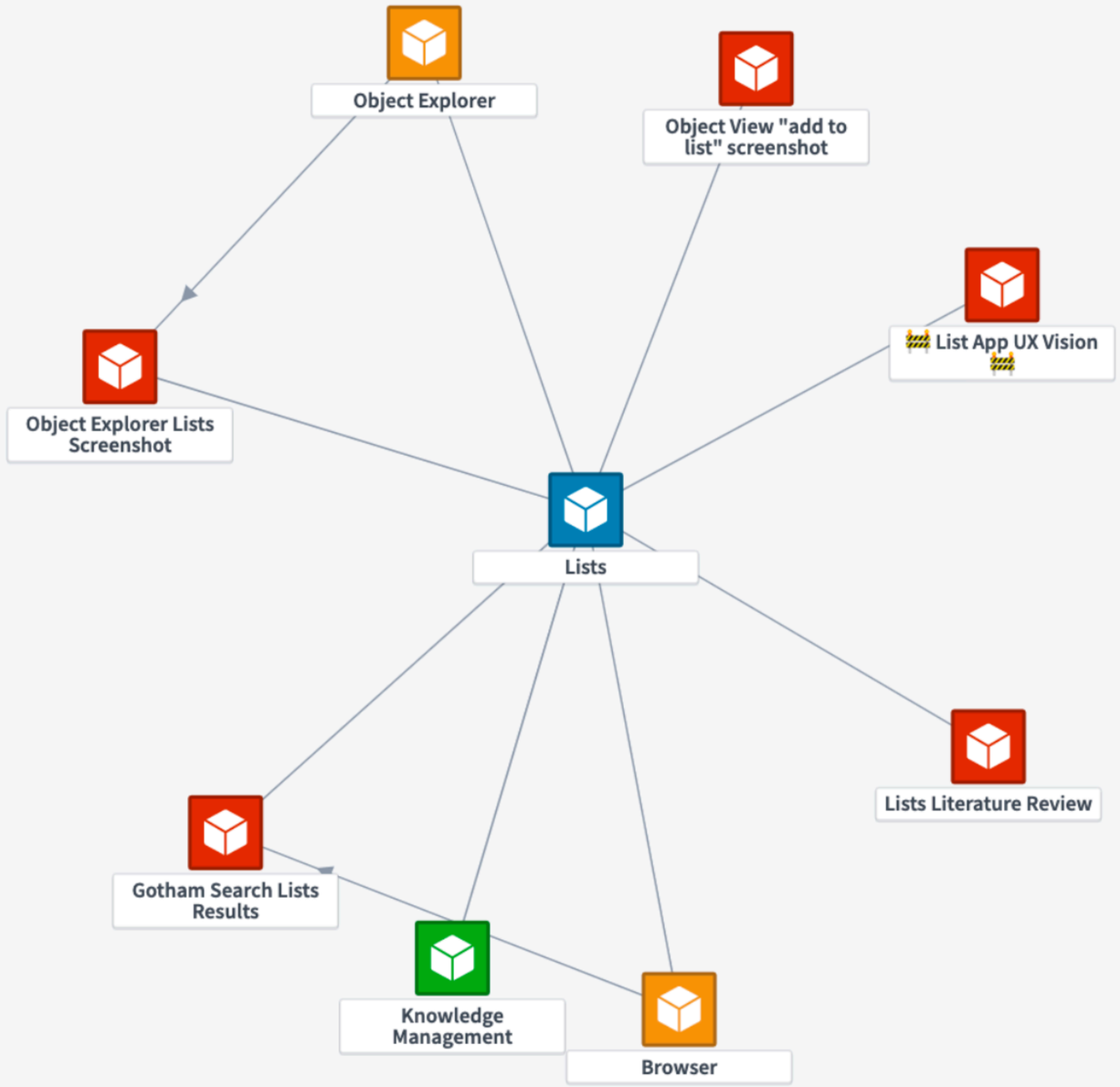
product families & catalogs

Daniel Jackson · Autodesk · Woodinville, WA · Dec 3-5, 2024

concepts at Palantir (2023)

Wilczynski et al, arxiv.org/abs/2304.14975



challenges they were facing
issues not attributable to modules or even products
inconsistent UX across products for similar functions
"conceptual entropy": growing complexity

what they did
integrated concepts into company knowledge base
leaders bootstrapped by writing initial concepts
exploiting existing documents
now 200 concepts recorded, 280 regular users

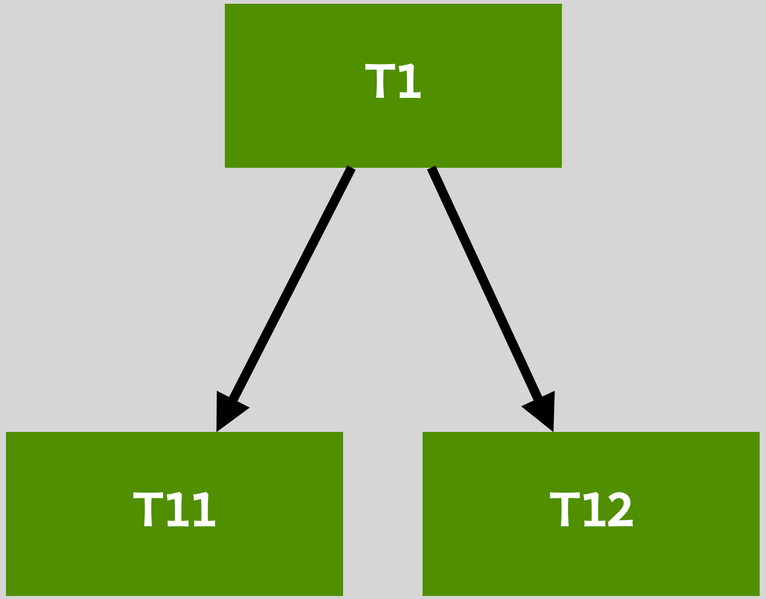
concepts go beyond engineering
concepts used in marketing; IP lawyers interested too

concepts empower PMs
new career path: PMs given ownership of concepts

anticipated impacts
cataloging key assets & avoiding rework
aligning concepts across products, reuse
aligning marketing/design/engineering

a history of programming in 5 minutes

the origins of the problem

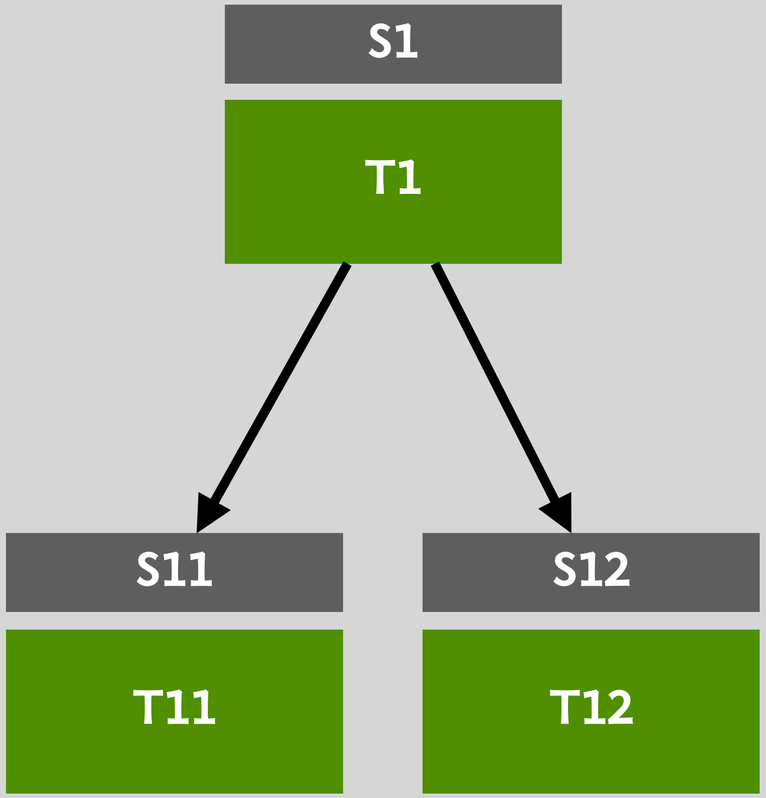


divide and conquer
break task T1 into subtasks T11, T12
implement as modules

a new problem: coupling
if T11 fails, T1 will fail too
to understand T1, you need to understand T11
if you change T11, may need to change T1 too

much of software engineering
is focusing on mitigating this problem

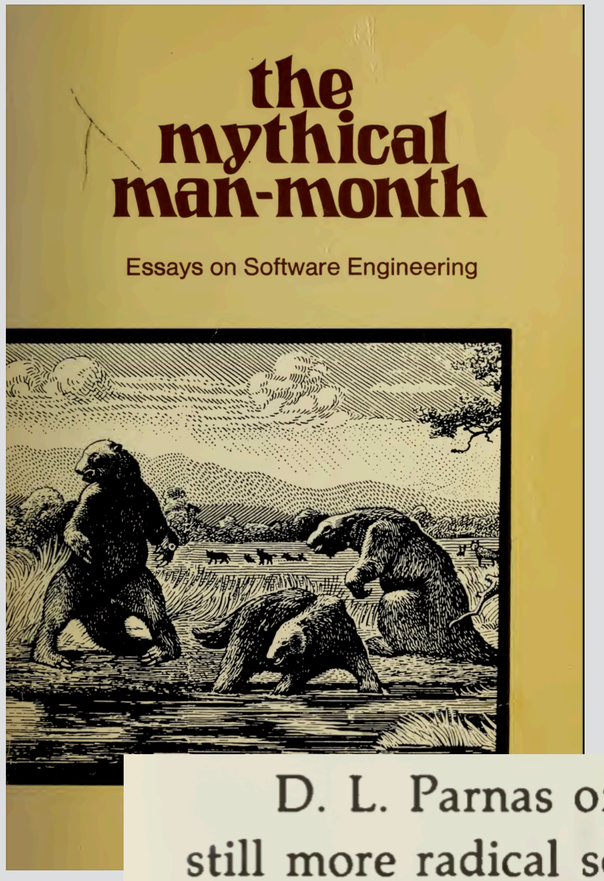
advance #1: specifications as firewalls



change the dependencies
T1 no longer depends on T11 and T12
instead it depends on the specs S11 and S12

modular reasoning
show that T1 satisfies S1 assuming S11 and S12
show that T11 satisfies S11, T12 satisfies S12

in 1975, this was controversial!

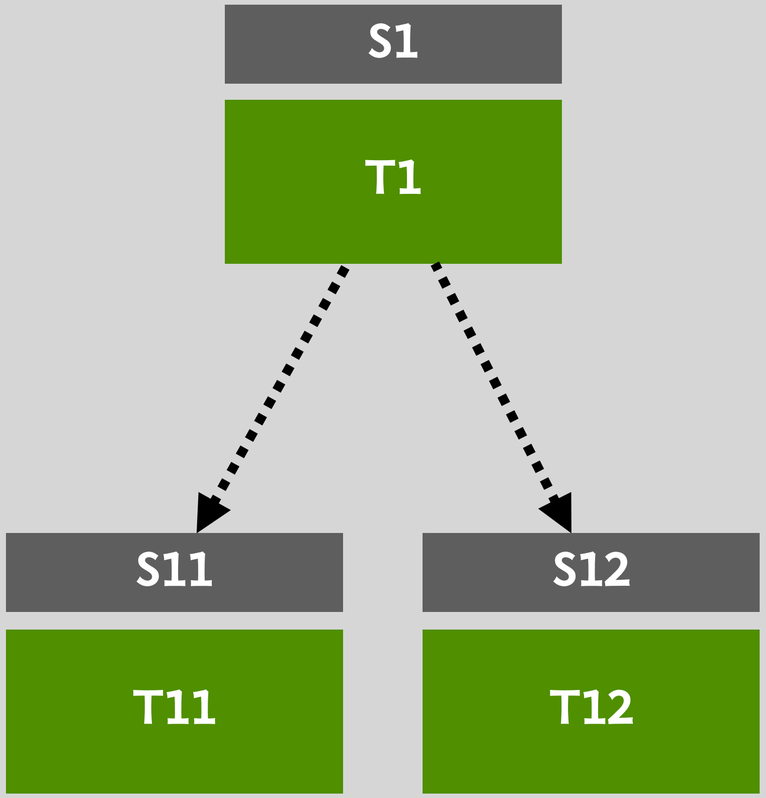


David Parnas was right, and I was wrong about information hiding. I am now convinced that information hiding, today often embodied in object programming, is the only way of raising the level of software design.

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.

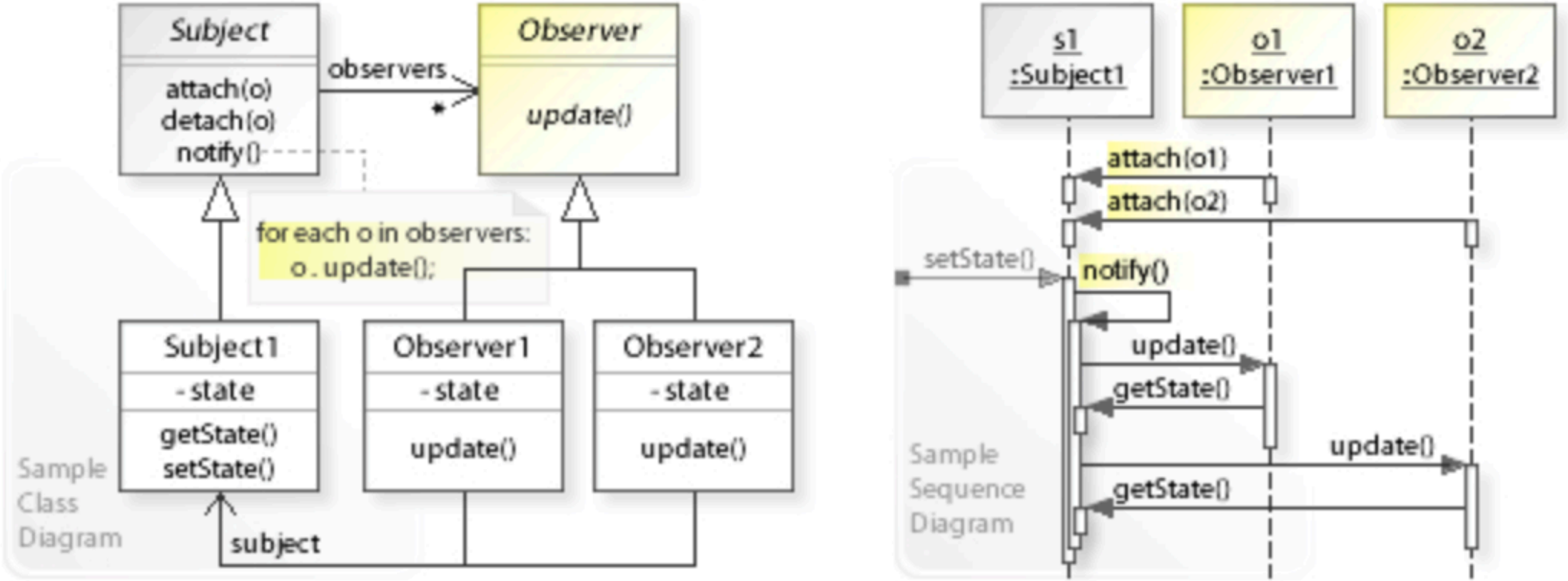
advance #2: OOP and dynamic configuration



since T1 only needs an S11 and an S12
don't need T11 and T12 in particular
can avoid naming T11 and T12 in T1
pass them in at runtime instead

a new problem
can no longer find dependencies statically

this is how “gang of four” patterns work



A sample UML class and sequence diagram for the observer design pattern. [6]

advance #3: design dependencies explicitly

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

provide guidance for which dependencies are ok

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;

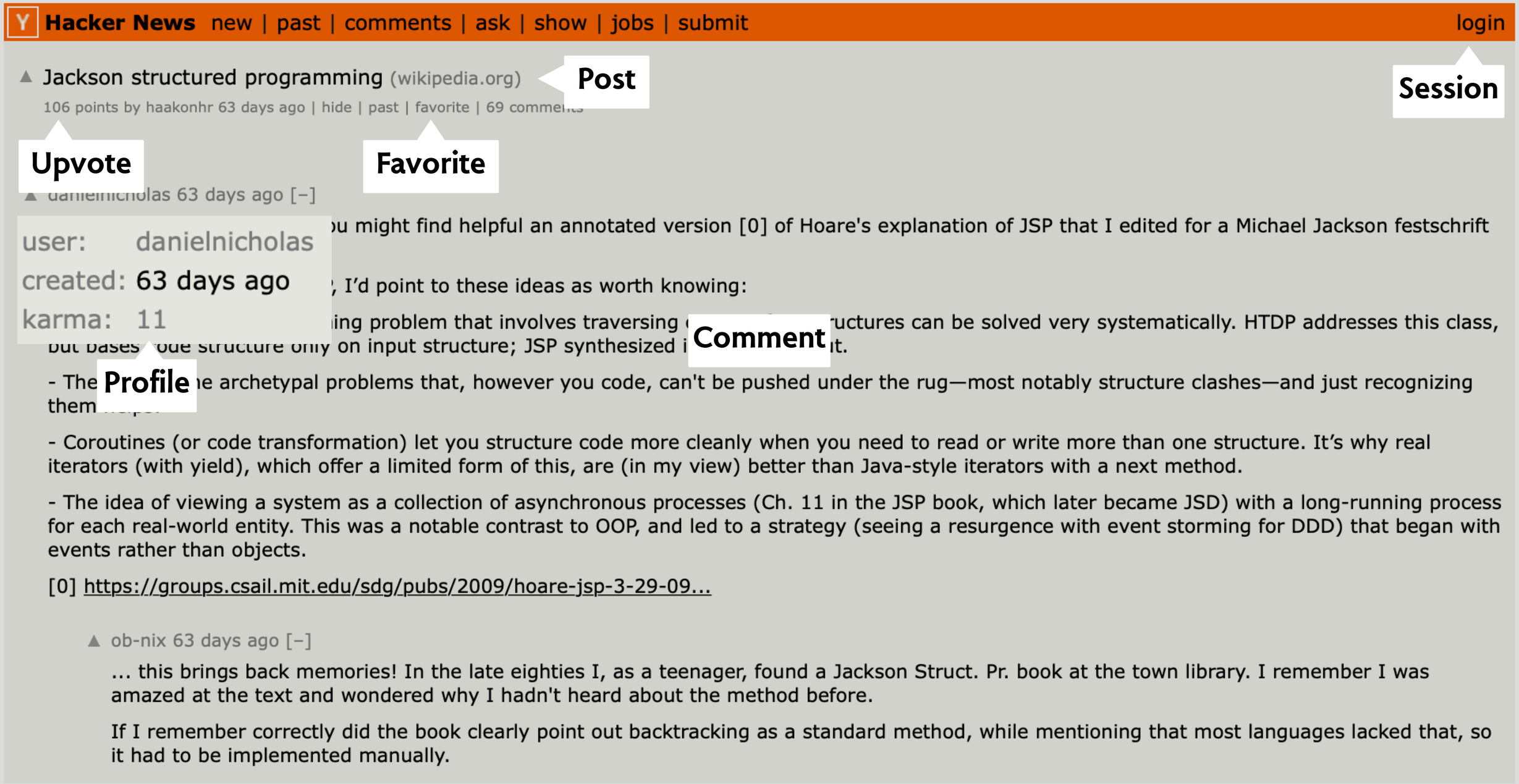
b) B is not substantially more complex because it is not allowed to use A;

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

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

how OOP encourages dependencies

most apps are made from familiar functions



let's build it with OOP

```
class User {
  String name;
  String password;
  User register (n, p) { ... }
  User authenticate (n, p) { ... }
 }
```

```
class Post {
  User author;
  String body;
  Post new (a, b) { ... }
}
```

adding upvoting

```
class User {
  String name;
  String password;
  User register (n, p) { ... }
  User authenticate (n, p) { ... }
}
```

```
class Post {
  User author;
  String body;
  Set [User] ups, downs;
  Post new (a, b) { ... }
  upvote (u) { ... }
  downvote (u) { ... }
}
```

adding karma

```
class User {
  String name;
  String password;
  int karma;
  User register (n, p) { ... }
  User authenticate (n, p) { ... }
  incKarma (i) { ... }
  bool hasKarma (i) { ... }
}
```

```
class Post {
  User author;
  String body;
  Set [User] ups, downs;
  Post new (a, b) { ... }
  upvote (u) { ... }
  downvote (u) {
    if u.hasKarma (10)  ... }
}
```

adding commenting

```
class User {
  String name;
  String password;
  int karma;
  User register (n, p) { … }
  User authenticate (n, p) { … }
  incKarma (i) { … }
  bool hasKarma (i) { … }
}
```

```
class Post {
  User author;
  String body;
  Set [User] ups, downs;
  Seq [Post] comments;
  Post new (a, b) { … }
  upvote (u) { … }
  downvote (u) {
    if u.hasKarma (10)  … }
  addComment (c) { … }
}
```

what's wrong with this code?

```
class User {
  String name;
  String password;
  int karma;
  User register (n, p) { ... }
  User authenticate (n, p) { ... }
  incKarma (i) { ... }
  bool hasKarma (i) { ... }
}
```

```
class Post {
  User author;
  String body;
  Set [User] ups, downs;
  Seq [Post] comments;
  Post new (a, b) { ... }
  upvote (u) { ... }
  downvote (u) {
    if u.hasKarma (10)  ... }
  addComment (c) { ... }
}
```

User authentication

Posting

Upvoting

Commenting

Karma

no separation of concerns
Post class contains posting, commenting, upvoting, karma

→ **classes are novel & not reusable**
Post class won't work in an app that doesn't have karma points

dependencies between files
Post class calls *User* class to get karma points

→ **can't be built independently**
to build *Post* class, need *User* class to have been built already

a different way

```
concept User {
  Map [User, String] name;
  Map [User, String] password;
  User register (n, p) { ... }
  User authenticate (n, p) { ... }
}
```

```
concept Karma [U] {
  Map [U, Int] karma;
  incKarma (u, i) { ... }
  hasKarma (u, i) { ... }
}
```

concerns now cleanly separated

coupling is gone: refs are polymorphic

```
concept Post [U] {
  Map [Post, U] author;
  Map [Post, URL] url;
  Post new (a, u) { ... }
}
```

```
concept Upvote [U, I] {
  Map [U, I] ups, downs;
  upvote (u, i) { ... }
  downvote (u, i) { ... }
}
```

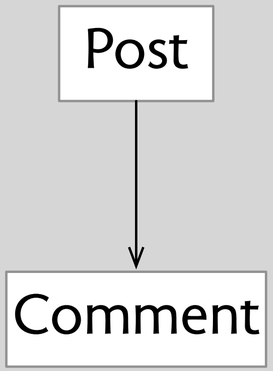
```
concept Comment [U, T] {
  Map [Comment, U] author;
  Map [Comment, T] target;
  Map [Comment, String] body;
  Comment new (a, t, b) { ... }
}
```

```
when HTTP.request (downvote, u, i)
sync
  Karma.hasKarma (u, 10)
  Upvote.downvote (u, i)
```

natural OOP coding produces bad dependencies

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

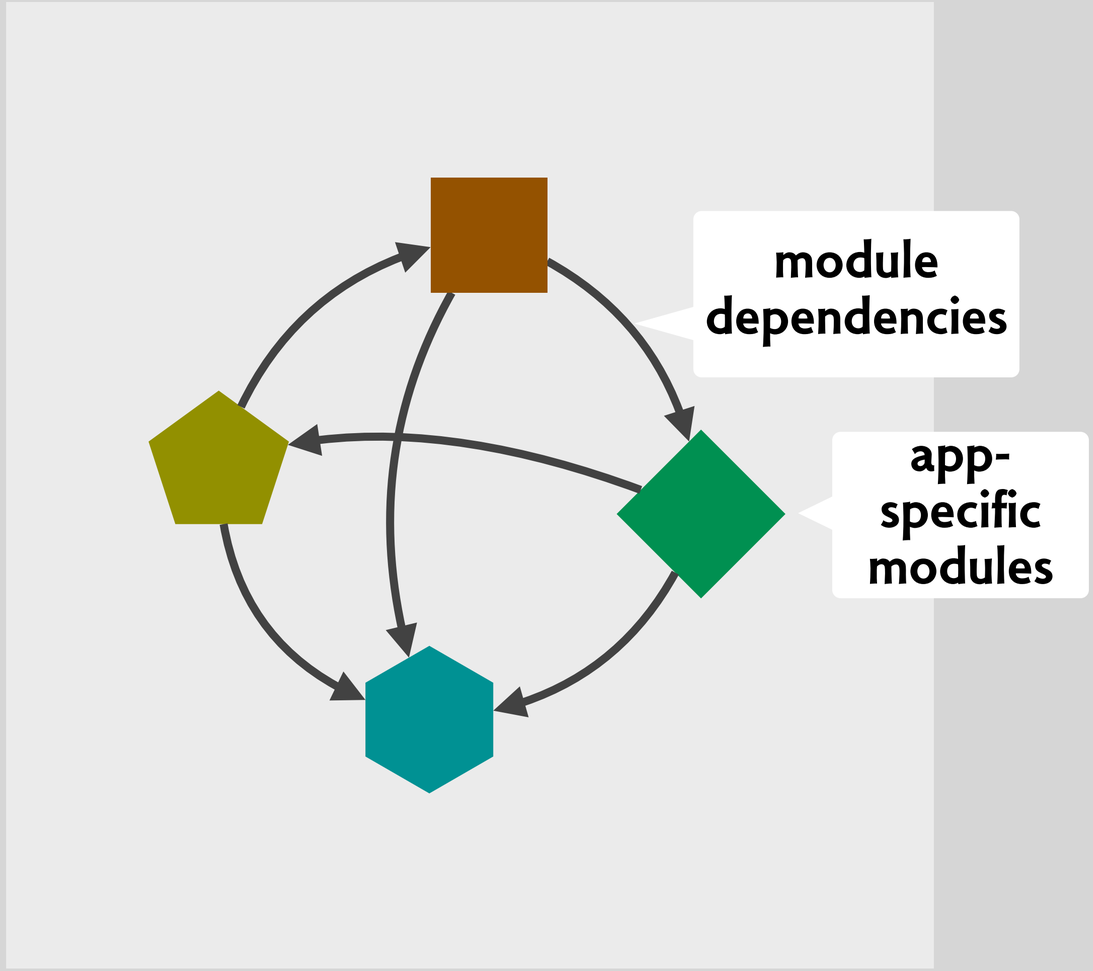
"Post uses comment"



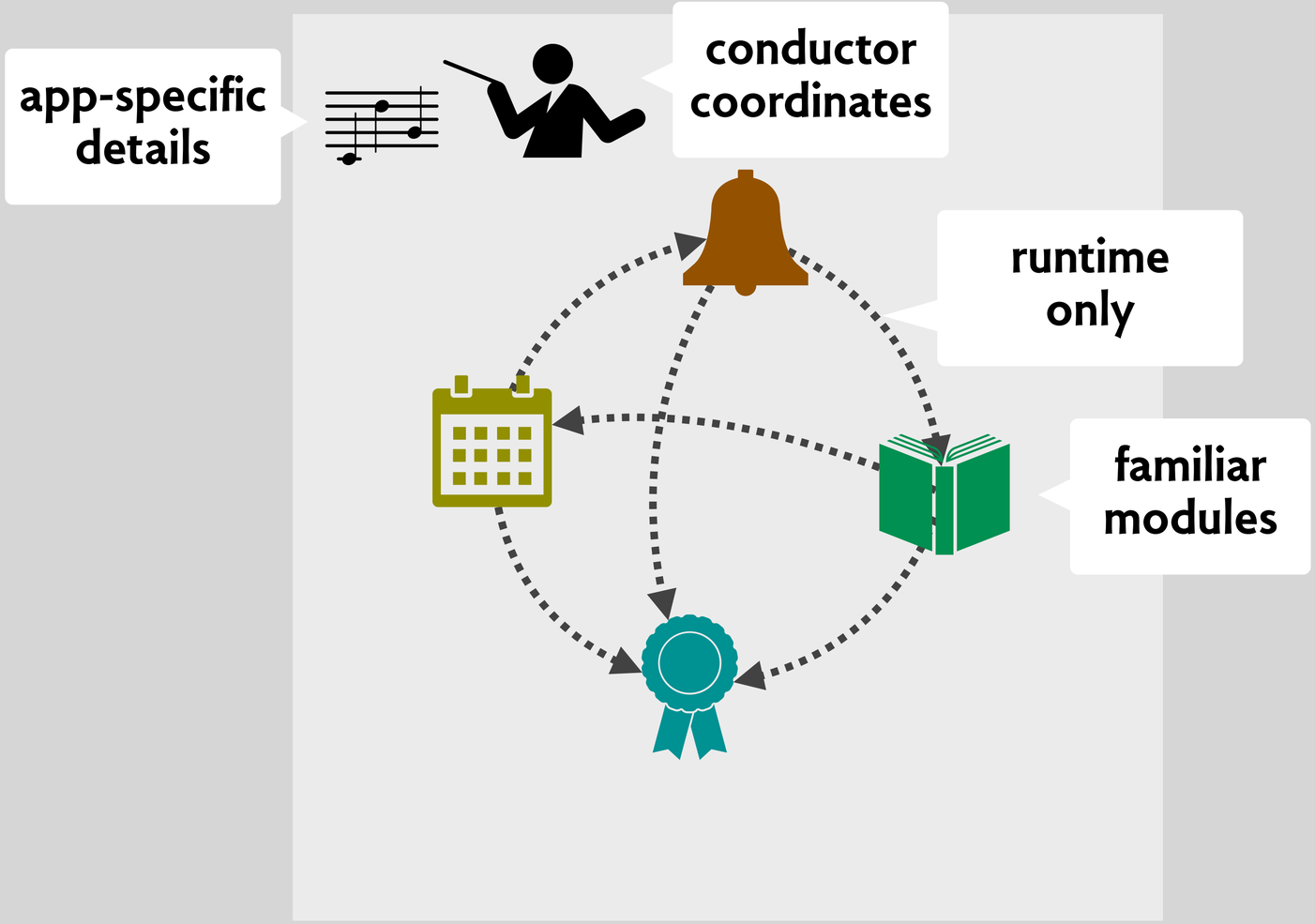
any app including Post
must include Comment too

dependencies & concepts

what conventional programming looks like

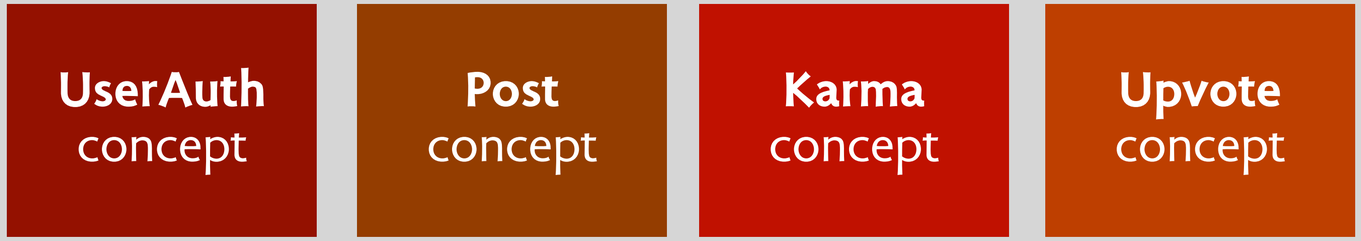


a different approach using concepts



icons by Luis Prado & Zach Bogart, Noun Project

concepts are free-standing

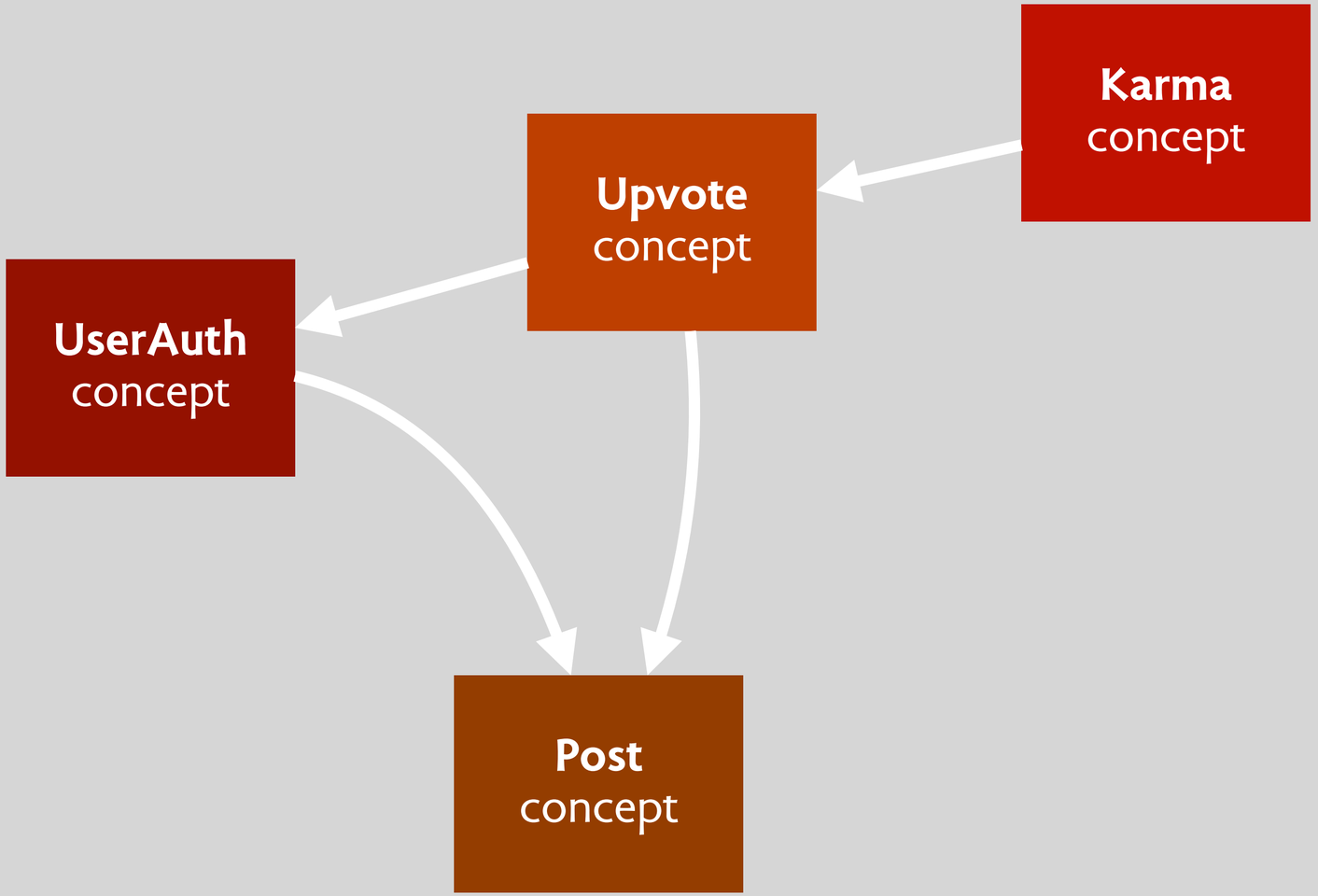


users can **understand** concepts independently

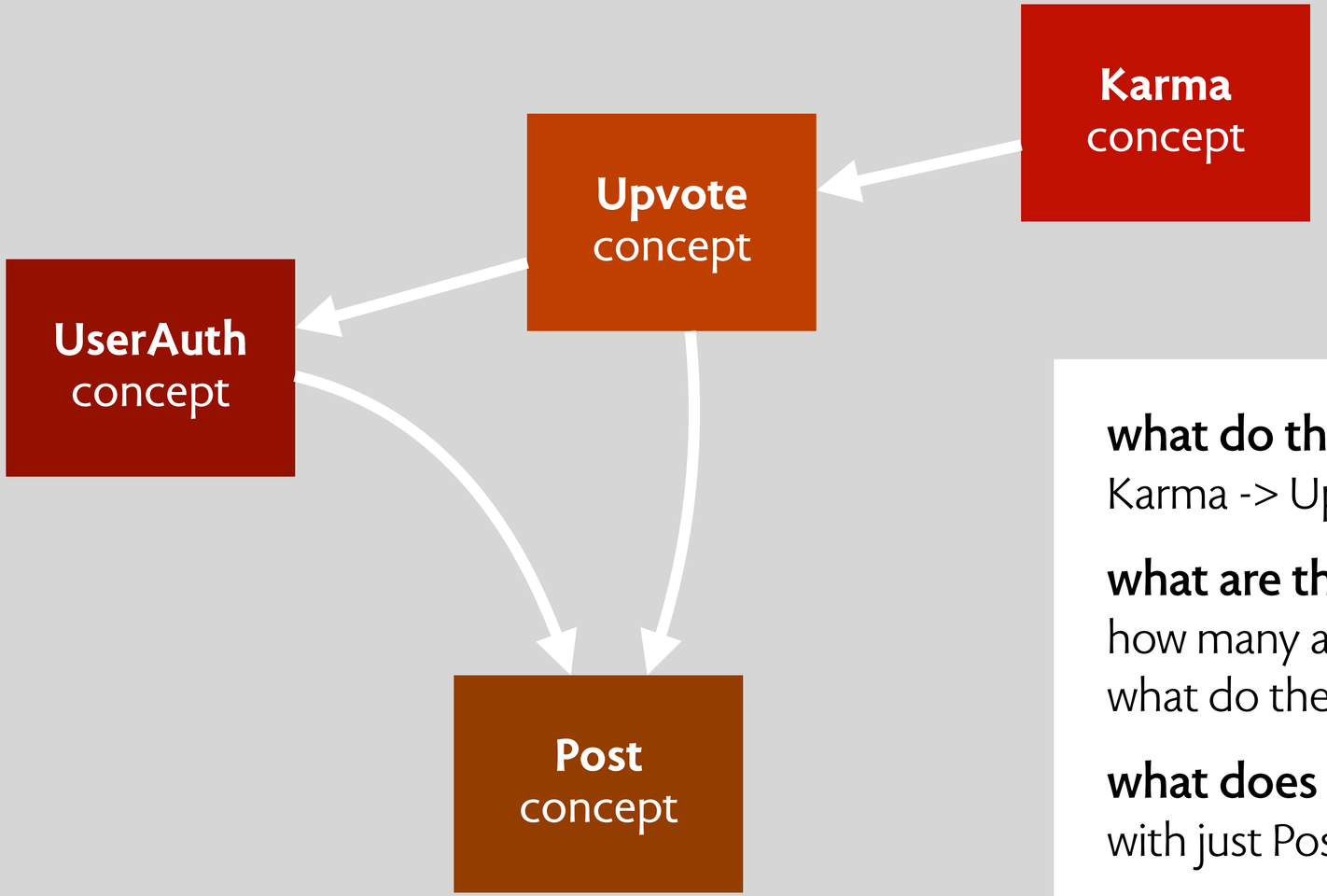
designers can **design** concepts independently

programmers can **code** concepts independently

but Parnas's subsets are still relevant



check your understanding



what do the arrows mean?
Karma -> Upvote?

what are the subsets?
how many are there?
what do they include?

what does an app look like
with just Post, eg?

concept instances & indexing

concept scoping principles

every concept can be
instantiated: perhaps many times
indexed: one some objects

small scope, many instances
simplifies concept definition
separation of concerns
opportunity for concurrency

larger scope, few instances
support more functionality

checklist: concept state

- ✔ enough for concept function
- ✔ but no more than needed

concept Labeling [Item]

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

example: how many labeling instances?
one for each macOS user, or one for the whole filesystem?

check your understanding: which is correct?

concept User

state
username: UserName
password: Password

concept UserAuth [User]

state
username: User -> **one** UserName
password: User -> **one** Password

checklist: concept state

- ✓ enough for concept function
- ✓ but no more than needed

check your understanding: which is best?

concept Labeling [Item]

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

checklist: concept state

- ✓ enough for concept function
- ✓ but no more than needed

one instance for all of Gmail

one instance for each Gmail user

a design puzzle: which is best?

concept Reservation

state
a set of resources
a set of bookings
for each booking
 a resource
 an owner

one instance for OpenTable

one instance for each restaurant

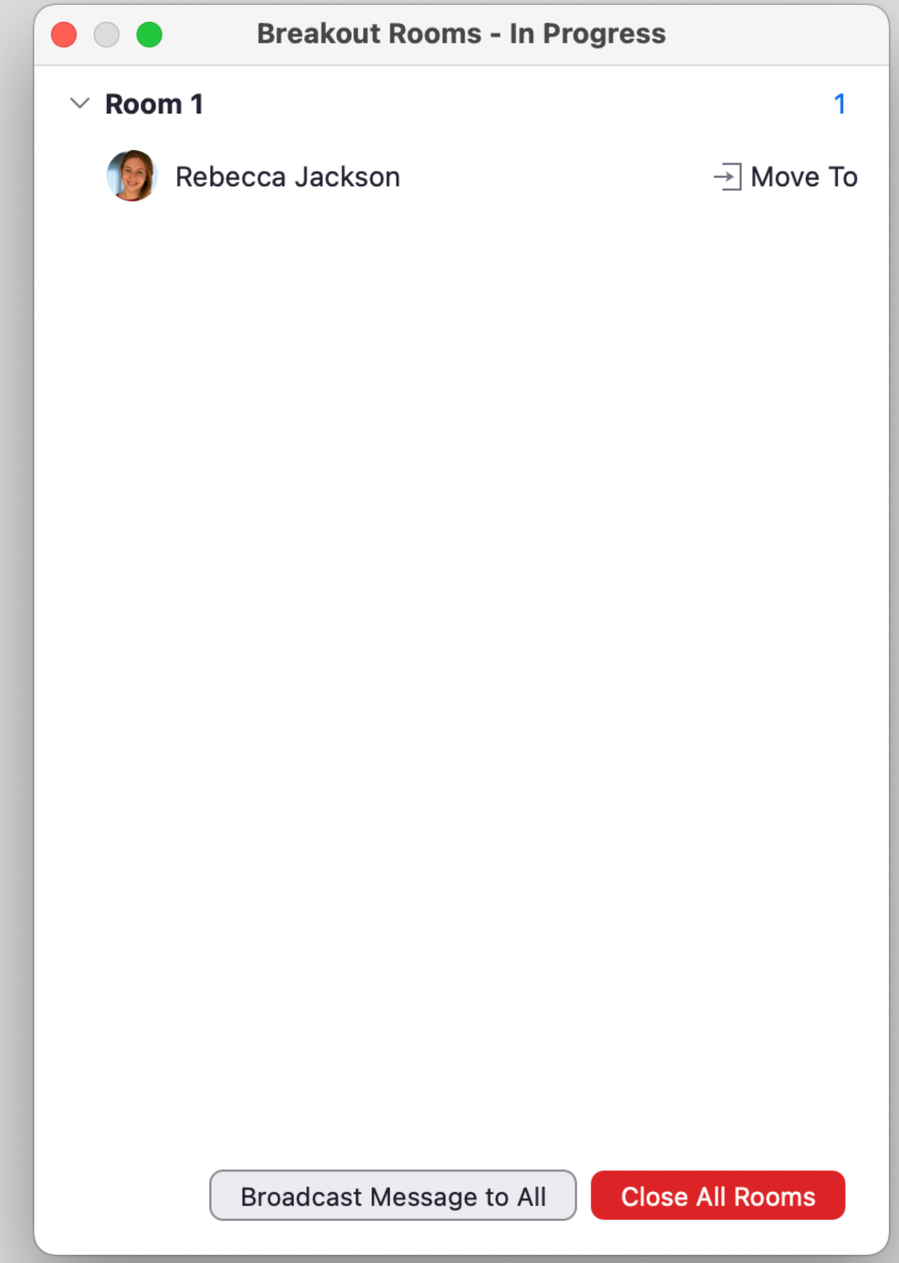
one instance for each restaurant/location pair

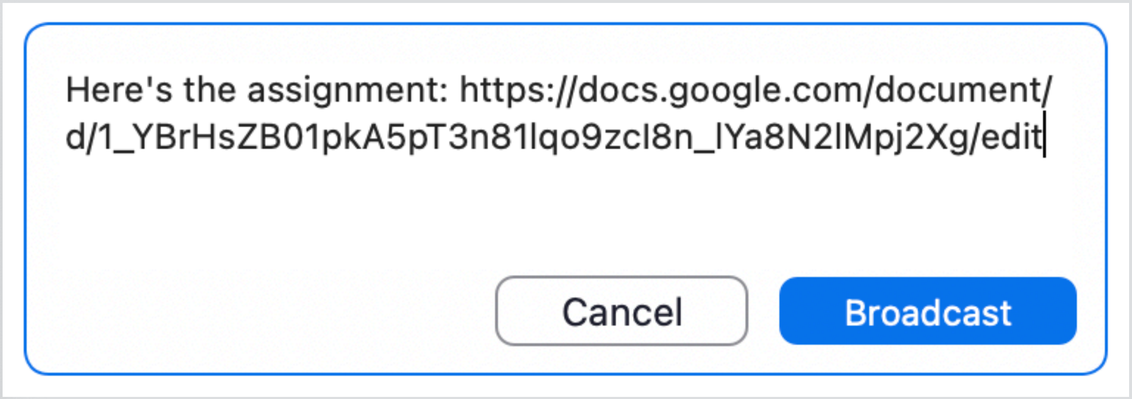
checklist: concept state scope

- ✓ enough for concept function
- ✓ but no more than needed

Zoom chat: design issues

breakout rooms, chat & broadcast





when in breakout room
chat is limited to members of the room
can't even message the host of the meeting
and host can't message all meeting participants

Zoom's solution
add a new concept called Broadcast
similar to Chat, but can't reply, click on links, or persist

what do you think is going on in this design?

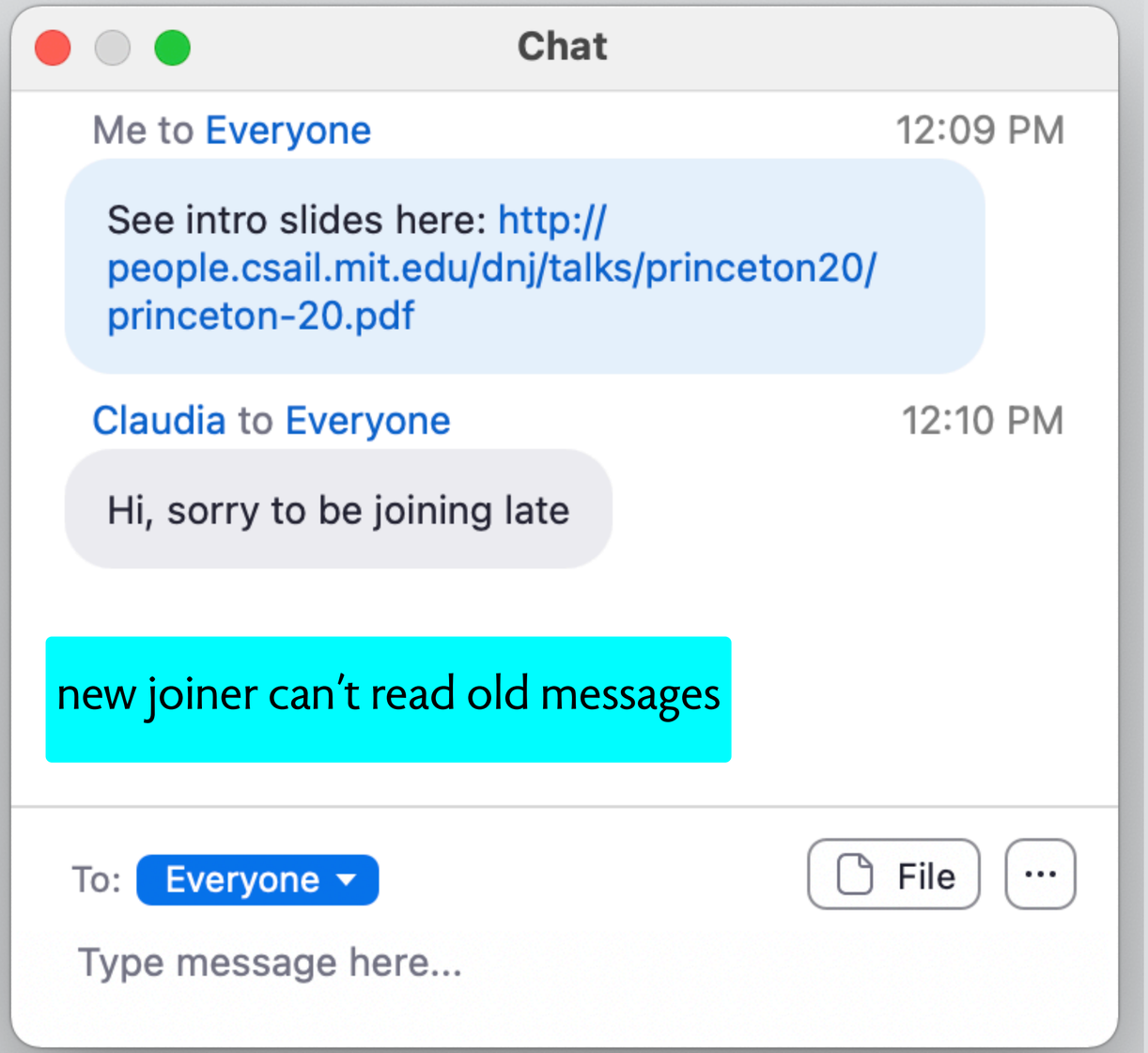
Broadcast a message to all breakout rooms

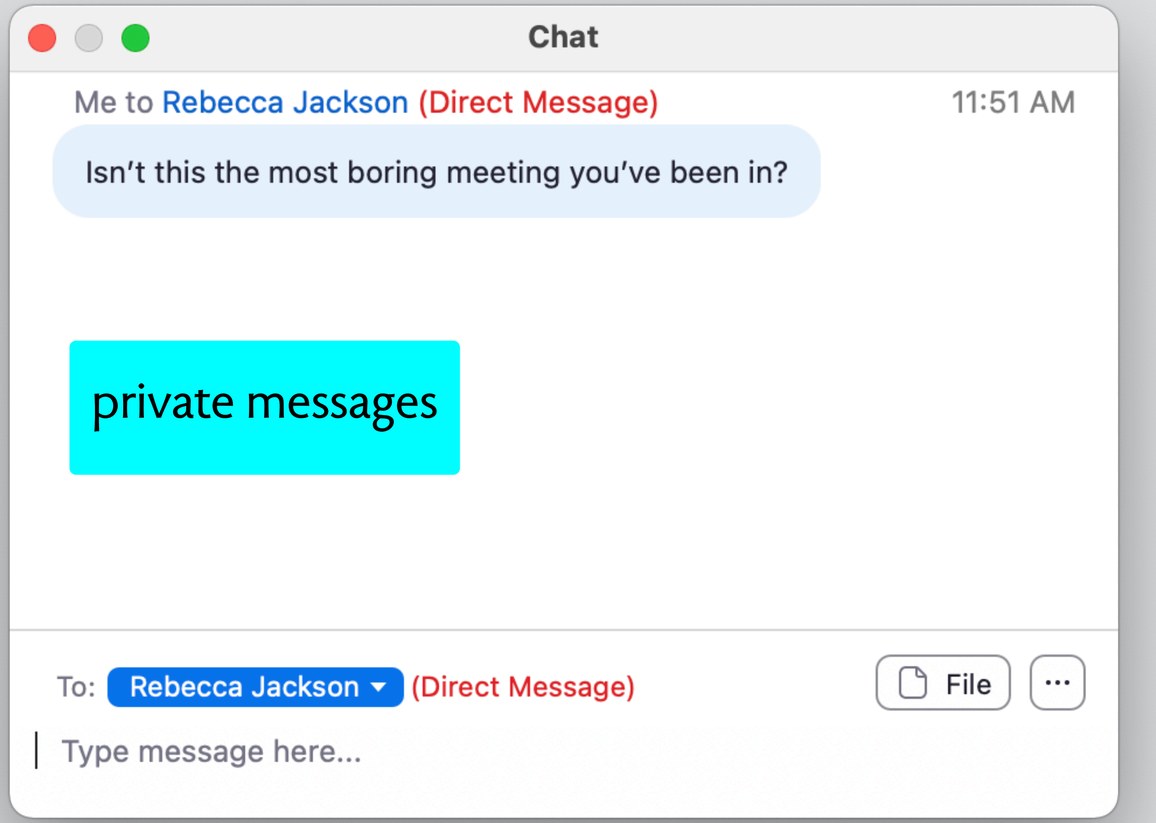
The host can broadcast a message to all breakout rooms, to share information with all participants.

Note: This must be enabled in your breakout room settings.

1. In the meeting controls, click **Breakout Rooms**  .
2. Click **Broadcast**, and select **Broadcast Message**.
3. Enter your message and click the send icon  .
   The message will appear for all participants in breakout rooms, and disappear after ~10 seconds.

other complications





a concept framing

concept Chat

state
a set of members
a set of messages
for each message
  a sender, a body, a time

for each member
  a join time

actions
join (u: User)
leave (u: User)
post (u: User, m: Text): Message
can_view (u: User, m: Message)

is this state sufficient?

how are chats indexed?

- one instance for each Zoom meeting occurrence
- one instance for each Zoom meeting, all occurrences
- one instance for each breakout room within a meeting

loss of design knowledge?

The Zoom Chat is broken (previous messages disappear after Breakout Rooms)

mrjankeck
Explorer

2023-01-07 02:47 AM

⚠️ The Zoom Meeting Chat is broken!!

Two of the Zoom Tech Hosts on my team have discovered that previous chat messages disappear once you enter or come back from a Breakout Room.

For anyone who has been sharing instructions in the chat before you send participants off to do an acti
has some very serious consequences.

original design
when move to breakout, chat from main room cleared
so how to share instructions for breakouts?

zoom fixes this
messages from chat copied to breakout room

“new meeting chat experience”
threads, quoting, formatting in chat

a regression
now messages no longer copied to breakout room

exercise

take a collection of Autodesk concepts
for now, don't worry too much about exact definitions
eg, Model, Analysis, Evaluation, Proposal, Template, ...

construct a subset diagram for them
does the diagram reflect the history of the product's development?
are all the sensible subsets realizable in practice?
what else can you learn from the diagram?

takeaways

concepts are independently defined
a concept can be reused in a different app
doesn't require the presence of other concepts

but in a single app
only certain combinations of concepts will make sense
these subsets define a family of possible applications

the subset dependency diagram
can clarify which concepts are core, what order to develop in, etc