concept design
part 2: behavior

Daniel Jackson · Autodesk Online Workshop · June 2025


Charles Eames Ray Eames 13721
The details are not details. They make the design. *Charles Eames*
Eames Contract Storage
1507 Folding bed unit; light shelf, bed platform, reading light. Accessories: mattress.
HERMAN MILLER INC.
ES 103

qualities of the language we're seeking

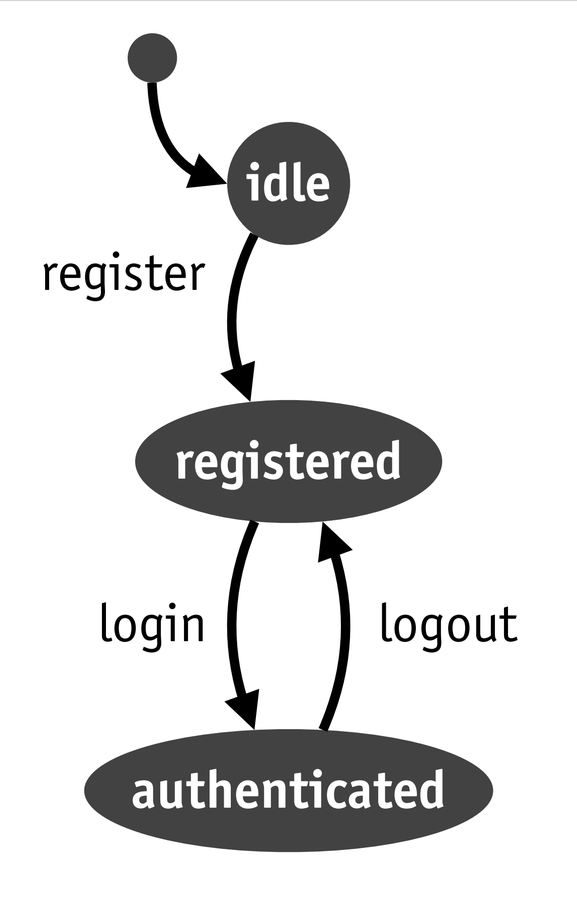
essential
observable behavior
not UI or code

precise
clear & objective
can express details

succinct
densely expressive
also can be distilled

introducing state machines

a state machine for user registration and sessions



a diagrammatic representation

state
status: {idle, registered, authenticated} := idle

actions

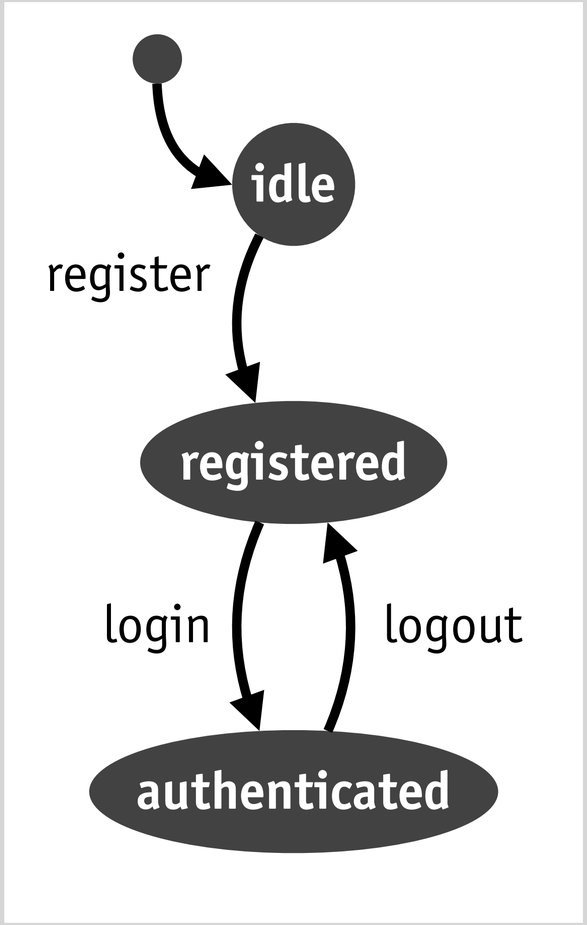
register
requires status = idle
ensures status := registered

login
requires status = registered
ensures status := authenticated

logout
requires status = authenticated
ensures status := registered

a textual representation

traces: histories of action occurrences



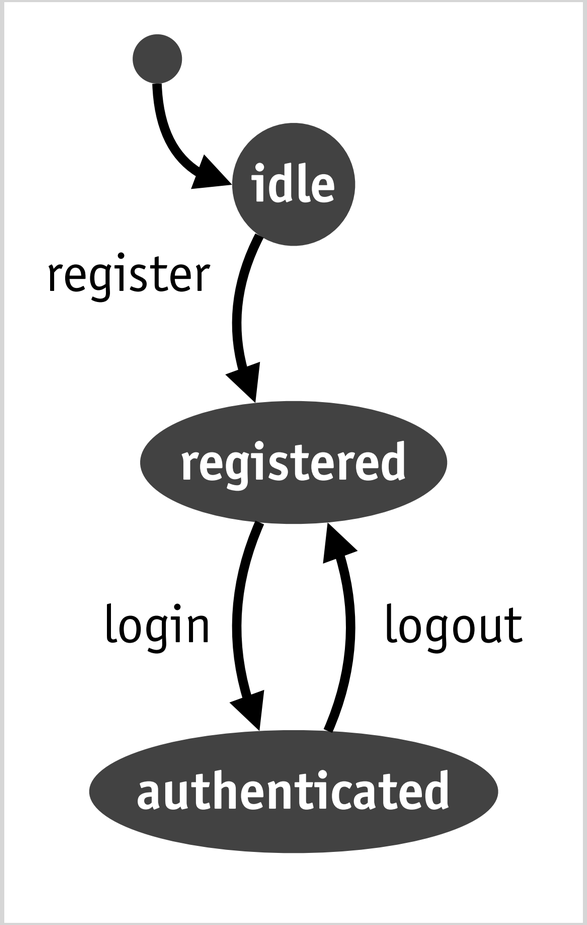
these are traces:

<>
<register>
<register, login>
<register, login, logout>
...

these are not traces:

<login>
<register, logout>

traces and their states



each trace results in a state:

<> [status = idle]
<register> [status = registered]
<register, login> [status = authenticated]
<register, login, logout> [status = registered]
…

why does this matter?
we can synchronize with other machines
“user can create post when authenticated”

a simple puzzle
are there non-empty traces leading to status= idle?
why might this be useful?
how would you change the design?

what if more than one user?

state
a set of registered users
for each registered user
  a username and a password
a set of active sessions
for each active session
  an authenticated user

actions

register (username, password: String): User
requires no existing registered user with username
ensures
  create a fresh user with username and password
  add to set of registered users

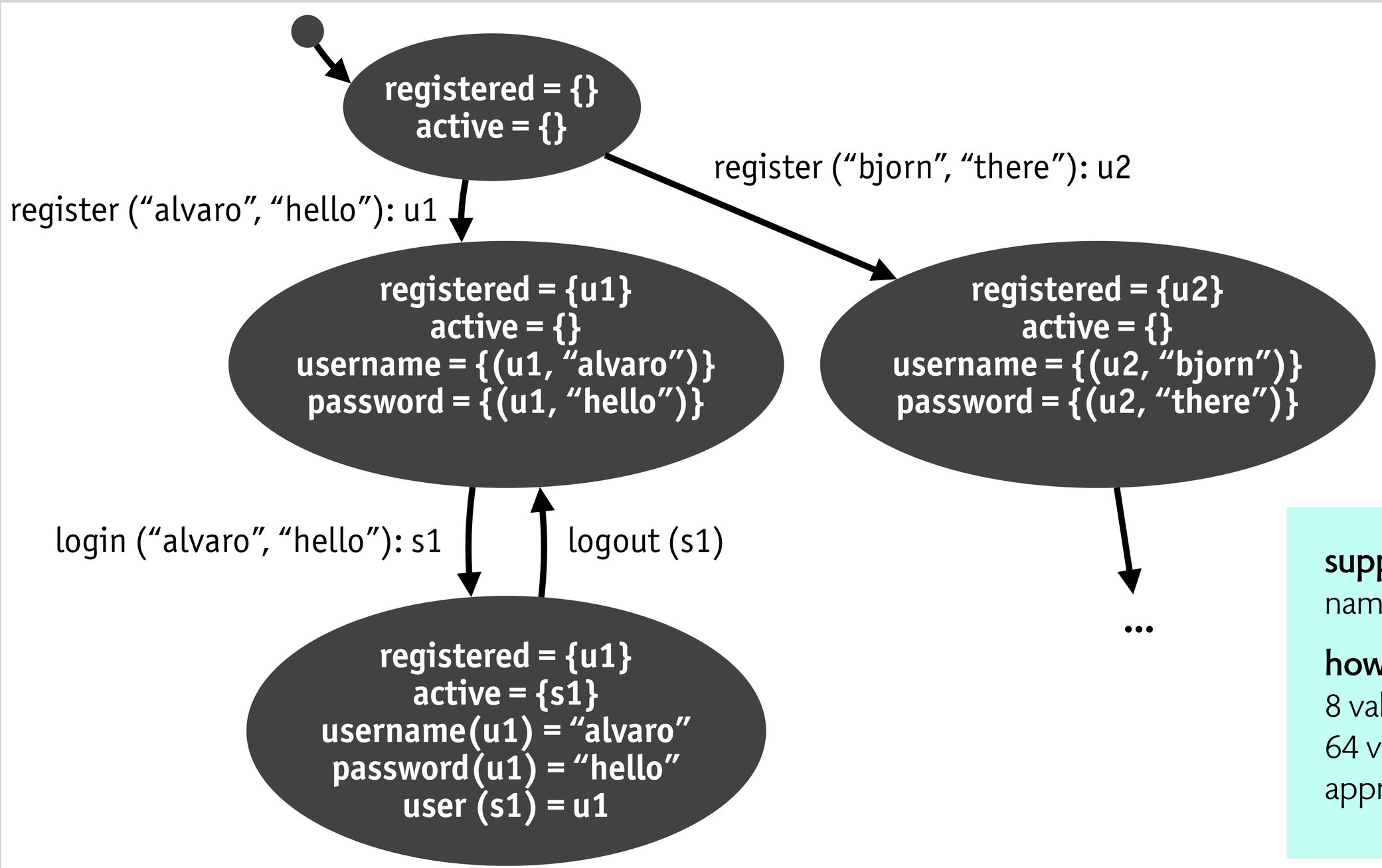
login (name, password: String): Session
requires some registered user with username and password
ensures
  create a fresh active session as a result
  associate the matching user with the session

logout (session: Session)
requires the session is active
ensures
  remove the session from the active sessions

why does the login action return the session?
because the client needs it to call logout!

why not just have login return the user instead?
because this design allows one user to have two sessions active

why diagrams no longer help



suppose scope of 3
names, users, passwords

how many states?
8 values of each set
64 values of each map
approx 17m states!

defining the state more formally

informal state declaration

state
a set of registered users
for each registered user
  a username and a password
a set of active sessions
for each active session
  an authenticated user

formal state declaration

state
registered: set User
username, password: User -> String
sessions: set Session
user: Session -> User

another possible formalization

state
registered: set User
for all registered
  username, password: String
sessions: set Session
for all sessions
  user: User

why do this?
lets you give a name and a type
more easily translated to code
gateway to nice diagrams

why not do this?
harder for less technical folks to read

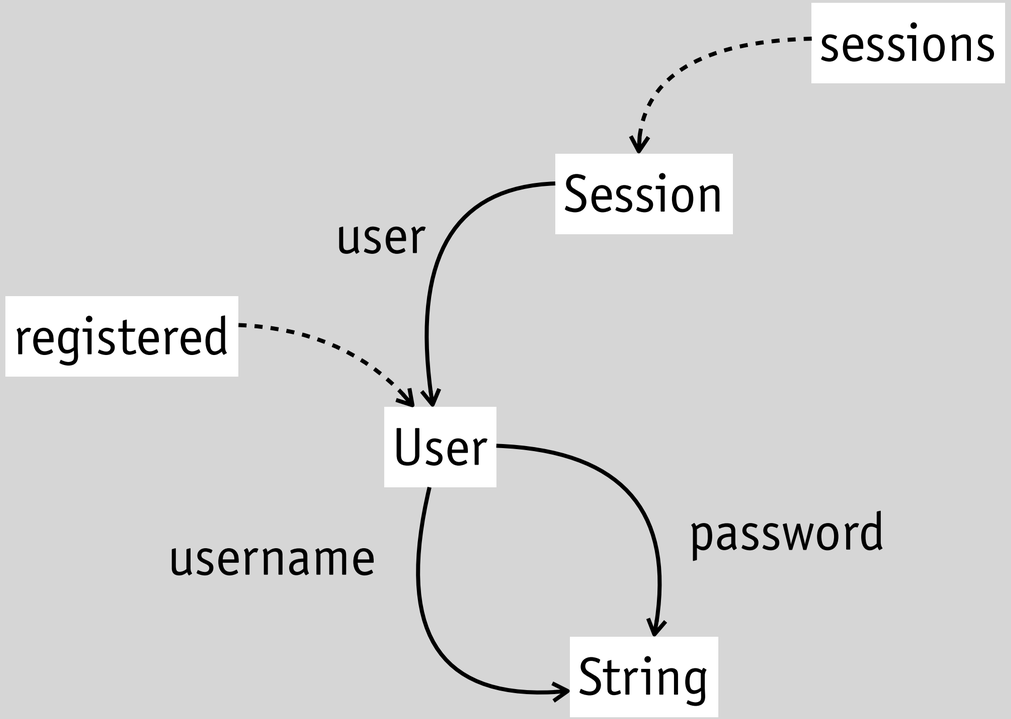
a diagrammatic form

state
a set of registered users
for each registered user
  a username and a password
a set of active sessions
for each active session
  an authenticated user

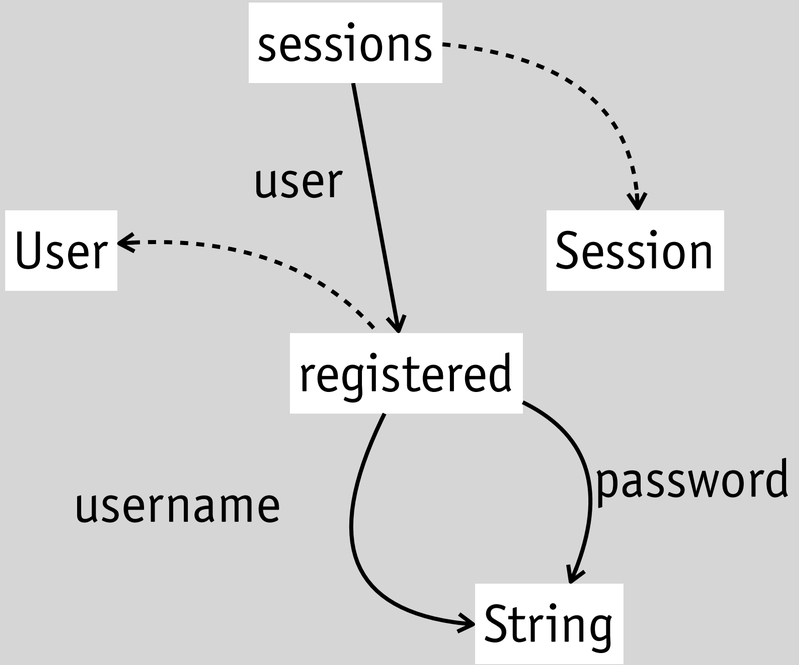
state
registered: set User
username, password: User -> String
sessions: set Session
user: Session -> User

diagram conventions
solid arrow is relation (a set of pairs)
dotted arrow is subset (is-a)

this is a data model
"extended entity-relationship model"



a diagrammatic representation of the state



a tighter representation

defining the actions formally

state
registered: set User
username, password: User -> String
sessions: set Session
user: Session -> User

why do this?
exposes subtle errors
can analyze automatically (eg, with Alloy)

why not do this?
more work and harder for some to read

actions

register (n, p: String): User
requires no u: registered | u.username = n
ensures some u: User - registered {registered += u; u.password := p; u.name := n; result := u }

login (n, p: String): Session
requires some u: registered | u.username = n and u.password = p
ensures some s: Session - sessions {sessions += s; s.user = u; result := s }

logout (s: Session)
requires s in sessions
ensures sessions -= s; s.user := none

traces and their states

each trace results in a state, now a rich structure:

<> [registered = {} and sessions = {}]
<register (n, p): u > [registered = {u} and u.username = n and u.password = p]
<register (n, p): u, login (n, p): s> [... and s in active sessions and s.user = u]
<register (n, p): u, login (n, p): s, logout (s)> [registered = {u} and ...]
...

can assert properties of the state instead:

<> [no registered and no sessions]
<register (n, p): u > [u in registered and u.username = n and u.password = p]
<register (n, p): u, login (n, p): s> [s in sessions and s.user = u]
<register (n, p): u, login (n, p): s, logout (s)> [s not in sessions]

check your understanding

Which of these is NOT true?

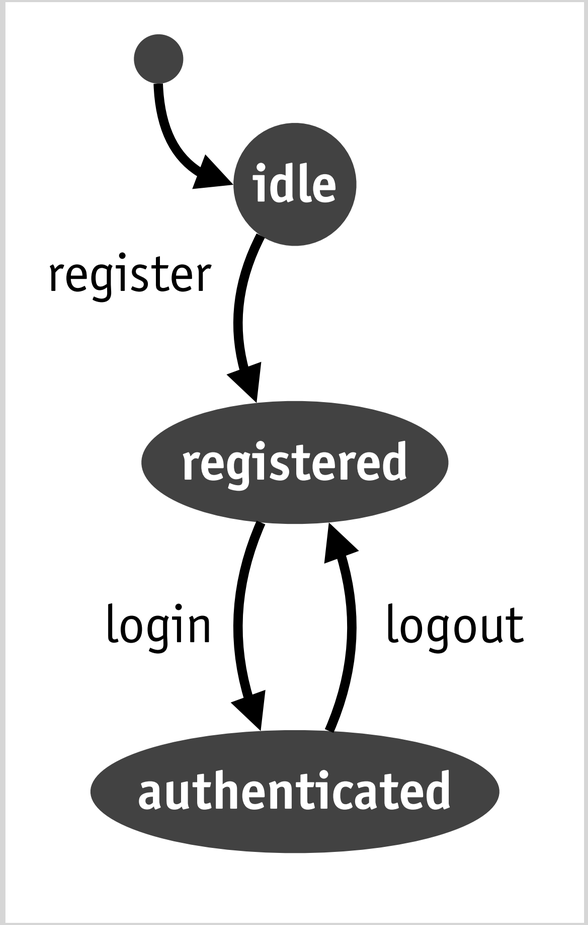
(a) State machines offer a precise but abstract way to describe behavior

(b) State machines are good for all kinds of mechanisms, not just concepts

(c) State machines always terminate eventually

concepts & objects

a common mistake: concept as object



one per user, augmented with username, password, etc?

state
username
password
session

actions

register (username, password: String): User
// creates user with name and password and no session

login (name, password: String): Session
// if name and password match,
// creates a session for the user and returns it

logout (session: Session)
// unsets session

looks appealing at first
reminiscent of OOP, matches diagram

limits scope of concept to one user
but will this actually work?
do the actions all make sense?

how many "objects" in this concept?

state
a set or registered users
for each registered user
  a username and a password
a set of active sessions
for each active session
  an authenticated user

actions

register (username, password: String): User
requires no existing registered user with username
ensures
  create a fresh user with username and password
  add to set of registered users

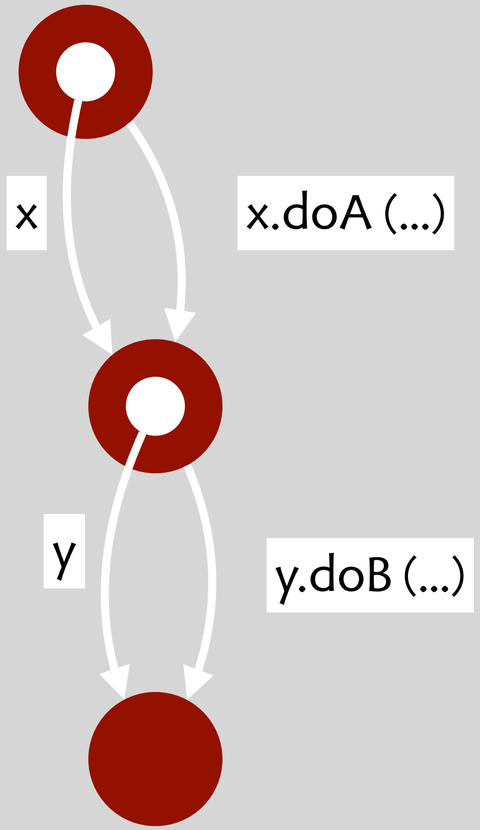
login (name, password: String): Session
requires some registered user with username and password
ensures
  create a fresh active session as a result
  associate the matching user with the session

logout (session: Session
requires the session is active
ensures
  remove the session from the active sessions

this concept creates users *and* sessions
unlike OOP classes, concepts not limited to one type of object

also
concepts capture relationships between objects

thoughts on OOP (for now, more later)



a great framework for programming
an effective way to organize code

the key idea of OOP
model computation as collection of objects
"unary" methods mutate object state
objects reference and call each other

can be limiting for coding
why people use functional languages, eg

even less applicable in design work
"unary methods on single objects"
not a helpful way to describe behavior
worse, conflation & fragmentation (later)

check your understanding

Which correctly relates objects to concepts?

(a) Typically, a system will have more concepts than object classes

(b) Concepts are expressed in terms of objects, so an OOP implementation is preferred

(c) The way objects encapsulate their state is a coding detail ignored in concept design

three examples

what the examples teach

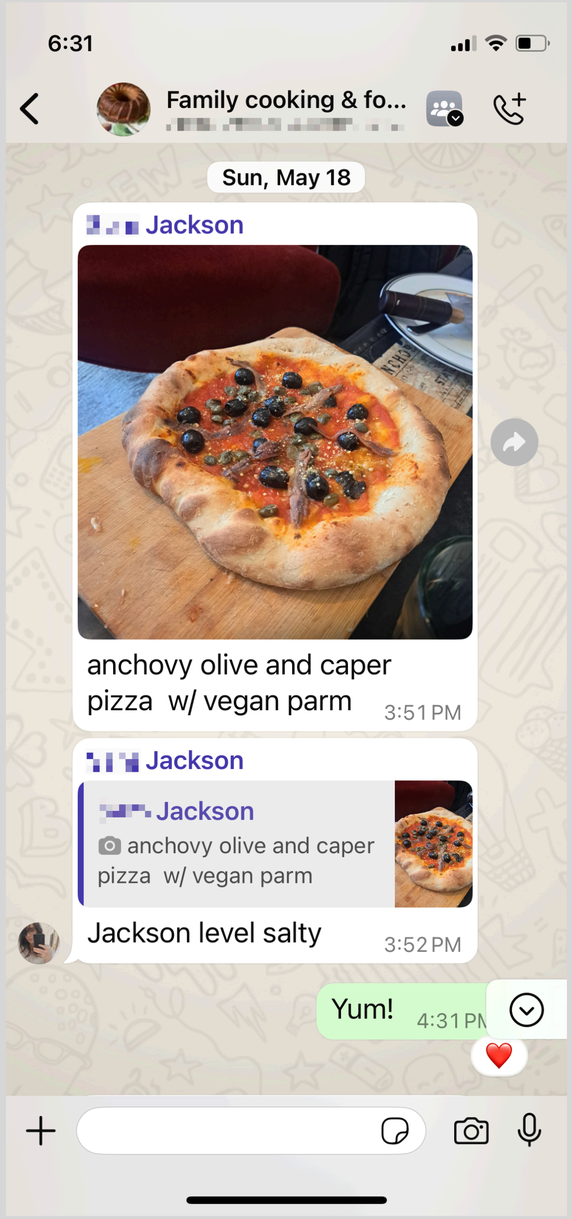
group chat (WhatsApp)
how defining the state helps you explore tricky behaviors

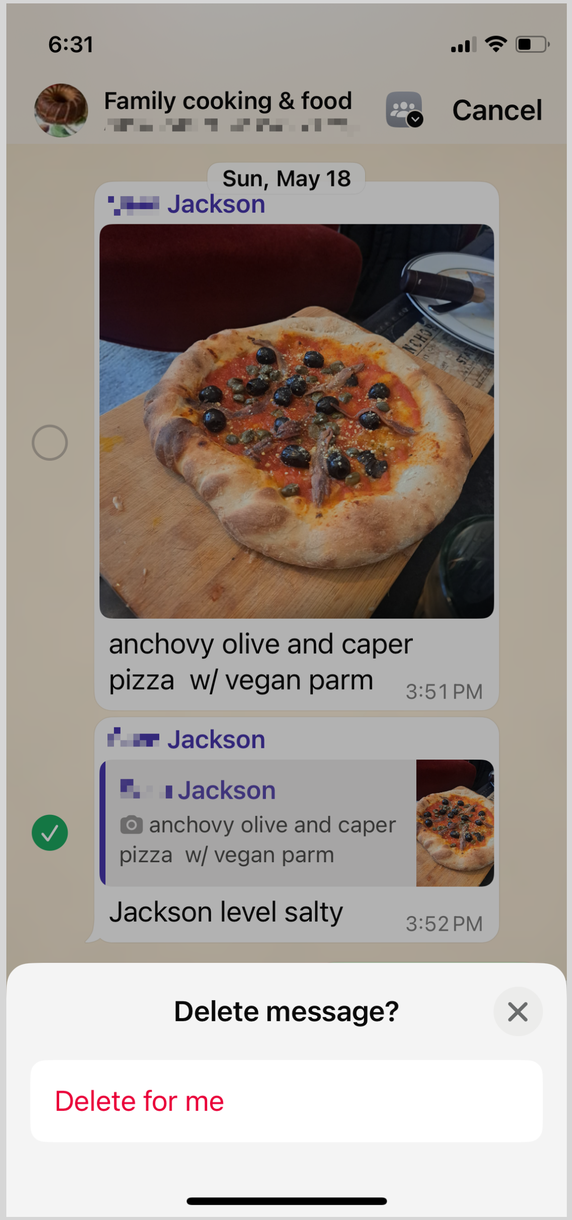
folder (Unix, Dropbox, etc)
how state structure leads to unexpected behaviors

file synchronization (Google Drive, Dropbox, etc)
how to model a distributed system
defining actions with behavior that isn't fully specified
implementing states in a clever way

chat concept in WhatsApp

group chat concept in WhatsApp





some features shown here

sent & received messages

replies to messages

deleting messages

why might group members see different messages?

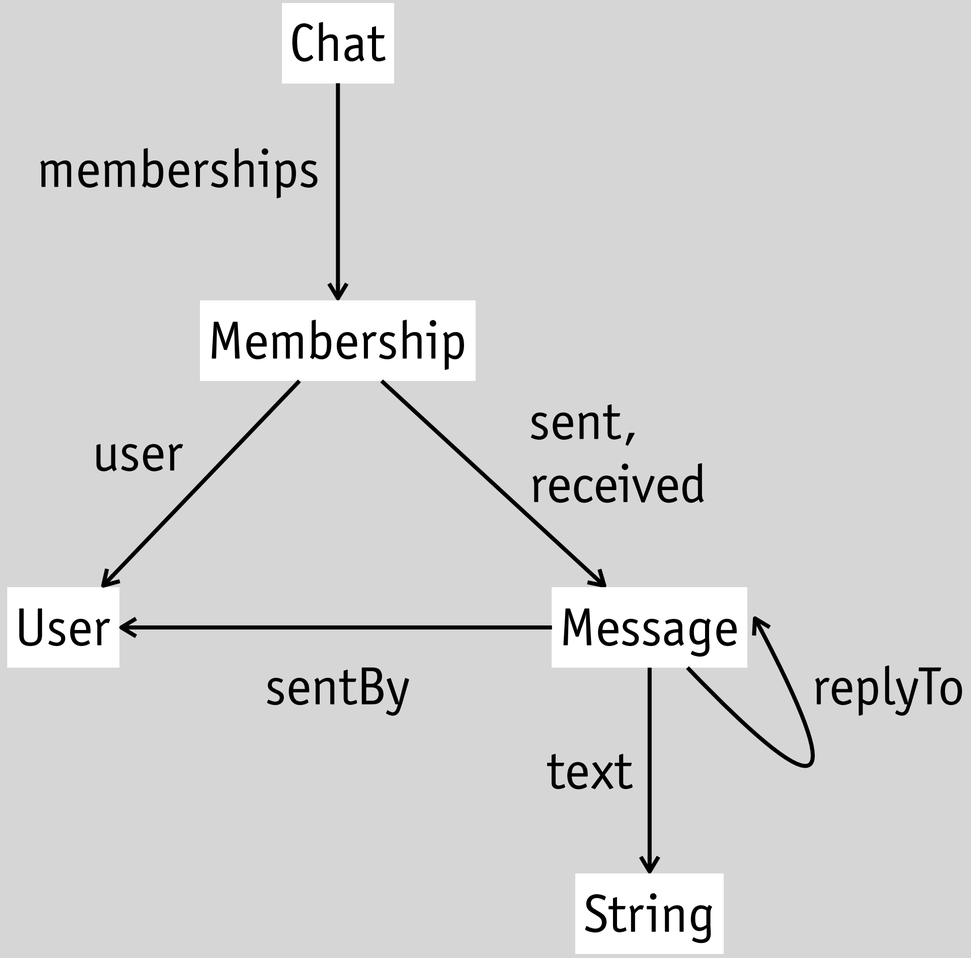
only see messages when member

you can "delete for me"

the state of group chat

concept GroupChat

state
a set of chats
for each chat
  a set of memberships
for each membership
  a user who is the member
  a set of sent messages
  a set of received messages
for each message
  the user who sent it
  the text content
  message it replies to [opt]



why memberships vs. users as members?

user has different messages in each chat

why sentBy if have user's sent messages?

user may have deleted message others still see

actions for group chat

concept GroupChat

state
a set of chats
for each chat
  a set of memberships
for each membership
  a user who is the member
  a set of sent messages
  a set of received messages
for each message
  the user who sent it
  the text content
  message it replies to [opt]

what actions are missing?
create, delete chat
leave, rejoin

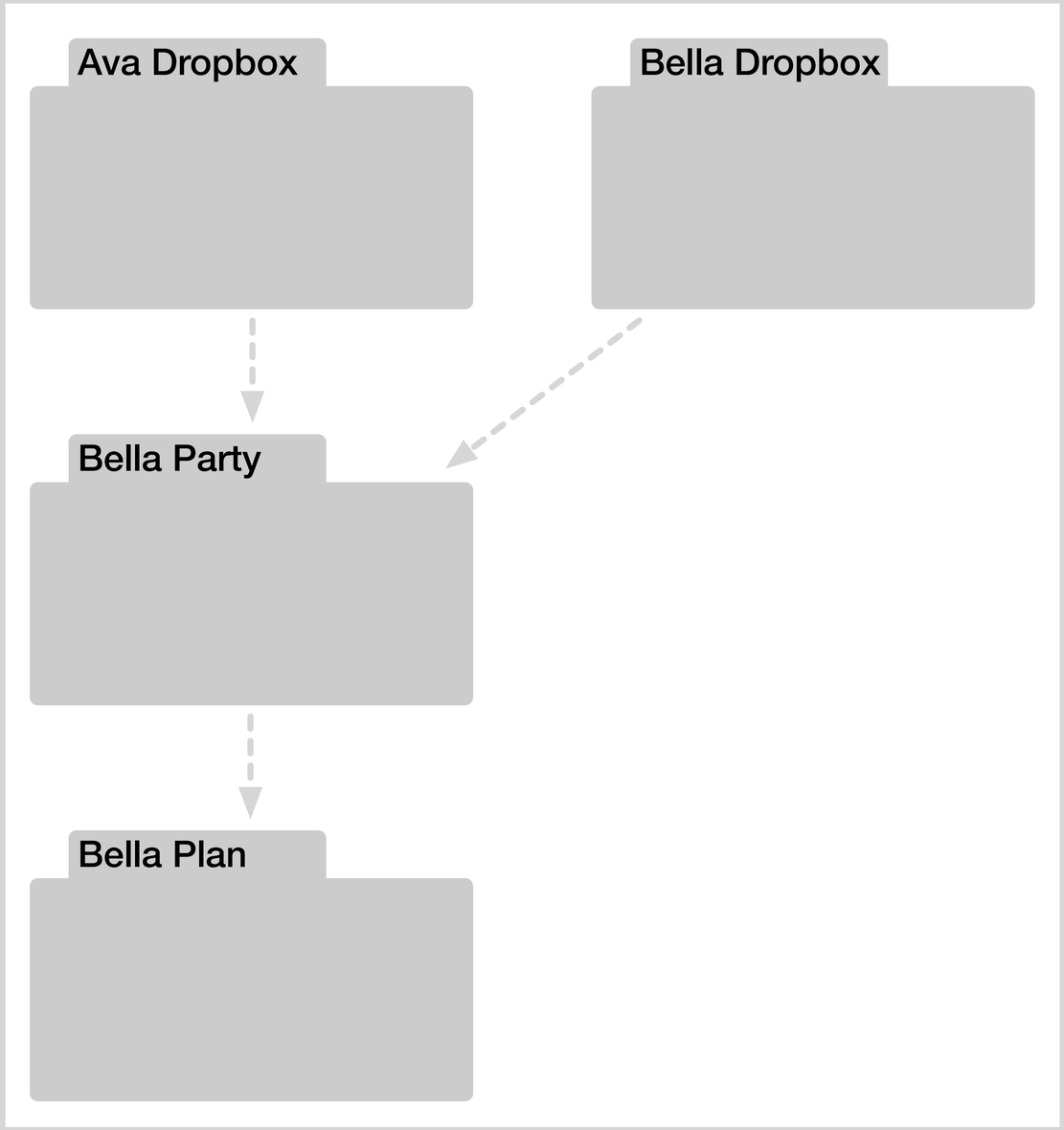
actions

join (user: User, chat: Chat)
requires no existing membership for user
ensures
  adds membership for user with no sent/received messages

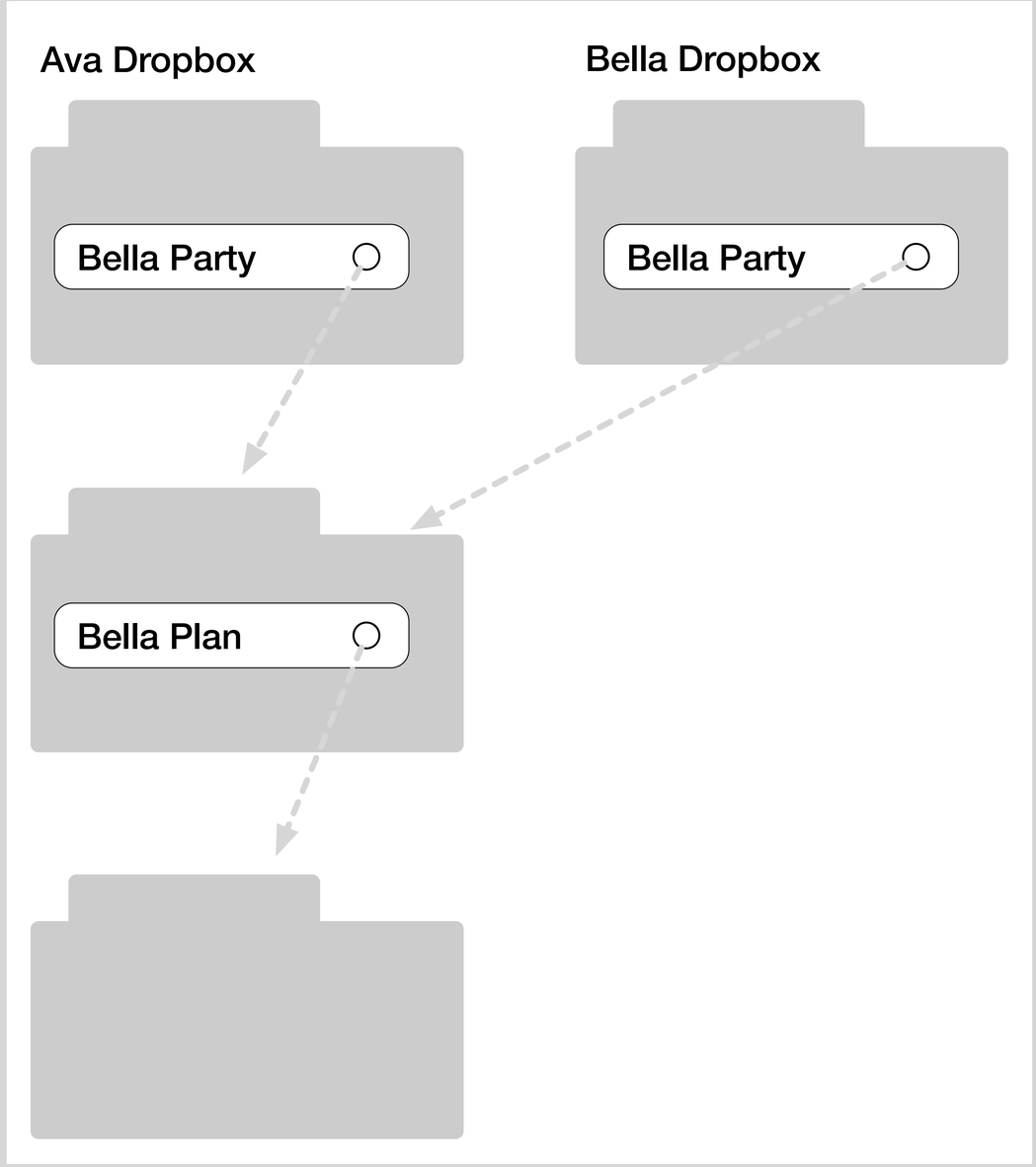
post (user: User, chat: Chat, text: String)
requires user is a member of the chat
ensures
  adds a message from user with given text
  to this user's sent messages, and to the received messages
  of all other users who are currently members of the chat

deleteForMe (user: User, message: Message, chat: Chat)
requires message is in user's sent messages for chat
ensures
  removes message from user's sent messages

folder concept
in Unix



how many users believe the folder concept works

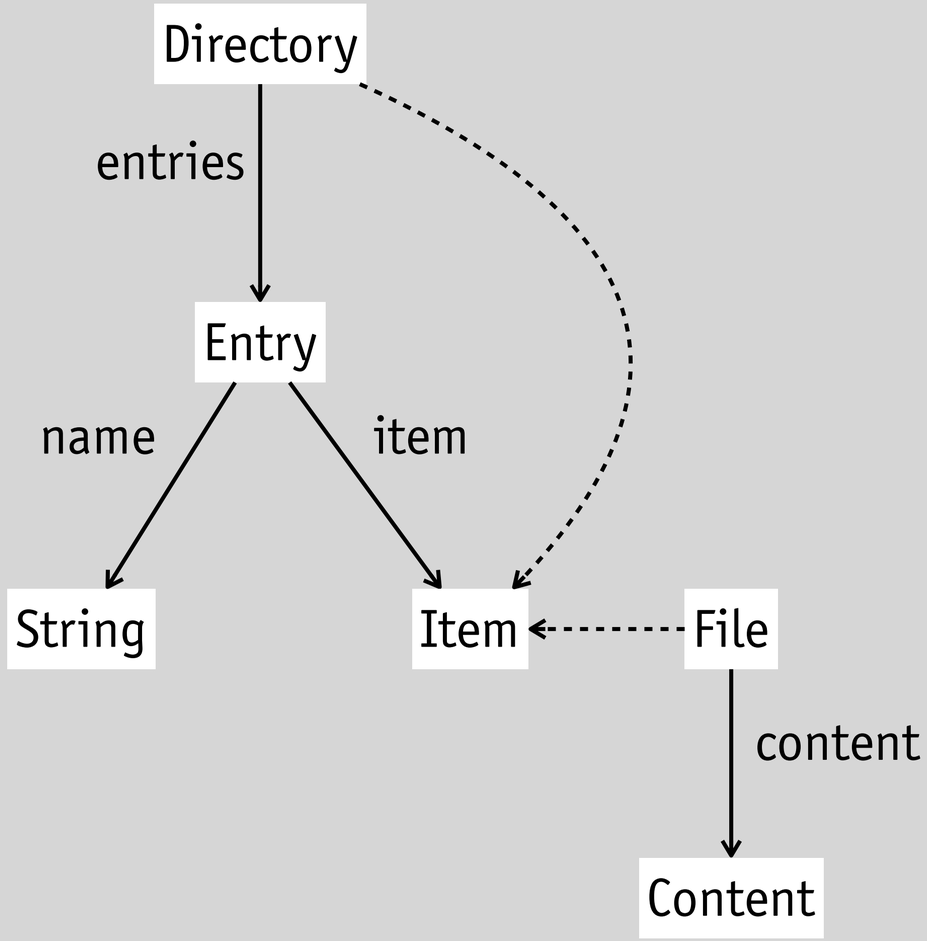


how folders actually work (in Dropbox, Unix, Multics)

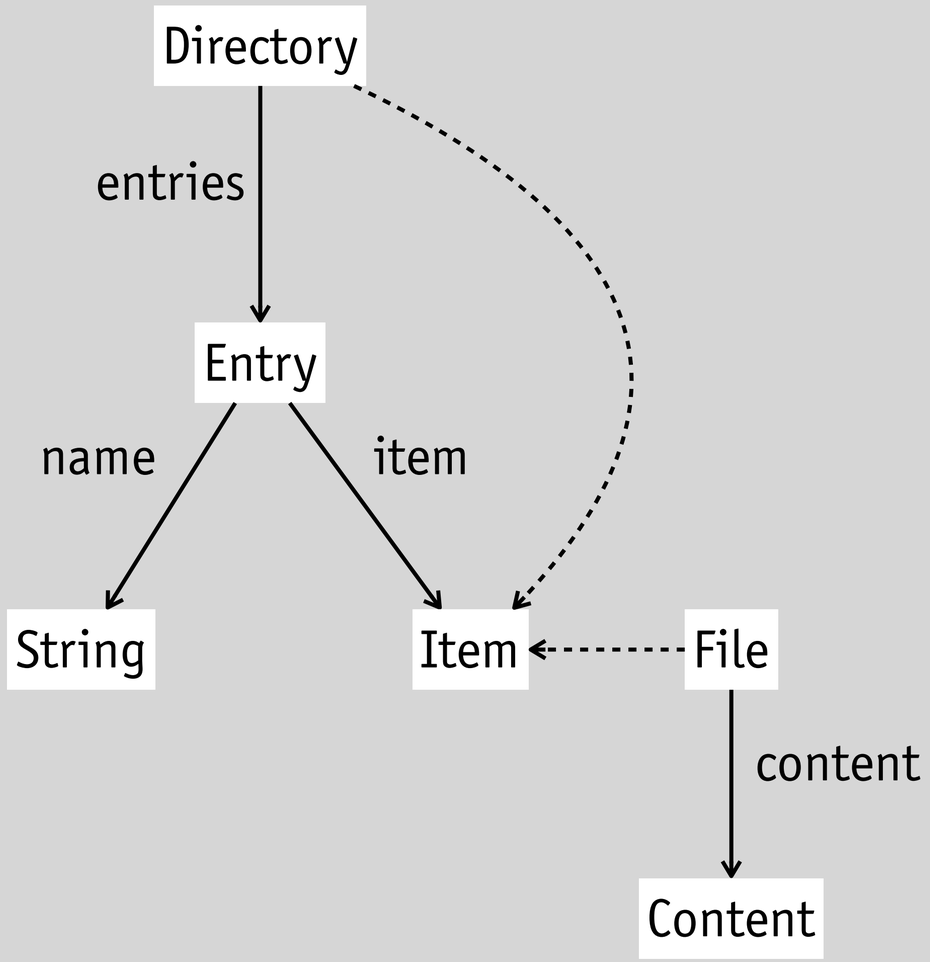
directories in unix

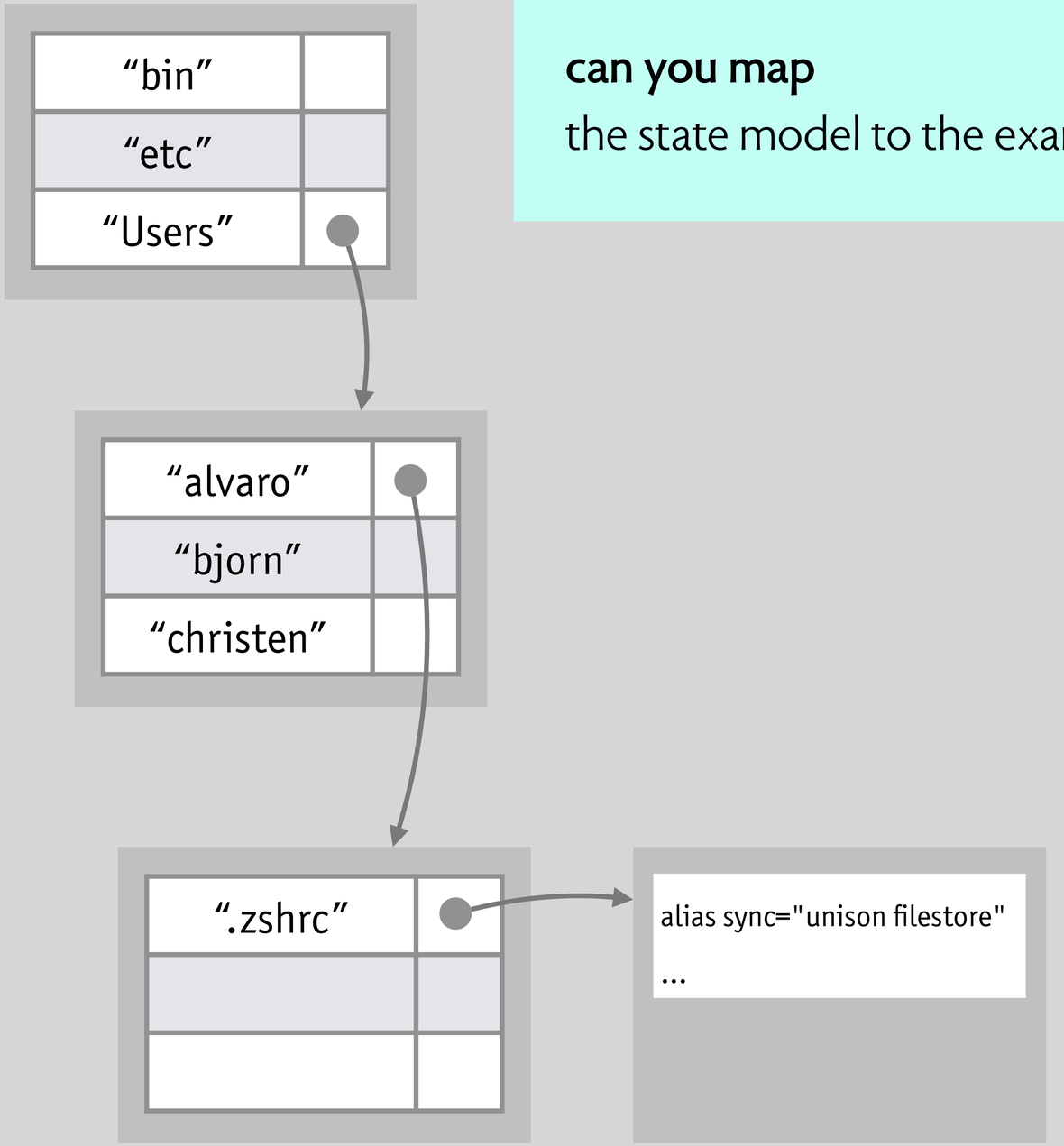
concept UnixDirectory

state
a set of directories
for each directory
  a set of entries
for each entry
  a name
  an item (file or directory)
for each file
  the content



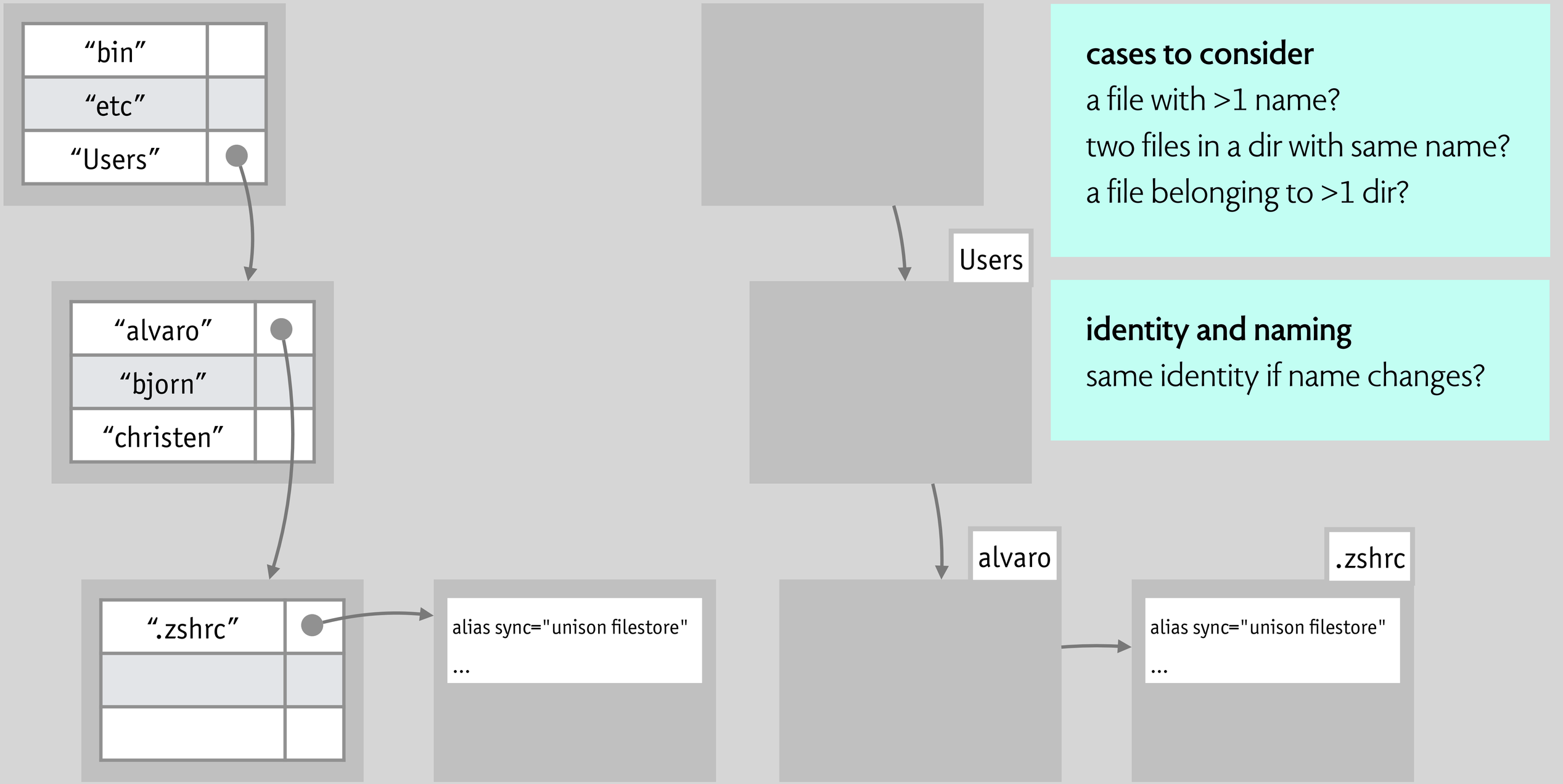
an example



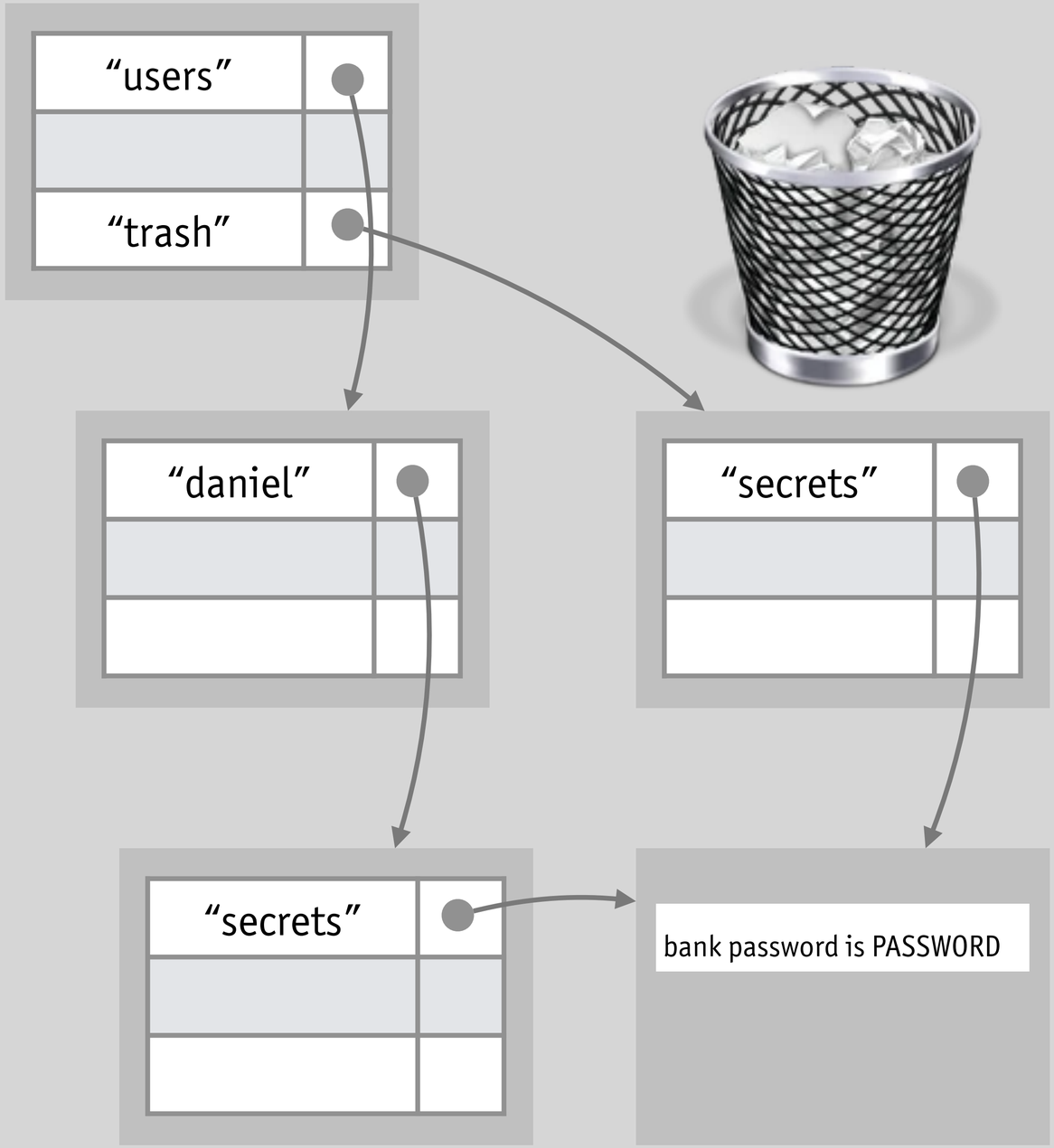


can you map
the state model to the example?

an alternative design



a unix puzzle: what happens when trash is emptied?



check your understanding

Which is true of the Unix directory concept?

(a) The name of a file is one of its (modifiable) metadata properties

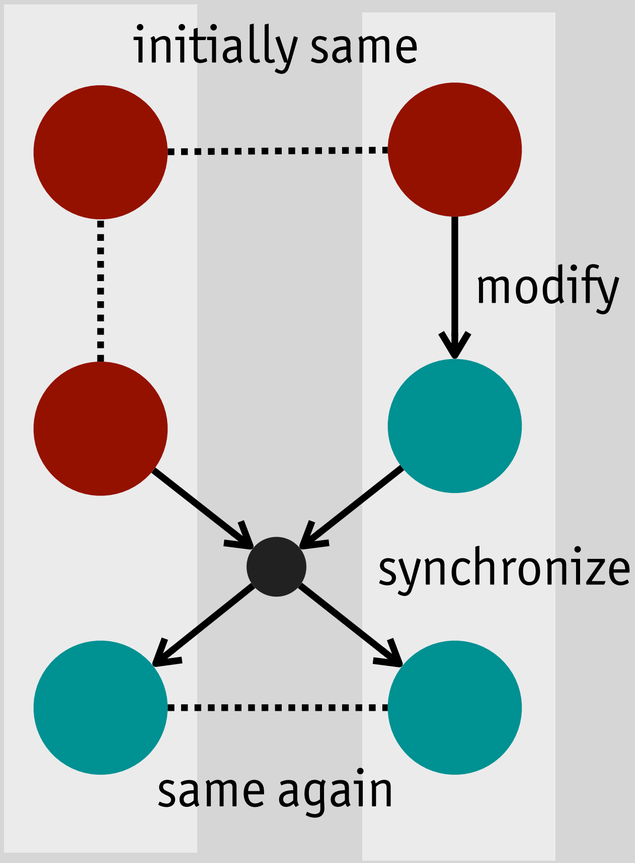
(b) Every file or directory has a single, unique pathname

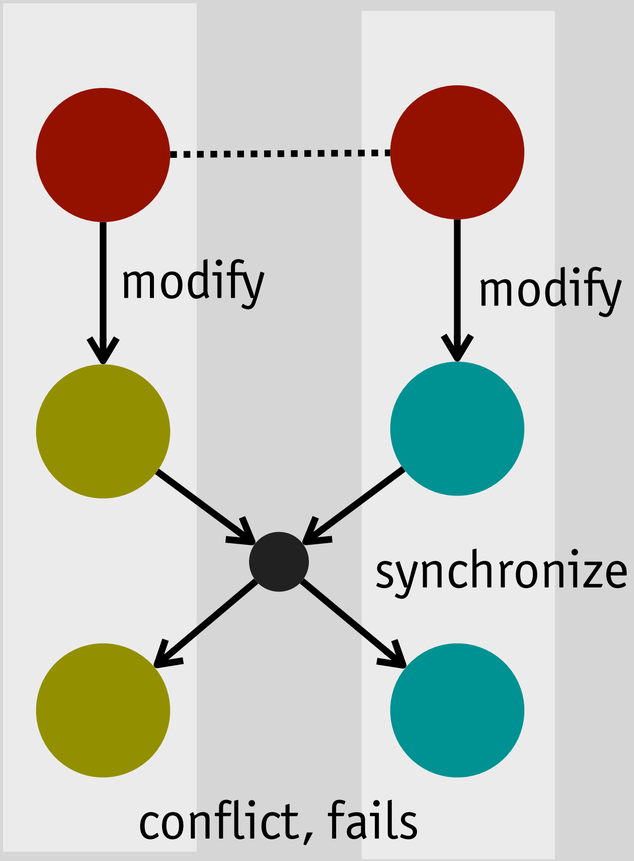
(c) Deleting a file removes a directory entry, not the file itself

file sync concept

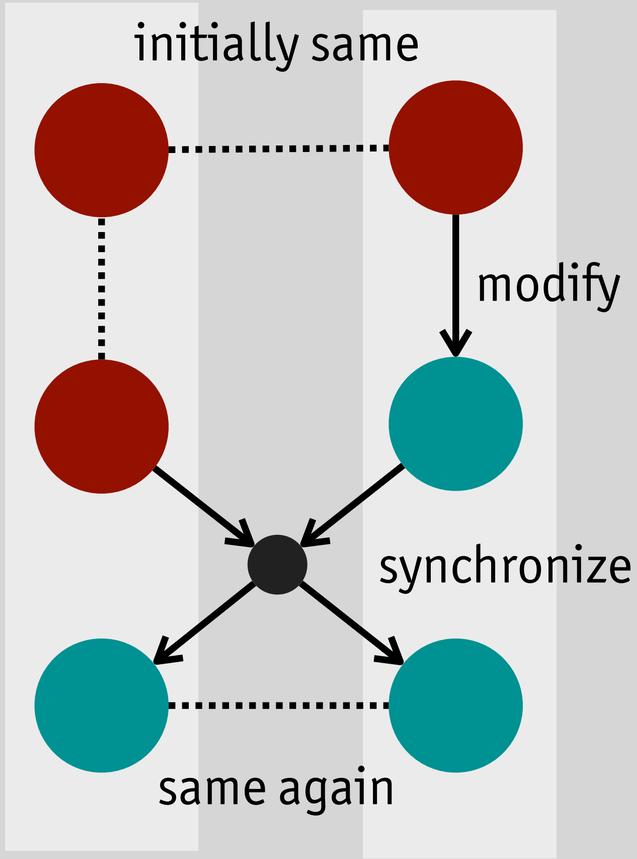
Box, Drive, etc

file synchronization concept





file sync concept state & actions



concept FileSynchronizer

state
a set of filenames
for each filename
 the previous contents of the file
 the contents in system A
 the contents in system B

state is distributed

why is this impractical?
storage? computation?

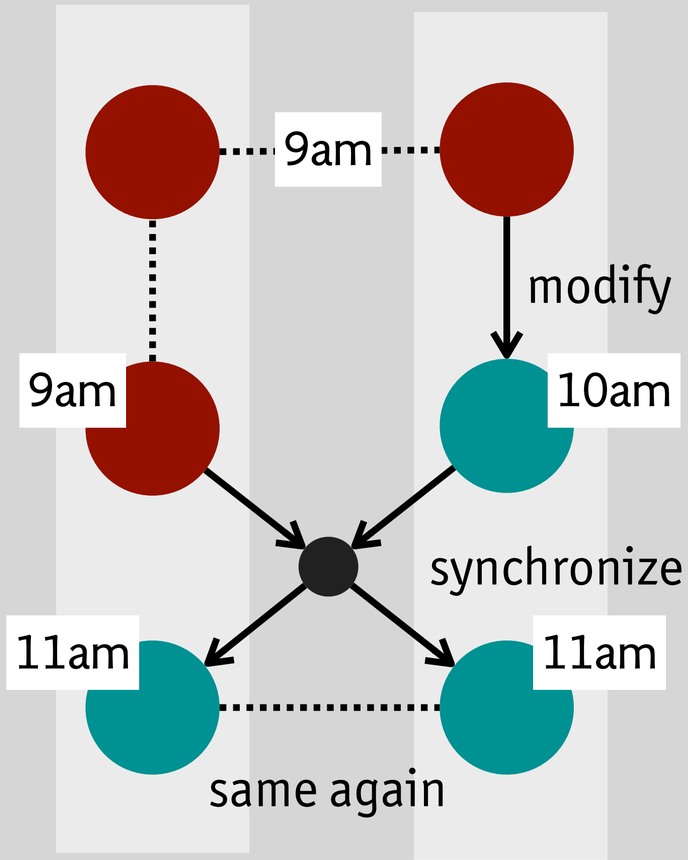
actions

modify (system: System, name: Name, contents: Contents)

synchronize (name: Name) : {success, conflict}
ensures
 if returns *success* and contentsA (name) = previous (name)
  then contentsA (name) := contentsB (name)
 elseif returns *success* and contentsB (name) = previous (name)
  then contentsB (name) := contentsA (name)
 else returns *conflict*

partial spec

an implementation



concept FileSynchronizer

state
a set of filenames
the date of the last sync
for each filename
 the contents in system A
  the date last modified in A
  the contents in system B
  the date last modified in B

state
a set of filenames
for each filename
  the previous contents of the file
  the contents in system A
  the contents in system B

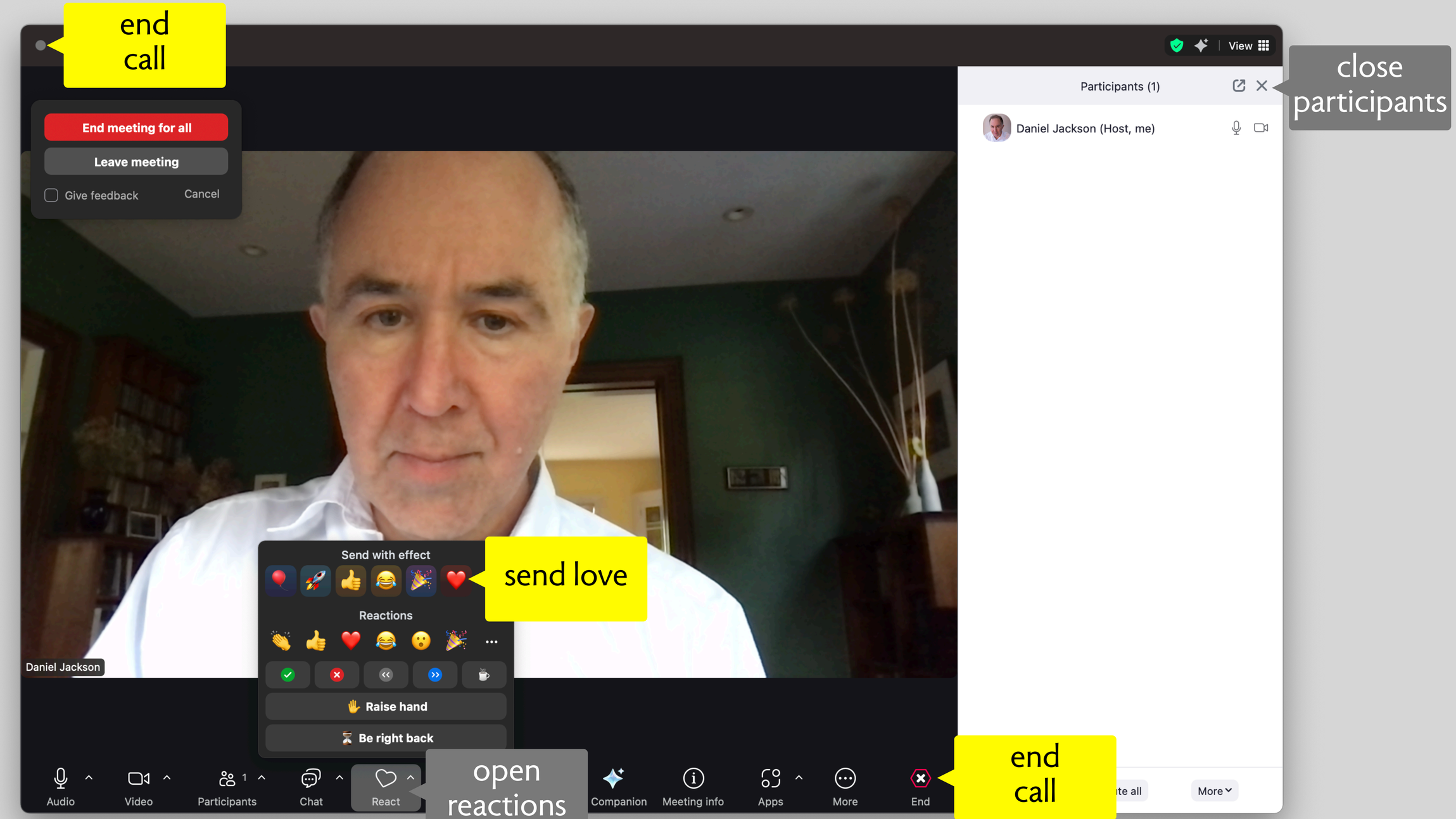
abstract state

actions

synchronize (name: Name) : {success, conflict}
ensures
 if B modified after last sync and A not modified after last sync
   then contentsA (name) := contentsB (name); return *success*
 elseif A modified after last sync and B not modified after last sync
   then contentsB (name) := contentsA (name); return *success*
 else returns *conflict*

heuristics
for states & actions

not all user interface “actions” are concept actions



do you have enough actions?

is purpose/value delivered?
note that being in the state may be enough

have you covered the whole life cycle?
is there an initial setup? a winding down?

are there ways to undo previous actions?
or to compensate if they were erroneous?

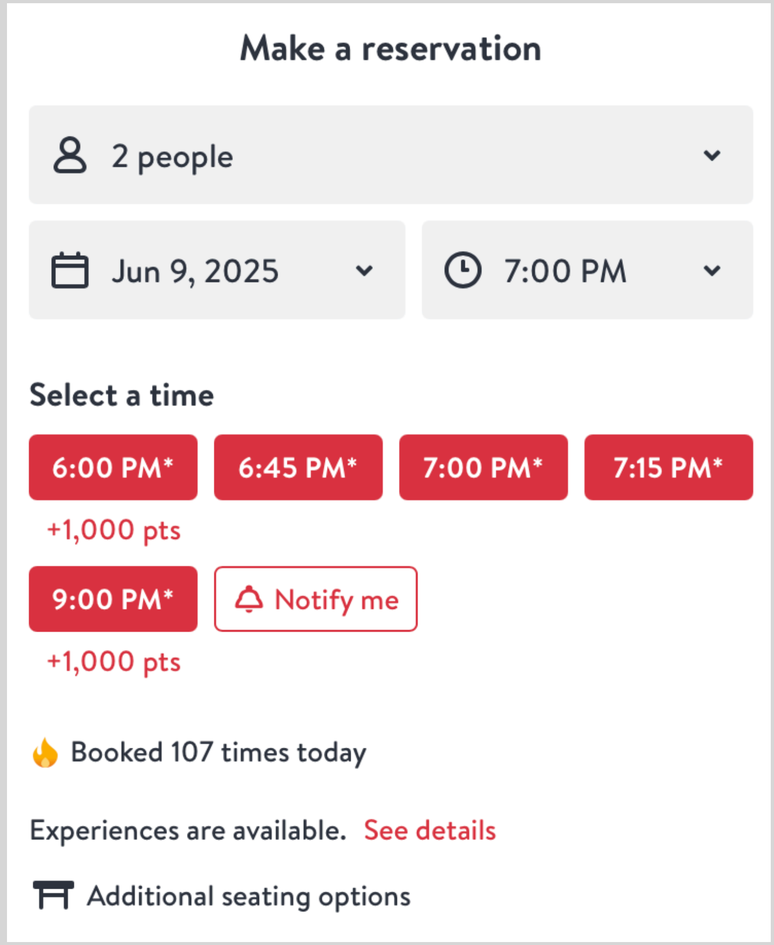
do all objects have create, update, delete?
for associated state, not literally objects

seat

(set availability)

unseat party
cancel reservation

change reservation



concept Reservation
actions reserve...

applying action heuristics to GroupChat

is purpose/value delivered?
note that being in the state may be enough

have you covered the whole life cycle?
is there an initial setup? a winding down?

are there ways to undo previous actions?
or to compensate if they were erroneous?

do all objects have create, update, delete?
for associated state, not literally objects

create group
delete group

delete post
leave group

edit post
add member
remove member

concept GroupChat
actions
join group
post message

applying action heuristics to FileSync

is purpose/value delivered?
note that being in the state may be enough

have you covered the whole life cycle?
is there an initial setup? a winding down?

are there ways to undo previous actions?
or to compensate if they were erroneous?

do all objects have create, update, delete?
for associated state, not literally objects

first time sync
disconnect?

revert?

create file or folder
delete file or folder

concept FileSync
actions
modify file
synchronize

do you have a rich enough state?

can you support all your actions?
determine if allowed, and generate results

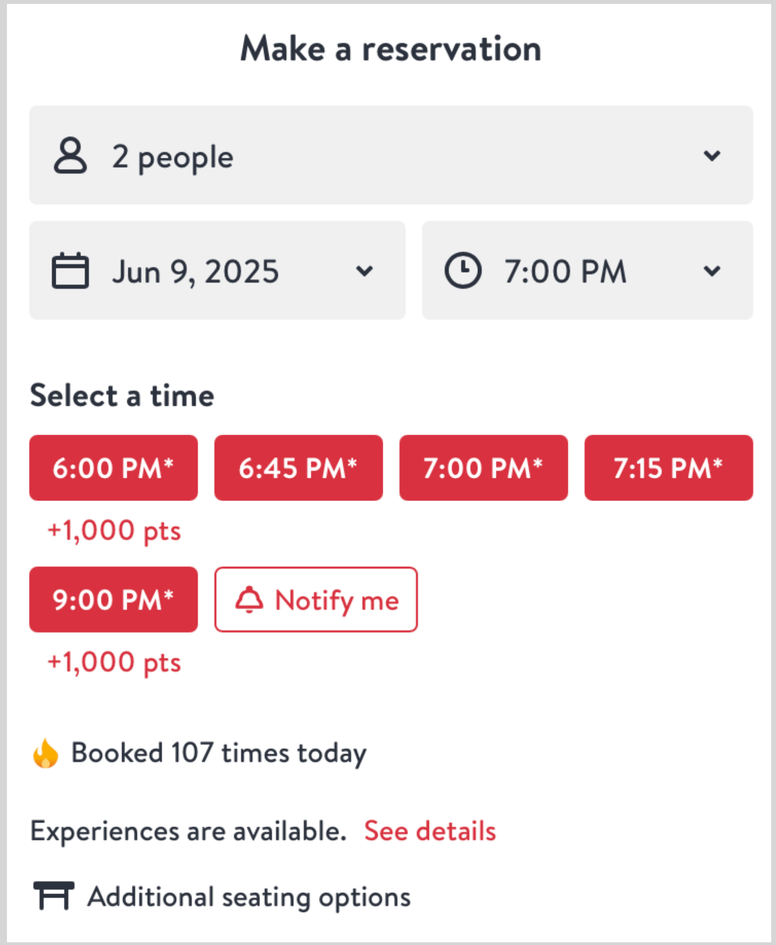
should you track history?
remember completions, deletions, undos?

what info about action occurrence?
maybe also who did it? when?

table sizes

retain after seat?

by vs. for?
time of reservation?



concept Reservation
actions createSlot, reserve, cancel, seat, unseat, no-show, ...

takeaways

what you learned today

what you learned today

state machines
how do model behavior with states and actions
a new take on data models: partitioned and action-driven

how detailing behavior helps
raises tricky design questions
exposes complexities that may confuse users
helps you explore the entire design

what I hope you can now do

design concepts in detail
with states and actions

produce behavior outlines
with data model diagrams & action lists

what's next?

what's next?

homework #1: post to our Slack group
what one idea did you find most useful, surprising, confusing?

homework #2: post to our Slack group
a state+action model of a concept, from Autodesk or not
(no need to finish it: just make a start so we can see where it's going)
or, apply heuristics to an Autodesk concept in the sandbox

plan for last session
how to break a system into concepts
modularity, purpose and synchronization