desperately seeking concepts
Conceptual integrity is the most important consideration in system design (1975)

I am more convinced than ever. Conceptual integrity is central to product quality (1995)
It is clear that users attempt to make sense—by building mental models—of the behavior of a system as they use it. If a simple model is not explicitly or implicitly provided, users formulate their own myths about how the system works... [I]f the user is to understand the system, the system has to be designed with an explicit conceptual model that is easy enough for the user to learn. We call this the intended user’s model, because it is the model the designer intends the user to learn.
When the designers fail to provide a conceptual model, we will be forced to make up our own, and the ones we make up are apt to be wrong. Conceptual models are critical to good design.
Conceptual modelling is the activity of formally describing some aspects of the physical and social world around us for purposes of understanding and communication...

We are interested in conceptual modelling because it is useful in rationalizing and supporting information system development.

John Mylopoulos. Conceptual modeling and Telos, 1992
where's the concept?

3 entities: how many concepts?

Customer \(\xrightarrow{\text{reservations}}\) Reservation \(\xrightarrow{\text{slot}}\) Slot

Customer \(\xrightarrow{\text{reservedSlots}}\) Slot

is the relation a concept?
The conceptual modelling community not only has no clear, general agreement on what its models model, it also has no clear picture of what the available options and their implications are. One common claim is that models represent concepts, but there is no clear articulation of what the concepts are.

Chris Partridge, Cesar Gonzalez-Perez and Brian Henderson-Sellers. Are Conceptual Models Concept Models? 2013
why it matters

modularity is the essence of design
provides separation of concerns & structure for reuse

without concepts, what are conceptual models?
like formal models of a domain in Alloy (or Z, or Statecharts...)

we have an intuition that concepts are distinct
restaurant reservation app based on concept of “reservation”?
Ava is a party planner

Bella is having a party

does the name change for Ava too?
answer: it depends

if Ava just shares Bella Plan with Bella and Bella renamed the folder, Ava sees no change

if Ava shares a folder Bella Party with Bella containing the folder Bella Plan, and Bella renamed Bella Plan then Ava does see the change
same two cases for deletion

Bella deletes Bella Plan from shared folder Bella Party

Bella deletes shared folder Bella Party
two concepts

name follows metadata concept

name is part of unixFolder concept
**concept** metadata

**purpose** tag items with properties for easy lookup

**structure**
  val: Item -> Property -> Value

**actions**
  define (i: Item, p: Property, v: Value)
    i.val[p] := v
  find (out is: Item, p: Property, v: Value)
    is = {i | i.val[p] = v}
  read (i: Item, p: Property, out v: Value)
    v := i.val[p]

**principle**
  define(i, p, v); **no** define(i, p,...); find(is,p,v)
  => i **in** is

---

**concept** unixFolder

**purpose** organize named items

**structure**
  member: Folder -> Name -> Item

**actions**
  add (i: Item, to: Folder, n: Name)
    to.member[n] := i
  rename (i: Item, f: Folder, old, new: Name)
    f.member := f.member - old->i + new->i
  find (f: Folder, n: Name, out i: Item)
    i := f.member[n]

**principle**
  add(i, f, n); **no** rename(i, f,...) **or** add(i',f,n);
  find(f, n, i') => i' = i
a real dropbox disaster

how to make space: find big files & delete ones you don’t recognize
Someone accidentally deleted thousands of files in my company Dropbox: how can I quickly undelete them?
Friends don't let friends delete shared Dropbox items

Christopher Breen
@BodyofBreen
Sep 9, 2013 5:00 AM

Reader Paul Cramblett has a problem with others who just don’t know how to share. He writes:

I maintain a Dropbox folder that I use to share files with a select group of friends. I've tried to explain how Dropbox works to these people but someone invariably drags all the files out of the folder, which means they're no longer available to the rest of us. Is there some way to prevent files from being removed by someone who doesn't understand the difference between “copy” and “move”?
Survey of Dropbox users (MIT CS undergrads)

Correctly predicting behavior

- Good knowledge: 60.0%
- Average knowledge: 20.0%
- Poor knowledge: 40.0%

Legend:
- Blue: Delete shared folder results in leaving
- Green: Delete shared subfolder removes it
the big picture
what caused the dropbox problem? not these things

- lack of technology
- bugs in the code
- classic UI design flaws
get the concepts right

metadata
unixFolder

for robust, usable software...

understand the user

design the user interface

avoid bugs in code
levels of UX design

physical
- color, size, layout, type, touch, sound
- Perceptual Fusion, Fitt's Law, Accessibility

linguistic
- icons, labels, tooltips, site structure
- Consistency, Info Foraging, Navigation Aids

conceptual
- semantics, actions, data model, purpose
- Undo, Norman's mapping, mental model alignment

concrete  abstract
a story of style
**Introduction**

How can we improve the quality of software? Make it more usable, robust and secure? Many responses to this challenge make a fundamental assumption: that quality is achieved by eliminating defects. It seems like a plausible enough idea. If you can find the parts of the interface that confuse users and polish or replace them, that will surely make it more usable. And if you can remove the bugs that cause the most frequent crashes, that should make it more robust. And how else to achieve security except by patching the vulnerabilities that hackers might exploit?

**Defects**

The assumption that defect elimination is the key to better software is so widespread that it is rarely questioned (and often not even explicitly articulated). Companies that make software like it because it can be applied incrementally, without major disruptions to their development process or to an often shaky codebase. Tool vendors promote it because it helps sell their products. Researchers adopt it because it makes their contributions easier to measure, and because they fear being accused of utopianism if they suggest avoiding defects in the first place.
**concept** style

**purpose** consistent formatting

**structure**
- defined: Style -> one Format
- style: Element -> one Style
- format: Element -> one Format = style.defined

**actions**
- define (s: Style, f: Format)
  - s.defined := f
- assign (e: Element, s: Style)
  - e.style := s

**principle**
- after define(s,f); assign(e1,s);
- assign(e2,s); define(s,f’)
- observe e1.format = e2.format = f’

---

name: essential for knowledge capture

purpose: why the concept exists

structure: localized data model

actions: observable & atomic

---

There is no problem in computer science that cannot be solved by introducing another level of indirection. *David Wheeler*

---

OP justifies & explains design how behavior fulfills purpose

---

Michael Polanyi operational principle
The invention of style

Tim Mott visits Ginn in 1974 and brings the idea of styles to PARC.

Charles Simonyi’s team implements style in the Bravo text editor.

Simonyi brings style to Microsoft in 1983.

Apple Pages 2005 adds style concept.

other instances of style

Powerpoint color schemes

Indesign swatches

Keynote image styles
non-instances: “pseudo-style”

Apple color swatches

TextEdit “styles”
a concept handbook

**Design variants**
- override formats
- style inheritance
- next style
- partial styles
- shareable stylesheet

**Typical uses**
- formatting paragraphs & characters
- formatting graphic objects
- Word, Pages, CSS, ...

**Concepts indexed by purpose**
- consistent formatting: style, template, copy settings, ...

**Known issues**
- deleting styles: what happens to elements?
- copying elements between documents
- need for “as is” values
- troublesome properties (eg, fontstyle)

**Implementation hints**

**Often used with**
- paragraph format
key properties of a concept: style as an example

- **inventive**: style has a long history of creativity & refinement
- **purposeful**: for consistency of formatting, esp. in large documents
- **behavioral**: “if you update the style of multiple paragraphs their formatting all changes in concert”
- **self-contained**: style concept independent of format, paragraph, typeface
- **reusable**: style in Keynote inspired by style in Pages, inspired by Style in Word...

-not domain entities that are just “out there”

-not arbitrary fragments of functionality

-not data models or ontologies

-not datatypes or modules

-often not domain-specific
composing concepts
weakest: existence coupling
most common: action synchronization
tightest: structure synchronization

folder sortable by volume!

property added in Lion (2011)
designing on purpose
understanding why: the key to usability

Macintosh Trash
- wrong purpose
- deleting things
- undeleting things
- right purpose

Photoshop Layers
- stacking objects
- non-destructive editing

Available Funds
- signal that deposits are safe
- permission to use

Smart Access
123-456 1234 567
Available funds
Account balance
$1,700.00
$1,700.00
The problem for Twitter is that the “favorite” function had developed a range of uses over time, many of which are known only to the journalists and social-media experts who spend all their time on the service. For some (including me), **clicking the star icon was a way of saving a tweet for later**, or of sending a link that was being shared to a service like Instapaper or Pocket. *Mathew Ingram*

I’ve favorited more than 60,000 tweets over the years, and in that time I’ve come to appreciate how versatile that little button is. I use it as **a kind of read receipt** to acknowledge replies; I use it whenever a tweet makes me laugh out loud; I use it when someone criticizes me by name in the hopes that seeing it’s one of my “favorite” tweets will confuse and upset them. *Casey Newton*

If Twitter integrated a simple heart gesture into each Tweet, engagement across the entire service would explode. More of us would be getting loving feedback on our posts and that would **directly encourage more posting** and more frequent visits to Twitter. *Chris Sacca*
confused concepts lead to confused users
how Twitter resolved the conceptual flaw
design rules
the specificity rule

- **specificity**
  - purposes:concepts are 1:1
  - example: category vs label in Gmail

- **redundancy**
  - >1 concept per purpose
  - example: page size vs feed in Epson

- **overloading**
  - >1 purpose per concept
Are you a Gmail user? Did you wake up a week or two ago to find that your new messages were now being automatically organized by Gmail into tabs of different, pre-determined categories? And, did you think, like me, that they were really ugly, stupid, and unnecessary? Here’s a quick tip on how to rid yourself of them!

initial reaction to categories
Using labels

Labels help you organize your messages into categories – work, family, to do, read later, jokes, recipes, any category you want. Labels do all the work that folders do, but with an added bonus: you can add more than one to a message.
overloading epson driver

result: can’t create custom size for front loading
also, page size presets in Lightroom hold feed setting
overloading commit concept

- feature setup
- backup just in case
- feature completion
the familiarity rule

familiarity
steal, don’t invent

needless specialization
custom concept, standard purpose

example
CollectionSet vs Folder in Lightroom
familiarity Lightroom’s collection (set) concept

✗ Lightroom: only collection sets can contain collections

✓ Zotero: collections can contain collections
familiarity Powerpoint's section concept

in Keynote

in Powerpoint

Powerpoint commands
familiarity Lightroom's export preset concept

- Ok, highlighting selects the preset.
- Huh, what are the checkboxes? And why the warning message?
the integrity rule

**integrity**

concepts safe when composed

- P1
- C1
- P2
- C2

**interference**

one concept breaks another

- P1
- C1
- P2
- C2

**example**

Label broken by Conversation in Gmail
integrity Gmail conversation breaks label concept

No messages matched your search. Try using search options such as sender, date, size and more.
Google Drive Sucks

Google Drive storage loses Google Docs data

I lost years of work and personal memories that I saved as Google Docs files because of a poor user interface.

What happened

I was organizing my files on my local computer. I moved them around and out of my Google Drive folder which syncs files. I didn't think anything of it. In the process I got an email from Google saying I'm running out of storage. So I go to the Google Drive site and empty the trash. I didn't think anything of it. I finish organizing my files.

The next morning, I go to open a .gdoc file and get this error:

![Google Drive error message](image)

Sorry, the file you have requested does not exist.

Make sure that you have the correct URL and that the owner of the file hasn't deleted it.

Get stuff done with Google Drive

Apps in Google Drive make it easy to create, store and share online documents, spreadsheets, presentations and more.

Learn more at [drive.google.com/start/apps](http://drive.google.com/start/apps).

My heart sank. What happened to the work from yesterday? I opened another file. Then another. All of them the same message. I was starting to freak out.
integrity cloudapp breaks sync concept

Google drive in cloud

- slides.gdoc
- slides.pdf

Google drive on client machine

- slides.gdoc
- slides.pdf

Another directory on client machine

- url
- slides.gdoc
- slides.pdf

move

sync
integrity proFont breaks toggleFormat concept

Hello there → Hello **there** → Hello there
conclusions
Design by concept is a new approach to creating software. A software product—whether an app, a service or a system—is viewed as a collection of interacting concepts, each with its own purpose, structure and behavior. Concepts can be invented afresh, but they can also be reused, exploiting the knowledge embodied in previous successful designs.

This book explains what concepts are and why they are central to software design; shows examples of concepts (from the most effective and ingenious to the most flawed and frustrating) taken from well-known applications; and presents design principles that can identify and eliminate flaws in existing and new designs.

Daniel Jackson is Professor of Computer Science, a MacVicar fellow, and Associate Director of the Computer Science and Artificial Intelligence Lab at MIT. His past research focused on software modeling and analysis; he is the creator of the Alloy language, and author of Software Abstractions: Logic, Language, and Analysis (MIT Press; second ed. two.o/zero.o/one.o/two.o). His current interests include software design for improved usability, security and safety, and new programming paradigms. He was a recipient of the two.o/zero.o/one.o/six.o ACM SIGSOFT Impact Award, the two.o/zero.o/one.o/seven.o ACM SIGSOFT Outstanding Research Award, and is an ACM Fellow.

https://deja-vu-platform.com

https://gitless.com
some research challenges

- formalizing design criteria
  - genericity, uniformity, decoupling
- smooth transition to code
  - new architectures, like microservices
- design language
  - an extension of Alloy? a logic for OPs
stay in touch!

register here for updates about the book etc:

https://tinyurl.com/conceptdesignlist
extra slides
apps = {concepts}
software app = \{ concepts \}

Finder (1984)  
folder, trash

Word (1983)  
paragraph, format, style

Photoshop (1988)  
pixelarray, layer, mask

Facebook (2004)  
update, friend, like

Drive (2012)  
synchronization, sharing

Google Doc (2009)  
edit (OT), cloud file
Software app class = \{concepts\}

text editor (e.g., Emacs)
  line, buffer

word processor (e.g., Word)
  paragraph, format, style

desktop publisher (e.g., Quark)
  page, textflow
concept choices within an app class

sharing content
post/comment/repost

controlling access
friend/follow/group/channel

how you react
upvote/rating/reaction

personal organizing
favorite/bookmark

shared organizing
hashtag/mention/label

concepts for social media apps
comparing apps via concepts

Lightroom
Photoshop
Capture One
Silver Efex

concepts for editing images?

action
tool
preset

adjustment
tool

layer/mask
tool

filter/preset/style
adjustment
control point
software that “just works”
Facebook has Zoom envy. A zillion companies are trying to eat Netflix’s lunch. Amazon isn’t the best place to shop, but it’s the king.

People — and I’m including myself — tend to overthink why some companies and products last and others wither. Being the first or even the best at something may not matter.

Simplicity is the overlooked secret to success. “It just works” are magic words.
“just works” is not so easy

- frictionless
- unobtrusive
- natural
- learnable

- predictable
- robust
- safe & secure
- error-tolerant

- powerful
- capable
- flexible
- efficient
what it’s not about

cool technology: cloud, machine learning, blockchain

removing or preventing bugs in code
structure
elements, relationships, composition

criteria
objective measures of goodness

patterns
capturing design experience
### Typography

**Structure**
- page, text block, margin
- glyph, ligature, alternate
- ascender, bowl, serif
- justification, spacing, alignment

**Criteria**
- readability: x-height, line length
- consistent color: italics not bold
- avoiding widows & orphans

**Patterns**
- classic text block ratios
- standard leading
- serif/sans pairings

### Bread Baking

**Structure**
- crust, interior, air pockets
- fermenting & raising agents
- flour varieties

**Criteria**
- shaping & elasticity
- density & crumb
- caramelization of crust

**Patterns**
- Lahey no-knead sourdough
- Irish soda bread
- pan cooked flat bread

### Software Engineering

**Structure**
- function, module, package
- closure, functional, callback
- loop, iterator, stream

**Criteria**
- encapsulation of rep
- simple interfaces
- avoiding dependences

**Patterns**
- layered architecture
- immutable datatype
- model-view-controller
- map/reduce/filter
concept structure & semantics
<table>
<thead>
<tr>
<th>concept</th>
<th>Style</th>
<th>name: essential for knowledge capture</th>
</tr>
</thead>
<tbody>
<tr>
<td>purpose</td>
<td>consistent formatting</td>
<td>purpose: why the concept exists</td>
</tr>
<tr>
<td>structure</td>
<td></td>
<td>structure: localized data model</td>
</tr>
<tr>
<td>defined</td>
<td>Style -&gt; one Format</td>
<td></td>
</tr>
<tr>
<td>style</td>
<td>Element -&gt; one Style</td>
<td></td>
</tr>
<tr>
<td>format</td>
<td>Element -&gt; one Format</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Format = style.defined</td>
<td></td>
</tr>
<tr>
<td>actions</td>
<td>define (s: Style, f: Format)</td>
<td>define(s,f); assign(e1,s);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>assign(e2,s); define(s,f');</td>
</tr>
<tr>
<td></td>
<td>assign (e: Element, s: Style)</td>
<td>assign(e1,s);</td>
</tr>
<tr>
<td></td>
<td>e.style := s</td>
<td>observe e1.format = e2.format = f'</td>
</tr>
</tbody>
</table>

There is no problem in computer science that cannot be solved by introducing another level of indirection. *David Wheeler*

**principle**

after define(s,f); assign(e1,s);
assign(e2,s); define(s,f');

observe e1.format = e2.format = f'
**concept** Style

**purpose** consistent formatting

**structure**
defined: Style -> **one** Format
style: Element -> **one** Style

maximal polymorphism: \( \text{one Format} = \text{style.defined} \)

**actions**
define \((s: \text{Style}, f: \text{Format})\)
\[ s.\text{defined} := f \]
assign \((e: \text{Element}, s: \text{Style})\)
\[ e.\text{style} := s \]

**principle**
\[ \text{after } \text{define}(s, f); \text{assign}(e_1, s); \text{assign}(e_2, s); \text{define}(s, f') \]
\[ \text{observe } e_1.\text{format} = e_2.\text{format} = f' \]

OP is an archetypal scenario
a theorem about behaviors
shows how purpose fulfilled
justifies packaging as concept
generalizes concept variants

Michael Polanyi
operational principle
**concept** AuthUser

**purpose** identify users

**structure**
- name, password: User -> **one** String
- sessions: Client -> **set** User

**actions**
- register(n: Name, p: String, **out** u: User)
- login (n: Name, p: String, c: Client)
- logout (c: Client)
- auth (c: Client, **out** u: User)

**principle**
- register(n,p,u); login(n,p,c); auth(c,u'); => u' = u

meaning is set of **traces**:

```
{
<>,
<register(n0,p0,u0)>,
<register(n0,p0,u0), login(n0,p0,c0)>,
<register(n0,p0,u0), register(n1,p1,u1)>,
...,
<register(n0,p0,u0), login(n0,p0,c0), auth(c0,u0)>,
...,
}
```

actually, transition **histories**:

trace `<register(n0,p0,u0)>` is projection of history `<
({name={}, password={}, sessions={}},
register(n0,p0,u0),
{name={u0->n0}, password={u0->p0, sessions={}}})
>`
**concept** Upvote

**purpose** track relative popularity

**structure**

votes: Item -> User

**actions**

upvote (i: Item, u: User)

votes += i->u

count (i: Item, out k: int)

k = #i.votes

**principle**

no upvote(i,u) then ...

count(i, k); upvote(i,u); count(i, k') => k'>k

**traces:**

```
{
<>, ...
<count(i0, 0)>, ...
<count(i0, u0)>, ...
<count(i0, u0), count(i0, 1)>, ...
<count(i0, 0), upvote(i0, u0), count(i0, 1)>, ...
}
```

**histories:**

```
{
<>,
<({votes={}}, upvote(i0,u0), {votes={i0->u0}})>
...
}
```
formalizing transitions, histories & traces

transitions
a transition is a triple (pre-state, action-with-args, post-state)
let pre(x), action(x), post(x) be the pre-state, action and post-state of x
let inits(c) and trans(c) be the initial states and set of transitions of concept c

histories
a history is a sequence of transitions
history h is consistent if for all f, g != <>, h = f ∧ g implies post(last(f)) = pre(first(g))

concept histories
histories(c), the histories of a concept c include:
(1) the empty history <>
(2) any <x> where x in trans(c) and pre(x) in inits(c)
(3) any consistent history f ∧ <x> where f in histories(c) and x in trans(c)

concept traces
if h in histories(c), map(h, action) in traces(c)

theorems
prefix closure: if f ∧ g in histories(c) then f in histories(c) [and same for traces]
complete state: if h and f ∧ g in histories(c), h ∧ g in histories(c) if it’s consistent
semantics of composition
We did a similar thing with a Scala -> Rust rewrite for the [http://prisma.io](http://prisma.io) query engine.

By rewriting small components and integrating them into the existing project using Java's native interface, our small team of 5 developers were able to pull off this massive rewrite in just under a year. The resulting codebase is rearchitected in a few very important ways, but mostly follows the same structure.

And because we kept and evolved our old Scala-based test suite, we have a very high confidence in the rewrite.

When Async/await finally landed, we could switch over very quickly, and it has been a joy to focus on benchmarks and performance over the last month. Spoiler: Rust is faster than Scala :-D

I promise that this is asked genuinely and isn't some sort of veiled "gotcha" (it's tough to tell on the internet sometimes); what was the reason for a change from Scala to Rust?

I ask because Scala already has a good type system and the JVM typically has good performance nowadays, particularly with something like GraalVM, so I am actually really curious to why you felt a Rust rewrite was a good idea.
making an app by composing concepts

```plaintext
concept Post
actions
new (a: Author, s: String, out p: Post)
edit (p: Post, s: String)
ge(t: Author, out ps: set Post)

concept Comment
actions
new (a: Author, s: String, t: Target, out c: Comment)
ge(t: Target, out cs: set Comment)

concept Upvote
actions
upvote (i: Item, u: User)
count (i: Item, out r: Int)
owns (o: Owner, i: Item)

concept Owner
actions
register (o: Owner, i: Item)
owns (o: Owner, i: Item)

class AuthUser
actions
register (n: Name, p: String, out u: User)
login (n: Name, p: String, c: Client)
logout (c: Client)
auth (c: Client, out u: User)

app HackerNews
includes Post, Comment, Upvote, AuthUser, Owner
synchronizes
newPost
AuthUser.auth (c, u)
Post.new(u, s, p)
Owner.register(u, p)
editPost
AuthUser.auth (c, u)
Owner.owns(u, p)
Post.edit(p, s)
newComment
AuthUser.auth (c, u)
Comment.new(u, s, p, x)
upvotePost
AuthUser.auth (c, u)
Upvote.upvote (p, u)
...
```
projecting transition

each transition in composite system
is interpreted as a transition in one of the concepts

Tony Hoare
CSP (1978)
check that projected transitions meet concept specifications

**Concept: AuthUser**
- AuthUser.register (n1, p1, u1)
- AuthUser.login (n1, p1, c1)
- AuthUser.auth (c1, u1)
- AuthUser.auth (c1, u1)

**Concept: Post**
- Post.new(u1, s1, p1)

**Concept: Owner**
- Owner.register(u1, p1)

**Concept: Upvote**
- Upvote.upvote (p1, u1)
formalizing composites histories & synchronizations

recall: transitions
trans(c) is the set of transitions of concept c [and trans(C) for concept set C]

composite histories
h is a composite history of an app made of concepts c in C if
   every transition in h is in trans(C) and the subhistory h@c is in histories(c)

composite transitions and synchronizations
a composite transition X for concepts C is a non-empty sequence of trans(C)
a synchronization S is a set of composite transitions
an execution of S is a concatenation of some members of S

app histories
the histories of an app composed of concepts C with sync S are
   the composite histories of C that are executions of S

not prefix-closed
note that the histories of an app are not generally prefix-closed
transitions of a composite transition must occur all-or-none
axes of synchronization
concept Post
actions
new (a: Author, s: String, out p: Post)
edit (a: Author, p: Post, s: String)
get (a: Author, out ps: set Post)

class AuthUser
actions
register (n: Name, p: String, out u: User)
login (n: Name, p: String, c: Client)
logout (c: Client)
auth (c: Client, out u: User)

sync post (c: Client, s: String, out u: User, out p: Post)
AuthUser.auth (c, u)
Post.new (u, s, p)

sync edit (c: Client, p: Post, s: String, out u: User)
AuthUser.auth (c, u)
Post.edit (u, p, s)
sync on actions & pre-state

**concept** Trash

**state**
all, trashed: set Object

**actions**
create (out o: Object)
delete (o: Object)
restore (o: Object)
emptyTrash ()

**concept** Folder

**state**
contents: Folder -> (File + Folder)

**static** root, trash: disjoint Folder

**initially** contents = root -> trash

**actions**
newFolder (parent: Folder, out f: Folder)
newFile (parent: Folder, f: File)
moved (o: File + Folder, to: Folder)
delete (f: File + Folder)

```plaintext
sync moveToTrash (o: File + Folder)
Folder.move (o, Folder.trash)
for x: o.*(Folder.contents) | Trash.delete (x)

sync empty ()
Trash.empty()
for x: Trash.trashed | Folder.delete(x)

sync restore (o: File + Folder, to: Folder)
{no (to + o.(Folder.parent)) & Trash.trashed}
Folder.move(o, to)
for x: o.*(Folder.contents) | Trash.restore (x)
```
sync on actions & post-state

**concept** Channel

**state**
rc, gc, bc: Image \(\rightarrow\) Channel
pixel: (Image + Channel) \(\rightarrow\) Coord \(\rightarrow\) Pixel

**static** red, green, blue: Pixel \(\rightarrow\) Pixel // color to greyscale
inv
all i: Image, c: Coord \(\mid\) i.pixel[c].red = i.rc.pixel[c] ...

**actions**
edit (x: Channel + Image, e: Coord \(\rightarrow\) Pixel)

**concept** Adjustment

**state**
npixel: Image \(\rightarrow\) Coord \(\rightarrow\) Pixel
adjFuns: Adjustment \(\rightarrow\) Param \(\rightarrow\) Pixel \(\rightarrow\) Pixel

**actions**
adjust (i: Image, a: Adjustment, p: Param)
concept polymorphism
<table>
<thead>
<tr>
<th>concept</th>
<th>Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>purpose</td>
<td>consistent formatting</td>
</tr>
<tr>
<td>structure</td>
<td>defined: Style -&gt; <strong>one</strong> Format</td>
</tr>
<tr>
<td></td>
<td>style: Element -&gt; <strong>one</strong> Style</td>
</tr>
<tr>
<td></td>
<td>format: Element -&gt; <strong>one</strong> Format = style.defined</td>
</tr>
<tr>
<td>actions</td>
<td>define (s: Style, f: Format)</td>
</tr>
<tr>
<td></td>
<td>s.defined := f</td>
</tr>
<tr>
<td></td>
<td>assign (e: Element, s: Style)</td>
</tr>
<tr>
<td></td>
<td>e.style := s</td>
</tr>
</tbody>
</table>

This concept is polymorphic in the types Style and Format: they are essentially type variables.
permuting transitions

typed transitions
the elements of each transition can be typed based on the decls

example
\{\text{defined}={} , \text{style}={} , \text{format}={} \} \\
\text{define}(s_0: \text{Style} , f_0: \text{Format}) \\
\{\text{defined}={} , s_0: \text{Style}\rightarrow f_0: \text{Format} , \text{style}={} , \text{format}={} \}

permuting a transition
given a permutation $\pi$ on type $T$, $\pi: T \rightarrow T$
permutation $\pi(t)$ of transition $t$ just lifts $\pi$ over $t$

example
\[ \pi: \text{Style} \rightarrow \text{Style} = \{s_0\rightarrow s_1, s_1\rightarrow s_0\} \] \\
$\pi(t) =$  \\
\{\text{defined}={} , \text{style}={} , \text{format}={} \} \\
\text{define}(s_1: \text{Style} , f_0: \text{Format}) \\
\{\text{defined}={} , s_1: \text{Style}\rightarrow f_0: \text{Format} , \text{style}={} , \text{format}={} \}
permutation invariance & polymorphism

**concept** Style

**purpose** consistent formatting

**structure**
- defined: Style -> **one** Format
- style: Element -> **one** Style
- format: Element -> **one** Format = style.defined

**actions**
- define (s: Style, f: Format)
  - s.defined := f
- assign (e: Element, s: Style)
  - e.style := s

**invariance & polymorphism**
a concept C is invariant (or polymorphic) in type T iff for any permutation \( \pi \) on type T, \( \pi: T \rightarrow T \) whenever t is a transition of C, \( \pi(t) \) is also

**what this means**
the concept just does database-like operations similar to Tarski’s notion of “logical operations”

**example**
Style concept is polymorphic in Style and Format
an example of a non-polymorphic type

Upvote is not polymorphic in the type int

example of non-invariant transition

\[ \pi: \text{int} \rightarrow \text{int} = \{0\rightarrow1, 1\rightarrow0\} \]

\{votes=\} count (i0:Item, 0:int) \{votes=\} is a transition
\{votes=\} count (i0:Item, 1:int) \{votes=\} is not a transition

note

A concept may be polymorphic in a primitive type but that indicates a specification error
an example of special values
this (very simplified) Format concept defines special values represented as variables of the state, set initially

an initialization subtlety
initial values aren’t given in the spec but they must be chosen in any implementation so Format concept is not polymorphic in the type Format

incomplete specification
this spec does not say what print does but implied that it italicizes text formatted as italic, etc

opaque types
call these non-polymorphic, non-primitive types “opaque” polymorphic type ~ type variable opaque type ~ abstract data type

---

concept Format

purpose stylize text

structure static Bold, Underline, Italic: disjoint Format format: Text -> set Format

actions apply (t: Text, f: Format)
  f in Bold + Underline + Italic
t.format :=
  f in t.format => t.format - f, t.format + f
print (t: Text) ...

implications of polymorphism

joining polymorphic types
polymorphic types can be joined in concept compositions
so AuthUser.User can be joined to Post.Author
this is how Deja Vu works

exposing implementation detail
AuthUser is polymorphic in String, so should be Password, say
(but if validated password, would no longer be polymorphic)

concept Post
actions
new (a: Author, s: String, out p: Post)
edit (p: Post, s: String)
get (a: Author, out ps: set Post)

concept AuthUser
actions
register (n: Name, p: String, out u: User)
login (n: Name, p: String, c: Client)
logout (c: Client)
auth (c: Client, out u: User)

sync
AuthUser.auth (c, u)
Post.new (u, s, p)
implications of opacity

joining opaque types
if opaque types are joined, concepts must share interpretation
not truly independent of each other

example
Channel and Adjustment both have Pixel as opaque
must have common interpretation of pixel values

concept Channel
state
rc, gc, bc: Image -> Channel
pixel: (Image + Channel) -> Coord -> Pixel

static red, green, blue: Pixel -> Pixel

actions
edit (x: Channel + Image, e: Coord -> Pixel)

concept Adjustment
state
pixel: Image -> Coord -> Pixel
adjFuns: Adjustment -> Param -> Pixel -> Pixel

actions
adjust (i: Image, a: Adjustment, p: Param)

sync
Adjustment.adjust (i, a, p)
Channel.edit (i, e)
{e = Channel.pixel[i]}
example: waze
**concept** CrowdsourcedConditionTracking

**purpose** track condition of a public resource

**structure**
reports: User -> Resource -> Condition -> Time
inferred: Resource -> Condition

**actions**
report (u: User, r: Resource, c: Condition, t: Time)
update () // compute inferred from reports

**principle**
with accurate reports and frequent updating, inferred condition reflects reality
**concept** ConditionPrediction

**purpose** predict future from past conditions

**structure**
- history: Resource -> Time -> **one** Condition
- predicted: Resource -> TimeSlot -> **one** Condition
- slot: Time -> **one** TimeSlot

**actions**
- report (r: Resource, t: Time, c: Condition)
- update () // compute inferred from reports

**principle**
with accurate reports and frequent updating, inferred condition reflects reality

which types are opaque in this concept?
example: group
## group concept

**concept** Group

**purpose** control access to shared assets

**structure**
- members: Group -> User
- assets: Group -> Asset

**actions**
- join (u: User, g: Group)
  - g.members += u
- contribute (u: User, g: Group, a: Asset)
  - u in g.members
  - g.assets += a
- access (u: User, a: Asset)
  - a in (members.u).assets

**principle**
- if you join a group and some contributes an asset, you can access it
concept Invitation

purpose grant optional access to resource

structure
pending, accepted: set Invitation
from, to: Invitation -> one User
for: Invitation -> Resource

actions
invite (inviter, invitee: User, r: Resource, out i: Invitation)
  i not in pending + accepted
  pending += i
  i.from := inviter; i.to := invitee; i.resource := r
accept (invitee: User, i: Invitation)
  i in pending and i.from = invitee
  accepted += i; pending -= i
access (u: User, r: Resource)
  some i: accepted | i.to = user and i.for = r
synchronizing group and invitation

**Group**
- join (u: User, g: Group)
- contribute (u: User, g: Group, a: Asset)
- access (u: User, a: Asset)

**Invitation**
- invite (inviter, invitee: User, r: Resource, out i: Invitation)
- accept (invitee: User, i: Invitation)
- access (u: User, r: Resource)

**sync**
- join (u, g) || accept (u, i) where Invitation.for[i] = g
purpose as design criterion
OP as a criterion for being a concept

if you can formulate a compelling OP, you have a concept

what’s compelling? intricate protocol non-trivial outcome

what’s not? entity with CRUD can’t stand alone

why does this matter? guides granularity, structure of design

social media
upvote: when you upvote, post ranked higher
friend: when you become friend, can access updates
post: after submitting post, people can read it
user account: when login, authenticated as particular user
user profile: : just a data structure without an OP
edit post: : just an action
timeline: an action? (show posts chronologically by author?)

image editing
image-local: when you edit pixels with local adjustment, get new image
image-global: when you apply global adjustment, image changes
image-channel: when you edit channel, whole image changes
channel, pixel, etc (alone): just data structures without an OP
brush, gradient, etc: just an action
some design criteria for reusability & simplicity

make concepts as polymorphic as possible
example: Group should not include user profiles (opaque)

break into smallest concepts you can
example: separate Invitation from Group

but not so small that OP is lost
example (good): Group
example (bad): Pixel
example (on the edge): UserProfile
gmail design issues
using labels to organize messages

- show messages with label hacking
- also implemented as a label

Gmail interface with labels highlighted.
<table>
<thead>
<tr>
<th>label:hacking</th>
<th>* me, Alyssa (12)</th>
<th>9:43 am</th>
</tr>
</thead>
</table>

| label:meetups | * me, Alyssa (12) | 9:58 am |

| label:hacking label:meetups | | No messages matched your search. Try using search options such as sender, date, size and more. |
What’s going on?

Labels are attached to messages: hacking, meetups.

1. Filter is applied to set of messages: some match.
2. Conversation appears if it includes a matched message.
so this is not a surprise
and this makes sense too (but order is special)
and this almost makes sense
**concept** Label

**purpose** organize items for easy retrieval

**structure**
label: Item -> one String

**actions**
mark (i: Item, p: Label)
i.label += p
unmark (i: Item, p: Label)
i.label -= p
find (ps: set Label): set Item
result = {i | ps in i.labels}

**story**
if mark(i,p); find(p):is then i in is
if no mark(i,p); find(p):is then i !in is

---

**composite system**

Label

- mark(m, sent)
- sendMsg(m)

Message

- send(m)

Database
when message m is sent
Label.mark(m, 'sent')
occurs implicitly

when Sent link is clicked
Label.find('sent'):ms
occurs

but ms includes messages never marked
why pick on gmail?

not a strawman!
about 1.5B users
20% of global market
27% of all email opens

do these nitpicks matter?

“The details are not the details; they make the product”—Charles and Ray Eames
trepansing: small symptoms of major surgery

Bronze Age skull with evidence of trepanning

The Extraction of the Stone of Madness, Hieronymus Bosch
font integrity example
pro fonts break integrity of format concept
synergy examples
what is design?

reusing concepts
using Style for color swatches

refining concepts
click to select Group elements

inventing concepts
Event Type in Calendly

daniel jackson
welcome to my calendly scheduling page. scheduling time with me is as easy as 1-2-3.

- phone call
- 60 minute meeting
- 30 minute meeting
- 15 minute meeting

synergy: merging concepts
channels in photoshop
the trash concept & its history

- **concept**: Trash
- **purpose**: undo deletion
- **structure**
  - all, inTrash: **set** Object
- **actions**
  - delete (o: Object)
  - empty ()
  - restore (o: Object)
  - new (o: Object)
  - exists (o: Object, **out** b: bool)
- **story**
  - delete(o); restore(o); exists(o, true)
  - delete(o); empty(); exists(o, false)

Microsoft MS-DOS 6 (1993): “DeleteSentry”
Apple vs. Microsoft (1994): Apple lost, but ©Trash

*holds files not folders, so can’t recover structure*
merging two concepts

**concept** Trash

**purpose** undo deletion

**structure**
- all, inTrash: set Object

**actions**
- delete (o: Object)
- empty ()
- restore (o: Object)
- new (o: Object)
- exists (o: Object, out b: bool)

**story**
- delete(o); restore(o); exists(o, true)
- delete(o); empty(); exists(o, false)

**concept** Folder

**purpose** local organization

**structure**
- root: Folder
- contents: Folder -> set (Folder + Object)

**actions**
- move (o: Object + Folder, to: Folder)
- new (p: Folder, out f: Folder)
- list (f: Folder, out os: set Object)
- delete (f: Folder)
- root (out f: Folder)

**story**
- list(f, os); move(o, to); list(f, os’)
  => if o not in os and to != f then os = os’
trash x folder

can contain folder

generalizes date deleted

handle volumes
trash x folder

synergies
trash is not a special thing  
all folder tools apply  
can put folder in trash  
move to trash = delete  
move from trash = restore  
date added = date deleted

anomalies
trash contains objects from >1 volume  
in trash folder, can group by volume  
delete immediately allows partial emptying  
trash folder has no path (path concept)  
can’t move trash folder or delete it

purpose: undo deletion  
purpose: local organization
email x server account
style/toc synergy
photoshop synergies

selection = mask = channel = image
how to sharpen an image using an edge mask
select channel with greatest contrast
duplicate selected channel
apply Filter > Stylize > Find Edges
apply Image > Adjustments > Invert
apply Filter > Other > Maximum
apply Filter > Noise > Median
apply Image > Adjustment > Levels
apply Filter > Blur > Gaussian Blur
right-click to make channel a selection
select image layer
apply Select > Inverse
apply Filter > Sharpen > Unsharp Mask

- treat channel as image
- make selection from channel
- apply filter using selection as mask
dropbox filename example
survey of dropbox users (MIT CS undergrads)

correctly predicting behavior

- delete shared folder results in leaving
- delete shared subfolder removes it

Kelly Zhang
a conceptual model of file names and deletion

- **rename**
  - slide.pdf → slides.pdf

- **delete**
  - slides.pdf
the actual model, courtesy of multics (1963-69!)
Principle: Choose metaphors that will enable users to instantly grasp the finest details of the conceptual model.
[T]o see what rate of progress one can expect in software technology, let us examine the difficulties of that technology. Following Aristotle, I divide them into essence, the difficulties inherent in the nature of software, and accidents, those difficulties that today attend its production but are not inherent.

The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract in that such a conceptual construct is
To design something really well, you have to get it. You have to really grok what it’s all about. It takes a passionate commitment to really thoroughly understand something, chew it up, not just quickly swallow it. Most people don’t take the time to do that.
Almost anything in software can be implemented, sold, and even used given enough determination... But there is one quality that cannot be purchased in this way—and that is reliability.

The price of reliability is the pursuit of the utmost simplicity. It is a price which the very rich find most hard to pay.
levels of UX design (export diagram)

- Physical: color, size, layout, type, touch, sound
- Linguistic: icons, labels, tooltips, site structure
- Conceptual: semantics, actions, data model, purpose

Concrete | Abstract
quality beyond correctness

“it’s not a bug, it’s a feature”

Storage Almost Full
You can manage your storage in Settings.

Done

Settings

iPhone: storage catch-22

crashplan: this is success?

Dropbox: deleting shared files
concept trash

purpose undo deletion

structure
   objects, trashed: set Object

actions
   delete (o: Object)
      o in objects - trashed => trashed += o
   empty ()
      objects -= trashed; trashed := none
   restore (o: Object)
      o in trashed => trashed -= o
   new (o: Object)
      o !in objects => objects += o

principle
   ... delete(o); restore(o) {o in objects - trashed}
   ... delete(o); empty() {o !in objects}
**concept** reservation

**purpose** consistent formatting

**structure**
- slots: Owner -> Slot
- holds: User -> Slot

**actions**
- create \((o: \text{Owner}, s: \text{Slot})\)
  
  \[\text{no slots}.s \Rightarrow \text{slots} + o \rightarrow s\]

- reserve \((u: \text{User}, o: \text{Owner}, s: \text{Slot})\)
  
  \[\text{no holds}.s \text{ and } o \rightarrow s \text{ in slots} \Rightarrow \text{holds} + u \rightarrow s\]

- cancel \((u: \text{User}, s: \text{Slot})\)
  
  \[u \rightarrow s \text{ in holds} \Rightarrow \text{holds} - u \rightarrow s\]

- use \((u: \text{User}, o: \text{Owner}, s: \text{Slot})\)
  
  \[u \rightarrow s \text{ in holds and } o \rightarrow s \text{ in slots} \Rightarrow\]

**principle**

if create and reserve and not cancel then can use

**name:** essential for knowledge capture

**purpose:** why the concept exists

**structure:** localized data model

**actions:** observable & atomic

**OP justifies design and explains it**

**shows how behavior fulfills purpose**
elements of a concept design method

**structure**: how to express & combine concepts

**principles**: applicable distillation of experience

**patterns**: handbook of known concepts & issues

**tools**: exploit computing for analysis & synthesis

**separation of concerns**: easier to focus, divide labor

**avoiding predictable pitfalls**, speeding up design

**capturing expertise and experience for better design**

**catching subtle flaws**, reducing manual effort
Design by concept is a new approach to creating software. A software product—whether an app, a service or a system—is viewed as a collection of interacting concepts, each with its own purpose, structure and behavior. Concepts can be invented afresh, but they can also be reused, exploiting the knowledge embodied in previous successful designs.

This book explains what concepts are and why they are central to software design; shows examples of concepts (from the most effective and ingenious to the most flawed and frustrating) taken from well-known applications; and presents design principles that can identify and eliminate flaws in existing and new designs.

Daniel Jackson is Professor of Computer Science, a MacVicar fellow, and Associate Director of the Computer Science and Artificial Intelligence Lab at MIT. His past research focused on software modeling and analysis; he is the creator of the Alloy language, and author of Software Abstractions: Logic, Language, and Analysis (MIT Press; second ed.). His current interests include software design for improved usability, security and safety, and new programming paradigms. He was a recipient of the ACM SIGSOFT Impact Award, the ACM SIGSOFT Outstanding Research Award, and is an ACM Fellow.
principle: make concepts modular

- concepts have **no dependences**
  - ✓ trash does not “use” deleted labels

- concepts **encapsulate decisions**
  - ✓ labels independent of folder structure
  - ✗ Facebook tags change access control

- concepts are **polymorphic**
  - ✓ label items not folders
  - ✗ Twitter tweet content determines if reply or not
**simple group functionality**

user can create a new group
other users can request to join
users can contribute posts to the group
and can read other user’s posts
**modularity**

**group, most granular concepts**

**concept** Group  
**state**  
owner, members: Group -> User  
assets: Group -> Asset  
**actions**  
create (owner: User, out g: Group)  
join (u: User, g: Group)  
contribute (u: User, g: Group, a: Asset)  
access (u: User, a: Asset)

**concept** Request  
**state**  
owns, requested, granted, denied: User -> Resource  
**actions**  
register (owner: User, r: Resource)  
request (u: User, r: Resource)  
respond (o, u: User, r: Resource, answer: bool)

**concept** Post  
**state**  
author: Post -> Author  
content: Post -> String  
**actions**  
new (a: Author, s: String, out p: Post)  
edit (p: Post, s: String)  
get (a: Author, out ps: set Post)

**sync** newGroup (o: User, out g: Group)  
Request.register(o, g)  
Group.create(o, g)

**sync** requestJoin (u: User, g: Group)  
Request.request(u, g)

**sync** join (o, u: User, g: Group)  
Request.respond (o, u, g, true)  
Group.join (u, g)

**sync** post (u: User, g: Group, s: String, out p: Post)  
Post.new (u, s, p)  
Group.contribute (u, g, p)
**REUSE**
what: break into concepts that can be used independently
when: new concept is more focused, stands alone, and usable in other contexts

**SEPARATE**
what: factor out disjoint functionalities into separate concepts
when: some subsets of actions and states are decoupled; unclear purpose

**ENCAPSULATE**
what: bring functionality together to localize design decisions
when: invariants and couplings cross concept boundaries, and complicate sync
modularity design moves for group/post/request concepts

- **ENCAP** (authors are group members)
- REUSE
- REUSE
- REUSE, SEPARATE
- ENCAP
- (no reqs from existing members)
- ENCAP
- REUSE
- REUSE
- ENCAP
- ENCAP
- ENCAP
- ENCAP

GPR

GP R

GR P

G PR

GPR
Those of you who read my “other” blog (at WindowsITPro.com) are probably aware of my views on Outlook’s continuing failure to be able to suppress or otherwise deal with the generous number of synchronization logs that the client generates. Last May, I wrote about the fact that it is impossible to use Exchange retention policies to eliminate the pesky logs and that the suggested registry settings prove to be as ineffective.

Now I see that the nice people who work in Microsoft Support have given up the ghost too and issued KB2686541 that explains that you might “notice that messages are being created in the Sync Issues folder” but that “MRM does not process or delete the items” because “the folder is a client-side folder only. In this context, MRM means “Messaging Records Management”, the Exchange subsystem devoted to controlling content in user mailboxes. It really means MFA, the Managed Folder Assistant, because that’s the Exchange 2010 server component that does the processing of retention policies and would very much like to get its hands on Outlook’s synchronization logs, if only they weren’t hidden away in that client-side folder.

synchronization logs are stored as messages in email folders naturally, not sync’d with server but create storage leak and can’t be accessed by admins