

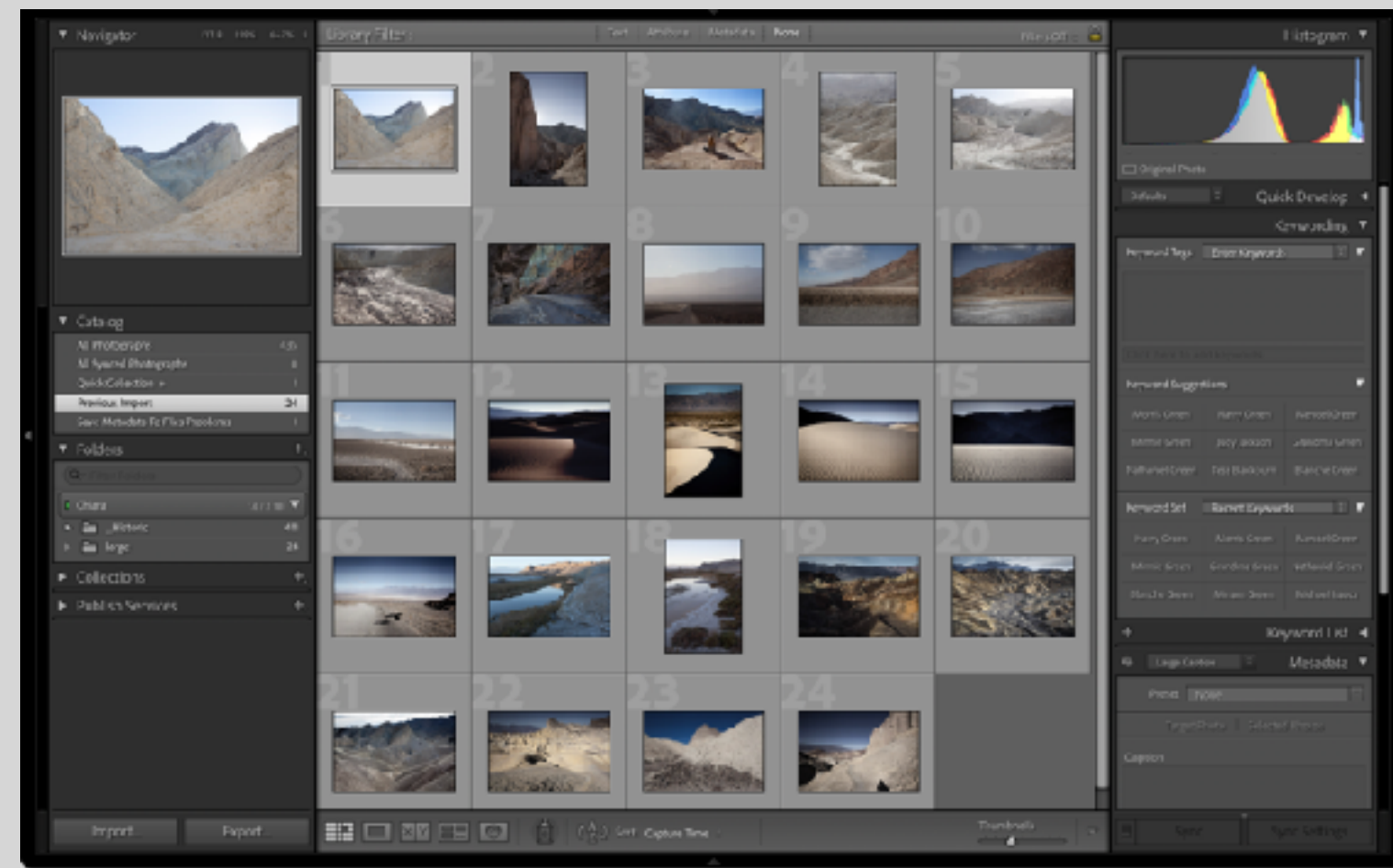
6.1040 · software studio · fall 2023

concept design basics

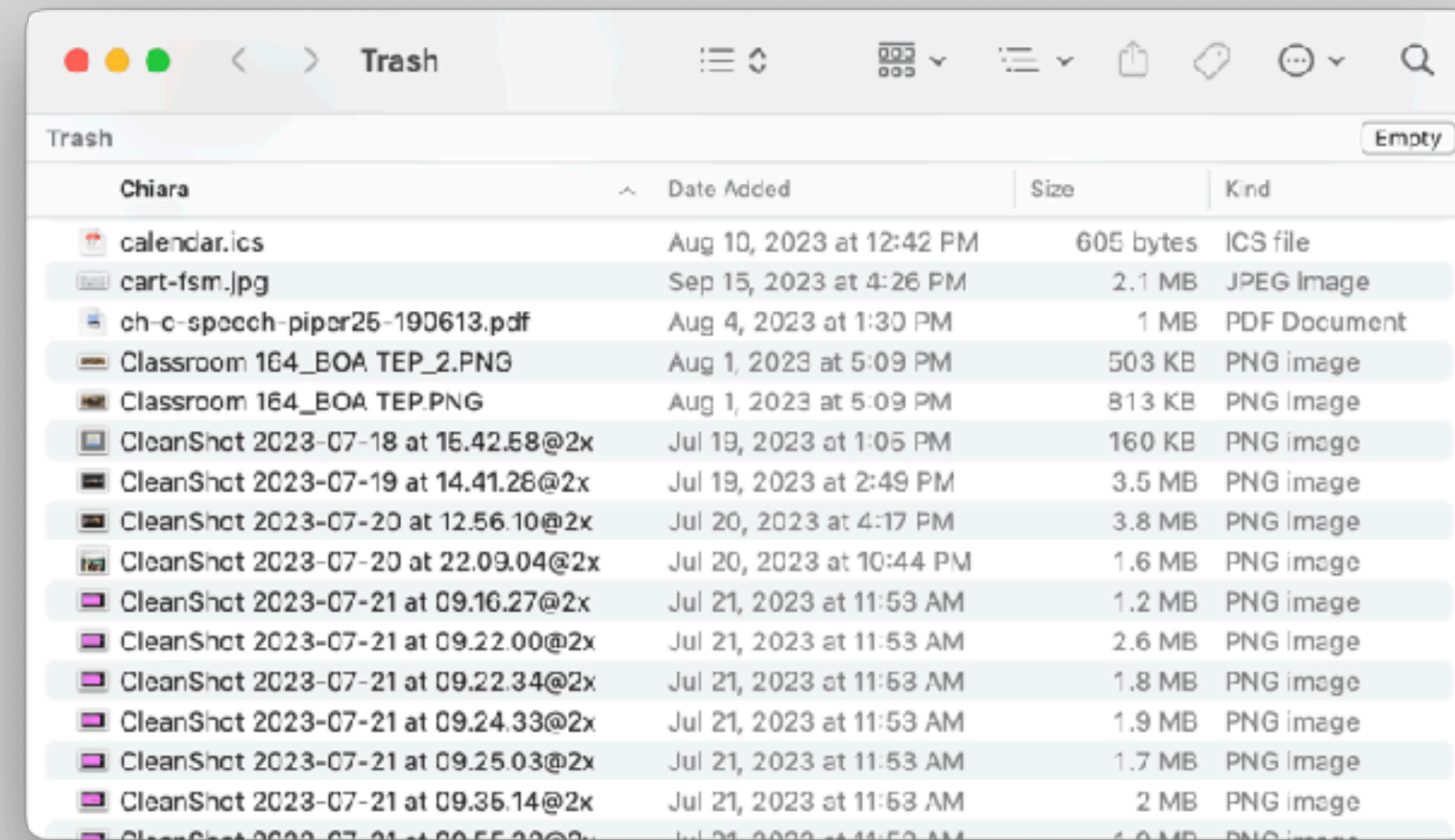
Daniel Jackson & Arvind Satyanarayan

how can you design
really great software?

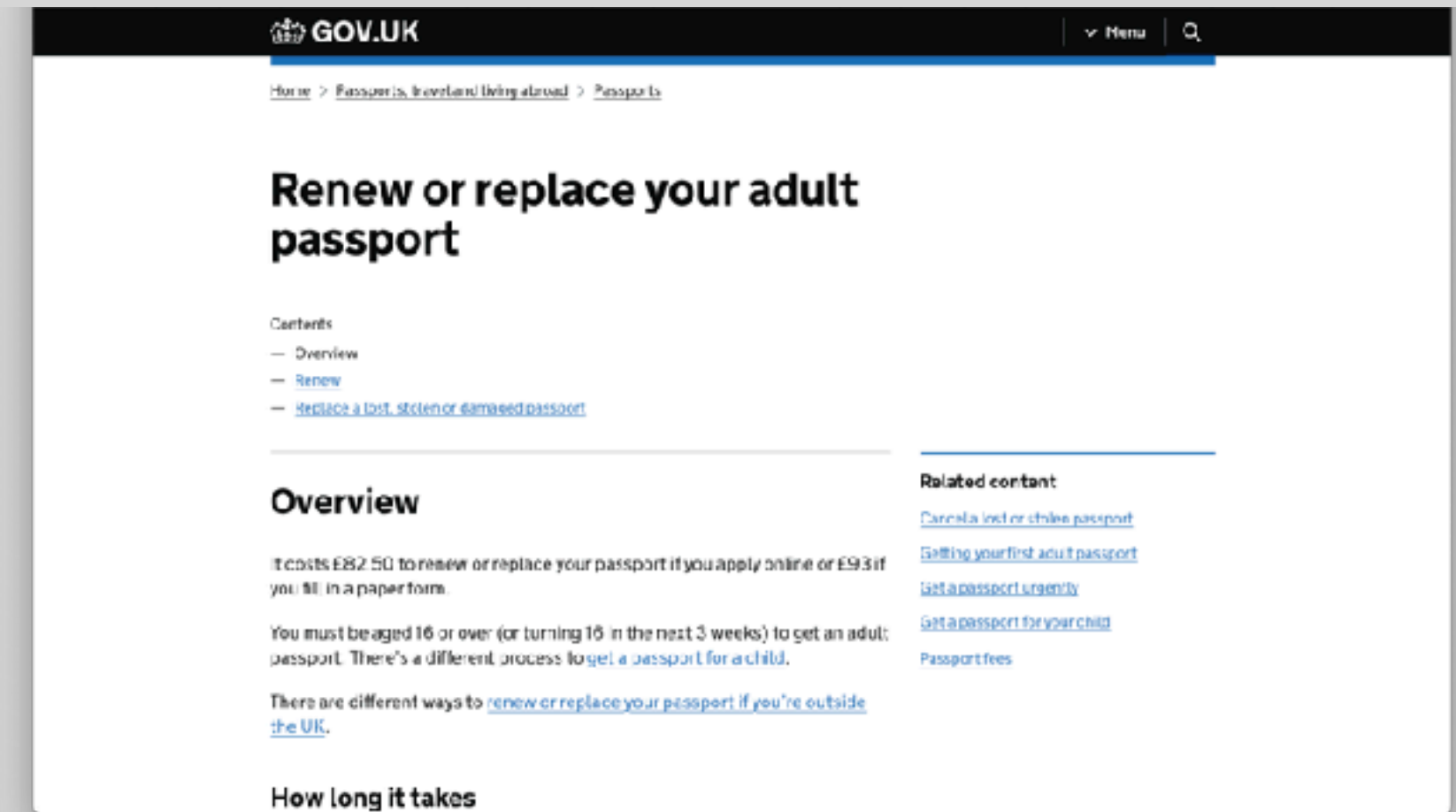
three examples of insanely good design



Adobe Lightroom



Apple MacOS Finder



UK Government Services



When you go to design a house you talk to an architect first, not an engineer. Why is this?

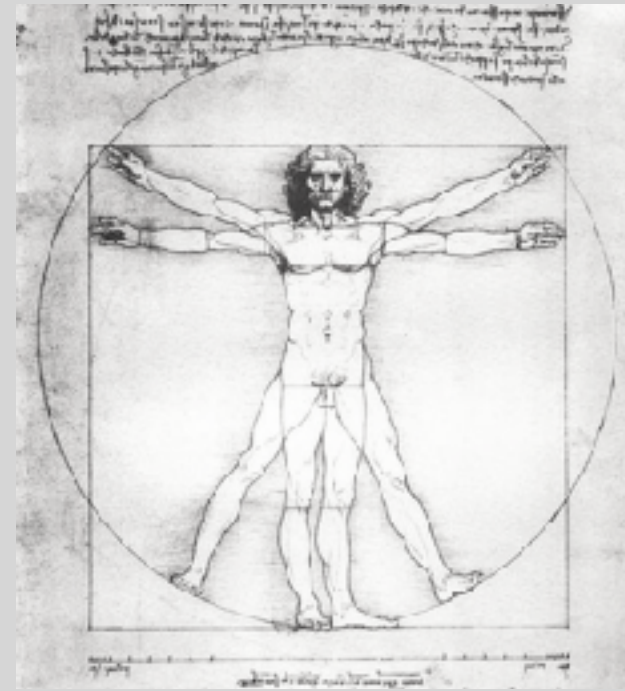
Because the criteria for what makes a good building fall outside the domain of engineering.

Similarly, in computer programs, the selection of the **various components** must be driven by the conditions of use.

How is this to be done? By software designers.

Mitchell Kapor, *A Software Design Manifesto* (1996)

levels of design



physical

color, size, layout,
type, touch



linguistic

icons, labels, tooltips,
site navigation

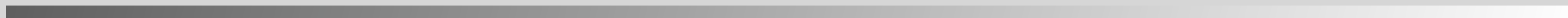


conceptual

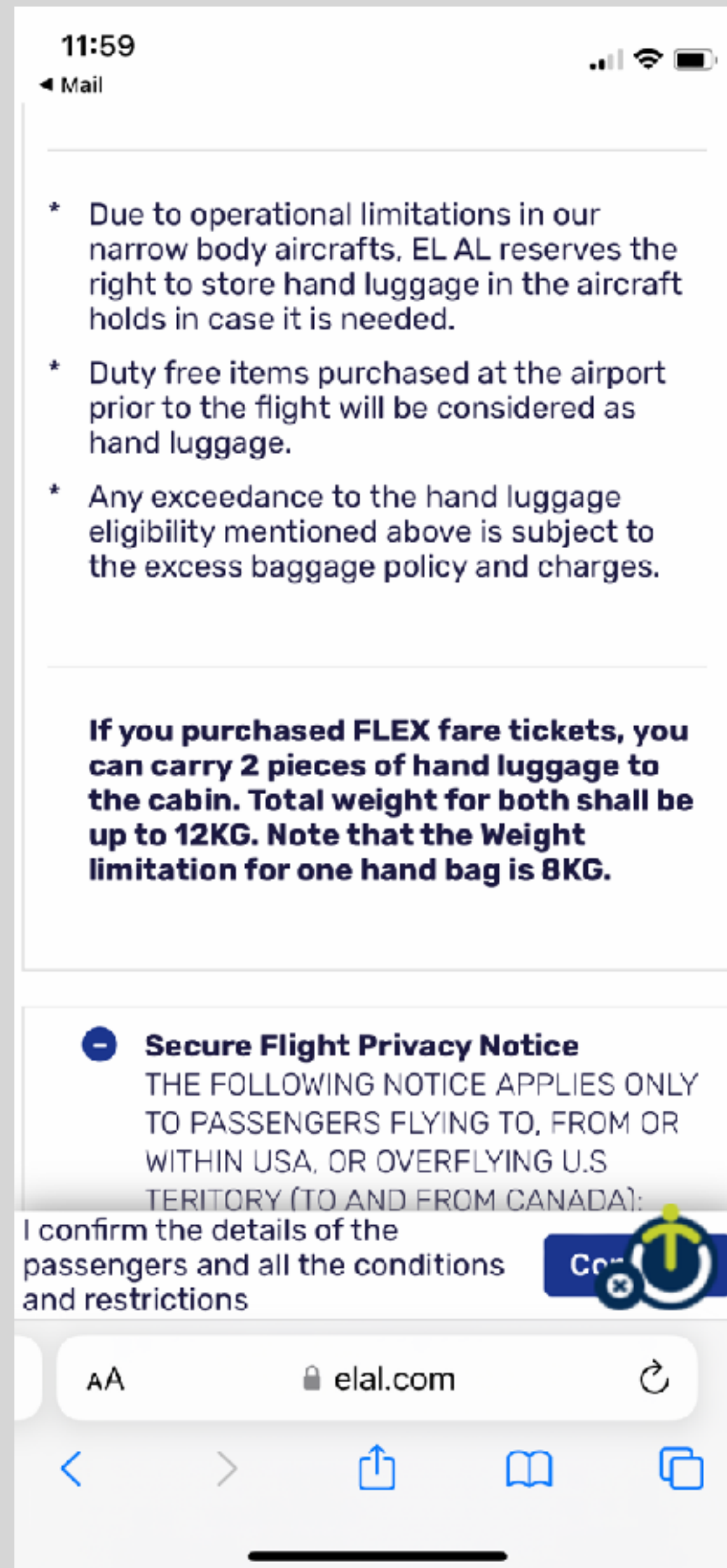
semantics, structure
& behavior

concrete

abstract



confirming checkin details on El Al














Select passengers for checkin

Daniel Jackson

continue


replacing a manuscript on Arxiv

Articles You Own  Replace  Withdraw  Cross list  Journal ref  Annotate  Link code & data				
Identifier	Primary Category	Title	Actions	Author
2304.14975	cs.SE	Concept-centric Software Development	     	Y

backing up on Backblaze

Backblaze

dnj@mit.edu



You are backed up as of: 5/17/23, 4:26 PM

Please Wait

Restore Options...

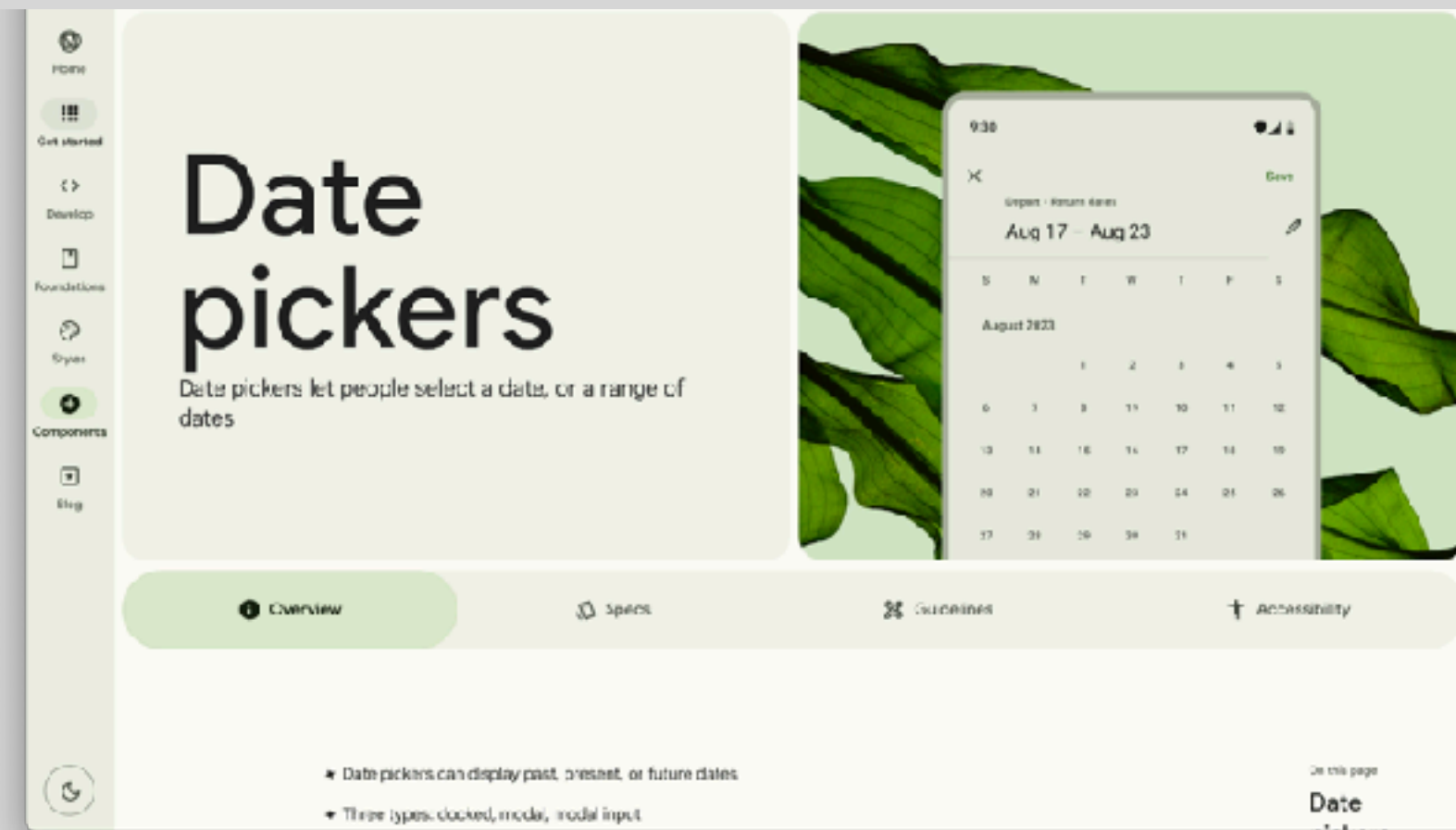
Settings...

Selected for Backup: 916,605 files / 211,505 MB
Backup Schedule: Continuously
Remaining Files: 916,605 files / 211,505 MB

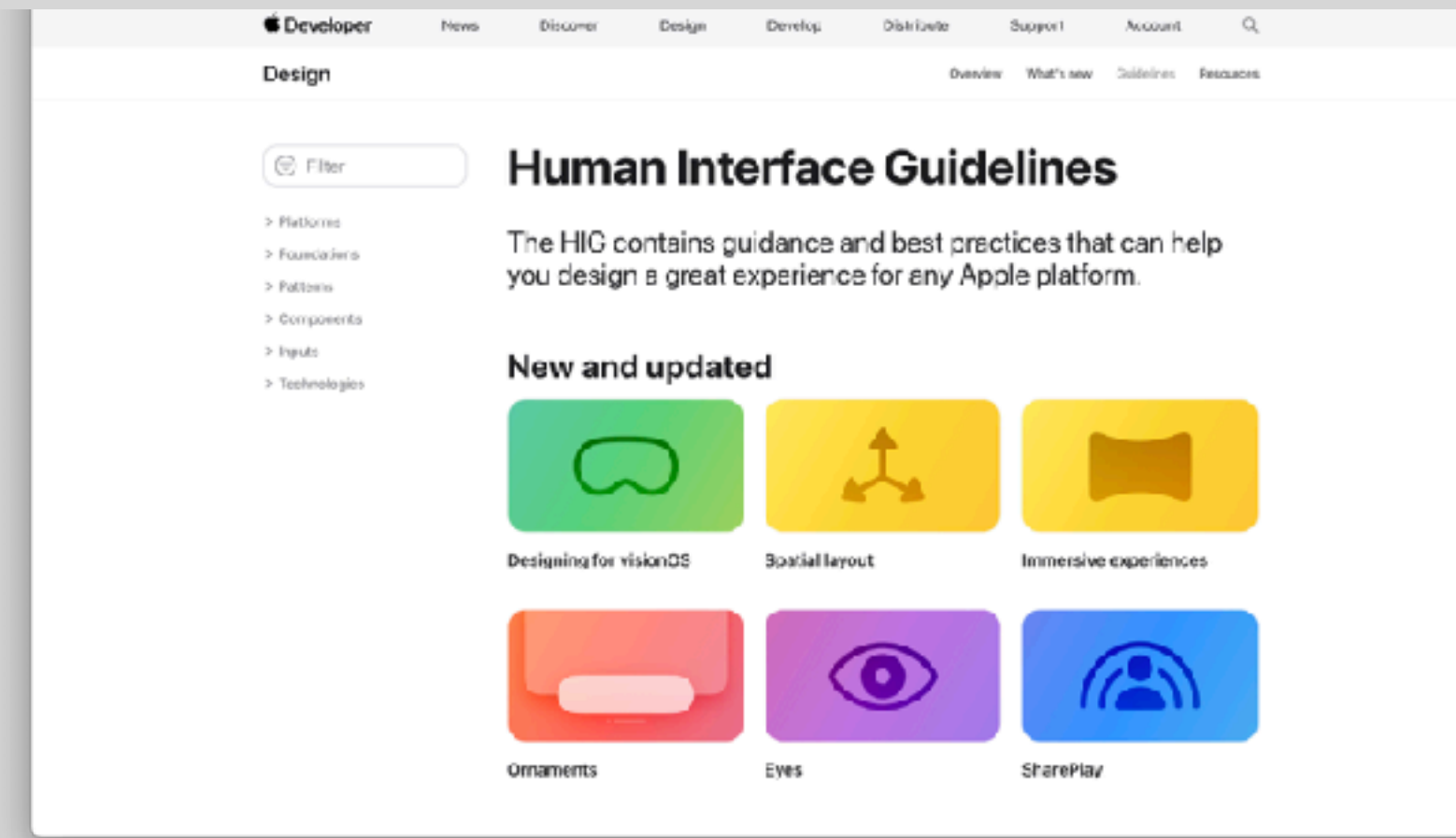
Version History: 30 days [Upgrade](#)
Manage account at [Backblaze.com](#)
Questions? [Help Center](#)

Your data is NOT backed up. [Buy](#) [Already bought?](#) ?

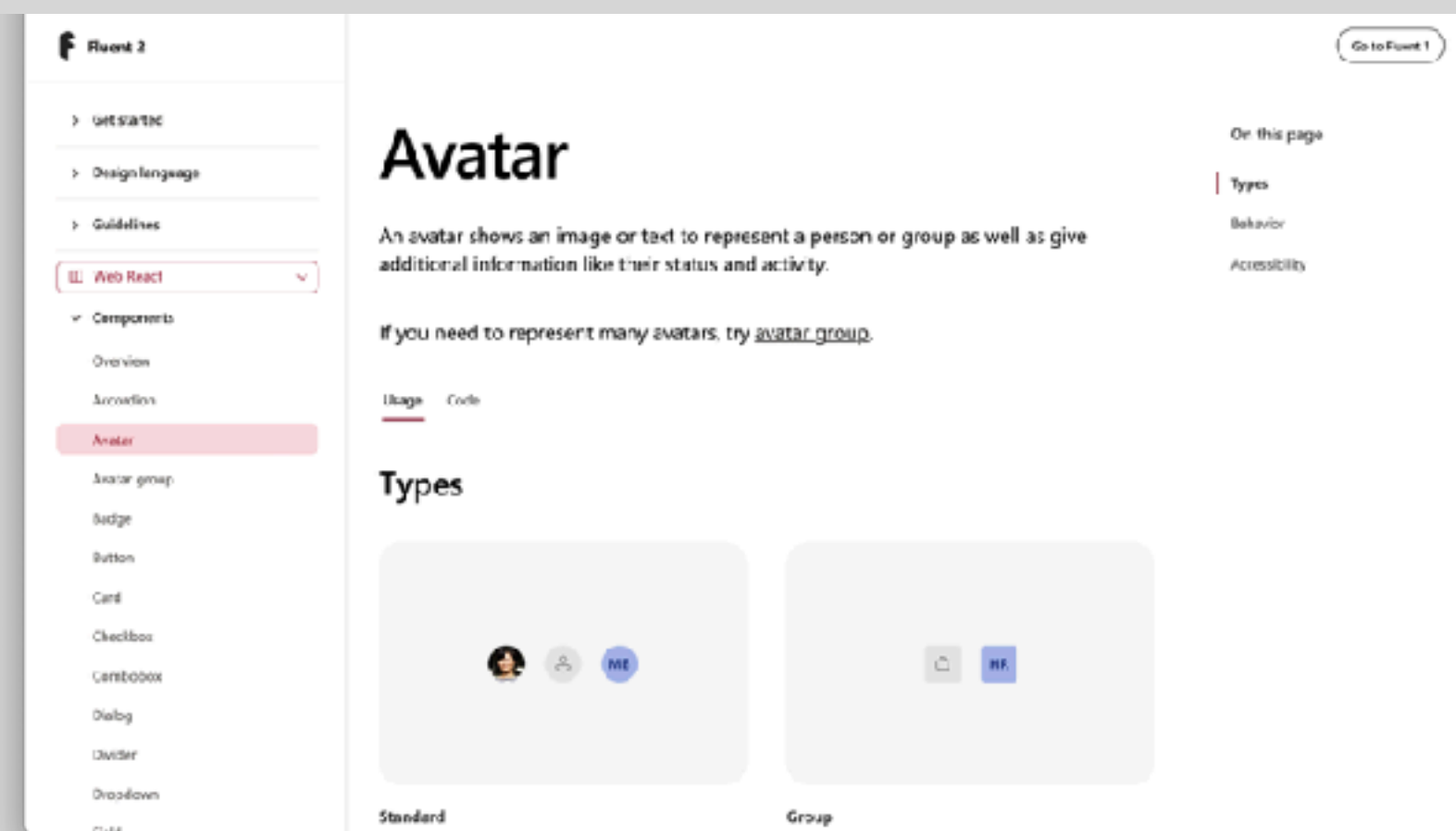
design systems



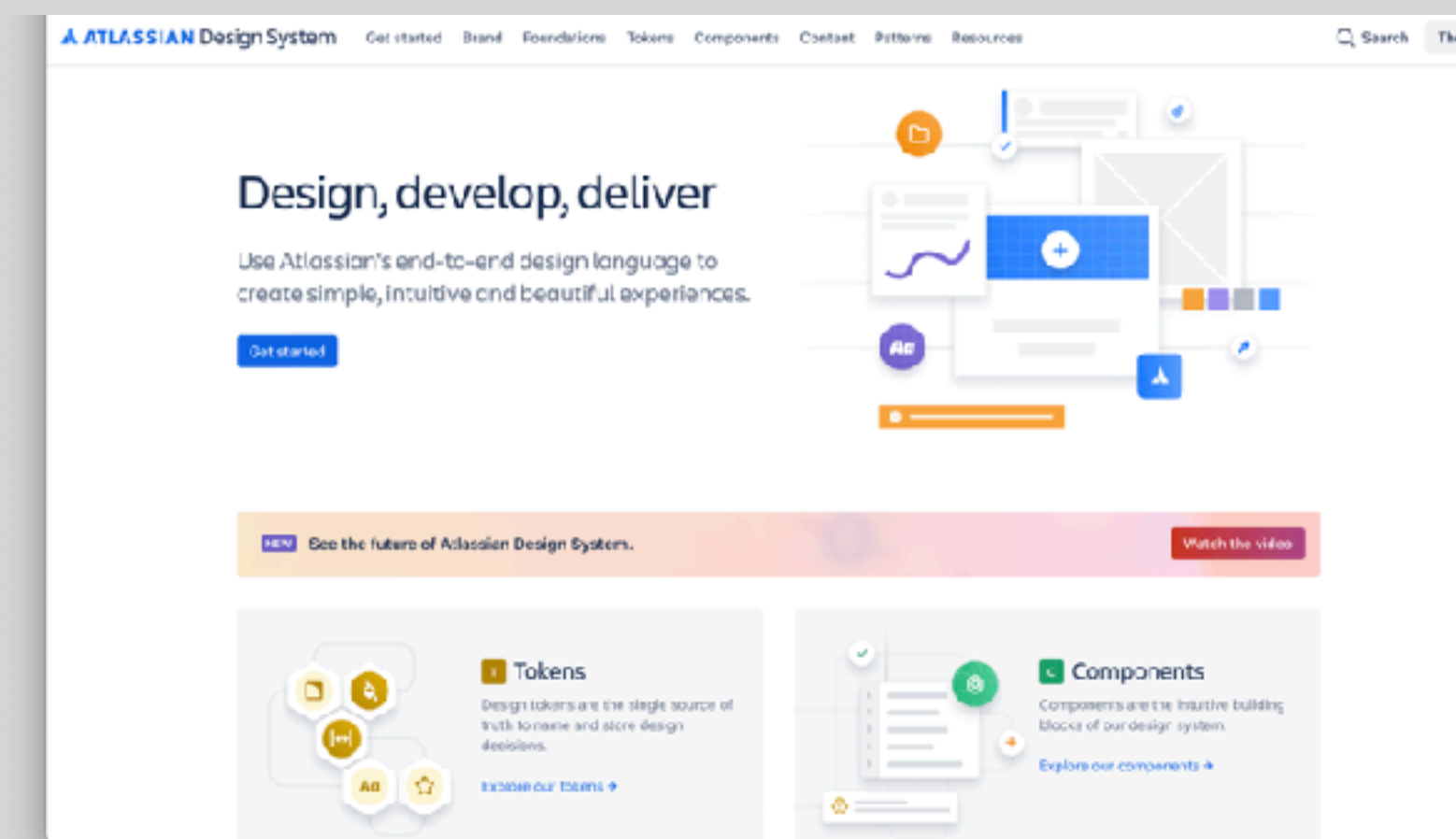
Google's Material Design



Apple's Human Interface Guidelines



Microsoft Fluent Design



Atlassian Design System

today's learning objectives

recognize levels of design in software

understand role of conceptual models

know how to model concepts as state machines

understand idea of factoring concepts into *patterns*

use synchronization to compose concepts into an app

conceptual
models

example: backblaze backup

The screenshot shows the Backblaze Backup application window. At the top, there are window control buttons (red, yellow, grey) and navigation arrows. The title bar reads "Backblaze Backup" and there is a search bar on the right. The user's email address "dnj@mit.edu" is displayed in the top right corner. A status message with a checkmark icon says "You are backed up as of: 6/6/22, 10:10 PM" and "Currently backing up newer files". Below this, there are two buttons: "Pause Backup" and "Restore Options...". To the right of these buttons is a diagram showing a computer monitor icon with an arrow pointing to a cloud icon with a red flame, representing the backup process. Below the diagram is a "Settings..." button. On the left side, there is a list of backup statistics: "Selected for Backup: 509,021 files / 2,379,995 MB", "Backup Schedule: Continuously", "Remaining Files: 0 files / 0 KB", and "Transferring: photo.0259-22.RAF". At the bottom, there is a link to "View files and manage account at: Backblaze.com" and a calendar icon with "1Y" and a help icon with a question mark.

Backblaze Backup

Search

dnj@mit.edu

You are backed up as of: 6/6/22, 10:10 PM
Currently backing up newer files

Pause Backup

Restore Options...

Selected for Backup: 509,021 files / 2,379,995 MB
Backup Schedule: Continuously
Remaining Files: 0 files / 0 KB
Transferring: photo.0259-22.RAF

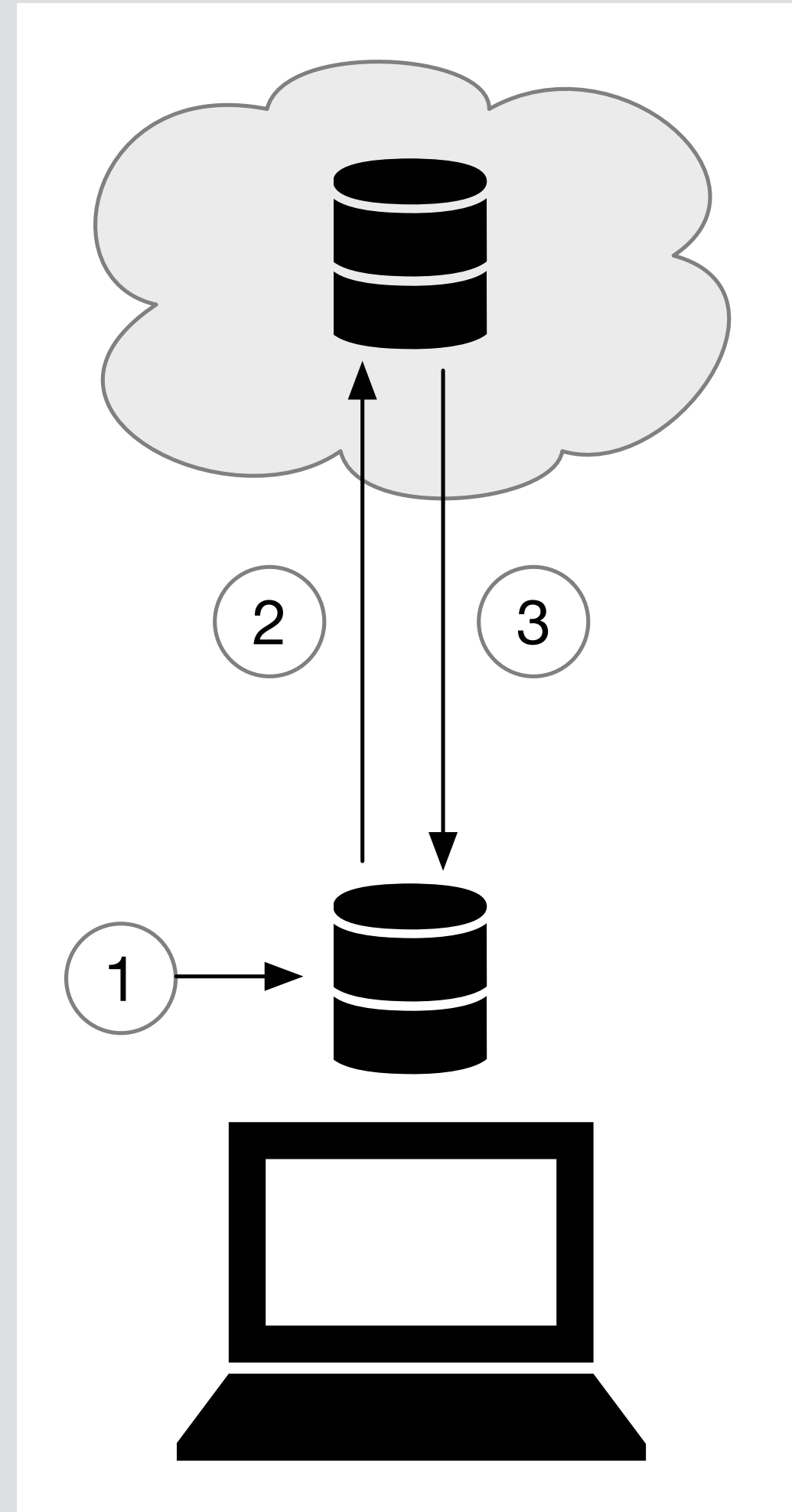
Settings...

What is being backed up?
How long will my first backup take?

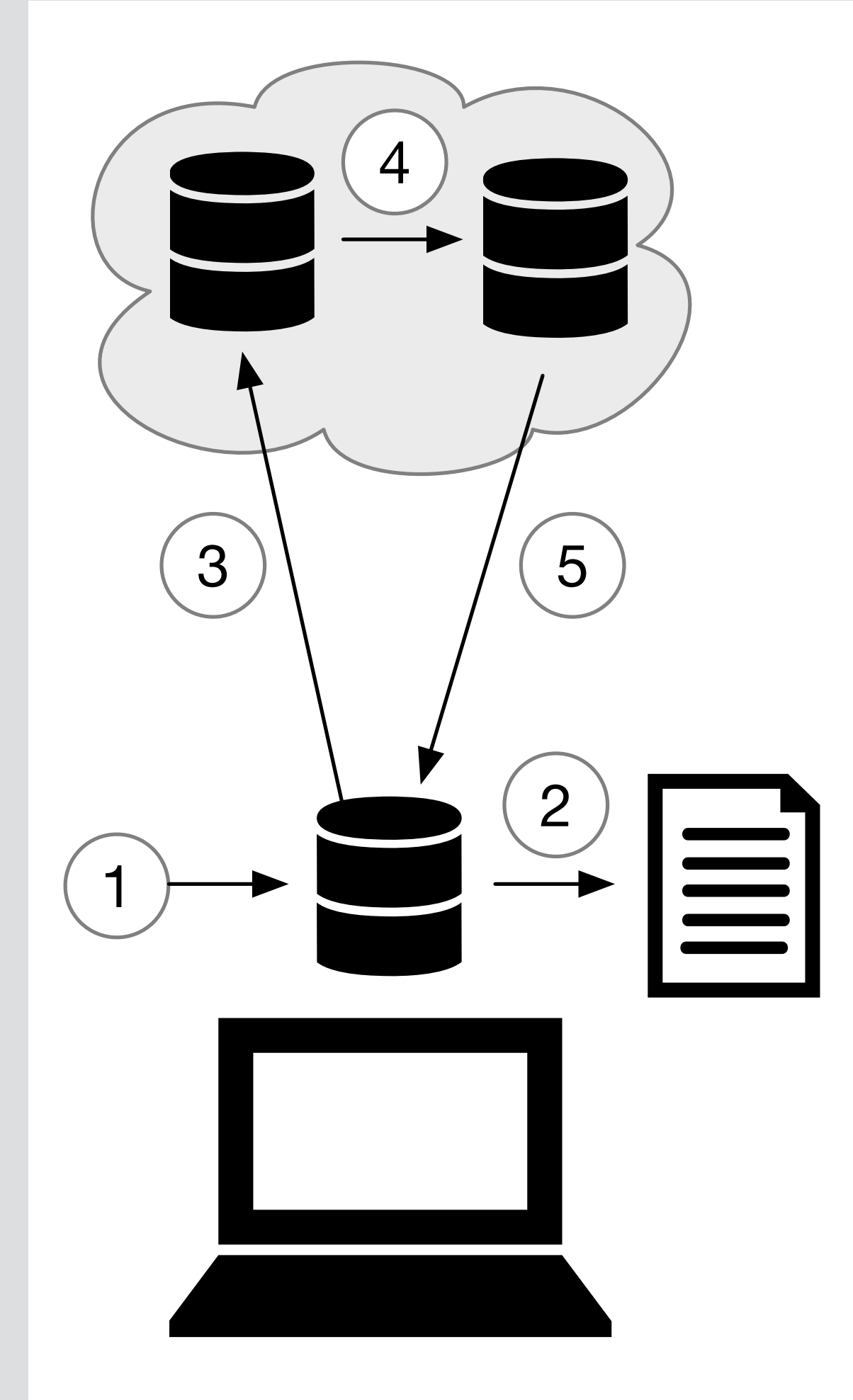
View files and manage account at: Backblaze.com

1Y ?

assumed vs. actual conceptual models

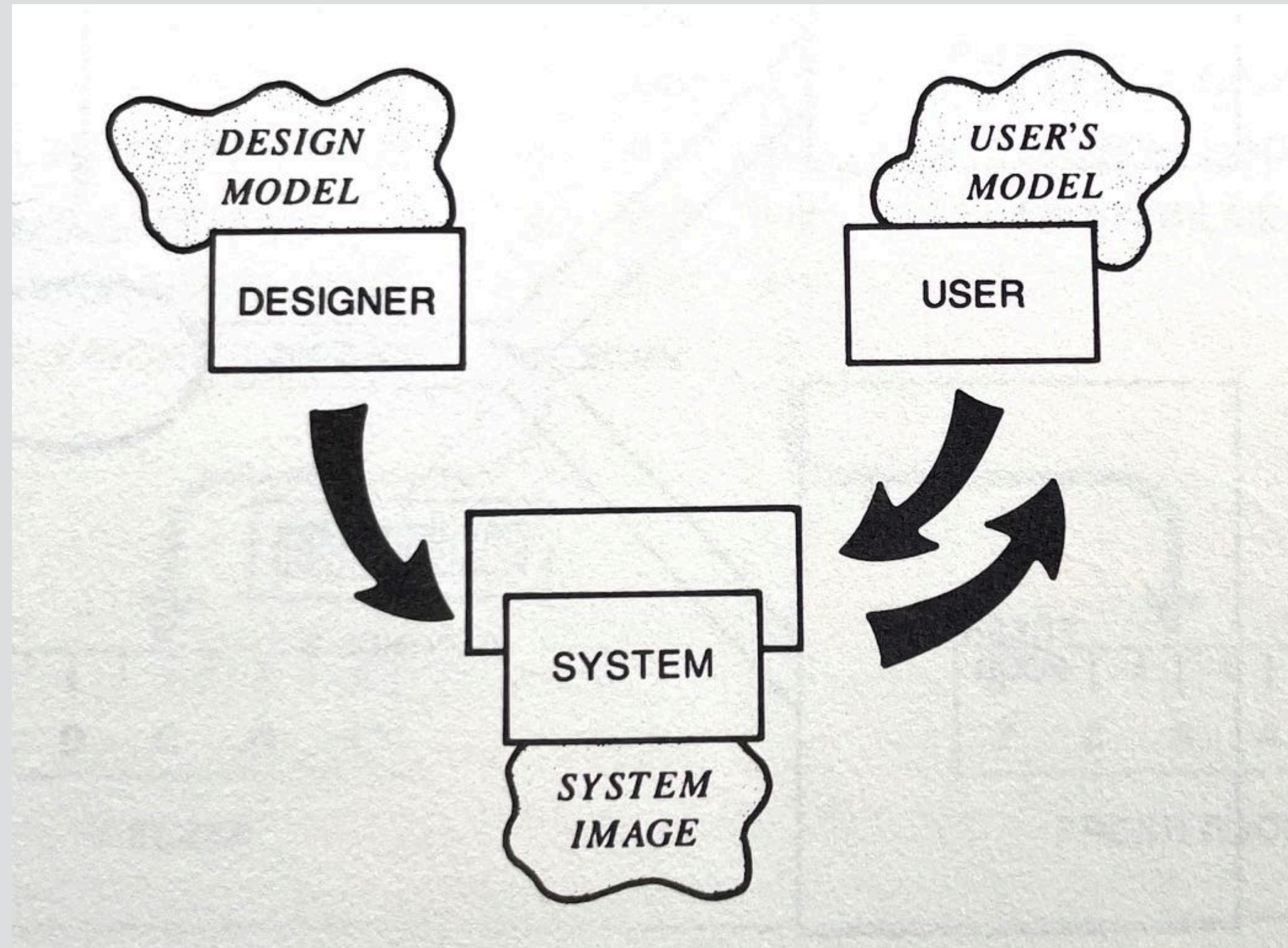


1. file modified
2. file backed up
3. file restored



1. file modified
2. list created
3. files backed up
4. files available
5. file restored

projecting an accurate system image



from *The Design of Everyday Things*

user-centered design (1980s)

concepts are a **byproduct** of design

designer's job: **shape UI** to project concepts

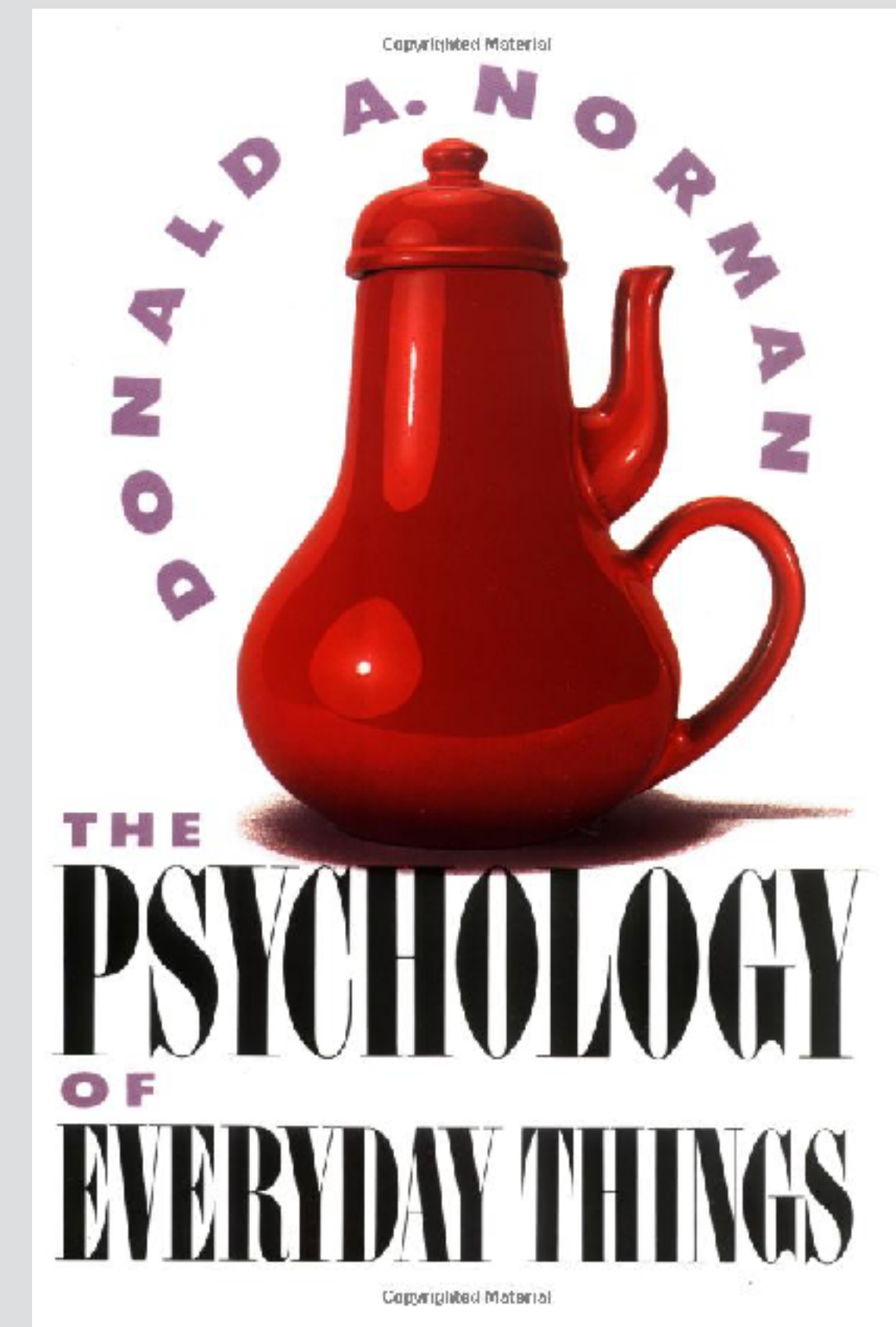
concepts are **psychological**

concept-based design

concepts are the **essence** of design

designer's job: **shape concepts**

concepts are **computational**



concepts as
state machines

yellkey: url to common word shortener.

yellkey
enter url and length of time for key to exist.

full url (e.g. http://www.google.com)

5 minutes

generate yellkey

your key is: **best.**
go to www.yellkey.com/best to use.

IMPORTANT:

yellkeys are **NOT** private. anyone can access your URL if they want to. please be careful what links you choose to share through yellkey.

try out our yellkey browser extensions for [Google Chrome](#), [Mozilla Firefox](#), and [Apple Safari](#)

the yellkey concept

concept Yellkey

purpose shorten URLs to common words

principle

after registering a URL u for time t and getting a shortening s
looking up s will yield u until the shortening expires time t later

can you identify actions?
which are by user? system?

actions

concept Yellkey

purpose shorten URLs to common words

principle

after registering a URL u for time t and getting a shortening s
looking up s will yield u until the shortening expires time t later

actions

register (u : URL, t : int, **out** s : String)

lookup (s : String, **out** u : URL)

system expire (**out** s : String)

what must be stored to support these actions?

a trace: $\langle \text{register}(u_1, t_1, s_1); \text{lookup}(s_1, u_1) \rangle$

state

concept Yellkey

purpose shorten URLs to common words

state

used: **set** String

shortFor: used -> **one** URL

expiry: used -> **one** Date

const shorthands: **set** String

actions

register (u: URL, t: int, **out** s: String)

lookup (s: String, **out** u: URL)

system expire (**out** s: String)

how to the actions
read/write the state?

concept Yellkey

state

used: **set** String

shortFor: used -> **one** URL

expiry: used -> **one** Date

const shorthands: **set** String

actions

register (u: URL, t: int, **out** s: String)

s in shorthands - used

s.shortFor := u

s.expiry := t secs after now

used += s

lookup (s: String, **out** u: URL)

s in used

u := s.shortFor

system expire (**out** s: String)

s.expiry is before now

used -= s

s.shortFor := none

s.expiry := none

non-
determinism

precondition

design question:
what if register replaced
existing shorthands for u
instead of adding one?

relational state

why write this

used: **set** String
shortFor: used -> **one** URL

rather than this?

Map [String, URL] shortFor;

sets & relations are simple and rep-independent

```
used = {"hello", "there"}  
shortFor = {"hello", "dnj.photo"}, {"there", "nytimes.com"}
```

can apply set & relation operators

```
findShorthandsFromURL (u: URL, out s: set String)  
s = u.~shortFor
```

what do updates mean?

s.shortFor := u means:
shortFor after is shortFor before, with all
pairs from s removed, and a new pair to u added

factoring concepts
into reusable patterns



Mont Saint Michel (1450–1521)



MIT (Bosworth, 1916)



Stata Center (Gehry, 2004)

A Pattern Language

Towns · Buildings · Construction



Christopher Alexander

Sara Ishikawa · Murray Silverstein

WITH

Max Jacobson · Ingrid Fiksdahl-King

Shlomo Angel

1977

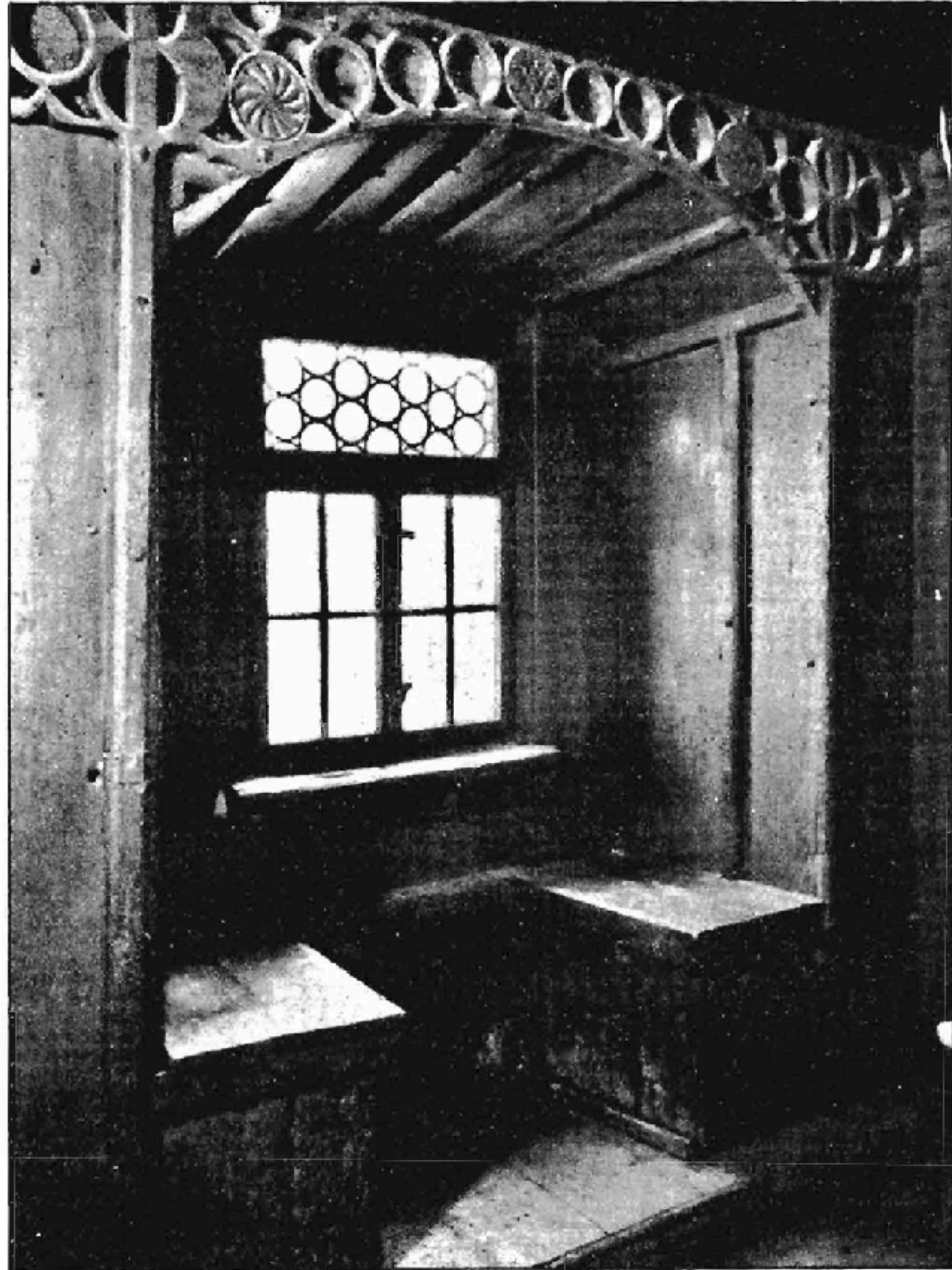
The Timeless Way of Building



Christopher Alexander

1979

180 WINDOW PLACE**



. . . this pattern helps complete the arrangement of the windows given by ENTRANCE ROOM (130), ZEN VIEW (134), LIGHT ON TWO SIDES OF EVERY ROOM (159), STREET WINDOWS (164). According to the pattern, at least one of the windows in each room needs to be shaped in such a way as to increase its usefulness as a space.



Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them.

can we factor yellkey into more familiar patterns?

concept Yellkey

purpose shorten URLs to common words

state

used: **set** String

shortFor: used -> **one** URL

expiry: used -> **one** Date

const shorthands: **set** String

actions

register (u: URL, t: int, **out** s: String)

lookup (s: String, **out** u: URL)

system expire (**out** s: String)

is there a separable
concept in here?

can you explain Yellkey in terms of these concepts?

concept Shorthand [Target] ← a polymorphic concept

purpose provide access via shorthand strings

principle

after registering a target t and obtaining a shorthand s ,
looking up s will yield t : $\text{register}(t, s)$; $\text{lookup}(s, t')$ $\{t' = t\}$

state

used: **set** String

shortFor: String \rightarrow **opt** Target

const shorthands: **set** String

actions

$\text{register}(t: \text{Target}, \text{out } s: \text{String})$

s in shorthands - used

$s.\text{shortFor} := t$; used += s

$\text{unregister}(s: \text{String})$

s in used

used -= s ; $s.\text{shortFor} := \text{none}$

$\text{lookup}(s: \text{String}, \text{out } t: \text{Target})$

s in used

$t := s.\text{shortFor}$

concept ExpiringResource [Resource]

purpose handle expiration of short-lived resources

principle

if you allocate a resource r for t seconds, after t seconds
the resource expires: $\text{allocate}(r, t)$; $\text{expire}(r)$

state

active: set Resource

expiry: Resource \rightarrow one Date

actions

$\text{allocate}(r: \text{Resource}, t: \text{int})$

r not in active

active += r ; $r.\text{expiry} := t$ secs after now

$\text{deallocate}(r: \text{Resource})$

r in active; active -= r ; $r.\text{expiry} := \text{none}$

$\text{renew}(r: \text{Resource}, t: \text{int})$

r in active; $r.\text{expiry} := t$ secs after now

system $\text{expire}(\text{out } r: \text{Resource})$

r in active; $r.\text{expiry}$ is before now;

active -= r ; $r.\text{expiry} := \text{none}$

a familiar concept has many uses

examples of uses of ExpiringResource

Wifi on airplane

discount coupon

credit card, passport, driving license

two factor authentication code

...

composition by
synchronization

adding a synchronization

concept Shorthand [Target]

purpose provide access via shorthand strings

principle

after registering a target t and obtaining a shorthand s ,
looking up s will yield t : $\text{register}(t, s)$; $\text{lookup}(s, t')$ $\{t' = t\}$

state

used: **set** String

shortFor: String \rightarrow **opt** Target

const shorthands: **set** String

actions

$\text{register}(t: \text{Target}, \text{out } s: \text{String})$

s in shorthands - used

$s.\text{shortFor} := t$; used += s

unregister ($s: \text{String}$)

s in used

used -= s ; $s.\text{shortFor} := \text{none}$

$\text{lookup}(s: \text{String}, \text{out } t: \text{Target})$

s in used

$t := s.\text{shortFor}$

concept ExpiringResource [Resource]

purpose handle expiration of short-lived resources

principle

if you allocate a resource r for t seconds, after t seconds
the resource expires: $\text{allocate}(r, t)$; $\text{expire}(r)$

state

active: set Resource

expiry: Resource \rightarrow one Date

actions

$\text{allocate}(r: \text{Resource}, t: \text{int})$

r not in active

active += r ; $r.\text{expiry} := t$ secs after now

$\text{deallocate}(r: \text{Resource})$

r in active; active -= r ; $r.\text{expiry} := \text{none}$

$\text{renew}(r: \text{Resource}, t: \text{int})$

r in active; $r.\text{expiry} := t$ secs after now

system $\text{expire}(\text{out } r: \text{Resource})$

r in active; $r.\text{expiry}$ is before now;

active -= r ; $r.\text{expiry} := \text{none}$

what other actions need
synchronizing?

synchronizing concepts

```
app YellKey
include HTTP
include Shorthand [HTTP.URL]
include ExpiringResource [String]
sync register (url: URL, short: String, life: int)
  when Shorthand.register (url, short)
  ExpiringResource.allocate (short, life)
sync expire (out short: String)
  when ExpiringResource.expire (short)
  Shorthand.unregister (short)
sync lookup (short: String, url: URL)
  Shorthand.lookup (short, url)
```


concept behavior is preserved!

concept Shorthand

concept ExpiringResource

register (url1, s1)

when register (u, t)
allocate (u)

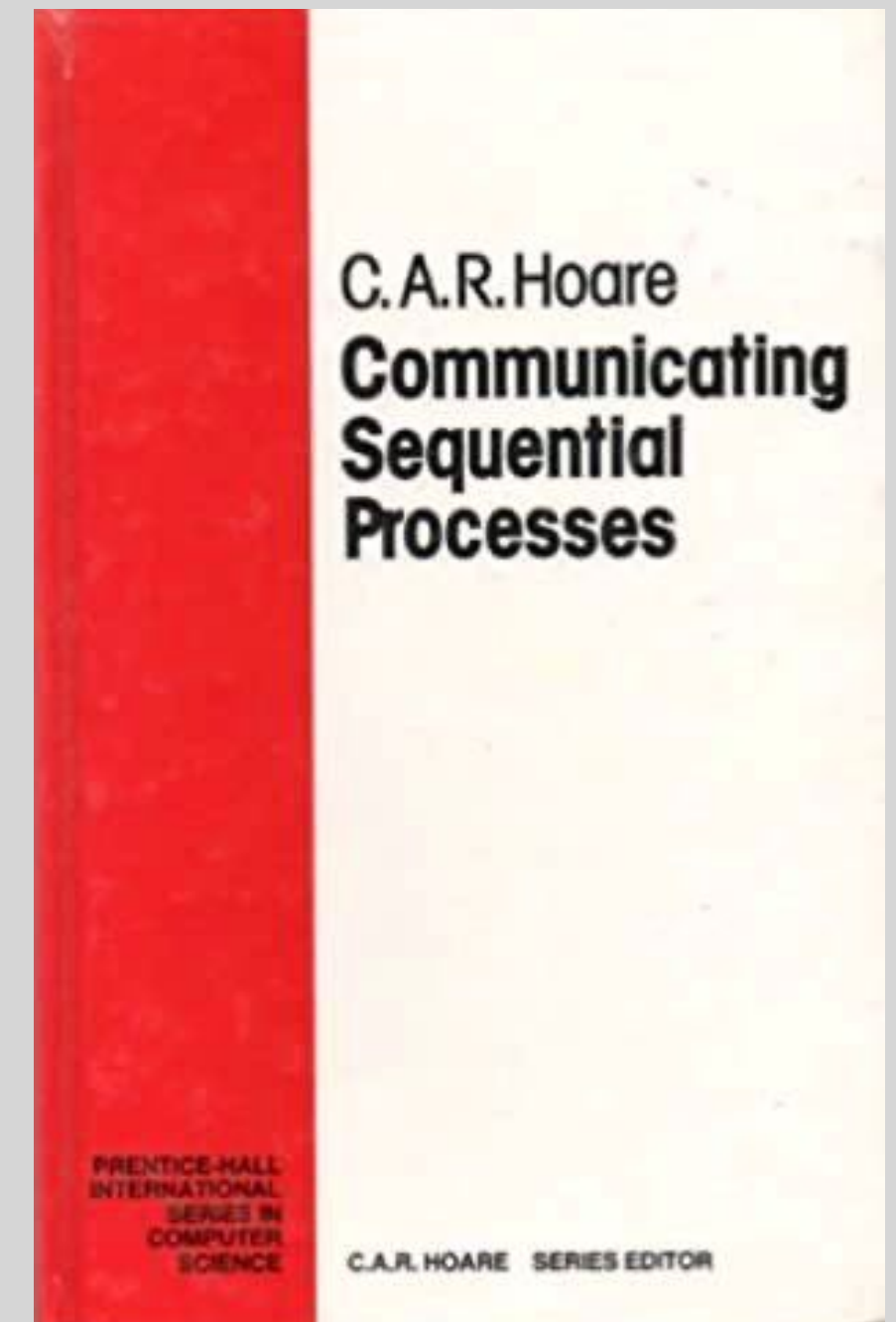
allocate (s1, 3600)

lookup (s1, url1)

unregister (s1)

when expire (r)
unregister (r)

expire (s1)



sync is from CSP

user sessions

▲ 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.

▲ 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

a familiar combination

concept User

purpose authenticate users

principle

after a user registers with a username and password, they can authenticate as that user by providing a matching username and password:

register (n, p, u); authenticate (n, p, u') {u' = u}

state

registered: **set** User

username, password: registered -> **one** String

actions

register (n, p: String, out u: User)

authenticate (n, p: String, out u: User)

concept Session [User]

purpose authenticate user for extended period

principle

after a session starts (and before it ends), the getUser action returns the user identified at the start:
start (u, s); getUser (s, u') {u' = u}

state

active: **set** Session

user: active -> **one** User

actions

start (u: User, **out** s: Session)

getUser (s: Session, **out** u: User)

end (s: Session)

where is one of these used without the other?

what syncs are needed?

do sessions last forever?

why two concepts are needed

application of User without Session:

authenticating one-off actions in operating systems

MacOS: authenticate when opening app for first time

Unix: executing command requiring superuser

reauthenticating mid-session for critical actions

confirming bank transfers

one time authentication in websites

when cancelling a subscription

application of Session without User:

authenticating by different means

biometrics such as facial recognition, fingerprint

unauthenticated sessions

in some games and chat apps, user just enters name

putting it all together

concept ExpiringUserSession

include User

include Session [User.User]

include ExpiringResource [Session.Session]

sync register (username, password: String, **out** user: User)

User.register (username, password, user)

sync login (username, password: String, **out** user: User, **out** s: Session)

when User.authenticate (username, password, user)

Session.start (user, session)

ExpiringResource.allocate (session, 300)

sync logout (s: Session)

when Session.end (session)

ExpiringResource.deallocate (session)

sync authenticate (s: Session, u: User)

Session.getUser (s, u)

sync terminate (s: Session)

when ExpiringResource.expire (session)

Session.end (session)

levels of design

convergent design

**composition by
synchronization**

**patterns &
factoring**

**faithful projection of
conceptual model**

**shaping a
conceptual model**

**concept as
state machine**