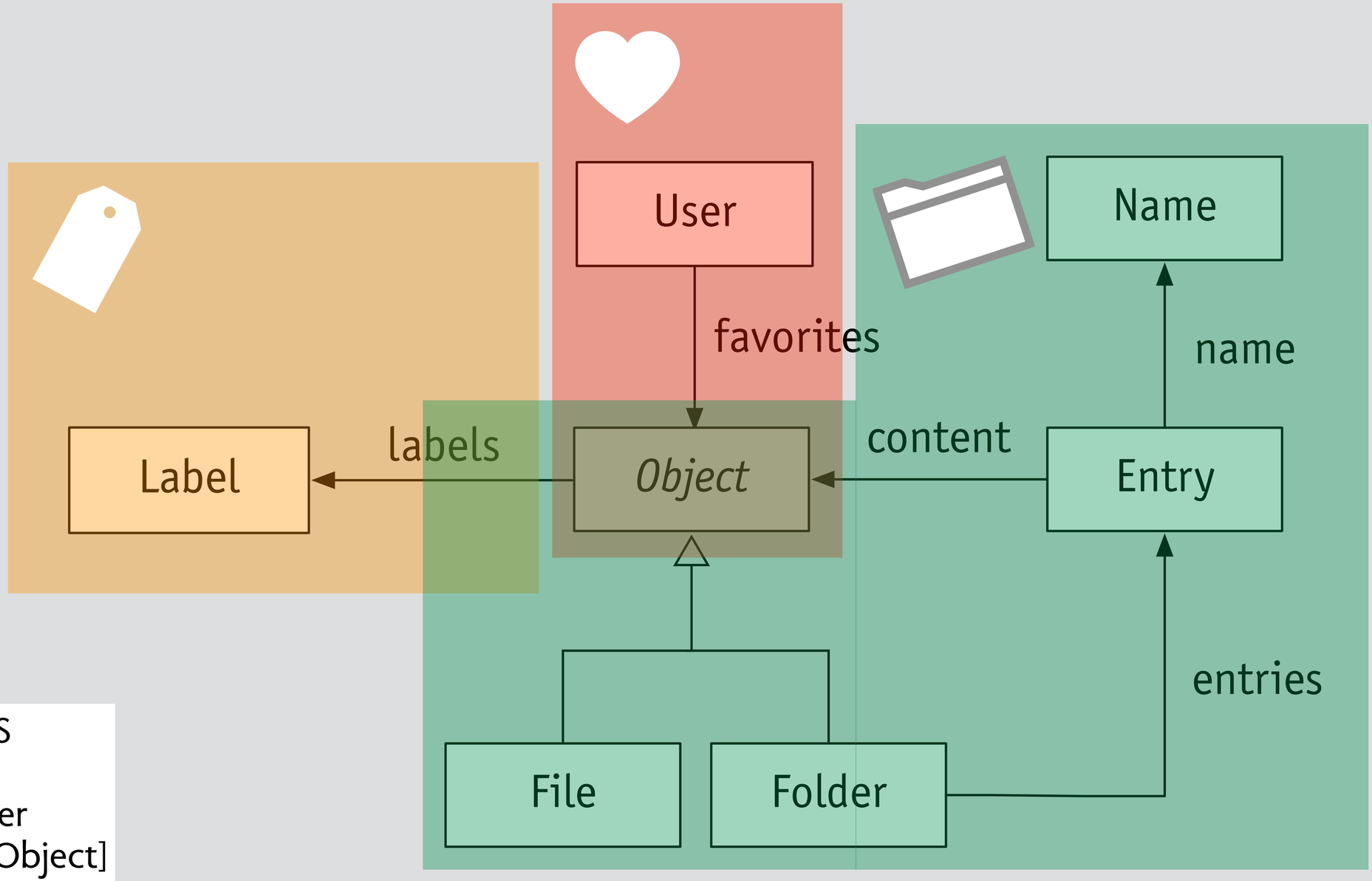
state
root: Folder
entries: Folder -> **set** Entry
name: Entry -> **one** Name
content: Entry -> **one** (File + Folder)

can we put these concepts together?





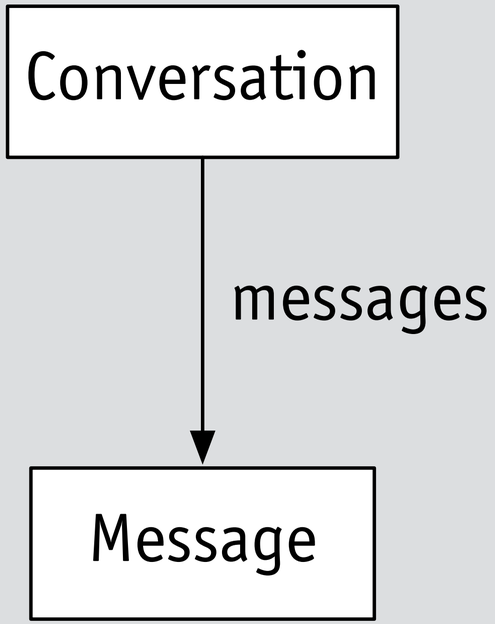
combining concept states



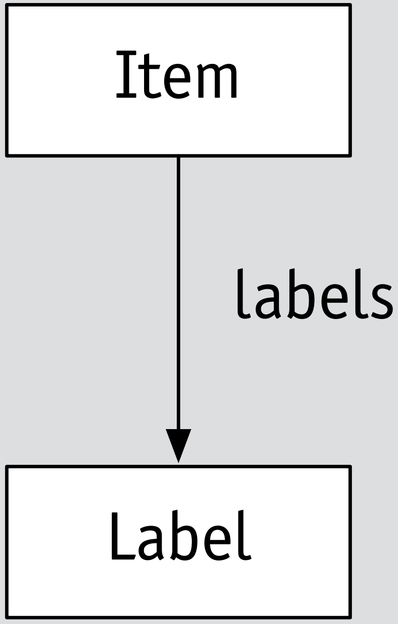
app MyOS
concepts
UnixFolder
Favorite[Object]
Label[Object]

combining Conversation and Label in Gmail

concept Conversation

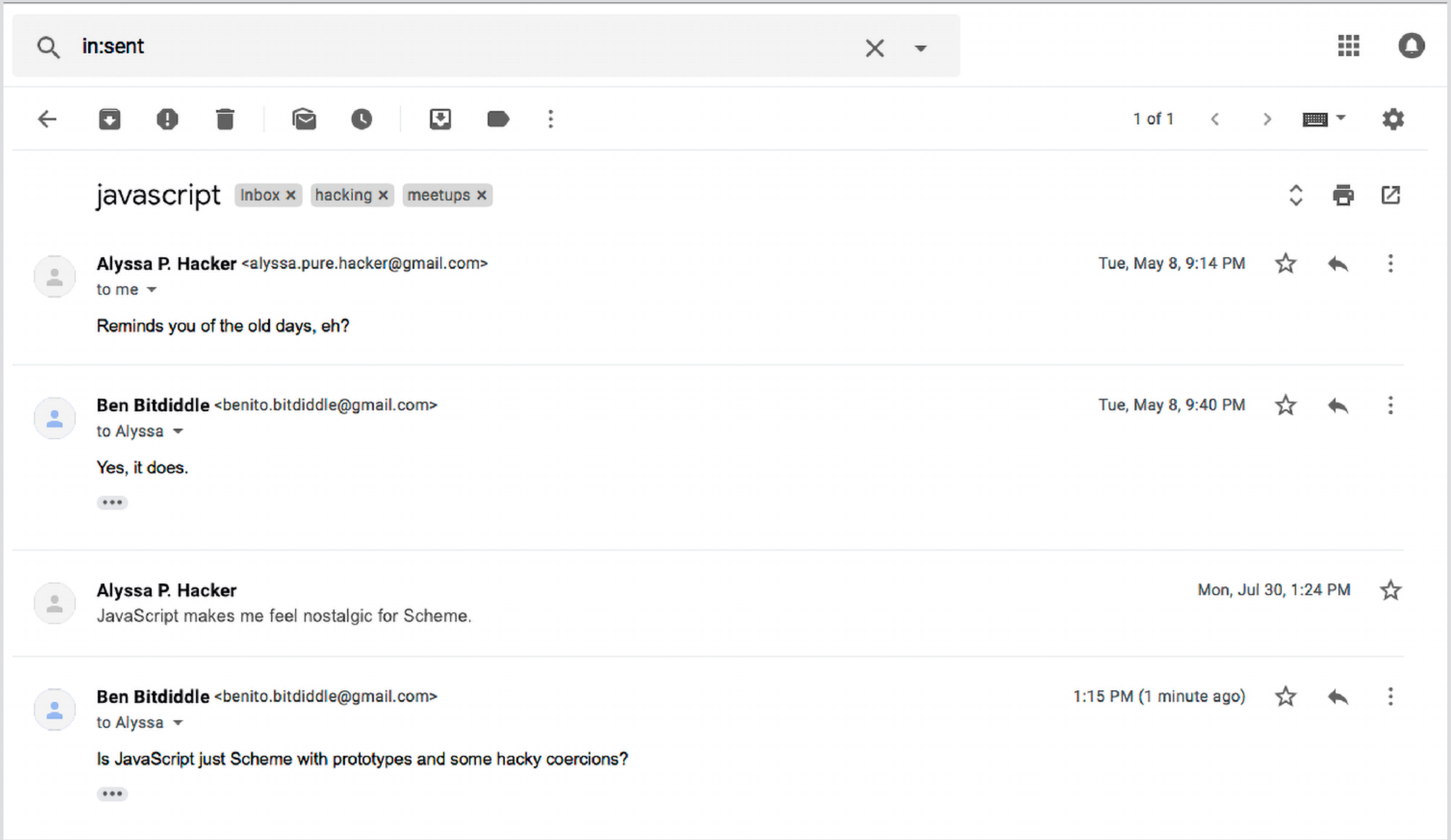


concept Label [Item]



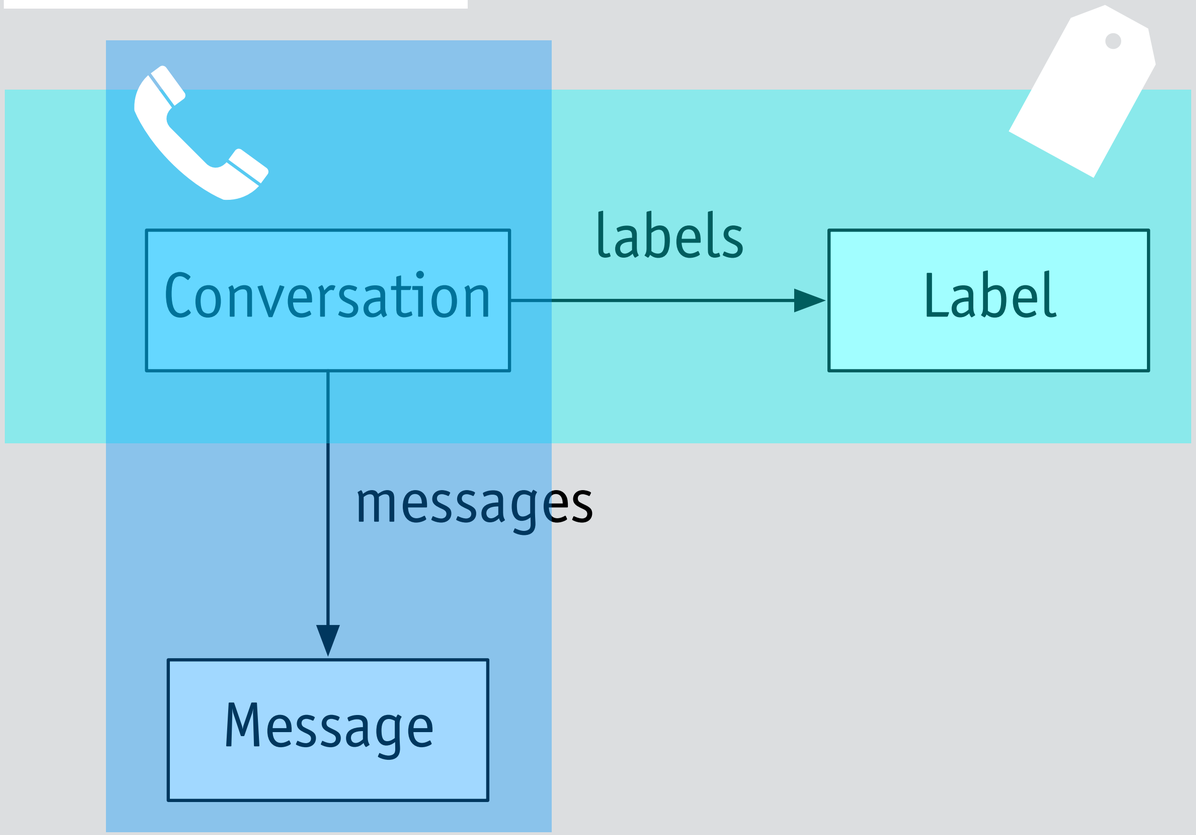
should labels be attached to conversations or messages?

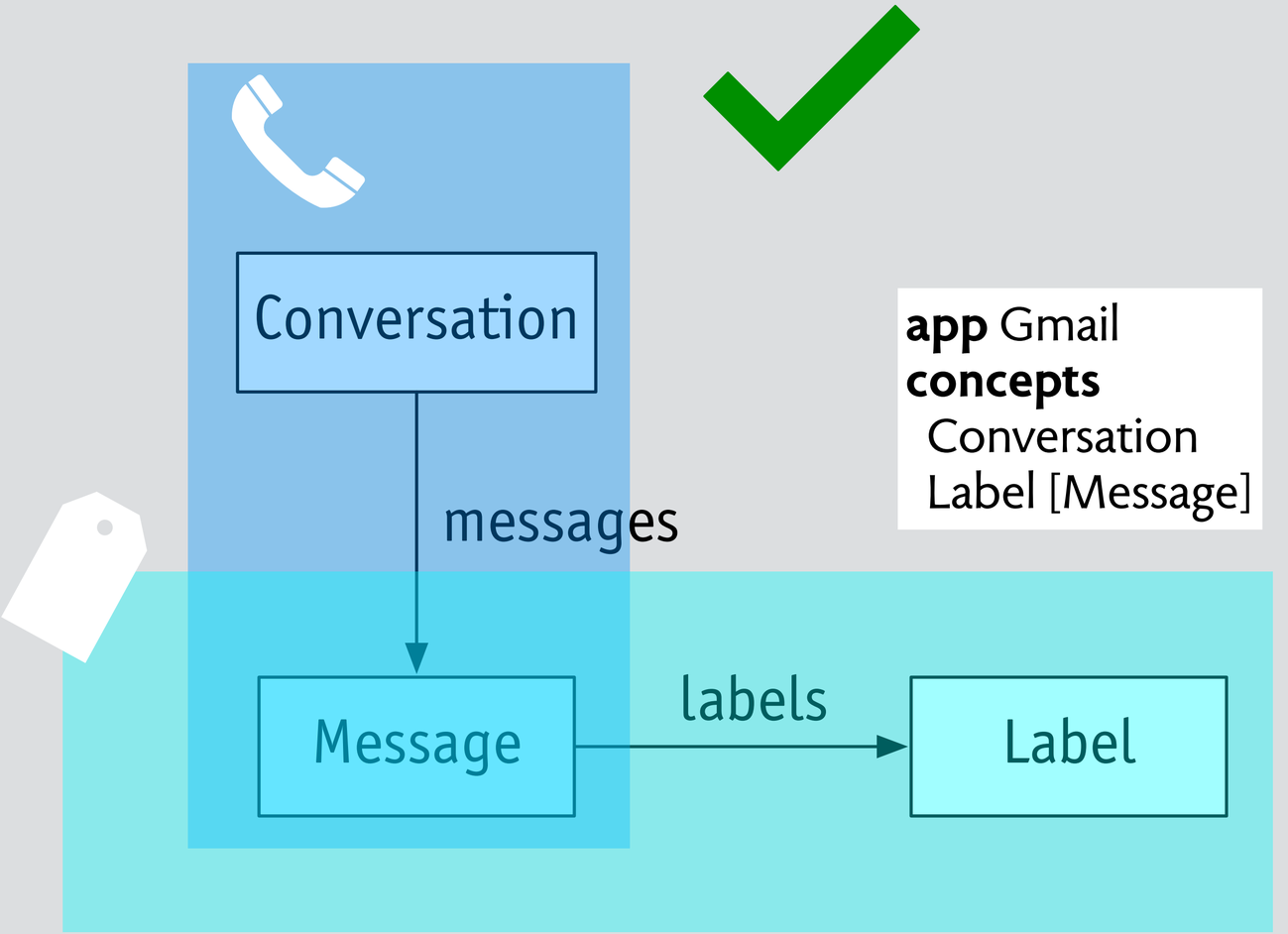
a conversation in Gmail with some labels



which is it?

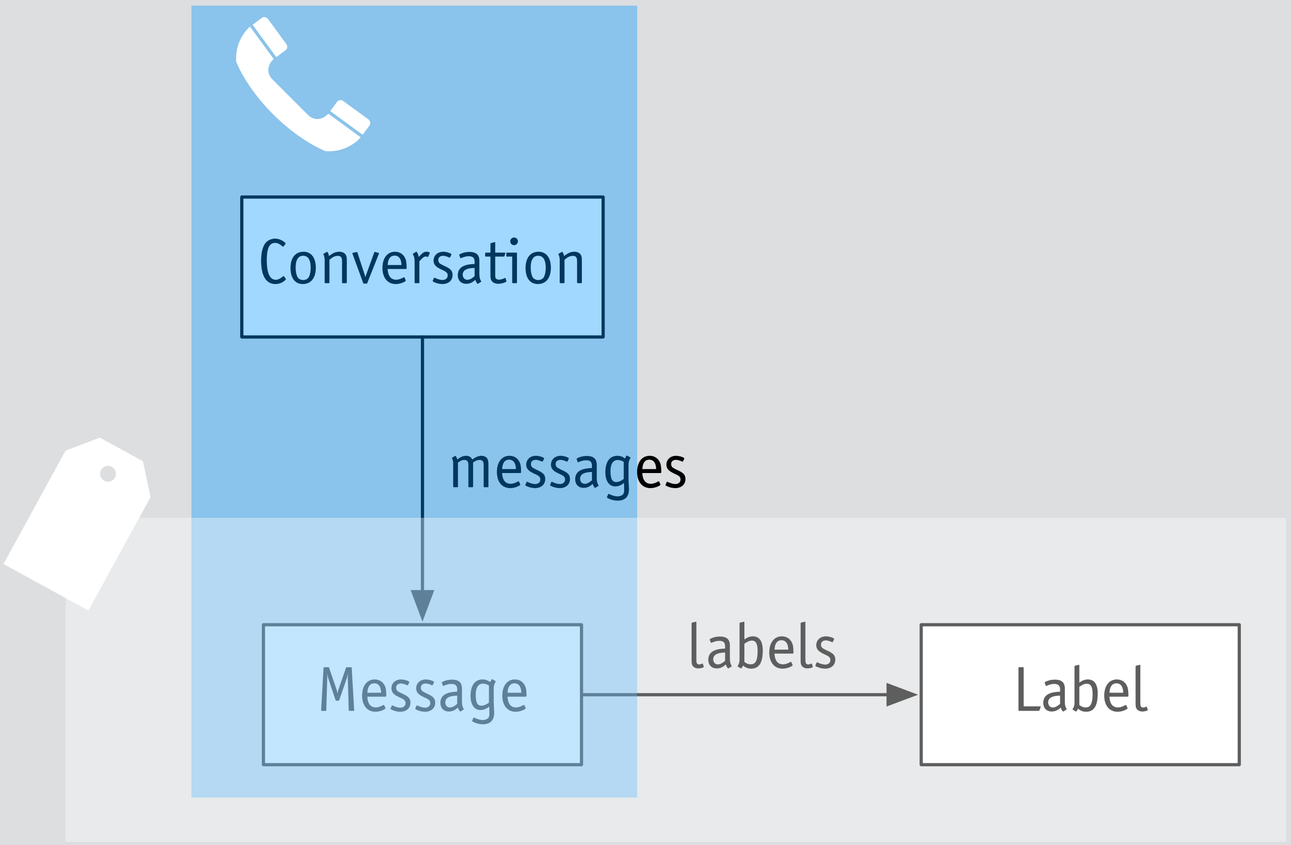
app Gmail
concepts
Conversation
Label [Conversation]





app Gmail
concepts
Conversation
Label [Message]

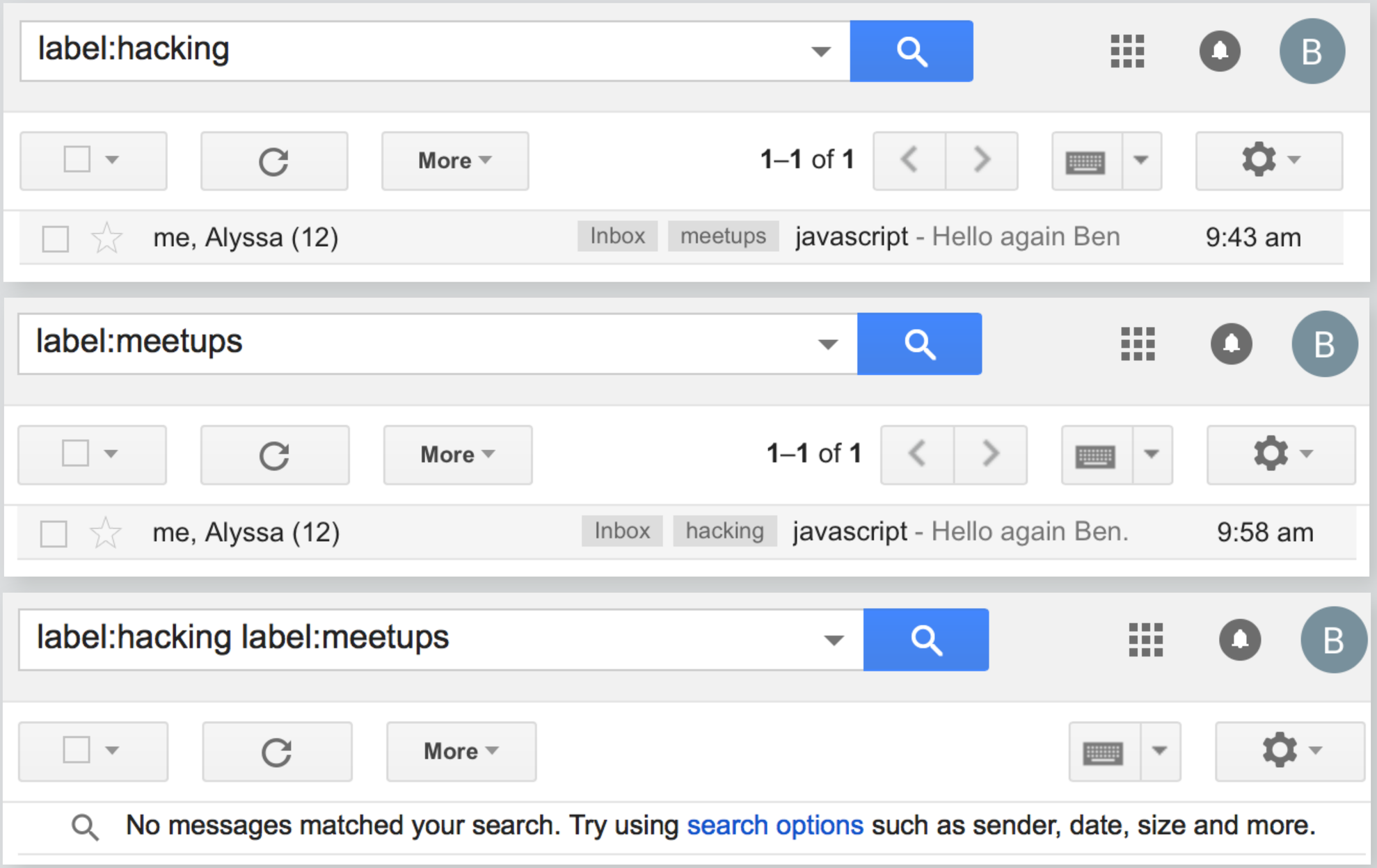
so what does Gmail display when you run Label.filter?



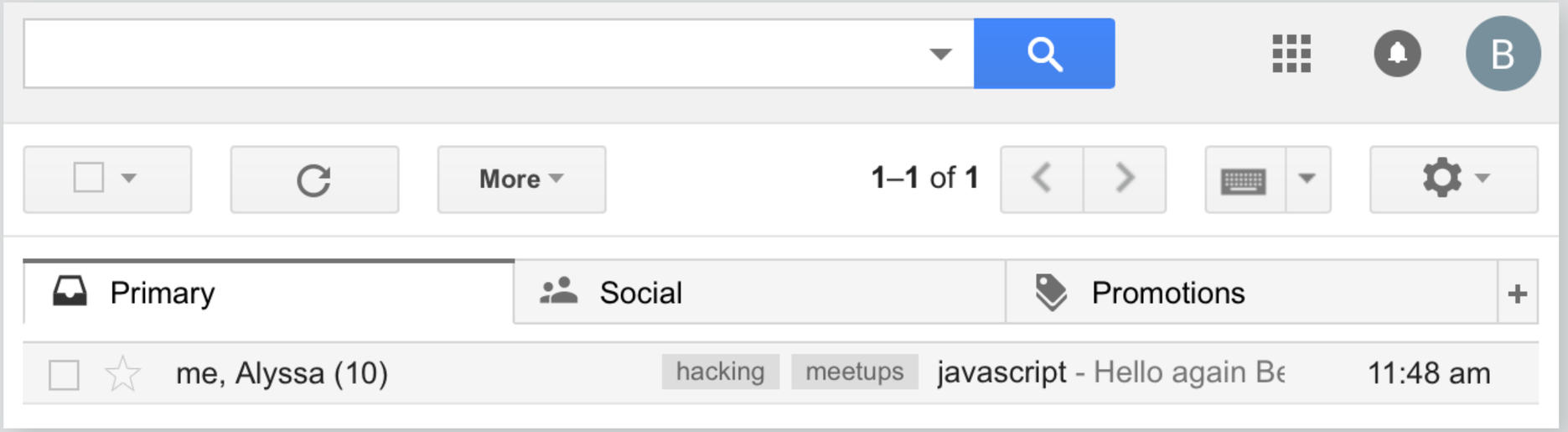
shows the set of Conversations that contain a message included in the filter results, **along with** all their messages!

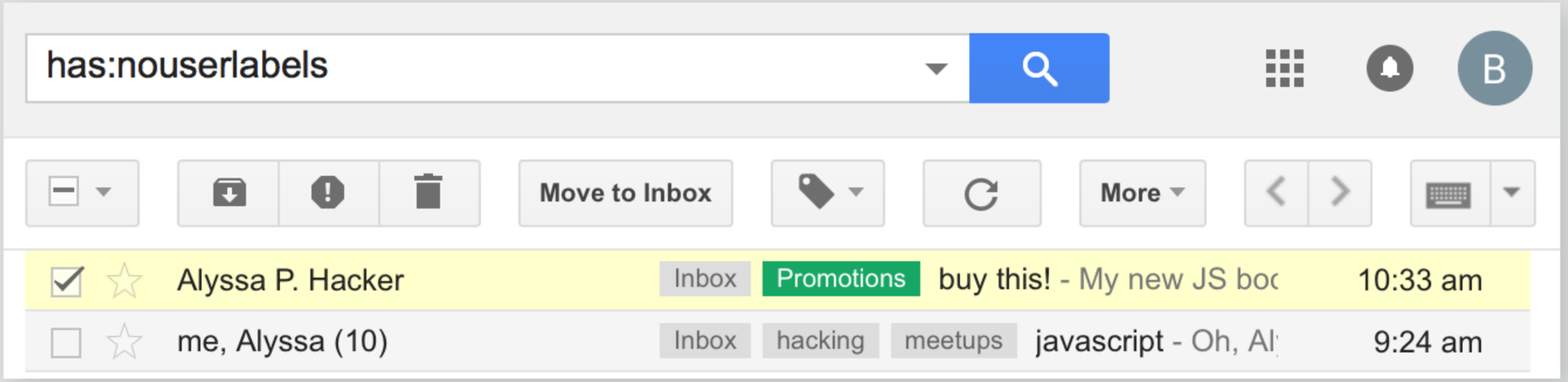
app Gmail
concepts
Conversation
Label [Message]

gmail funnies (a)



gmail funnies (b)





making concepts generic

making polymorphism explicit

concept Upvote

purpose rank freets by popularity

state
upvote, downvote: Freet -> **set** User
rank: Freet -> Int

app Fritter
concepts
Freet
Upvote

A reasonable concept design
while you're exploring Fritter

concept Upvote [Item]

purpose rank items by popularity

state
upvote, downvote: Item -> **set** User
rank: Item -> Int

app Fritter
concepts
Freet
Upvote[Freet.Freet]

A better concept design
that makes **polymorphism** clear

fixing a flawed concept description

concept Upvote

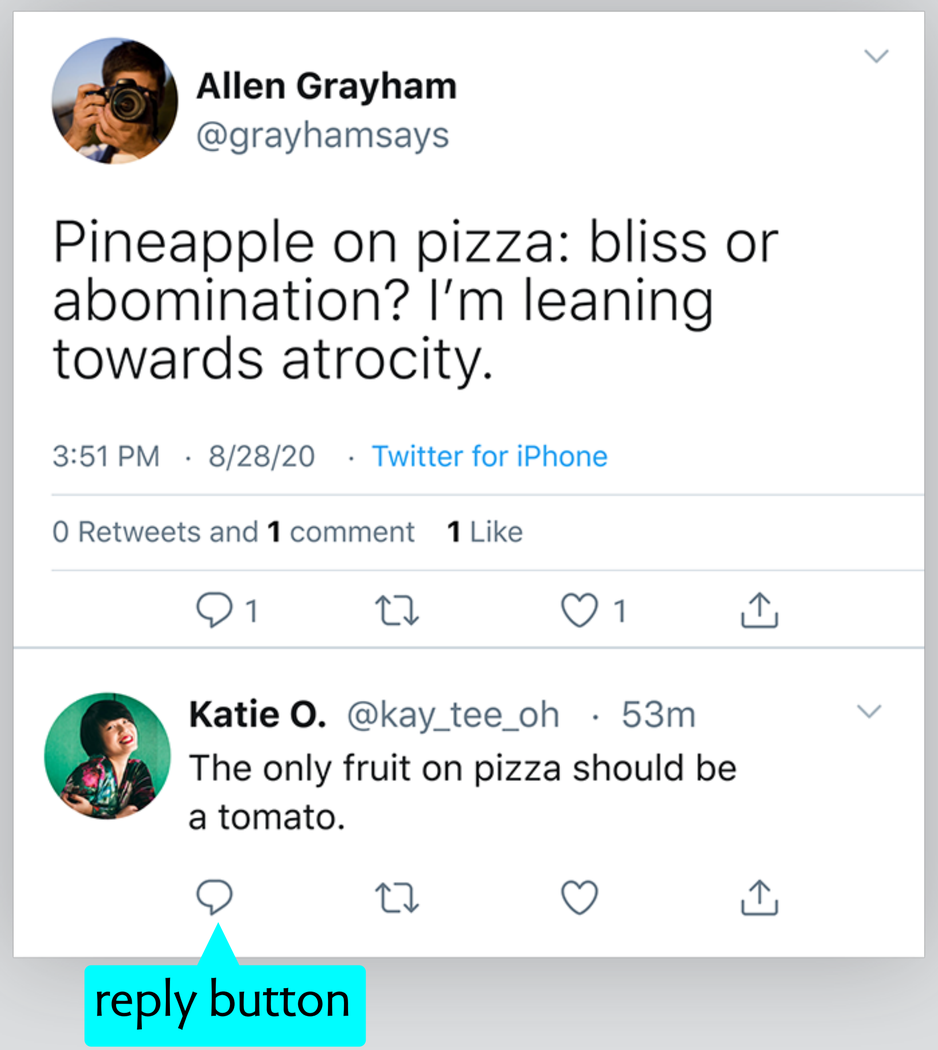
purpose rank freets by popularity

state
upvote, downvote: Freet -> **set** User
rank: Freet -> Int
created: Freet -> Date

actions
update ()
 // update rank based on freet ages

can't be supported by state:
Freet is just a reference!
need to augment the state

why not just "use" the Freet concept inside Upvote?
this would undermine concept independence
it would introduce implementation dependences
and wouldn't scale anyway

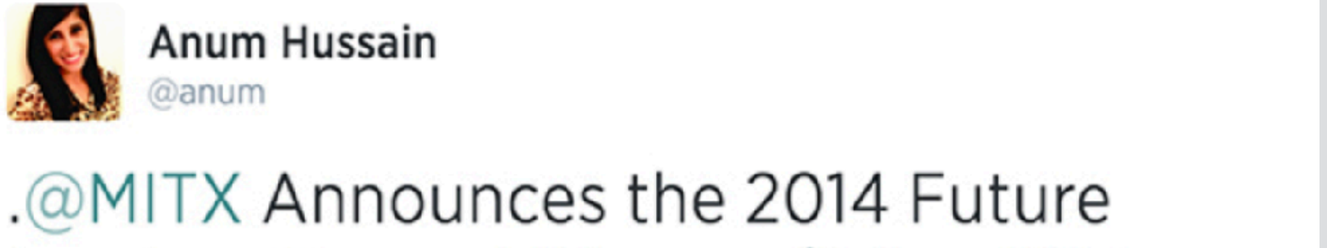


is Tweet a polymorphic concept?
can we separate out a Content concept?

seems a bit pedantic?
in many blogging platforms, full blown rich text editor for posts

prior to 2016
pressing reply called an action: Tweet.new (t: Tweet)
whether t was a reply depended on the text inside t

Want everyone to see your Tweets? Or perhaps you're not using @username to reply to someone, but rather to begin a sentence with her handle. In situations like these, use a period — "." — in front of your @reply to allow your entire following to see the Tweet, as in the following example.



from The #1 Twitter Mistake (dummies)

takeaways

what we learned today

relational view of state
a powerful and simple way to think
can then translate into any rep

state invariants
stronger implies simpler but less flexible state
and may imply more work to maintain

joining concepts
about instantiating generics
can draw a diagram to show connections
but can implement independently

generic concepts
bring simplicity and separation of concerns
try to be as generic as possible