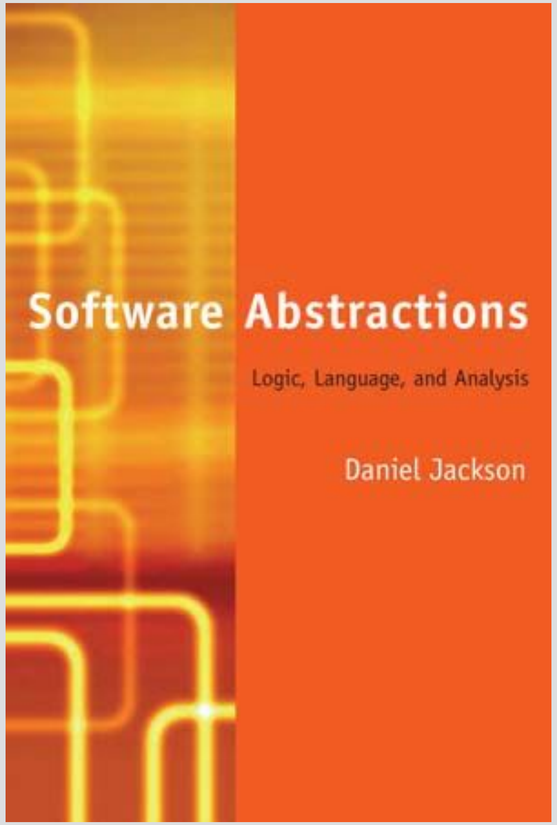
6.1040 · software studio · fall 2023

diverge/converge: features to concepts

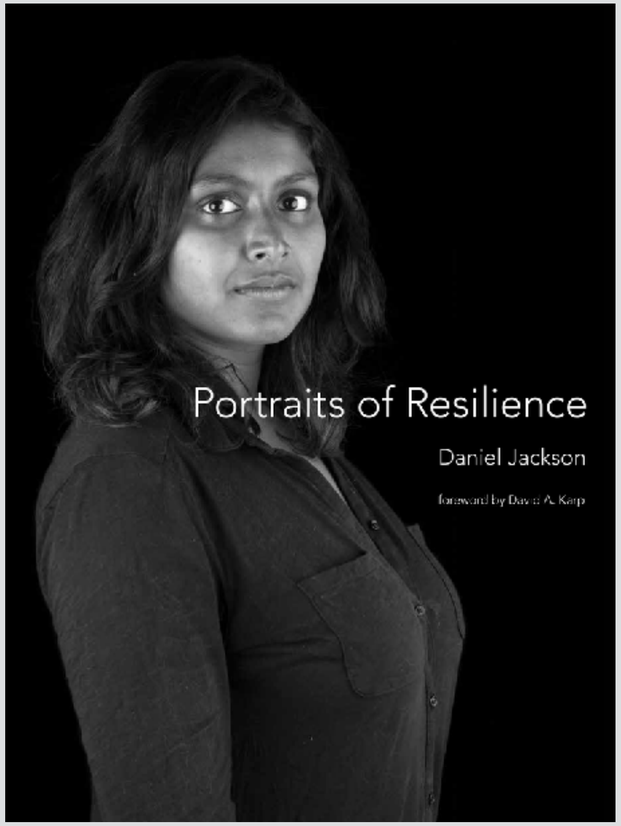
Daniel Jackson & Arvind Satyanarayan

introduction

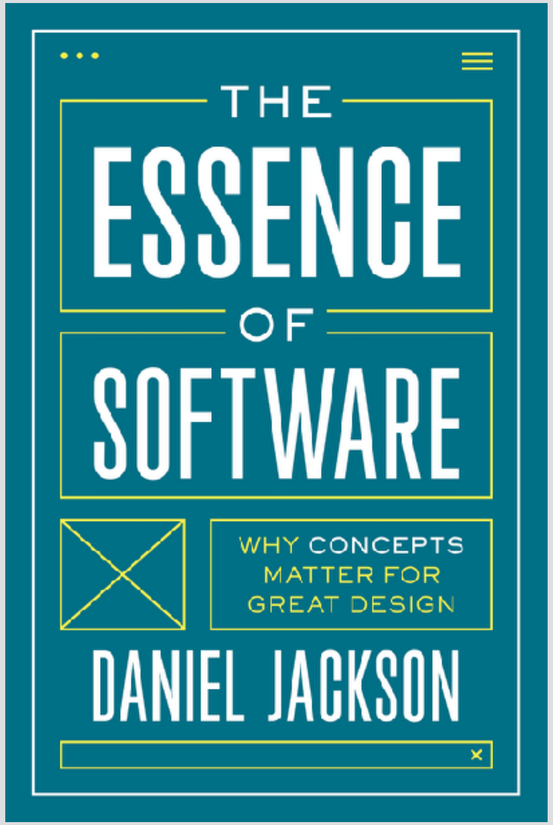
what I like to do (or, my career in books...)



Alloy: a design language (2006)



Mental health at MIT (2017)



Concept design (2021)

my passion outside work: photography

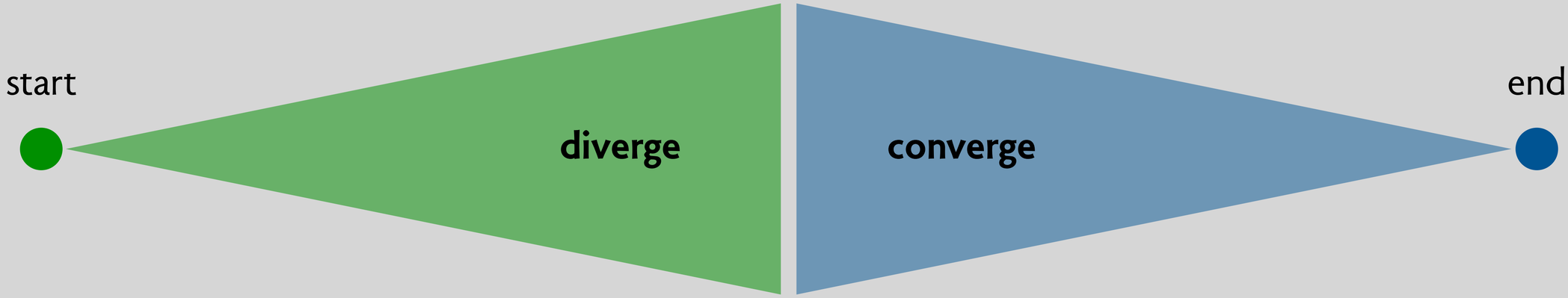
today's learning objectives

get the idea of diverge/converge
have some practice doing divergent design
learn the basic idea of concepts as software modules
have some practice doing convergent design
learn how to express subsets with dependency diagrams

my second favorite classic software engineering idea

diverge/
converge

two modes of thinking, two phases of design



generating ideas freely
expanding possibilities
finding new directions
ignoring barriers
goal: space of options

coalescing & unifying
reducing & simplifying
identifying problems
resolving conflicts
goal: coherent design

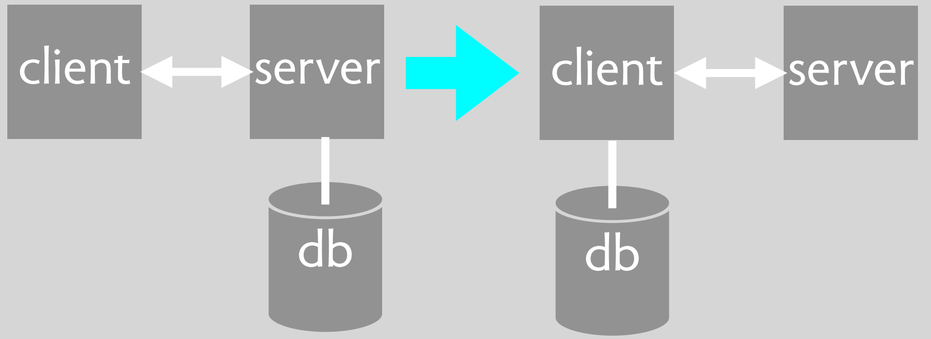
techniques for divergent design

brainstorming
generating lists of feature ideas
working collaboratively
taking improv posture “yes and”

lateral thinking
provocative idea generation: take a bad idea & pursue it
identify and challenge assumptions
focus on overlooked aspects of problem

foraging for inspiration
perusing books in stores and libraries
going down internet rabbit holes
fixating on unusual things

an example of a bad idea



https://riffle.systems

let's brainstorm!

an example app

name **SiteSpot**

audience **architecture enthusiasts**

purpose **self-guided architectural tours**

when you visit a neighborhood, helps you learn about the design and history of the buildings as you walk around

brainstorming features

info pops up automatically when in front of building
play audio so you can use app without your phone out
...
scan building and it starts tour with AR (Yilin)
ghost view of internal structure from 3d models (Jin)
recommendations of things to see based on so far (Gowri)
passers by leave facts geocaching (Amanda)
crowdsourced reviews (Ethan)
bookmark locations for coming back late (Oomi)
contribute descriptions, maybe by owner (Luca)
recommend based on aggregated crowd behavior (Shayla)
create route past favorite locations (Cal)

gap analysis
what's missing in existing apps?

viability
is critical mass needed?
who will generate content?
where will revenue come from?

analogies
are there similar apps or features?

exploring social/ethical values

in convergent design
to constrain options, tweak behavior of concepts

in divergent design
to suggest new features, add safeguards

stakeholders
users with different abilities
considering children
indirect stakeholders (not users)

time
impact on friendships

pervasiveness
other countries and geographies

values
autonomy and community
experience of values
environmental impacts

some features I came up with before class, roughly organized into areas

customized routes based on user prefs, starting point
show elevation, walk difficulty, safety of neighborhood
locate bike rental stations

track members of the group and show locations on map
introduce members of different groups/tours
live chat with other walkers

find coffee and ice cream shops on the way
link to websites for parks, museums and other sights
offer discount coupons for local stores & cafes
reviews and ratings of sights, city blocks, neighborhoods

stories of local inhabitants and artifacts about them
connect to local politics (eg, zoning, urban planning)

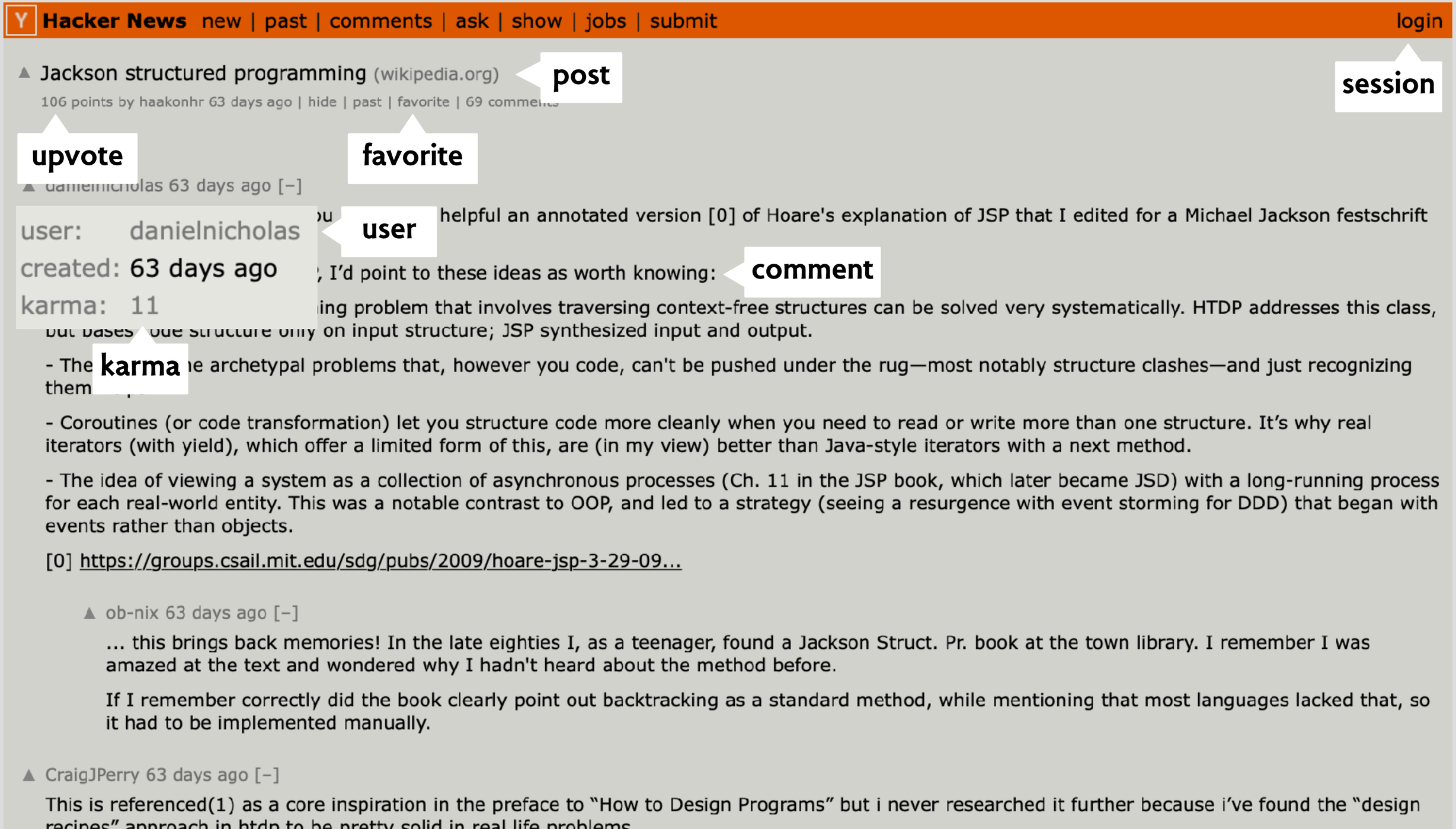
info pops up automatically when in front of building
play audio so you can use app without your phone out
show historical photos of the same site
augmented reality overlay of architectural features
point at building and have app tell you about it

filter landmarks or routes based on period, style, history

record where you've been, shows what you missed
save site as a favorite so you can review it later
send postcard to a friend using an image of the site

kids' mode: simpler explanations
gamifying: points for spotting features or visiting sites

concepts


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
session
upvote
favorite
danielnicholas 63 days ago [–]
user: danielnicholas
created: 63 days ago
karma: 11
user
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:
comment
ing problem that involves traversing context-free structures can be solved very systematically. HTDP addresses this class,
but bases ode structure only on input structure; JSP synthesized input and output.
karma
- The e 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.
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

upvote: a sample concept

concept Upvote

purpose rank items by popularity

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

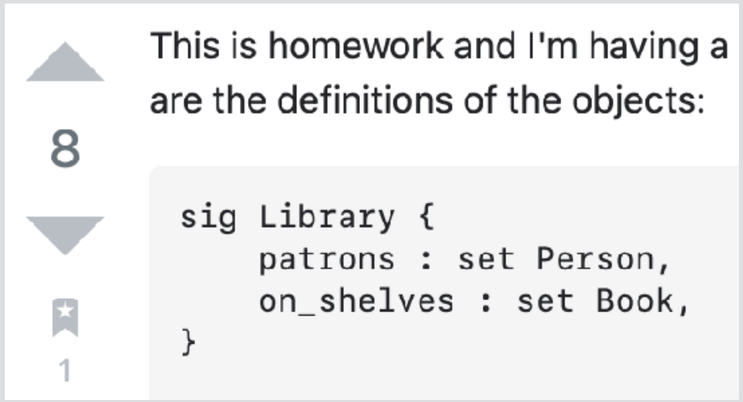
Michael Polanyi (1891-1976)

similar UIs, very different concepts

concept Upvote

purpose rank items by popularity

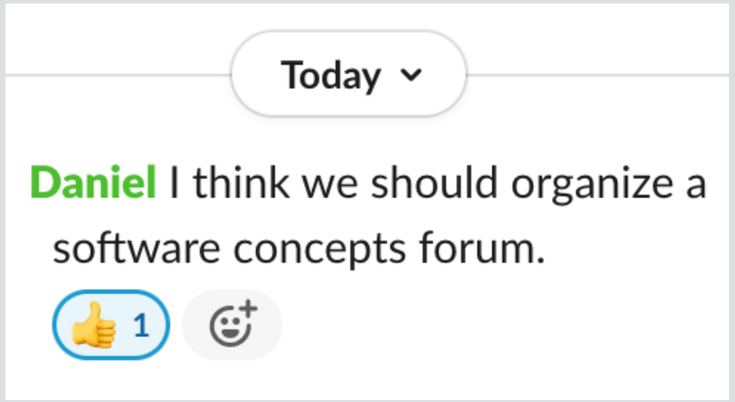
principle after series of upvotes of items, the items are ranked by their number of upvotes



concept Reaction

purpose send reactions to author

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

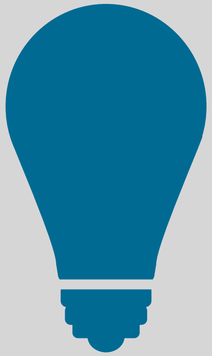


concept Recommendation

purpose use prior likes to recommend

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

what's a concept?



semantic
user facing, not internal
not UI, but underlying function
behavioral not just structural

purposive
fulfills a user need
included for a reason
end-to-end, not a fragment

modular
mutually independent
generic (polymorphic)
reusable in other apps

semantic
user facing, not internal
not UI, but underlying function
behavioral not just structural

purposive
fulfills a user need
included for a reason
end-to-end, not a fragment

modular
mutually independent
generic (polymorphic)
reusable in other apps

login?	User, Session
submit?	Post, Moderation
comment count?	Comment
navbar? breadcrumb?	*X*

converging
on concepts

suppose we consider these features

info pops up automatically when in front of building
save site as a favorite so you can review it later
play audio so you can use app without your phone out
show historical photos of the same site
locate bike rental stations
find coffee and ice cream shops on the way
link to websites for parks, museums and other sights
stories of local inhabitants and artifacts about them

tactics in identifying concepts

fill in missing basic concepts: Building
use a familiar concept: LinkedArticle, Favorite
generalize: LocalBusiness covers museum, cafe, bike station
make it generic: Building assets are photos, audio, etc;
Map points of interest can be buildings or businesses

Building

purpose: collect media assets around sites
principle: basic info & assets stored w/site, then shown when site is selected later

LocalBusiness

purpose: offer basic info about local businesses
principle: store & retrieve by location/category

Map

purpose: show nearby points of interest
principle: after point of interest is registered, it will appear on the map if near current location

LinkedArticle

purpose: make textual info navigable
principle: if you request article, content is displayed with links you can follow to other articles or external assets

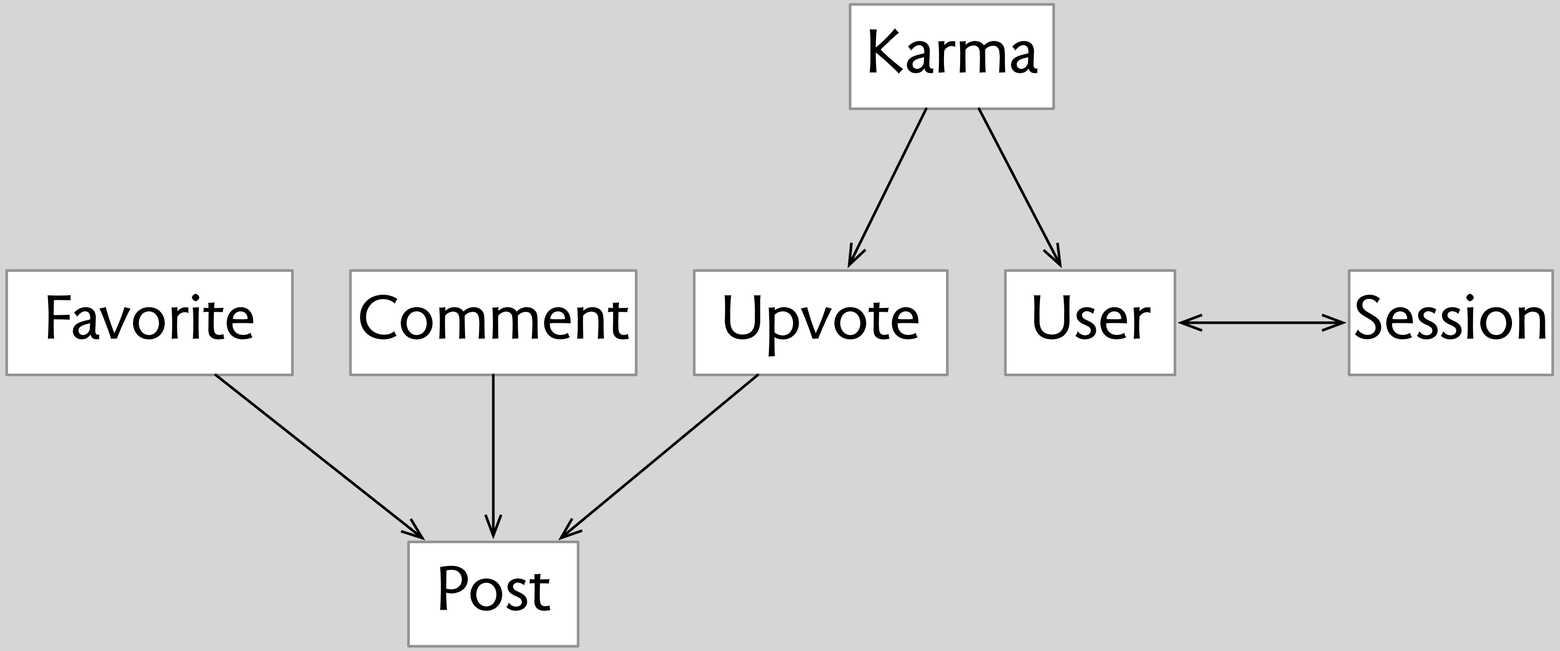
Favorite

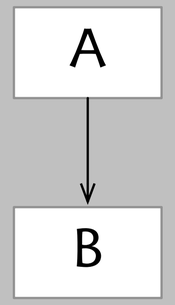
purpose: save items for later review
principle: if mark item, can select and view later

Photo, Audio (standard concepts)

dependency diagrams

hacker news in one diagram





an app that includes
A must include B too

subsets

{Post}

{Post, Upvote}

{Post, Upvote, Karma} ...

what are some dependencies for SiteSpot?

Building

Map

LocalBusiness

LinkedArticle

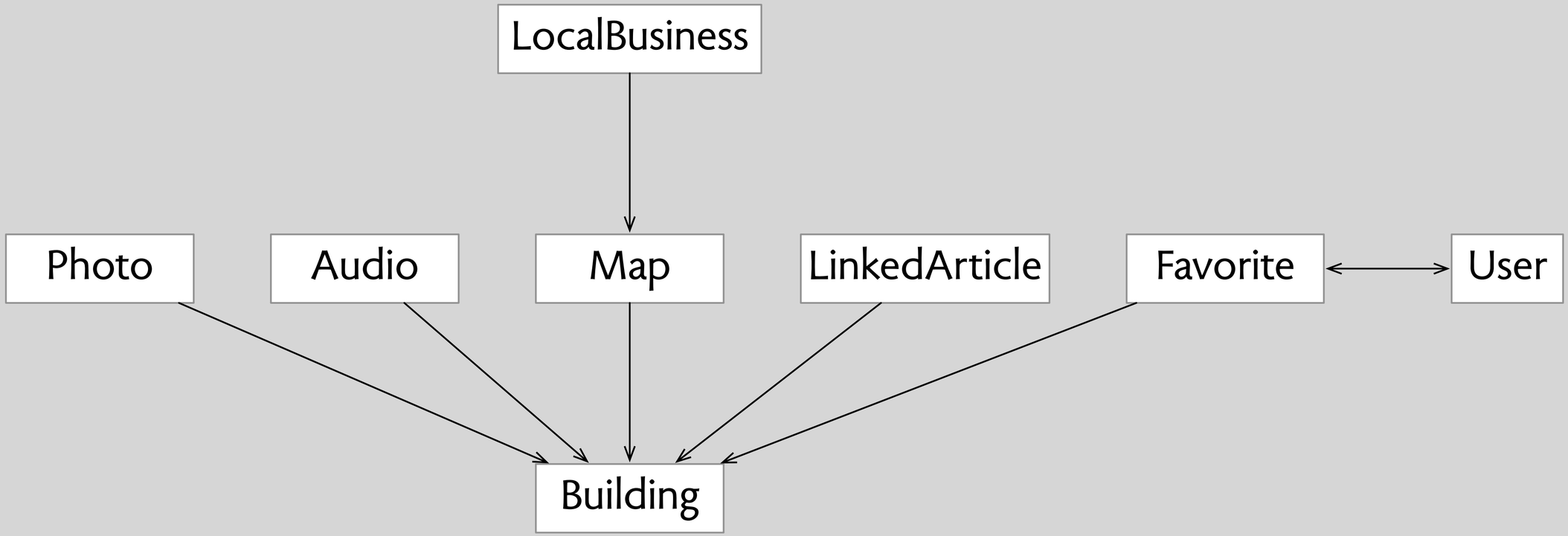
Photo

Audio

Favorite

Coupon

dependencies for SiteSpot



the origin of dependencies

if the dependency diagram shows dependencies what does it mean for concepts to be independent?

two kinds of dependency
extrinsic: from the context of usage
intrinsic: from the software component itself

concepts are decoupled
they have no intrinsic dependencies

examples of intrinsic dependencies
a function that calls another
an object oriented class that references another

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.

Manuscript received June 7, 1978; revised October 26, 1978. The earliest work in this paper was supported by NV Phillips Computer Industrie, Apeldoorn, The Netherlands. This work was also supported by the National Science Foundation and the German Federal Ministry for Research and Technology (BMFT). This paper was presented at the Third International Conference on Software Engineering, Atlanta, GA, May 1978.

The author is with the Department of Computer Science, University of North Carolina, Chapel Hill, NC 27514. He is also with the Information Systems Staff, Communications Sciences Division, Naval Research Laboratory, Washington, DC.

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."

4) "Our SYSGEN was intended to allow us to tailor a system to our customers' needs but it was not flexible enough to suit us."

After studying a number of such systems, 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.

0098-5589/79/0300-0128$00.75

a criterion for allowing intrinsic dependencies

3) The criteria to be used in allowing one program 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;

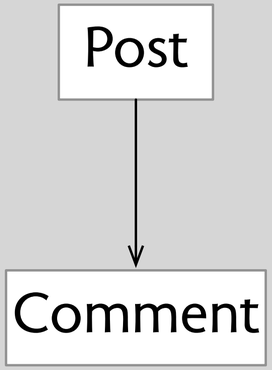
c) there is a useful subset containing B and not A;

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

Parnas's strategy

```
class Post {
  List<Comment> comments;
  ...
  }
```

"Post uses comment"



any app including Post must include Comment too

😢

study the code and extract all the intrinsic dependencies

draw a dependency diagram of all intrinsic dependencies

check that every dependency is acceptable as an extrinsic dependency

summary of what you learned

diverge/converge: two modes of design thinking
values: consider during brainstorming too
concepts: a way to structure functionality
dependences & subsets: viewing a program as a family

let us know what's working for you and what isn't.
we'll use feedback to adjust week by week!
https://tinyurl.com/6104-feedback