Software against humanity?

An Illichian perspective on the industrial era of software

Stephen Kell

S.R.Kell@kent.ac.uk

University of Kent

Introductions

- about me
- about you?

Then

- Illichian ideas outlined
- software as an industrial institution
- the institution not working
- transferring the Illichian

About me

Most of my research is ‘core CS’:

- programming language implementation
- operating systems
- software extensibility, debuggability, liveness…


GALLEY EDITOR
SMALLTALK-80
VSI

How did I get here?

I dabble with history and philosophy of CS, because

- it's interesting!
- (+ an accident)

but also from frustration with core CS. . . :

- a-historical
- 'chasing the stick'
- indifferent to 'big questions'

Apology for an impressionistic talk

Reasons to be sceptical (part 3?)

Examples of research malaise:

- software performance viewpoint unchanged from 1970s
- interoperability problems remain ‘black sheep’
- paradigms increasingly entrenched

Examples of practical malaise:

- increase of ‘hello-world complexity’
- hardware advances soaked up, increasingly invisible
- subversion of 1960s–80s idealism
  - open-source, internet, …

Technology's headline capabilities continue to improve.

But the *distribution* of those abilities

- across scenarios
- across people

. . . seems to be stagnant or worsening.

- not just in equitability of share
- . . . in absolute capability of the median constituent!

We performed a ‘blank string’ search against the Users table. . . . Ultimately we found that our self-imposed response time threshold of 3 seconds was crossed at 3000 users.

Hello World

Let's get started by creating a "Hello World" service that runs on your local machine and communicates with IFTTT.

Sign in or sign up before following this tutorial.

This tutorial and code sample will help get you up and running on the IFTTT Platform quickly and show you how to verify that your service is working correctly using the IFTTT endpoint tests. If you'd prefer, feel free to skim over this section or dive right into the Service API Reference!

Download the Rails app

To get started, copy the following to a file named "hello_world.rb" in your home directory:

(a 123-line Ruby file. . .)

“A falling tide sinks all boats.”

Ivan Illich (1926–2002)

"A few patients
survived longer with transplants
of various organs. On the other
hand, the total social cost exacted by
medicine ceased to be measurable in
conventional terms. Society can have
no quantitative standards by which
to add up the illusion, social control,
prolonged suffering, loneliness,
genetic deterioration, and frustration
produced by medical treatment."

—from 'Tools for Conviviality' (1973)

Criticism of institutions

Illich most famously critiqued three institutions:

- institutionalised education
- modern medicine
- car-based transportation & planning

He observed that each was poor at its stated ends…

- the means and ends had become confused!
- can still be self-sustaining
- can still *claim* advances *by its own criteria*

Design of our institutions is key: technical + political

‘It is not strictly necessary to accept 1913 and 1955 as two watershed years in order to understand that early in the century medical practice emerged into an era of scientific verification of its results. And later medical science itself became an alibi for the obvious damage caused by the medical professional.’

“The invention of the ball-bearing… signaled a true… political choice … between more freedom in equity and more speed. The bearing is an equally fundamental ingredient of two new types of locomotion … symbolized by the bicycle and the car. The bicycle lifted man’s auto-mobility into a new order, beyond which progress is theoretically not possible. In contrast, the accelerating individual capsule enabled societies to engage in a ritual of progressively paralyzing speed.”

—from ‘Energy and Equity’ (1974)

‘Bicycles for the mind’

Maybe we’ve got ‘cars for the mind’ instead?

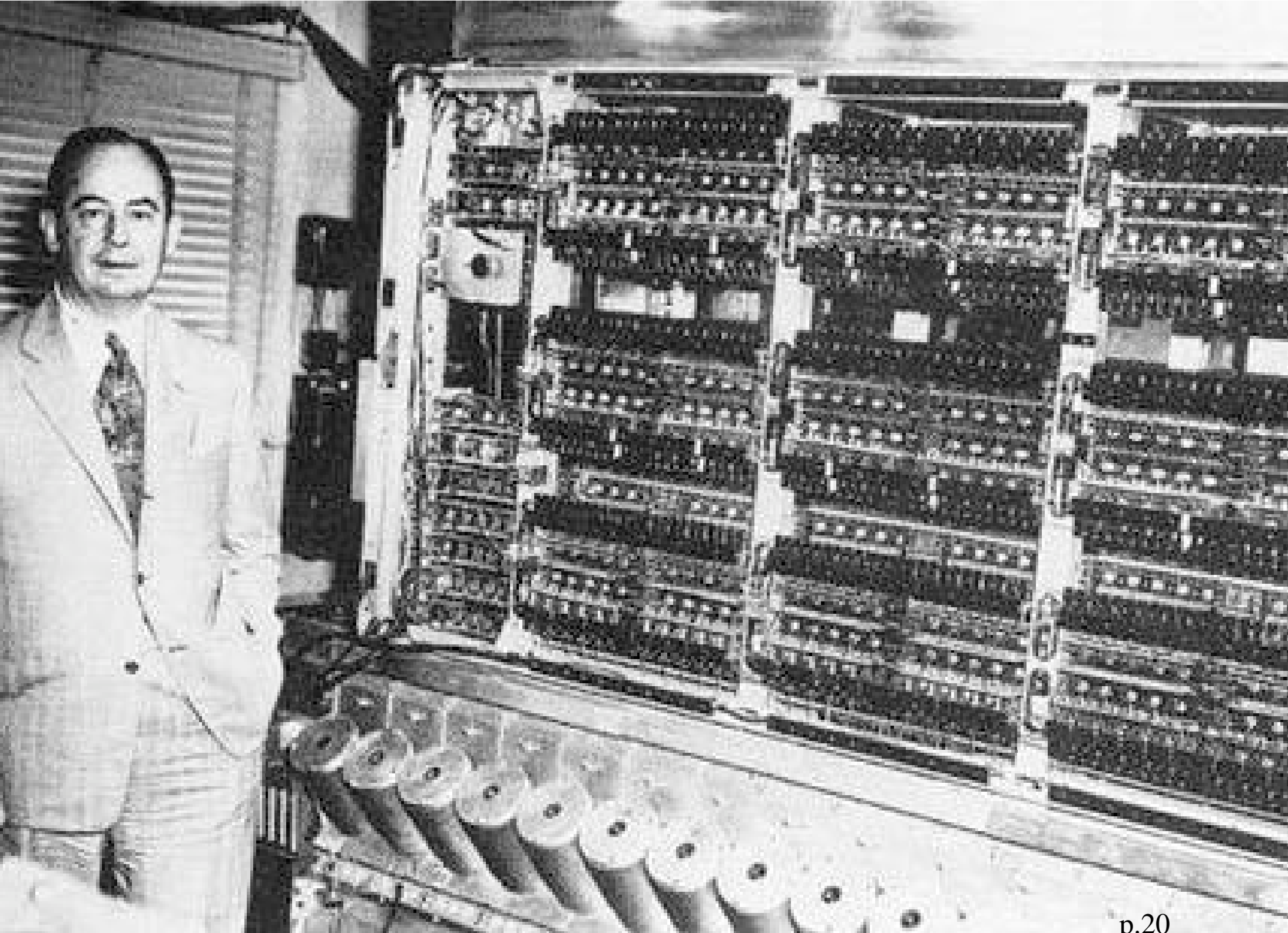
- ‘progressively paralyzing’ computational power
- ‘one class…monopolizes…’
- ‘create distances for all and shrink them for only a few’

Some Illichian phenomena:

- creeping yet ‘watershed’ transitions…
- … from real to counter-productivity…
- … of *institutions*
- societal cost/benefit vs governing elites
- ‘radical monopoly’—the exclusion of alternative means

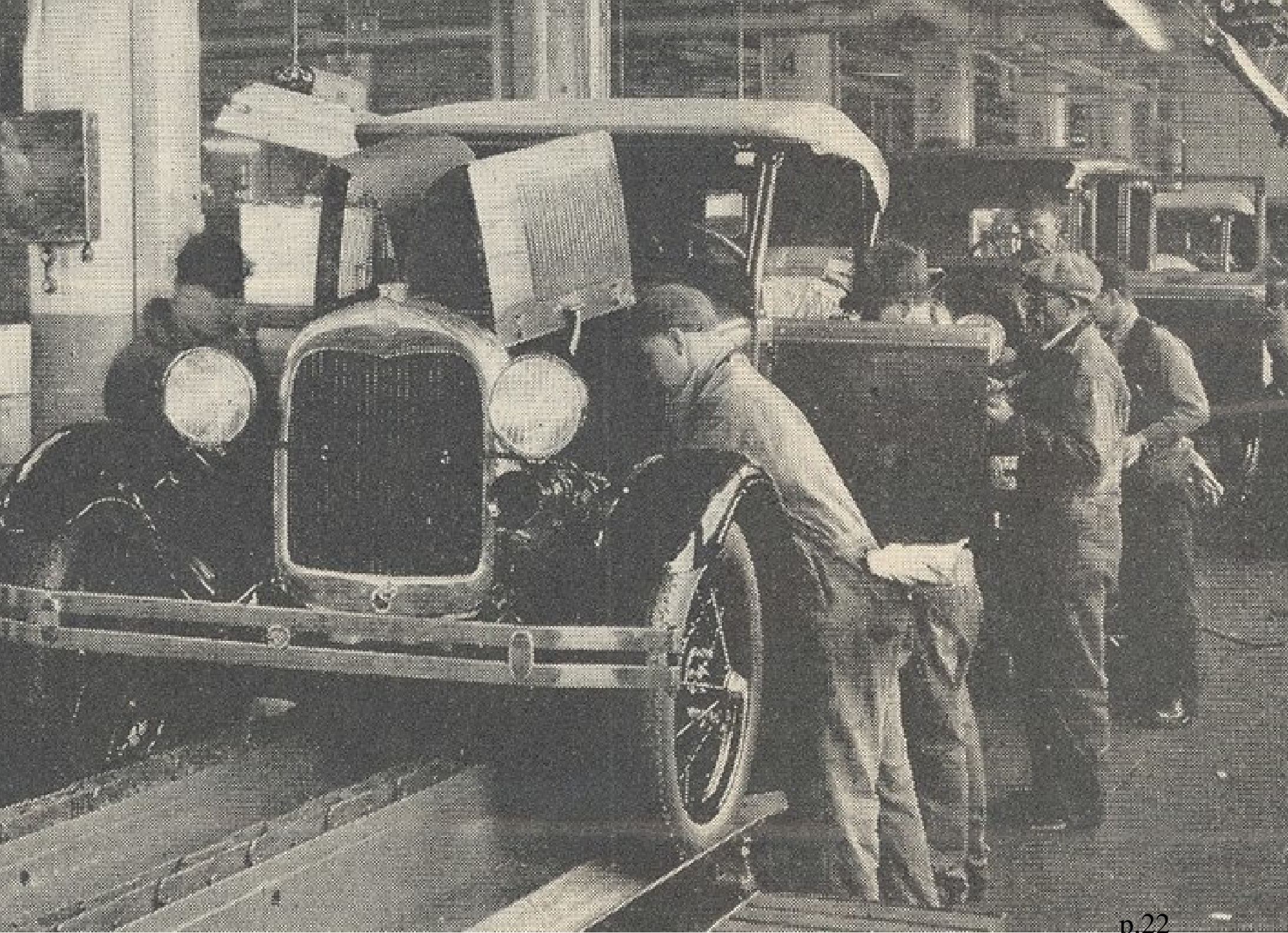
Some software phenomena:

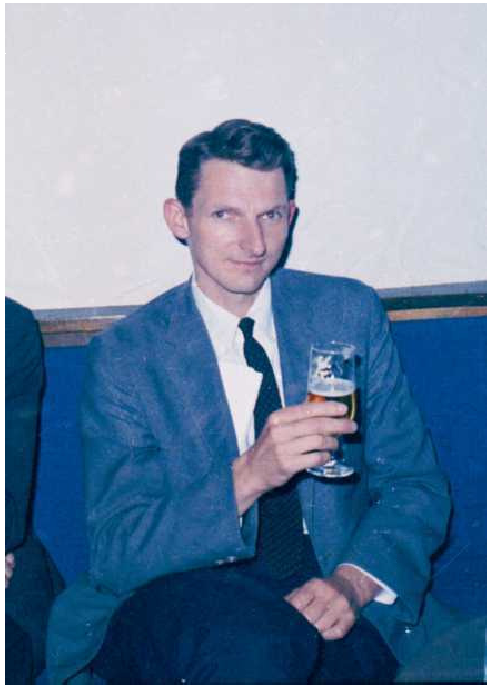
- bootstrapping, recursion… (self-application)
- a tendency to expand over time
- a tendency to induce demand for itself
- a tendency to create exclusive institutions
- consumed by many, controlled by few
- creeping transition…
- … from ‘net enabling’ to ‘net enslaving’?



Some software hypotheses:

- ‘code complexity per unit value’ is increasing
- overriding research culture is one of ‘escalation’
  - applying more software to the problems of software
  - . . . believed will overcome, not worsen, problems
- culture and technology form a feedback loop
  - e.g. additiveness and monotonicity in programming
  - (cf. differencing or reconciliation. . .)
- de-escalating has potential value
  - ‘doing more with less’, cf. more with more





“When we undertake to write a compiler, we begin not by saying ‘What table mechanism shall we use?’ but ‘What table mechanism shall we build?’ … [My vision is that the builder] will be able to say ‘I will use a String Associates A4 symbol table, in size 500x8,’ and therewith consider it done. As a bonus he may later experiment with alternatives to this choice, without incurring extreme costs.”

Some parts of McIlroy's vision *did* come to pass

- extensive software libraries

Some didn't

- *fine-grained* libraries
- 'alternatives… without extreme costs'

Some other things happened:

- industrial 'optimisation mindset'
- means and ends confused

Bret Victor
@worrydream

Following

help i've forgotten what computers are for

HELLO we're symbolics, it's 1982, we just made the greatest personal mind-amplifier that the world will ever know, here's what you can use it for

Symbolics, Inc. designs, manufactures, sells, and supports advanced state-of-the-art, high-performance, single-user computer systems that feature a highly interactive man/machine interface. These systems were designed in response to the growing demand for increasing the productivity of highly skilled professional staff in various high technology disciplines. Present applications include the design of very large scale integrated (VLSI) circuits, symbolic mathematical analysis, genetic engineering, seismic studies for oil and mineral exploration, training simulation, software production, and artificial intelligence research and development. The system design objective, achieved to an extent never before offered commercially, has been to greatly enhance programmer and user productivity.

Symbolics
who designe
Institute of T
and its opera
will enable S
future techn
and software

making more computers

burning the earth to power computers

computers

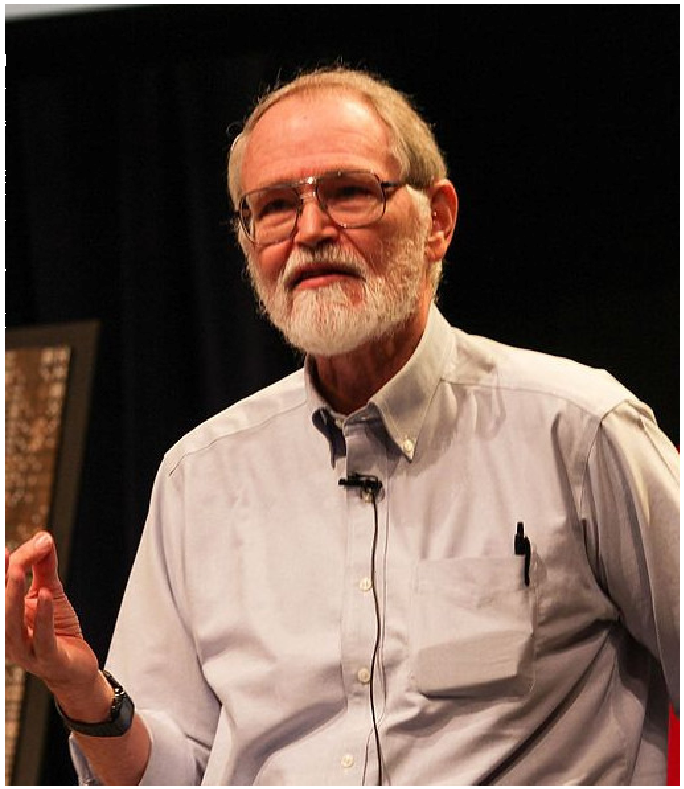
creepy shit

killin' dudes

dehumanization of society

10:52 PM - 11 May 2017

Some well-known programming wisdom:



"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as [possible] when you write it, how will you ever debug it?"

—Brian Kernighan
from *The Elements of Programming Style* (with P.J. Plauger)

Some well-known programming wisdom:

> “Everyone knows that debugging
> is twice as hard as writing a program
> in the first place. So if ~~you’re~~ your
> compiler is as clever as [possible]
> when ~~you write~~ it optimises
> it, how will you ever debug it?”

Compilers are very advanced machines

A tiny example due to Chris Lattner...

```
void contains_null_check (int *P) {
    int dead = *P;
    if (P == 0)
        return;
    *P = 4;
}
```

After optimization, it becomes (effectively)

```
void contains_null_check (int *P) {
    *P = 4;
}
```

Why? 'It's permitted by *undefined behaviour* in C.'

Why really?

C compilers have become extreme ‘performance squeezers’.

They don’t *have* to be. It’s counterproductive!

- greater effort per unit product
- harder to debug → workarounds, not fixes

And so it escalates:

- increasing ‘expertise’ required of programmer
- more {complex, fragmented} tooling
- generate more work to ‘rewrite the C’
- more code → more demand for optimisation (!)

We become invested deeper and deeper in this cycle.

Software performance is no longer about infrastructure!

It's a systemic problem of how software is developed.

New roads induce new traffic. *Systemic*, not 'choice'.

Infrastructure gains are soaked up by a 'software sponge'.

It is not simply 'saving time to spend on features'.

Escalation ensures features *remain costly to implement*.

The malaise is with the industrial roots of software culture.

Functional languages are no better

Lest you think I was just ranting about the madness of C…

We've proved 'well-typed programs don't go wrong'!

Let's get rid of those run-time tags…

The concern of machine efficiency has trumped all others…

Even ones we all agree are more important!

Assumption is always: *the next* software will fix this.

'Solving a crisis by escalation.'

“A few patients
survived longer with transplants
of various organs. On the other
hand, the total social cost exacted by
medicine ceased to be measurable in
conventional terms. Society can have
no quantitative standards by which
to add up the illusion, social control,
prolonged suffering, loneliness,
genetic deterioration, and frustration
produced by medical treatment.”

—from ‘Tools for Conviviality’ (1973)

The blame game

Escalators can often be identified by *blaming the human*.

‘Fix your code!’

‘Remember: we work for the machines!’

Another escalator: 'let's make a new X'

Maybe you don't like C. So create a new language!

How will people interface with older code? Hmm...

```
struct Point
{
    int x_;
    int y_;
};
```

```
Local<Value> GetPointX(Local<String> prop,
                       const AccessorInfo &info) {
  Local<Object> self = info.Holder();
  Local<External> wrap = Local<External>::Cast(self->
     GetInternalField(0));
  void* ptr = wrap->Value();
  int value = static_cast<Point*>(ptr)->x_;
  return Integer::New(value);
}
void SetPointX(Local<String> prop, Local<Value> value,
                       const AccessorInfo& info) {
  Local<Object> self = info.Holder();
  Local<External> wrap = Local<External>::Cast(self->
     GetInternalField(0));
  void* ptr = wrap->Value();
  static_cast<Point*>(ptr)->x_ = value->Int32Value();
}
```

What just happened

We revere the *internal* and denigrate the *external*.

- special word: *legacy*

This disregard is not shared by empirical science

- 'external validity'

Nor is it shared by all engineers

- design as a discipline

Massively counterproductive.

- ++integration_cost, ++reimplementation
- --tool_power, --maintainability

Monotonicity yes; reconciliation no

Forking off a new $whatever$ is just ‘what we do’.

It is perceived as a free operation.

Integration is someone else’s problem. . .

. . . and affects only people who have themselves to blame.

They should have used the shiny new thing from the start!

It’s the future!

Better ways: *possible*, but still not *done*

> "Integration is linking your .o files together, freely intercalling functions… … you don't have a foreign loader, you don't coerce types across function-call boundaries, you don't make one language dominant, and you don't make the woes of your implementation technology impact the entire system.
>
> "[All these] can be addressed in a Lisp implementation. This is just not the way Lisp implementations have been done…"

—Richard P. Gabriel
"Lisp: good news, bad news, how to win big"
AI Expert, 1994

Re: [v8-users] Re: Making v8::Persistent safe to use

Jun 21, 2013 1:34 AM

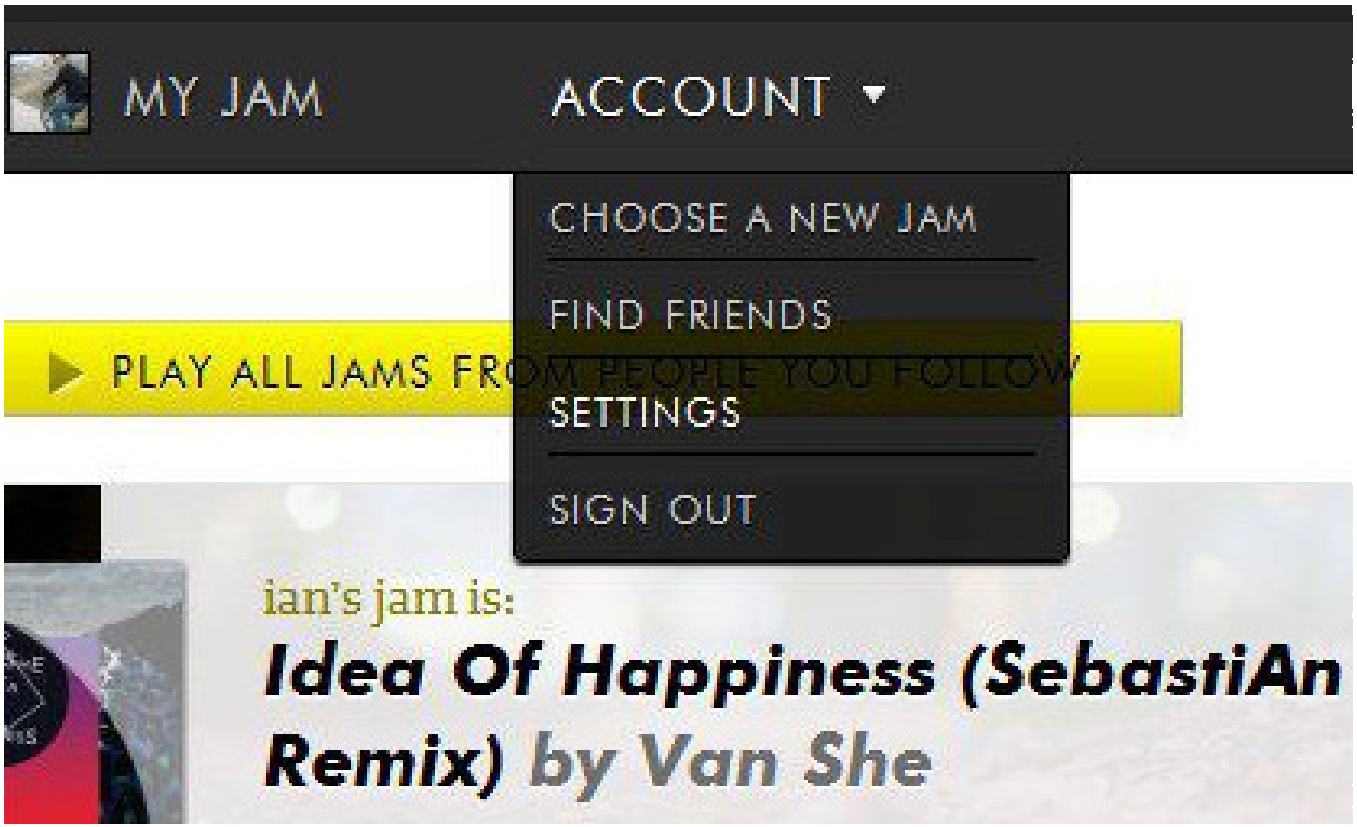
Posted in group: **v8-users**

On Fri , Jun 21, 2013 at 9:19 AM, Dan Carney <dca...@chromium.org> wrote:

> The transition from Local to Handle won't happen for a while. It's more of a cleanup step after everything else is done, and there's no urgency since there shouldn't be any performance impact.

The callback signature changes alone break almost every single line of v8-using code i've written (tens of thousands of them), and i am still

This Is My Jam will become a read-only time capsule on **September 26, 2015**. This means you won't be able to post anymore, but you'll be able to browse a new archive version of the site.



But keeping the jams flowing doesn't just involve our own code; we interoperate
with YouTube, SoundCloud, Twitter, Facebook, The Hype Machine, The Echo Nest, Amazon, and more. Over the last year, changes to those services have meant instead of working on Jam features, 100% of our time's been spent updating years-old code libraries and hacking around deprecations just to keep the lights on. The trend is accelerating with more breaking/shutting off each month, soon exceeding our capacity to fix it.

‘I speak about radical monopoly when one industrial production process exercises an exclusive control over the satisfaction of a pressing need, and excludes nonindustrial activities from competition. Cars thus monopolize traffic. … That motor traffic curtails the right to walk, not that more people Chevies than Fords, constitutes radical monopoly.’

Illich would say…

Among linked software, there is a *radical monopoly*

It is a monopoly of the *recent*.

‘If you can’t keep up with change, that’s your problem.’

This affects anyone on a budget (including researchers).

It’s not ‘your’ problem; it’s one of technologies and tools.

… and the culture which created them

… and the culture which they create.

Energy as inequity

Sometimes, a little project will become ‘hot’.

Investment of effort in a codebase is good, surely?

Maybe not, if it lessens others’ ability to benefit

The more power expended on a codebase

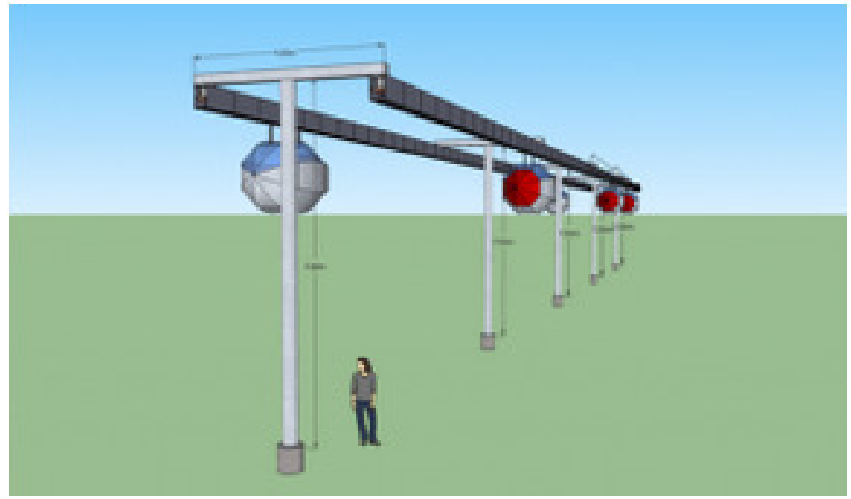
… the more power is needed to *use* or *contribute*

Think: Linux kernel, Android, LLVM, …

What can we *do* about all this?

- opt out of society?
- take shelter from the worst?
- join in, and enjoy job security?

Yet more advanced technology. . . ?



The founder of Mint.com, Aaron Patzer, has been researching alternative urban transportation under a company called Swift over the past six months, but he has determined that the personal maglev system he had been envisioning is economically not viable for a company to produce. Patzer described all of his findings and development in a blog post (hat tip to Tech Crunch), including the high economics of such a transportation network.

Illich: “I have chosen ‘convivial’ as a technical term to designate a modern society of **responsibly limited tools**.”

“Commuter transportation leads to negative returns when it admits, anywhere in the system, speeds much above those reached on a bicycle.”

‘Responsible self-limitation’

We are quite used to this idea.

One example: information hiding

Another example: pure functional programming

These are evidently not the only limitations needed.

They may not even be among the best ones to choose.

To advance, we need new ways to limit ourselves.

Self-limitation 1: against performance-squeezing

It is hard to definitively forbid ‘performance squeezing’.

One idea: for language impls, insist on debuggability.

Ask: what are the *externalities*?

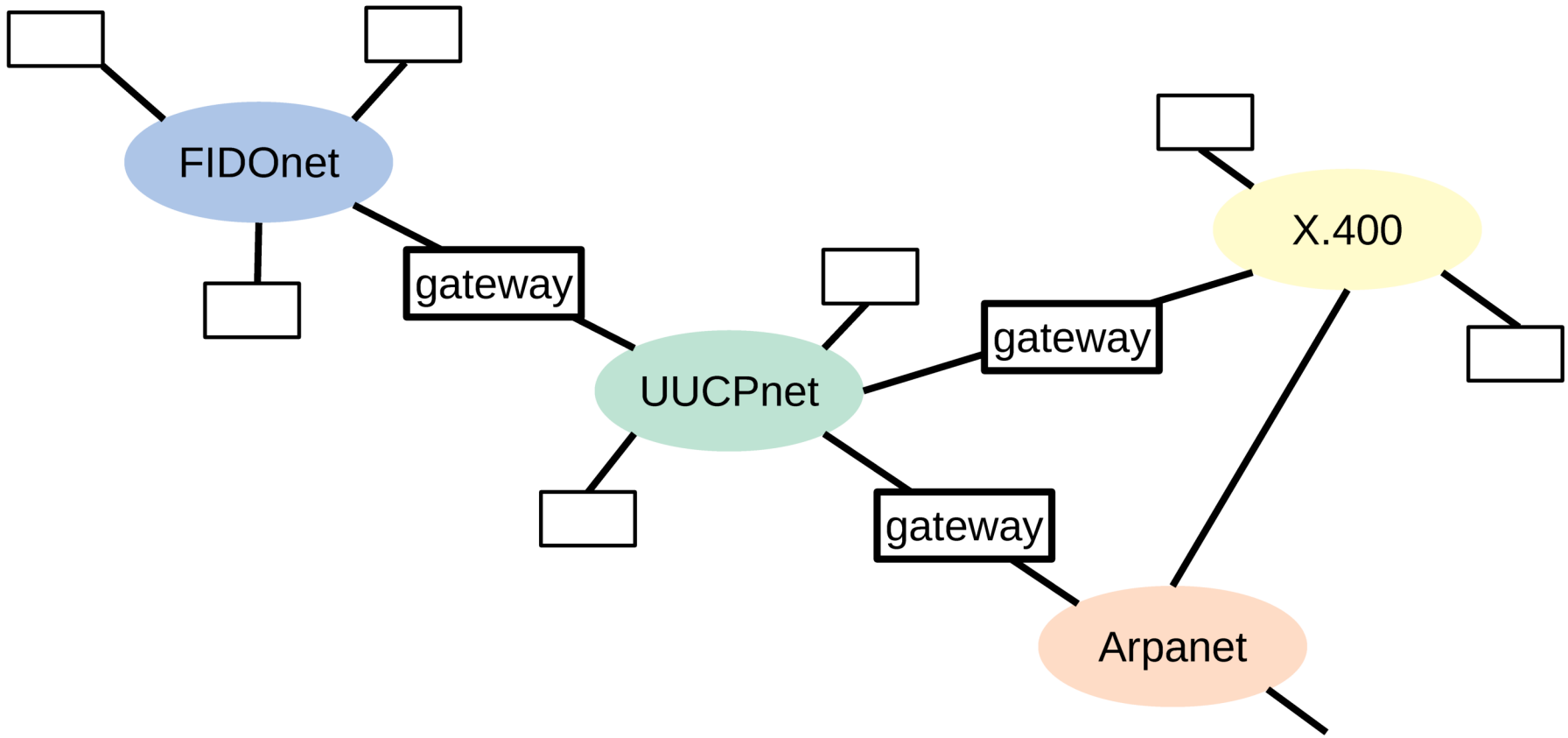
Ask: what story do the metrics not tell?

Performance comes at what cost? (aside: or what COST?)

Self-limitation 2: if it stacks, it must federate

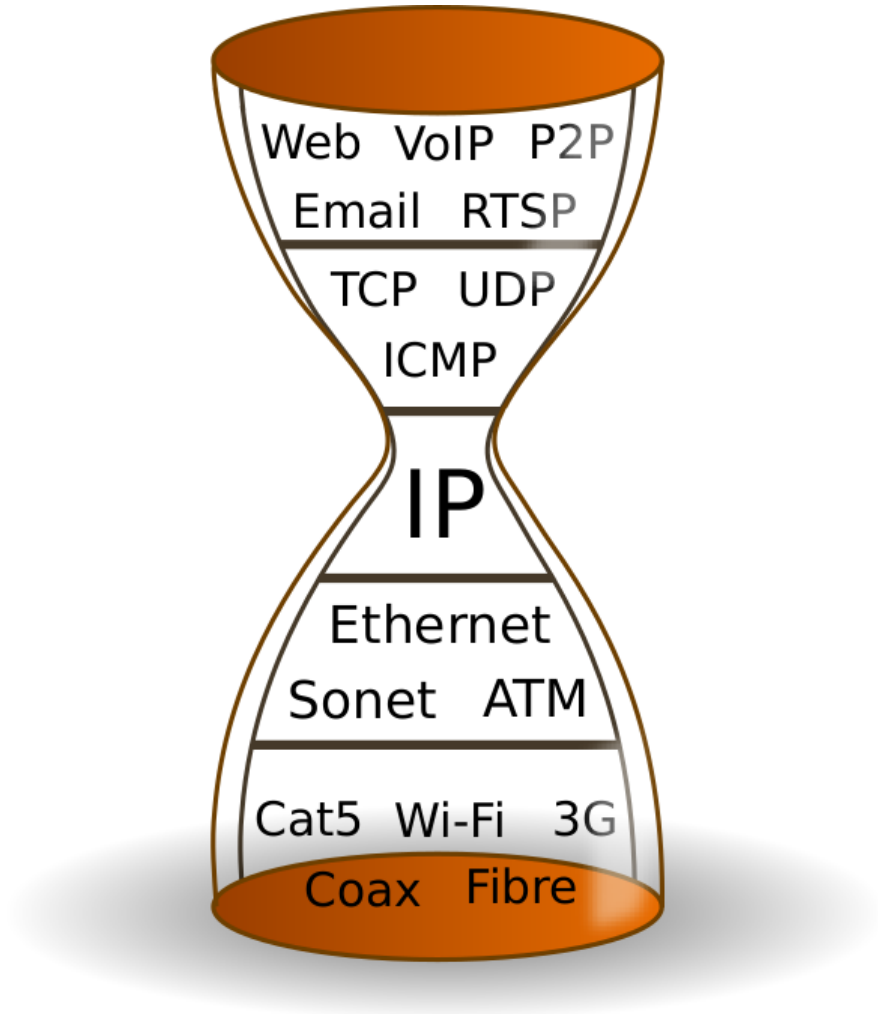
Pre-Internet, sending e-mail across networks was *possible*

- . . . if the right *gateways* were available + running



Deploying *new applications* was beyond the means of most

Self-limitation 2: if it stacks, it must federate



IP: an interface that *federated* the network abstraction

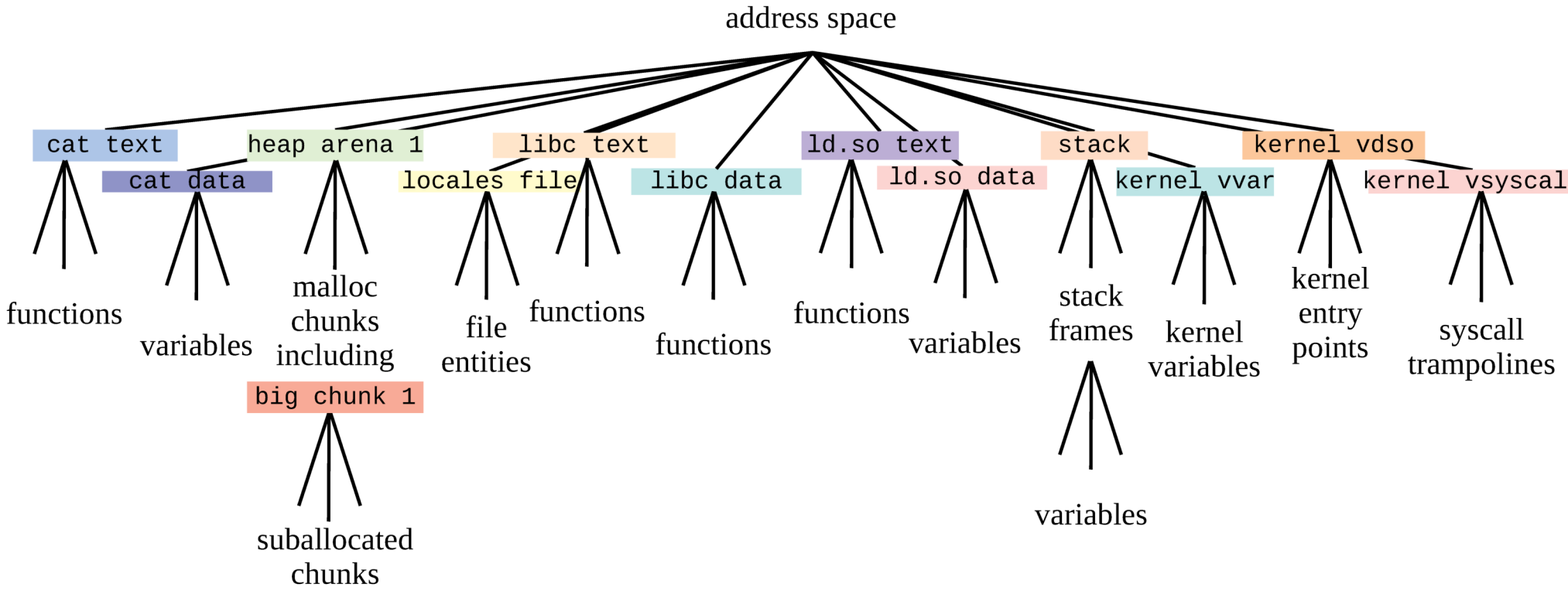
- obviated the escalating need for ALGs

What else can we federate?

Self-limitation 2: if it stacks, it must federate

My own liballocs project federates memory abstractions

- Unix memory is no longer raw bytes; ‘typed allocations’
- a step towards federating high-level language impls



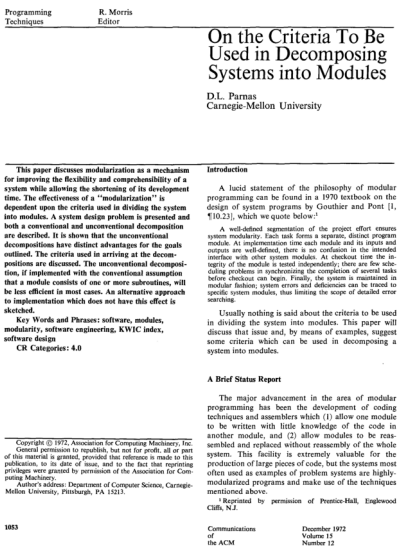
Federability is also what separates O-O from ADTs…

- Cook, Onward! 2009

‘Interoperability’ has been named the essence of object-orientation

- Aldrich, Onward! 2013

Self-limitation 3: degradeable hiding



Programming Techniques — R. Morris Editor

On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

Key Words and Phrases: software, modules, modularity, software engineering, KWIC index, software design
CR Categories: 4.0

Author's address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

1053

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:[1]

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

Usually nothing is said about the criteria to be used in dividing the system into modules. This paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules.

A Brief Status Report

The major advancement in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system. This facility is extremely valuable for the production of large pieces of code, but the systems most often used as examples of problem systems are highly-modularized programs and make use of the techniques mentioned above.

[1] Reprinted by permission of Prentice-Hall, Englewood Cliffs, N.J.

Communications of the ACM — December 1972 Volume 15 Number 12

"The formats of control blocks used in queues in operating systems and similar programs must be hidden within a 'control block module'. It is conventional to make such formats the interfaces between various modules. Because design evolution forces frequent changes on control block formats, such a decision often proves extremely costly."

D.L. Parnas

On the criteria to be used in decomposing systems into modules

CACM, December 1972

Tracing a Paradigm for Externalization: Avatars and the GPII Nexus

COLIN CLARK, OCAD University
ANTRANIG BASMAN, Raising the Floor

We will situate the concept of an **avatar** *(a working simulacrum of part of a system separated from it in space or time) with respect to traditional concepts of programming language and systems design. Whilst much theory and practice argues in favour of* **insulation** *(the creation of architectural boundaries prohibiting the leakage of information) we will find that many successful systems take a diametrically opposed approach. We name this family of systems as those based on* **externalised state transfer***. Rather than hiding implementation details behind APIs, object interfaces or similar, these systems actively advertise their internal structure and its coordinates via data and metadata. Examples of these systems include RESTful web applications, MIDI devices, and the DWARP debugging file format. We discuss such systems and how we can purposefully design new systems embodying such virtues in a more distilled form.*

1 AVATARS

An avatar is a part of a system of which a working simulacrum can be naturally embodied elsewhere. The simulacrum is a functioning replica of the remote system, which can stand in for it to some level of faithfulness without having to ask questions of it at each interaction. The notion of "naturalness" we hope to establish, however, cannot be determined by examining properties of a single group of interacting systems. Instead, we will need to consider ecologies of such related systems, their users, and networks of other authors who work with them. By exhibiting different kinds of systems which we consider meet and do not meet our definition, we hope to sharpen understanding of the distinction we are interested in, as well as promote new development techniques that make the implementation of avatars natural and idiomatic.

We have borrowed and extended the term "avatar" from its traditional use describing a visual or schematised representation of a user of a system. In our usage, an avatar is a more or less symmetrical concept: two systems may be avatars of one another if they meet the criterion of faithfulness or representation without one of them being necessarily designated as "primary" or "real".

The most common appearance of the avatar pattern is in supporting a remote user interface, whereby a single system exposes the potential for control to different locations, separated by a network of some kind. To the extent that the remote system might be agnostic as to whether the controlled system is local or remote, it could be said to be working with an avatar, or functioning local replica of the controlled system. However, whilst it is one of the most convenient, the avatar pattern is an uncommon choice for implementing such a remote UI. A more common pattern has the remote UI bound to the controlled system by a message bus along which passes essentially arbitrary messages — that is, messages with individual schemas whose content do not correspond to an integrated view of the system state. We could describe this pattern as a remote procedure call (RPC) or API idiom where the message bus simply stands in for whatever function calls the two parts of the architecture found it necessary to make to each other from time to time. Examples of this remoting pattern are the X Windows protocol, the Microsoft Windows Remote Desktop protocol, Virtual Network Computing, etc.

1.1 Avatars for the future

There are numerous other productive uses of the avatar model, which need not be limited to scenarios where the separation is in space. The avatar model could be used to support advanced live programming scenarios [3], where a system that is being authored can be asked, explicitly or implicitly, to speculate about states it might reach in the future as a result of different authorial gestures. This supports the highest level of liveness (L6) in Church et al. [4]'s classification of live programming paradigms. In this case, part of the current system becomes an avatar for part of its state in the future — it is a working simulacrum allowing the user to see some or all of its behaviour, but which is integral in itself and may be immediately discarded without any side effects on the behaviour of the current system.

1.2 Avatars from REST

However, it's not accidental that by far the most prolific technology for supporting remote user interfaces, the web, is also the birthplace of a software idiom, REST [6], which is a foundational part of the avatar model. In REST, "representations of state are transferred" — that is, the remote system does not simply answer arbitrary messages, but instead guarantees to transfer complete (exhaustive) representations of part of its application state to the UI client. The expectation is that the client then operates on this state in the form of a local avatar of some greater or lesser degree of faithfulness, as a functioning replica of some aspect of the remote system. The extension that the avatar model represents with respect to REST is that we provide for transfer of representations capable of dynamic evolution of behaviour. This distinction is elaborated in Basman et al. [3] where we describe the extension that a *potentia* represents with respect to a system, like the web, with the document object model (DOM) for its pages transmitted over HTTP as HTML, which only provides for representing an *extensa*.

1.3 Avatars not from objects

There are descriptions of systems which appear to conform to our description of an avatar, but which we recognise as part of a distinct conception. Bank [2] contains a now 'legendary interview by Wired Magazine with James Gosling, in which he describes an experience listening to a concert, watching "semirobotic lights which seemed to dance to the music" and being inspired by them to think differently about "making behaviour flow through networks". The resulting language that he designed, Java, is a paradigm member of what we named the API idiom for distributed behaviour. But there is a subtlety to be worked out here — since Java, Smalltalk *et al* do indeed allow behaviour to be shipped wholesale around the network and reconstituted at a remote site for further execution. Why do

"One of the reasons why many old MIDI instruments continue to be musically viable is… due [to] a means for externalizing the complete state of a musical device: all its patches, voice parameters, and settings. MIDI's designers only anticipated [these messages'] use as a means for loading and saving patches to and from external storage. In practice, however, this [also] enabled an unexpected ecosystem of third-party, software-based patch editors and alternative control hardware to emerge. "

Colin Clark and Antranig Basman

Tracing a Paradigm for Externalization, 2017

Information hiding is a heuristic based on anticipation.

‘I predict these details might change. Hide them.’

‘I predict these details won’t change. Expose them.’

What if our predictions are wrong?

We get this wrong all the time. Interface churn!

‘Hard’ abstraction is a recipe for disposability.

‘Soft’ abstraction provides a *separate* door exposing details

Tools for de-escalation

"We must guard against falling into the damaging rejection of all machines as if they were works of the devil."

It is not a contradiction that software can help de-escalate itself.

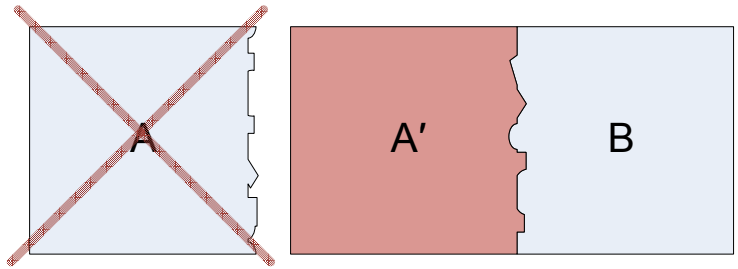
Such software should engender *much less future programming*

Tools for reconciliation

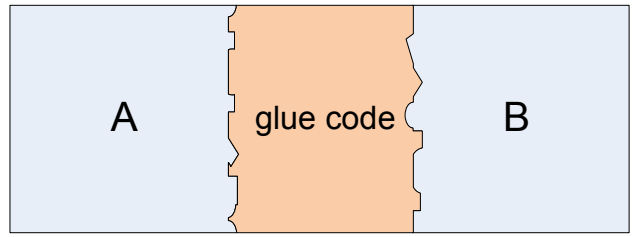
Constantly spawning: abstractly similar, *concretely different*

How can we reconcile them? Currently: at great cost.

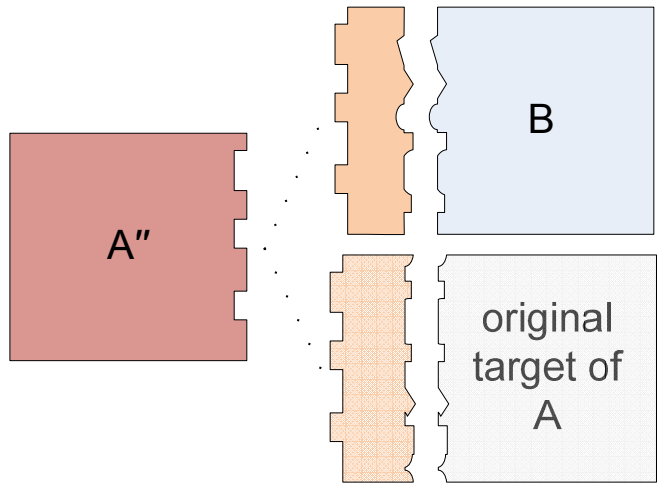
edit or patch



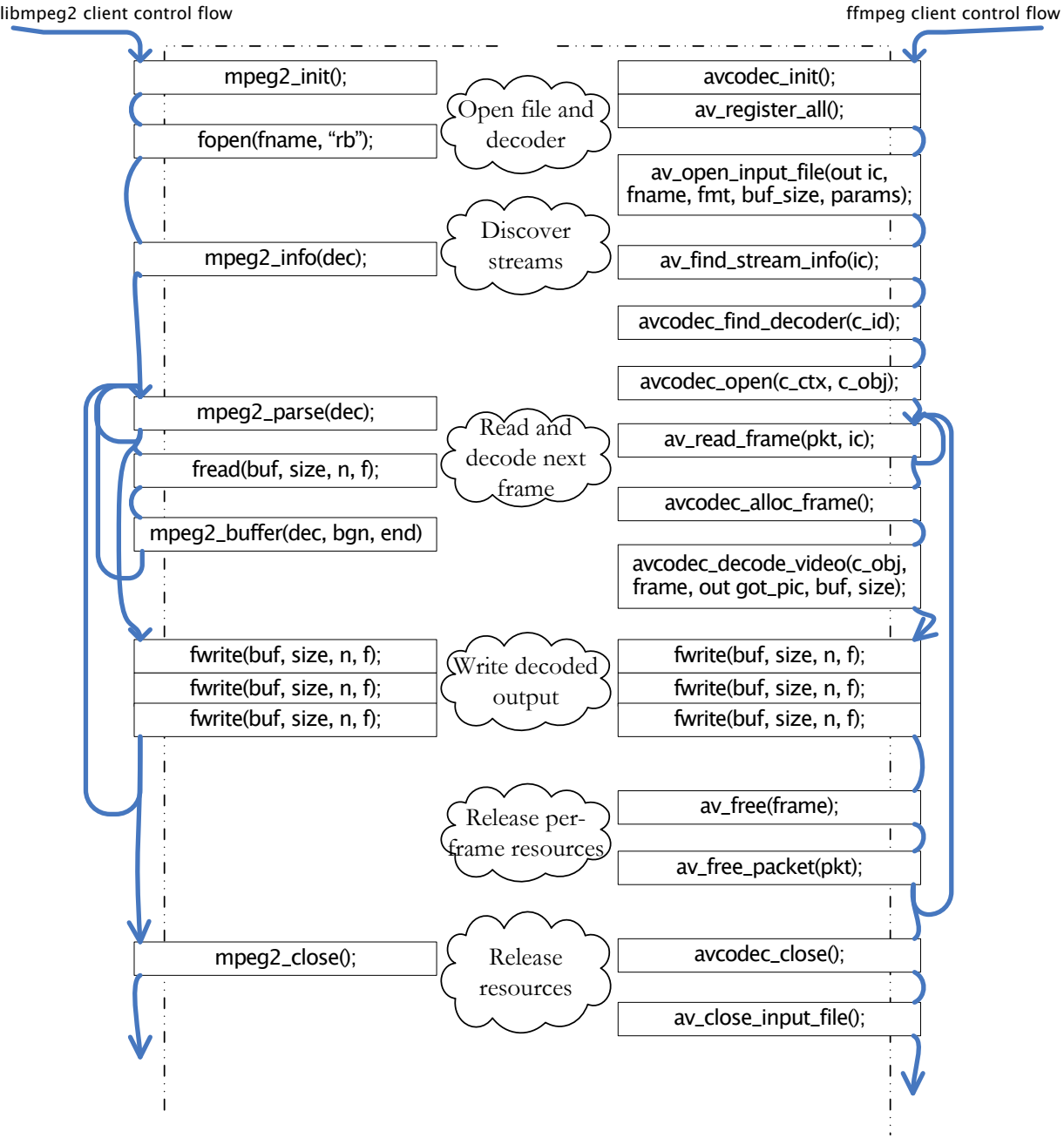
glue coding



abstraction layer



McIlroy wanted interchangeable ‘at reasonable effort’

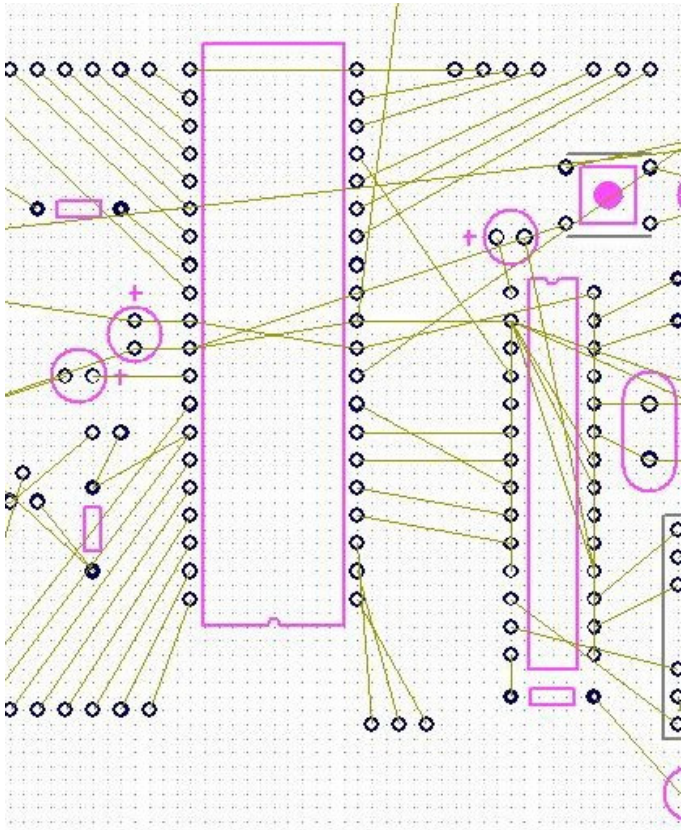


Problems:

- non-1-to-1 mappings
- context-sensitive
- data, not just code

Need tools
which (*semi*-)automate
the reconciliation
of interface differences.

Tools for integration



Hardware (and other domains)

- chip *invents its view* on outside
- keeps components simple
- … and composable

Software:

- no equivalent

Tools for description

```
$ man 5 proc
...
       /proc/[pid]/maps
              A file containing the currently mapped memory
              regions and their access permissions.

              The format of the file is:
...
```

If ‘format’ were machine-readable, I wouldn’t have to write:

```
int  nfields  = sscanf( linebuf ,
    "%lx-%lx %c%c%c%c %8x %2x:%2x %d %4095[\x01-\x09\x0b-\xff]\n",
    &entry_buf->first, &entry_buf->second, &entry_buf->r, &entry_buf->w,
    &entry_buf->x, &entry_buf->p, &entry_buf->offset, &entry_buf->devmaj,
    &entry_buf->devmin, &entry_buf->inode, entry_buf->rest);
```

. . . nor be fragile to changes in this format.

Culture for de-escalation

"Cultural change" is a problem, not a solution

We need a culture that values empowering individuals

. . . not providing warm bodies to feed the beast.

There's a lot of wall to tear down. How?

An unsuccessful tactic: pleading

"With Project Oberon we have demonstrated that flexible and powerful systems can be built with substantially fewer resources in less time than usual. The plague of software explosion is not a 'law of nature'. It is avoidable, and it is the software engineer's task to curtail it."

—Wirth, *A Plea for Lean Software*. *Computer*, 1995

No doubt deliberate effort *can* build simple software, but

- A new, parallel ecosystem won't shift culture.
- It contributes to the escalation!

Probably also unsuccessful: embarrassing

Culture for de-escalation

Those of us who are teachers wield enormous power.

The norm is to teach the 1970s industrial view of software.

… without even acknowledging this as a culture!

Wanted: not just ‘shaping the future’…

… ‘shaping the shaping of the future’!

Programming languages, programming culture.

Teaching for conviviality

We mostly teach {internal, industrial} viewpoints

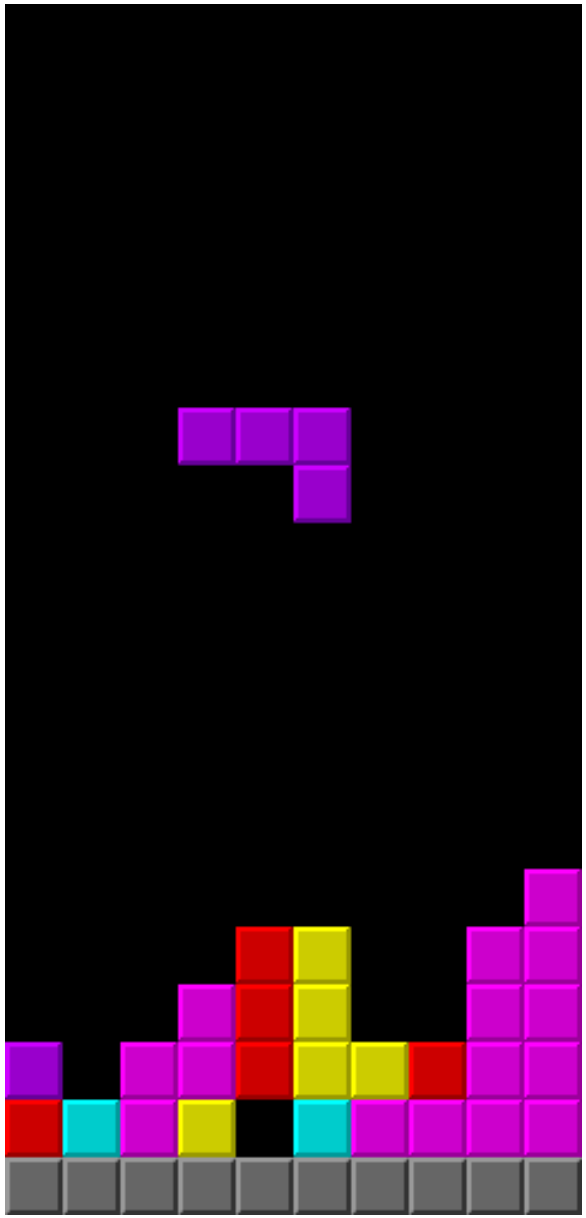
“a project”, “a client”, build “a system”

Performance and reliability seen as internal…

… not systemic effects

‘I optimised it and it runs faster!’

‘I proved it correct!’



The moral of Tetris:

“Development
is only sustainable if it makes
efforts to conserve complexity”

It is a game we will continue to lose.

Thank you for your indulgence.

Picture credits

Ken and Den – Peter Hamer (CC BY-SA)

Xerox Alto: PARC

stuffed Eeyore: ChipmunkRaccoonOz (CC BY-SA)

Illich – unknown (fair use)

Mac University Consortium logo: via folklore.org

Von Neumann – Alan Richards (via the IAS and CHM)

Ford assembly line – Literary Review (public domain)

traffic – public domain (US EPA)

McIlroy speaking – Brian Randell

McIlroy relaxing – Brian Randell

escalators at Lloyd's – phogel (CC BY-SA)

Brian Kernighan – Ben Lowe (CC BY 2.0)

bicycle – own work

e-mail gateways – own work

hourglass – Xander89 (CC BY-SA 2.0)

rat's nest – Lohray (CC BY-SA 3.0) [adapted from]