6.1040: Software Design

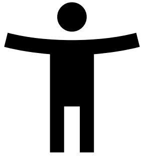
Concept Basics

Arvind Satyanarayan & Max Goldman

with material by Daniel Jackson

Fall ’24

Levels of design

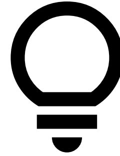


physical

color, size, layout,
type, touch, sound

linguistic

icons, labels, tooltips,
site structure



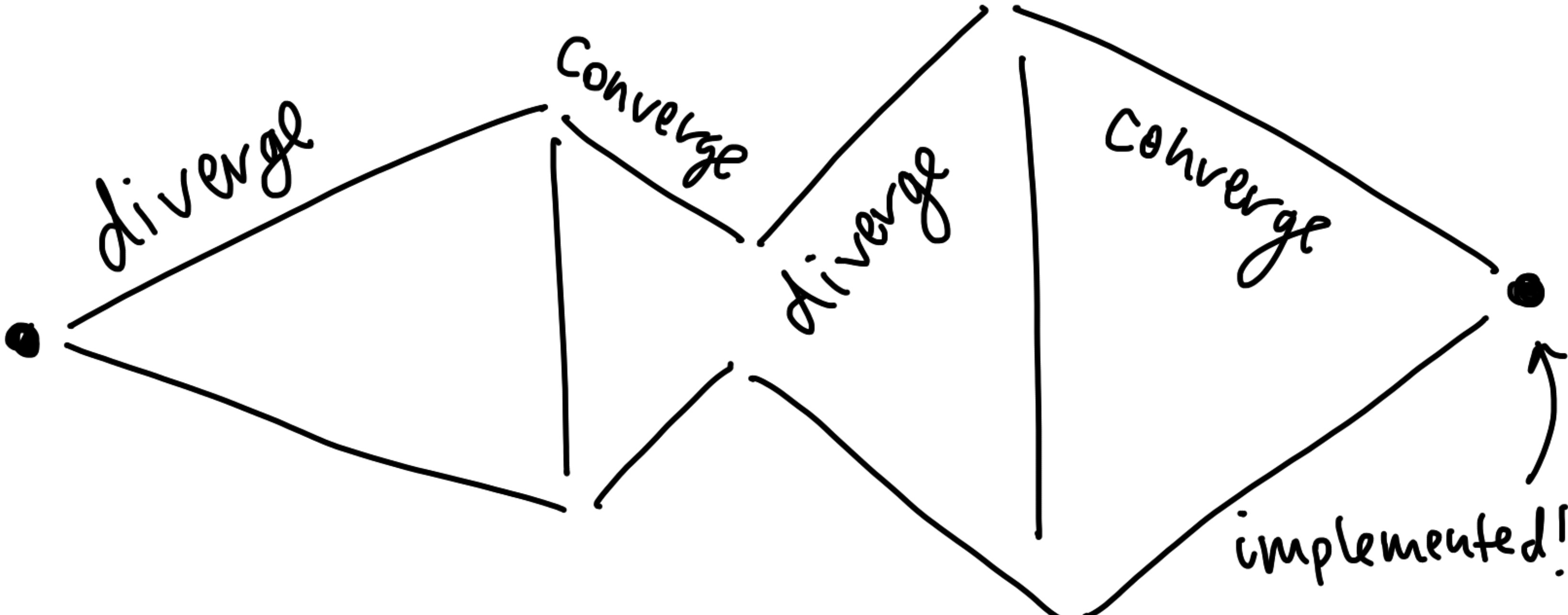
conceptual

semantics, actions,
data model, purpose

software concepts

Semantic
Purposive
Modular

Diverging and converging



www.yellkey.com

yellkey: url to common word shortener.

yellkey

enter url and length of time for key to exist.

http://web.mit.edu/

1 hour

generate yellkey

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.

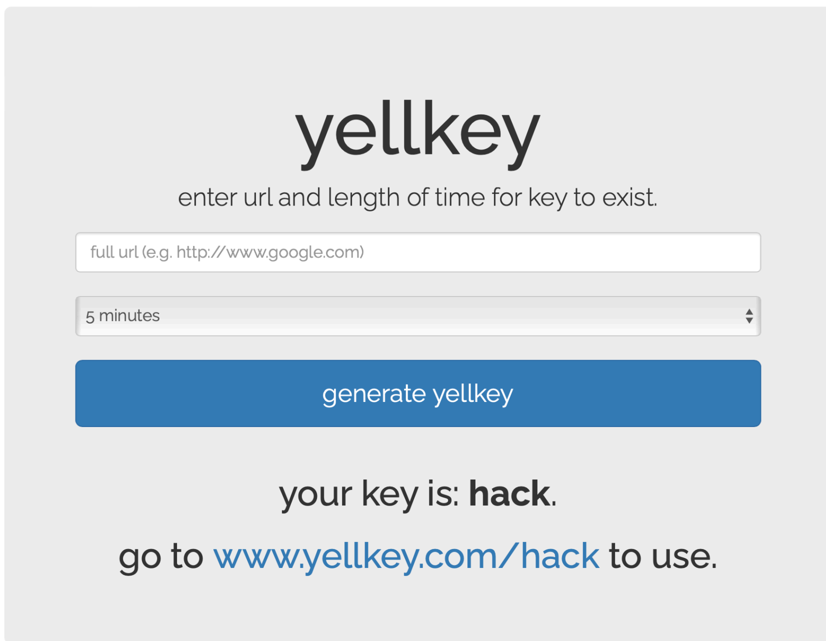
try out our yellkey browser extensions for Google Chrome, Mozilla Firefox, and Apple Safari from sarah lim and andrew finke

made with ♡ by delta lab

a very big thank you to chad etzel, the creator of shoutkey and inspiration for yellkey.

www.yellkey.com/generateKey

yellkey: url to common word shortener.



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

concept Yellkey

purpose what is it for?

principle

a small story that explains how it works

concept Yellkey

purpose shorten URLs to common words

principle

after registering a URL and getting a shorthand for it,
looking up that shorthand will yield the URL, until expiry

actions

what behaviors do users experience?

concept Yellkey

purpose shorten URLs to common words

principle

after registering a URL and getting a shorthand for it,
looking up that shorthand will yield the URL, until expiry

actions

register (url: URL, time: int, **out** short: String)
lookup (short: String, **out** url: URL)
system expire (**out** short: String)

concept Yellkey

purpose shorten URLs to common words

principle

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

actions

register (url: URL, time: int, **out** short: String)
lookup (short: String, **out** url: URL)
system expire (**out** short: String)

concept Yellkey

purpose shorten URLs to common words

principle

after register (u, t, s) then lookup (s, u) until expire (s)

actions

register (url: URL, time: int, **out** short: String)
lookup (short: String, **out** url: URL)
system expire (**out** short: String)

treating inputs &
outputs uniformly

concept Yellkey

purpose shorten URLs to common words

principle

after register (u, t, s) then lookup (s, u) until expire (s)

state

what must be stored to support the actions?

actions

register (url: URL, time: int, **out** short: String)
lookup (short: String, **out** url: URL)
system expire (**out** short: String)

concept Yellkey

purpose shorten URLs to common words

principle

after register (u, t, s) then lookup (s, u) until expire (s)

state

const shorthands: **set** String
used: **set** String
shortFor: used → **one** URL
expiry: used → **one** Date

every String in used is associated with exactly one URL & Date

actions

register (url: URL, time: int, **out** short: String)
lookup (short: String, **out** url: URL)
system expire (**out** short: String)

concept Yellkey

purpose
shorten URLs to common words

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

state

const shorthands: **set** String
used: **set** String
shortFor: used → **one** URL
expiry: used → **one** Date

invariants

... what must be stored
to support the actions?

actions

what behaviors do users experience? ...

register (url: URL, time: int, **out** short: String)
pick short from the set shorthands - used
update shortFor so that short → url
update expiry so that short → time sec. after now
add short to used

nondeterminism

lookup (short: String, **out** url: URL)
require short in used
url is the URL associated with short by shortFor

preconditions

system expire (**out** short: String)
require expiry of short is before now
remove short from used
update shortFor and expiry so that short → *none*

postconditions

concept Yellkey

purpose
shorten URLs to common words

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

state

const shorthands: **set** String
used: **set** String
shortFor: used → **one** URL
expiry: used → **one** Date

actions

register (url: URL, time: int, **out** short: String)

short in shorthands - used
short . shortFor := url
short . expiry := time sec. after now
used += short

lookup (short: String, **out** url: URL)

short in used
url := short . shortFor

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

short . expiry < now
used -= short
short . shortFor := none
short . expiry := none

Relational state

shortFor: String → **one** URL

shortFor is a binary relation from String to one URL, *e.g.*

{ ("hack", http://web.mit.edu/),
("punt", https://61040-fa24.github.io/),
("never", https://www.youtube.com/watch?v=MA_v0YMPN9c) }

Relational join

lookup (short: String, **out** url: URL)

url := short . shortFor

{ ("hack") } . { ("hack", mit.edu), ("punt", 61040), ("never", youtube) } = { (mit.edu) }

Relational state

shortFor: String → **one** URL

shortFor is a binary relation from String to one URL, *e.g.*

{ ("hack", http://web.mit.edu/),
("punt", https://61040-fa24.github.io/),
("never", https://www.youtube.com/watch?v=MA_v0YMPN9c) }

Relational join

lookup (short: String, **out** url: URL)

url := short . shortFor

{ ("hack") } . { ("hack", mit.edu), ("punt", 61040), ("never", youtube) } = { (mit.edu) }

Relational update

register (url: URL, time: int, **out** short: String)

short . shortFor := url

shortFor ′ = shortFor – (short, *) + (short, url)

new shortFor → (shortFor ′)

← old shortFor (shortFor)

← remove old pairs ((short, *))

Relational state

shortFor: String → **one** URL

shortFor is a binary relation from String to one URL, *e.g.*

{ ("hack", http://web.mit.edu/),
("punt", https://61040-fa24.github.io/),
("never", https://www.youtube.com/watch?v=MA_v0YMPN9c) }

Relational join

lookup (short: String, **out** url: URL)

url := short . shortFor

{ ("hack") } . { ("hack", mit.edu), ("punt", 61040), ("never", youtube) } = { (mit.edu) }

& other operations, *e.g.*

allShorthands (url: URL, **out** shorts: **set** String)

shorts := url . ~shortFor ← *inverse of shortFor*

{ (mit.edu) } . { (mit.edu, "hack"), (61040, "punt"), (youtube, "never") } = { ("hack") }

concept Yellkey

purpose
shorten URLs to common words

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

state

const shorthands: **set** String
used: **set** String
shortFor: used → **one** URL
expiry: used → **one** Date

actions

register (url: URL, time: int, **out** short: String)

short in shorthands - used
short.shortFor := url
short.expiry := time sec. after now
used += short

lookup (short: String, **out** url: URL)

short in used
url := short.shortFor

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

short.expiry < now
used -= short
short.shortFor := none
short.expiry := none

Alternative design: What if we modified register (...) so that it **replaced** any existing shorthand for the URL, instead of adding?

Something unsatisfying about *concept* Yellkey

Shortening and Expiring are both patterns we have seen elsewhere

They can be expressed generically

And we can describe Yellkey as a combination of Shortening + Expiring

Semantic

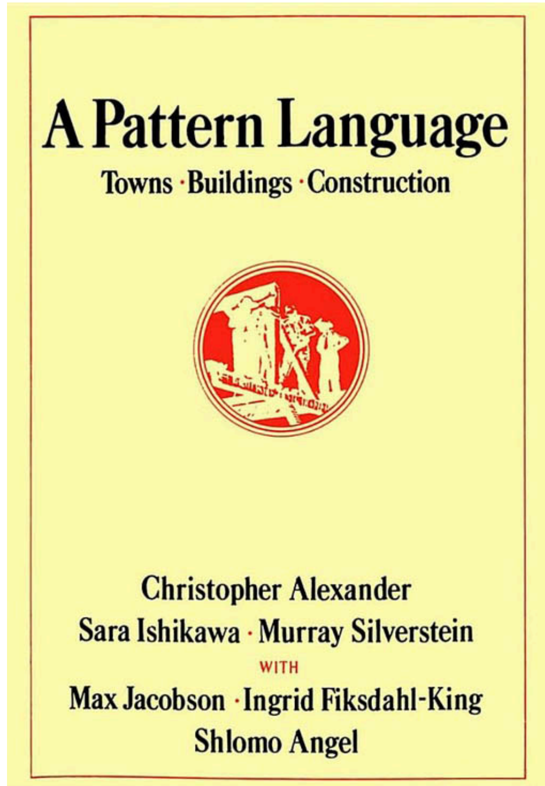
↳ about underlying behavior users experience
Not internals, user-facing
Not UI, but underlying function
Not just structure, behavior

Purposive

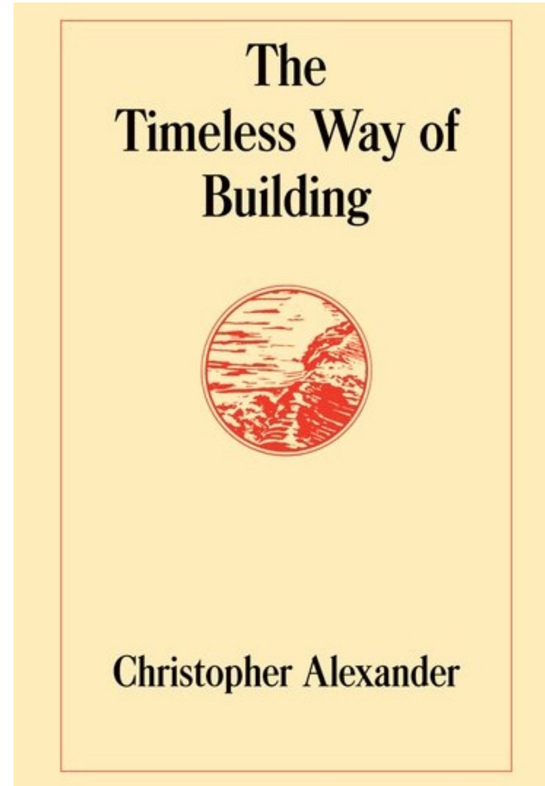
↳ fulfills an entire user need
Included for a reason
End-to-end, not just a fragment

Modular

↳ mutually independent
Generic (using polymorphic parameters)
Reusable within and across apps



(1977)



(1979)

concept Yellkey

purpose
shorten URLs to common words

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

state

const shorthands: **set** String
used: **set** String
shortFor: used → **one** URL
expiry: used → **one** Date

actions

register (url: URL, time: int, **out** short: String)

short in shorthands - used
short.shortFor := url
short.expiry := time sec. after now
used += short

lookup (short: String, **out** url: URL)

short in used
url := short.shortFor

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

short.expiry < now
used -= short
short.shortFor := none
short.expiry := none

Factoring out Shortening

concept Shortening *first draft*

purpose
provide access via short strings

principle
after registering a URL u
and getting a shorthand s,
looking up s will yield u until
???

state

const shorthands: **set** String
used: **set** String
shortFor: used → **one** URL

actions

register (url: URL, **out** short: String)

short in shorthands - used
short.shortFor := url
used += short

lookup (short: String, **out** url: URL)

short in used
url := short.shortFor

???

used -= short
short.shortFor := none

concept Shortening [Target]

purpose
provide access via short strings

principle
after registering a target t
and getting a shorthand s,
looking up s will yield t,
until s is unregistered

state

const shorthands: **set** String
used: **set** String
shortFor: used → **one** Target

actions

register (target: Target, **out** short: String)

short in shorthands - used
short . shortFor := target
used += short

lookup (short: String, **out** target: Target)

short in used
target := short . shortFor

unregister (short: String)

short in used
used -= short
short . shortFor := none

concept Shortening [Target]
purpose provide access via short strings
principle after registering a target t and getting a shorthand s, looking up s will yield t
state

- **const** shorthands: **set** String
- used: **set** String
- shortFor: used → **one** Target

actions

register (t: Target, **out** s: String)
- s in shorthands - used ; s.shortFor := t ; used += s

lookup (s: String, **out** t: Target)
- s in used ; t := s.shortFor

unregister (s: String)
- s in used ; used -= s ; s.shortFor := none

concept Expiring [Resource]
purpose handle expiration of short-lived resources
principle if you allocate a resource r for t seconds, after t seconds the resource expires

state

- active: **set** Resource
- expiry: active → **one** Date

actions

allocate (rsrc: Resource, time: int)
- rsrc not in active
- active += rsrc
- rsrc.expiry := time sec. after now

system expire (**out** rsrc: Resource)
- rsrc in active ; rsrc.expiry is before now
- active -= rsrc ; rsrc.expiry := none

concept Shortening [Target]
purpose provide access via short strings
principle after registering a target t and getting a shorthand s, looking up s will yield t
state

- **const** shorthands: **set** String
- used: **set** String
- shortFor: used → **one** Target

actions

register (t: Target, **out** s: String)

- s in shorthands - used ; s . shortFor := t ; used += s

lookup (s: String, **out** t: Target)

- s in used ; t := s . shortFor

unregister (s: String)

- s in used ; used -= s ; s . shortFor := none

concept Expiring [Resource]
purpose handle expiration of short-lived resources
principle if you allocate a resource r for t seconds, after t seconds the resource expires

state

- active: **set** Resource
- expiry: active → **one** Date

actions

allocate (rsrc: Resource, time: int)

- rsrc not in active
- active += rsrc
- rsrc . expiry := time sec. after now

system expire (**out** rsrc: Resource)

- rsrc in active ; rsrc . expiry is before now
- active -= rsrc ; rsrc . expiry := none

Synchronizing concepts to build an app

```
app Yellkey
    include Shortening [URL]
    include Expiring [String]

    sync register (url: URL, time: int, out short: String)
        Shortening.register (url, short)
        Expiring.allocate (short, time)

    system sync expire (out short: String)
        Expiring.expire (short)
        Shortening.unregister (short)

    sync lookup (short: String, out url: URL)
        Shortening.lookup (short, url)
```

```
concept Shortening [Target]
purpose provide access via short strings
principle after registering a target t and getting
a shorthand s, looking up s will yield t
state
    const shorthands: set String
    used: set String
    shortFor: used → one Target
actions
    register (t: Target, out s: String)
        s in shorthands - used ; s . shortFor := t ; used += s
    lookup (s: String, out t: Target)
        s in used ; t := s . shortFor
    unregister (s: String)
        s in used ; used -= s ; s . sh
```

```
concept Expiring [Resource]
purpose handle expiration of short-lived resources
principle if you allocate a resource r for t seconds, after
t seconds the resource expires
state
    active: set Resource
    expiry: active → one Date
actions
    allocate (rsrc: Resource, time: int)
        rsrc not in active
        active += rsrc
        rsrc . expiry := time sec. after now
    system expire (out rsrc: Resource)
```

Timelines of actions

concept behavior is always preserved

concept Shortening

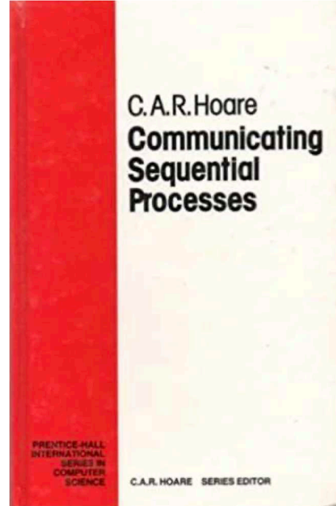
concept Expiring

register(mit.edu, "hack") ... when register(u,s) allocate(s) .... allocate("hack", 3600)

lookup("hack", mit.edu)

unregister("hack") ..... when expire(r) unregister(s) ... expire("hack")

TIME



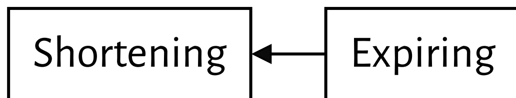
Formulation of synchronization due to Tony Hoare

Dependencies and subsets

Yellkey

We have designed our concepts so they have no intrinstic dependencies

What are the extrinsic dependencies?



"If we include Expiring, we must also include Shortening"

Designing Software for Ease of Extension and Contraction

DAVID L. PARNAS

Abstract—Designing software to be extensible and easily co
discussed as a special case of design for change. A number of
extension and contraction problems manifest themselves
software are explained. Four steps in the design of softw
more flexible are then discussed. The most critical step is th
a software structure called the "uses" relation. Some criteria
decisions are given and illustrated using a small example. I
that the identification of *minimal* subsets and *minimal* exte
lead to software that can be tailored to the needs of a broad
users.

Index Terms—Contractibility, extensibility, modularity, so
gineering, subsets, supersets.

1) "We were behind schedule and wanted to deliver an early release with only a <proper subset of intended capabilities>, but found that that subset would not work until everything worked."
2) "We wanted to add <simple capability>, but to do so would have meant rewriting all or most of the current code."

this simplification we would have had to rewrite major sections of the code."

... I have identified some simple concepts that can help programmers to design software so that subsets and extensions are more easily obtained. These concepts are simple if you think about software in the way suggested by this paper. Programmers do not commonly do so.

Manuscript received June 7, 1978; revised October 26, 1
earliest work in this paper was supported by NV Phillips Co
dustrie, Apeldoorn, The Netherlands. This work was also sup
the National Science Foundation and the German Federal M
Research and Technology (BMFT). This paper was presen
Third International Conference on Software Engineering, At
May 1978.

The author is with the Department of Computer Science,
of North Carolina, Chapel Hill, NC 27514. He is also with th
tion Systems Staff, Communications Sciences Division, Nava
Laboratory, Washington, DC.

0098-5589/79

Designing Software for Ease of Extension and Contraction

DAVID L. PARNAS

Abstract—Designing software to be extensible and easily contracted is discussed as a special case of design for change. A number of ways that extension and contraction problems manifest themselves in current software are explained. Four steps in the design of software that is more flexible are then discussed. The most critical step is the design of a software structure called the "uses" relation. Some criteria for design decisions are given and illustrated using a small example. It is shown that the identification of *minimal* subsets and *minimal* extensions can lead to software that can be tailored to the needs of a broad variety of users.

Index Terms—Contractibility, extensibility, modularity, software engineering, subsets, supersets.

I. Introduction

THIS paper is being written because the following complaints about software systems are so common.

1) "We were behind schedule and wanted to deliver an early release with only a <proper subset of intended capabilities>, but found that that subset would not work until everything worked."

2) "We wanted to add <simple capability>, but to do so would have meant rewriting all or most of the current code."

3) "We wanted to simplify and speed up the system by removing the <unneeded capability>, but to take advantage of this simplification we would have had to rewrite major sections of the code."

The criteria to be used in allowing one [module] to use another:

We propose to allow A "uses" B when all of the following conditions hold:

a) A is essentially simpler because it uses B;

b) B is not substantially more complex because it is not allowed to use A;

c) there is a useful subset containing B and not A;

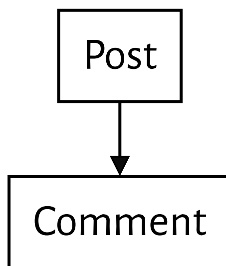
d) there is no conceivably useful subset containing A but not B.

Dependencies and subsets

Hacker News

posting.ts

```
class Post {
    readonly comments: Comment[];
}
```



allow Post uses Comment when...
(c) there is a useful subset containing Comment and not Post;
(d) there is no useful subset containing Post but not Comment

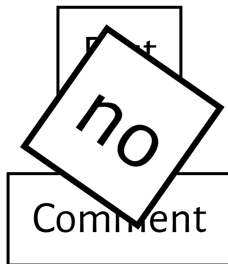
The criteria to be used in allowing one [module] to use another:
We propose to allow A “uses” B when all of the following conditions hold:
a) A is essentially simpler because it uses B;
b) B is not substantially more complex because it is not allowed to use A;
c) there is a useful subset containing B and not A;
d) there is no conceivably useful subset containing A but not B.

Dependencies and subsets

Hacker News

posting.ts

```
class Post {  no
    readonly      nts: Comment[];
}
```



exactly backwards!

allow Post uses Comment when…
(c) there is a u no bset
containing Co and not Post;
(d) there is no subset
containing Post but not Comment

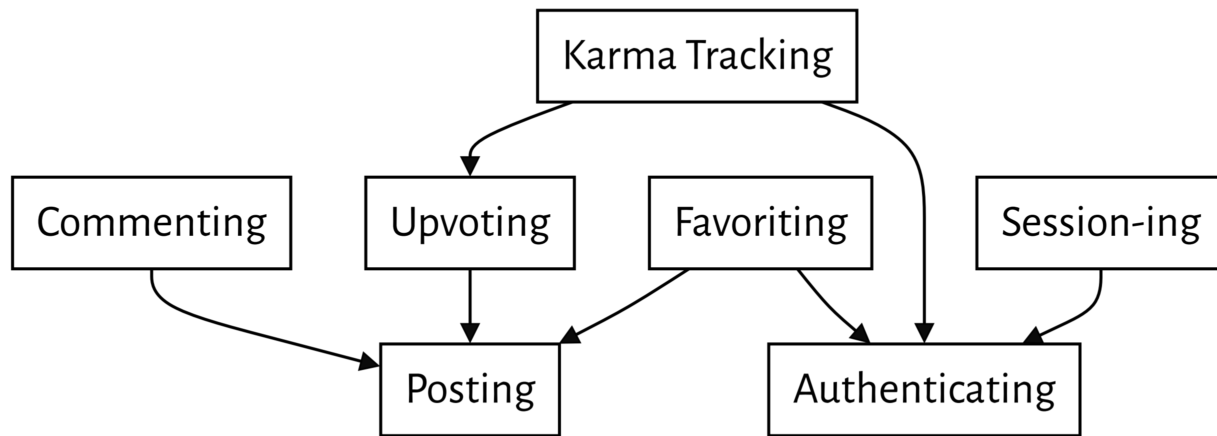
The criteria to be used in allowing one [module] to use another:
We propose to allow A “uses” B when all of the following conditions hold:
a) A is essentially simpler because it uses B;
b) B is not substantially more complex because it is not allowed to use A;
c) there is a useful subset containing B and not A;
d) there is no conceivably useful subset containing A but not B.

Modular

Dependencies and subsets

By eliminating *intrinsic* dependencies when there is no *extrinsic* dependency, we are prepared to build subsets and extensions

Hacker News



e.g.
{ Posting }
{ Posting, Commenting }
{ Posting, Upvoting, Karma Tracking, Auth-ing }
~~{ Karma Tracking }~~

A → B means: *an app that includes A must also include B*

(do Upvoting, Commenting, and Posting also depend on Authenticating? maybe!)

Another example

Expiring Authentication Sessions

Expiring shows up in many contexts

Authenticating *without* sessions?

Session-ing *without* authentication?

Another example

Expiring Authentication Sessions

concept Authenticating
purpose authenticate users so that app users correspond to people
principle after a user registers with a username and password pair, they can authenticate as that user by providing the pair:
register (n, p, u); authenticate (n, p, u') {u' = u}

state

registered: **set** User
username, password: registered → **one** String

actions

register (name, pass: String, **out** user: User)
authenticate (name, pass: String, **out** user: User)

Another example

Expiring Authentication Sessions

concept Authenticating
purpose authenticate users so that app users correspond to people
actions
register (name, pass: String, **out** user: User)
authenticate (name, pass: String, **out** user: User)

concept Session-ing [User]
purpose enable authenticated actions for an extended period of time
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 (user: User, **out** sess: Session)
getUser (sess: Session, **out** user: User)
end (sess: Session)

Another example

Expiring Authentication Sessions

concept Authenticating
purpose authenticate users so that app users correspond to people
actions
register (name, pass: String, **out** user: User)
authenticate (name, pass: String, **out** user: User)

concept Session-ing [User]
purpose enable authenticated actions for an extended period of time
actions
start (user: User, **out** s: Session)
getUser (s: Session, **out** user: User)
end (s: Session)

concept Expiring [Resource]
purpose handle expiration of short-lived resources

actions
allocate (r: Resource, time: int)
deallocate (r: Resource)
system expire (**out** r: Resource)

Another example

Expiring Authentication Sessions

concept Authenticating
purpose authenticate users so that app users correspond to people
actions
register (name, pass: String, **out** user: User)
authenticate (name, pass: String, **out** user: User)

concept Session-ing [User]
purpose enable authenticated actions for an extended period of time
actions
start (user: User, **out** s: Session)
getUser (s: Session, **out** user: User)
end (s: Session)

concept Expiring [Resource]
purpose handle expiration of short-lived resources
actions
allocate (r: Resource, time: int)
deallocate (r: Resource)
system expire (**out** r: Resource)

```
app ...
    ... including other concepts ...
    include Authenticating, Sessioning [Authenticating.User], Expiring [Sessioning.Session]

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

    sync login (username, password: String, out user: User, out session: Session)
        Authenticating.authenticate (username, password, user)
        Sessioning.start (user, session)
        Expiring.allocate (session, 300)

    sync authenticate (session: Session, out user: User)
        Sessioning.getUser (session, user)

    sync logout (session: Session)
        Sessioning.end (session)
        Expiring.deallocate (session)

    system sync expire (session: Session)
        Expiring.expire (session)
        Sessioning.end (session)
```

concept ExpiringAuthenticationSessions ← slow down, start with one flat collection, not a hierarchy

include Authenticating, Sessioning [Authenticating.User], Expiring [Sessioning.Session]

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

sync login (username, password: String, **out** user: User, **out** session: Session)
  Authenticating.authenticate (username, password, user)
  Sessioning.start (user, session)
  Expiring.allocate (session, 300)

sync authenticate (session: Session, out user: User)
  Sessioning.getUser (session, user)

sync logout (session: Session)
  Sessioning.end (session)
  Expiring.deallocate (session)

system sync expire (session: Session)
  Expiring.expire (session)
  Sessioning.end (session)

Keeping it simple

part I: concepts

Recognize existing patterns and factor them out

Converge on concepts where each fulfills exactly 1 purpose: not 2, not ½

Reuse existing knowledge *-and-* consider broader design implications

part II: synchronizations

Atomic (like individual concept actions): all or nothing

Not the place for complexity

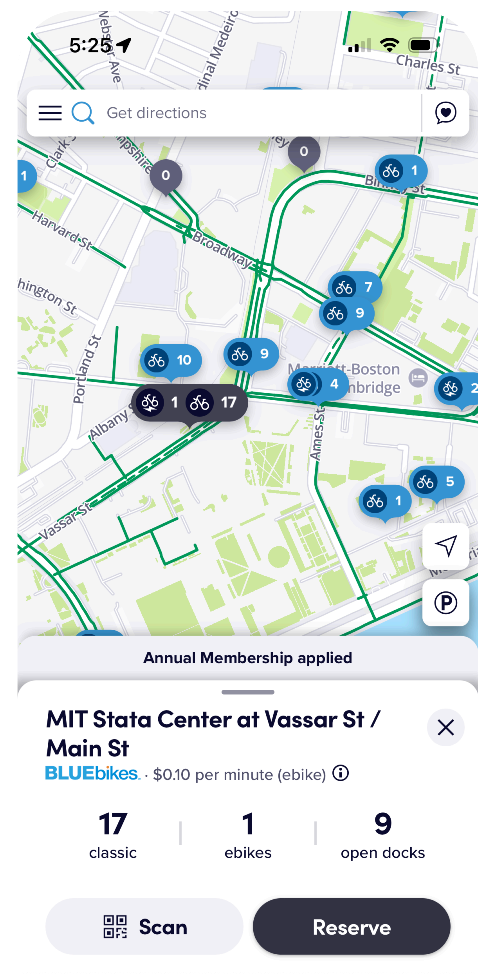
Simple concepts *plus* simple synchronizations *can equal* interesting novel behavior

One last example

🚲 Bluebikes

Ebikes can be reserved for for up to 10 minutes, at the same price per minute as riding

concept Reserving [Resource]



Today

Concepts as state machines with relational state

For structuring functionality: Semantic Purposive Modular

Dependencies and **subsets**

Extrinstic dependency graph shows us coherent subsets of an application

Patterns

Identifying and factoring out reusable concepts

Composition by **synchronization**

Behavior of individual concepts is preserved

Looking ahead

Concept design moves
Designing data and services