
C++ now
2025
Keynote:
Are We There Yet?
Sean Parent

At the first BoostCon in 2007, before it was C++Now, I gave a keynote, "A Possible Future of Software Development". I first gave the talk at Texas A&M when Bjarne invited me to speak at an industry affiliate event. I gave the talk several more times including to IBM Research, the F-35 Joint Strike Fighter software development team at Lockheed Martin, internally at Adobe, and as a Google Tech talk. The latter was recorded, and you can find it on YouTube.

I’ve heard from people who saw that talk that this had an influence on the development of the Elm language, Prezi (the presentation software), and the UI software for Tesla and WebOS (in Palm->HP->LG TVs).

Dave suggested I give the talk this year, since 18 years later, few of you have heard it (this was long before "That's a Rotate"). How many here saw the original or the YouTube video? This talk looks at what has changed and is an updated version of A Possible Future.

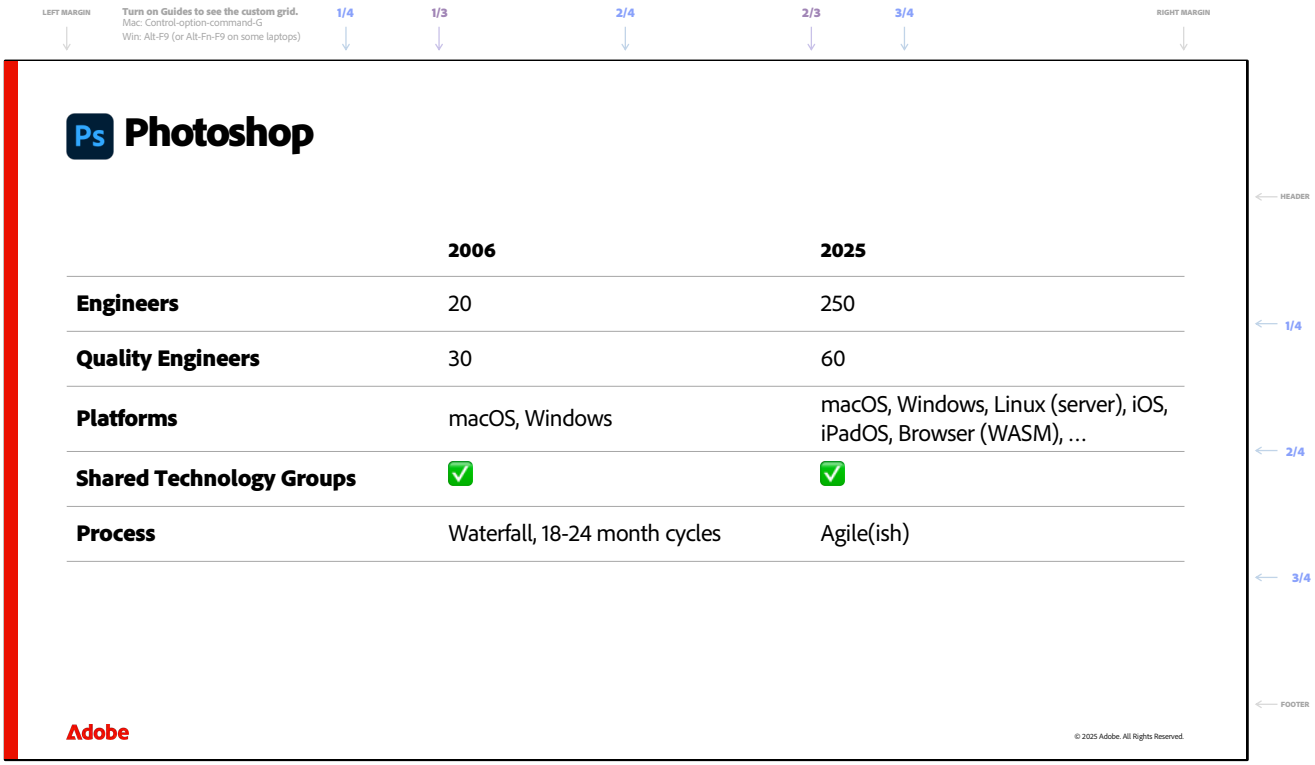
Two years after I gave the talk, the original Adobe Software Technology Lab ended, and I landed at Google working on ChromeOS. I returned to Adobe and worked on a stream of products (Revel, Lightroom Mobile, Lightroom Web, Photoshop Mobile, and Photoshop Web).

And a few years ago, I reformed Adobe's Software Technology Lab. One of our projects is to pick up the work that led to this talk. Project Code Less...

Industry Developments

The original audience for this talk was students, and I started with some of my background. And stats on Adobe.

The company has grown a bit - 5x the number of employees and nearly 8.5x revenue.



	2006	2025
Engineers	20	250
Quality Engineers	30	60
Platforms	macOS, Windows	macOS, Windows, Linux (server), iOS, iPadOS, Browser (WASM), …
Shared Technology Groups	✅	✅
Process	Waterfall, 18-24 month cycles	Agile(ish)

Photoshop has 12.5x the number of engineers and 2x the number of QE (more on that).

Fun fact: Photoshop 3.0 shipped on Sun and SGI Unix. But that was *way* back in 1994.

How we deal with shared technologies internally is entirely different now than before. I couldn't come up with any meaningful comparisons. We still have a heavy reliance on shared tech. The core of Photoshop itself is shipped as a library included in

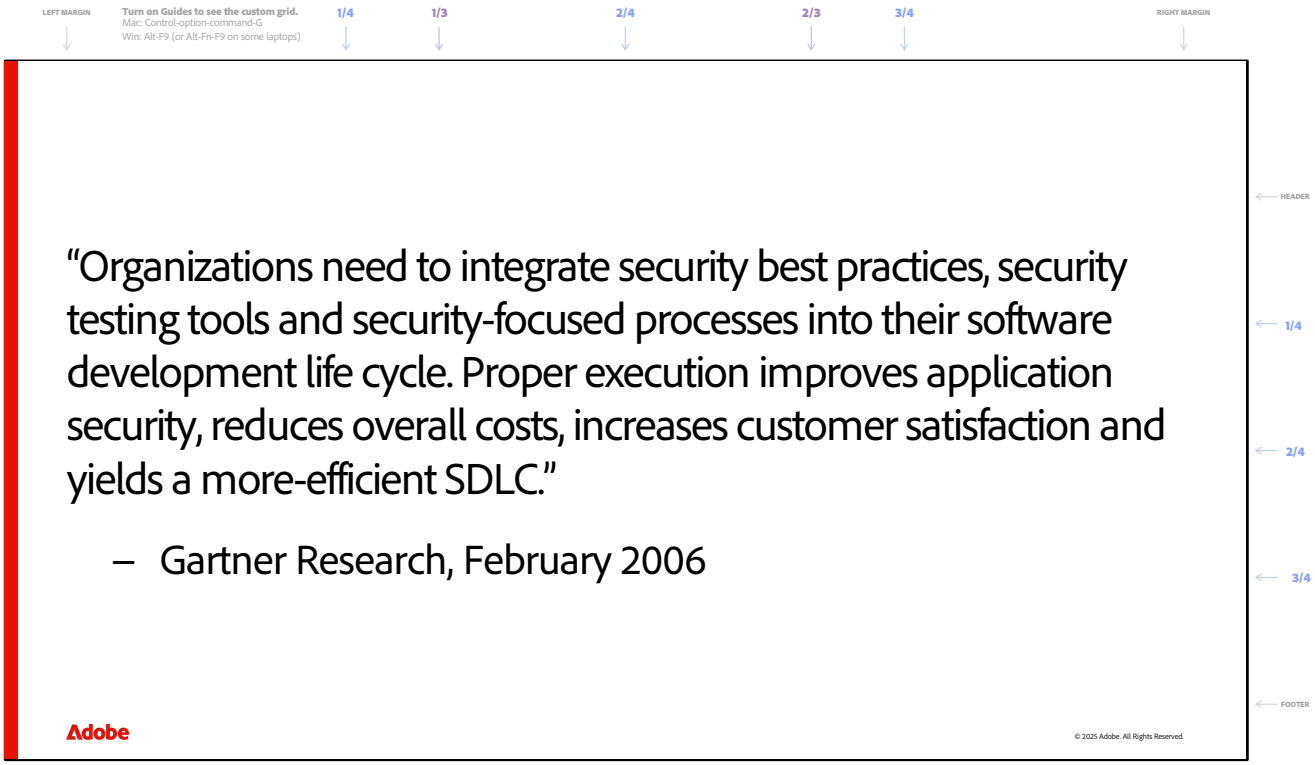
many of our other products (and the same is true of those products in Photoshop).

We didn’t follow a waterfall process out of ignorance. Significant constraints, such as the lead time for printing manuals and box art, and booking manufacturing time for CDs, pushed the processes to front-load features.

Analysts (Then)

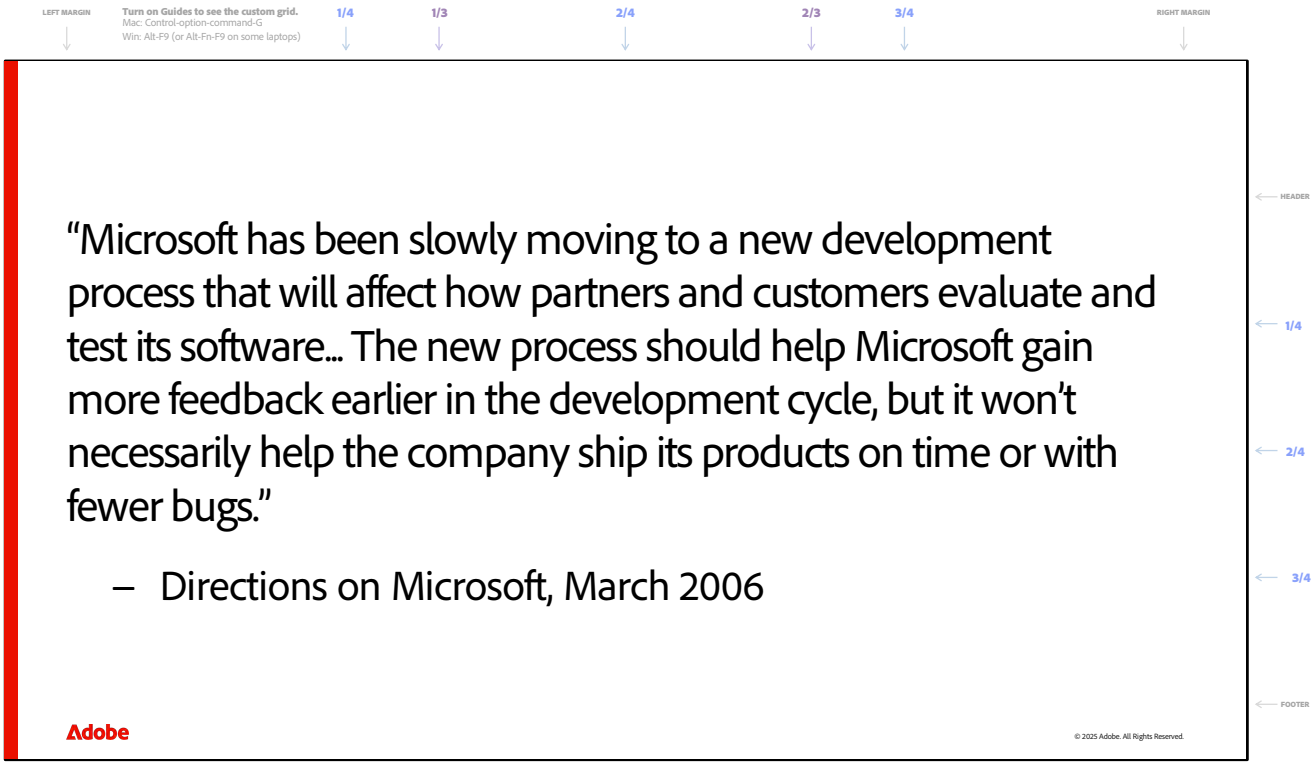
Let's look at what some analysts said in 2006 about where software was headed.

Large Quote



We consider security and memory safety a current hot topic. Adobe acquired Macromedia (and Flash) in 2005. You can look it up if you don't know what happened with Flash. Acrobat Reader has millions of users, and we've announced that Reader will be the default PDF viewer in Microsoft Edge starting this September. Security, both client and server side, remains a hot topic today.

Large Quote



[10:00 – 80:00]

I found this quote entertaining – I don't work at Microsoft, so I can't comment on how their development practices are going, but my sense as a user is they have improved (although I ran into several bugs in PowerPoint just putting this presentation together).

Large Quote

Why the Status Quo Will Fail (2006)

"I've assigned this problem [binary search] in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert the description into a programming language of their choice; a high-level pseudo code was fine... **Ninety percent** of the programmers found bugs in their programs (and I wasn't always convinced of the correctness of the code in which no bugs were found)."

– Jon Bentley, Programming Pearls, 1986

Adobe

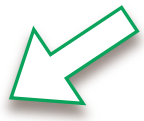
Jon Bentley's Solution (translated to C++)

```
int binary_search(int x[], int n, int v) {
    int l = 0;
    int u = n - 1;

    while (true) {
        if (l > u) return -1;

        int m = (l + u) / 2;

        if (x[m] < v) l = m + 1;
        else if (x[m] == v) return m;
        else /* (x[m] > v) */ u = m - 1;
    }
}
```



Adobe

Jon Bentley's solution is considerably longer and arguably incorrect.

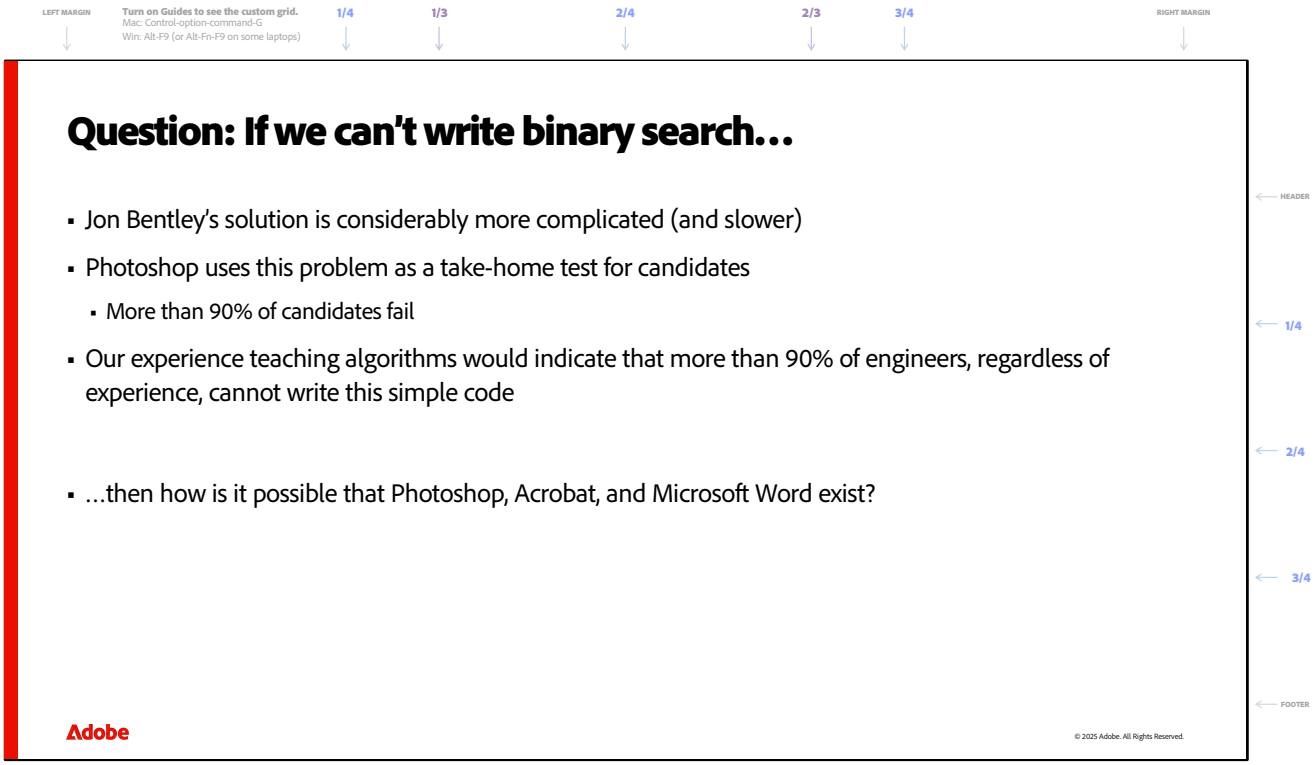
1. Throws away information
2. Possible overflow
3. Extra comparison
4. Requires signed values

Binary Search Solution (from 2006 talk)

```
int* lower_bound(int* first, int* last, int x) {
    while (first != last) {
        int* middle = first + (last - first) / 2;
        if (*middle < x)
            first = middle + 1;
        else
            last = middle;
    }
    return first;
}
```


Adobe

This would be a better solution to the problem.



This slide is from my original talk and still holds as generally true. I haven't measured it, but I feel the percentage of engineers who can write this code has increased. We have moved beyond the "Java Dark Ages," where Java, Design Patterns, and Object-Oriented Programming were taught at the exclusion of machine architecture, algorithms, and data structures. I credit programs like Google's Summer of Code and competitive coding contests for the change.

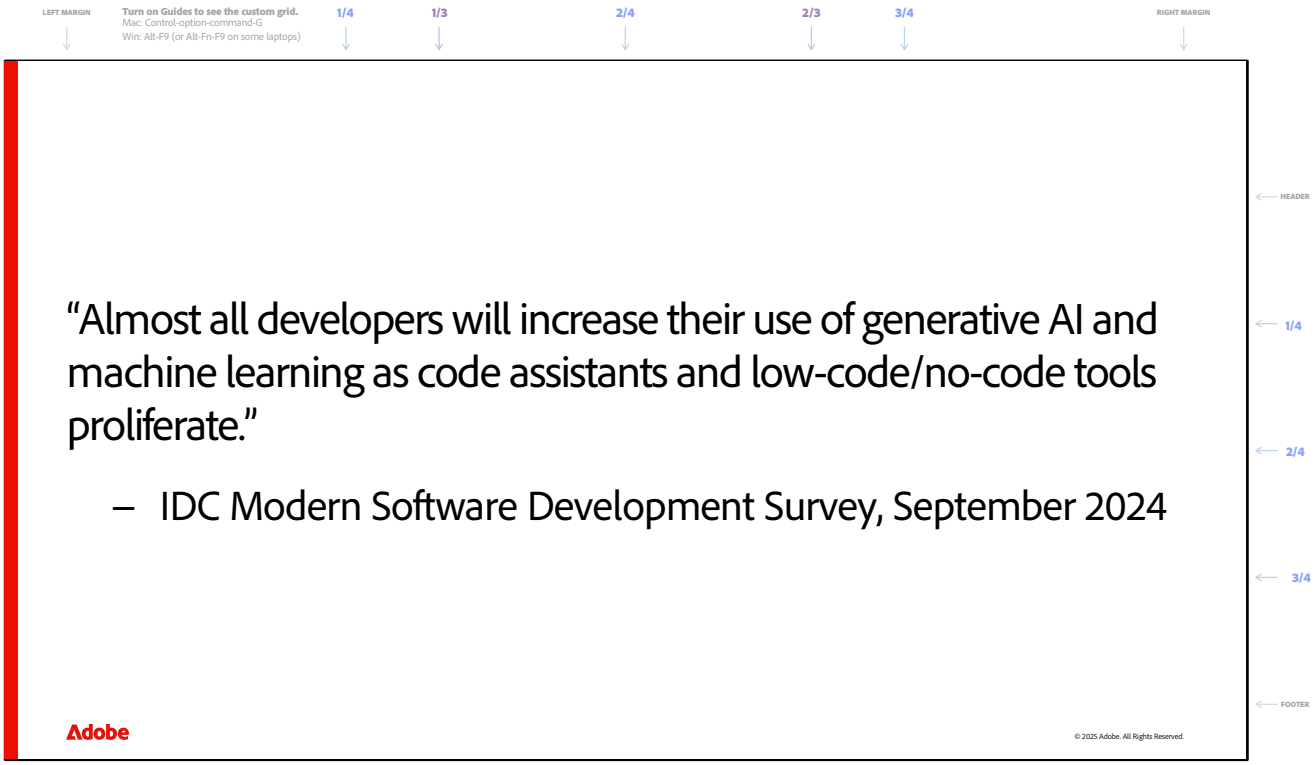
However, some of my colleagues believe we are entering a new dark age driven by AI and JavaScript, creating a generation of developers who don’t understand how the code they write works (or doesn’t). I haven’t been involved with hiring new grads in a while, so I have no firsthand knowledge.

Before answering the question, let’s look at what analysts say today.

Analysts (Now)

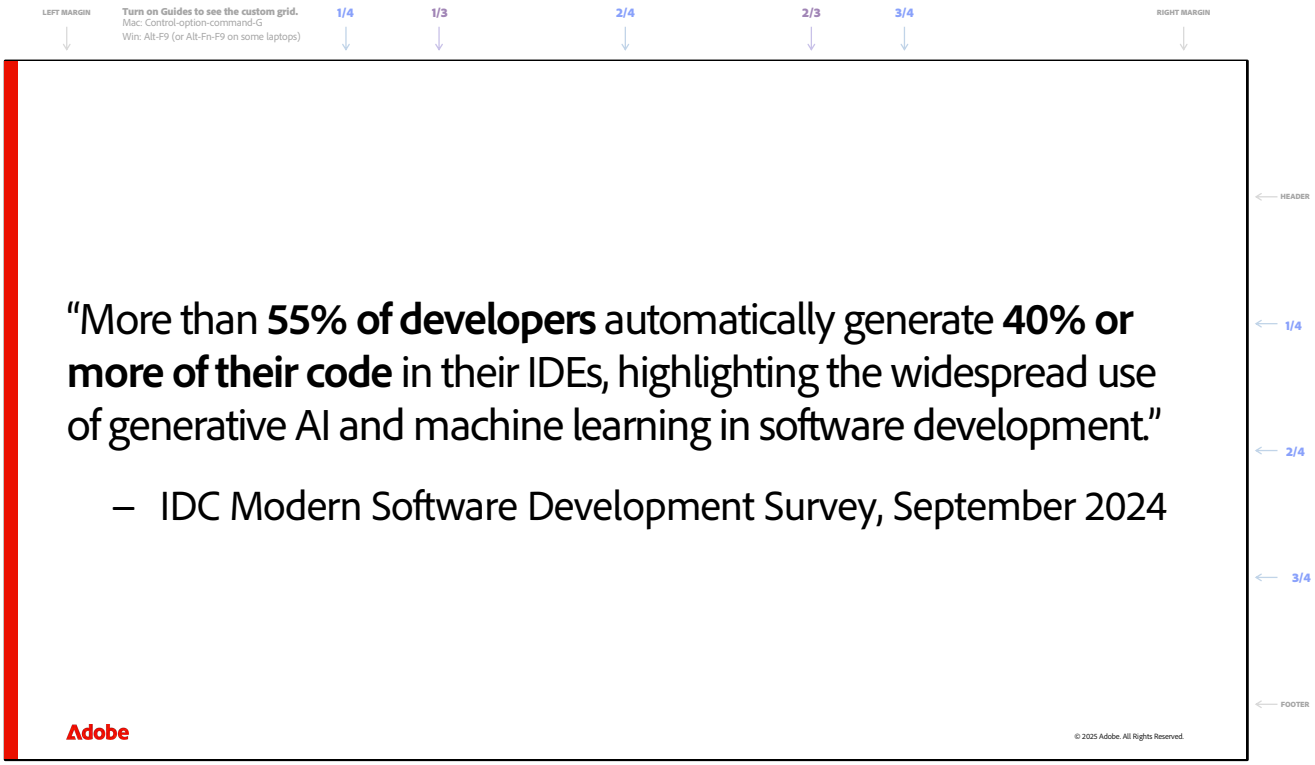
Even though I expected this, I was still a bit surprised…

Large Quote

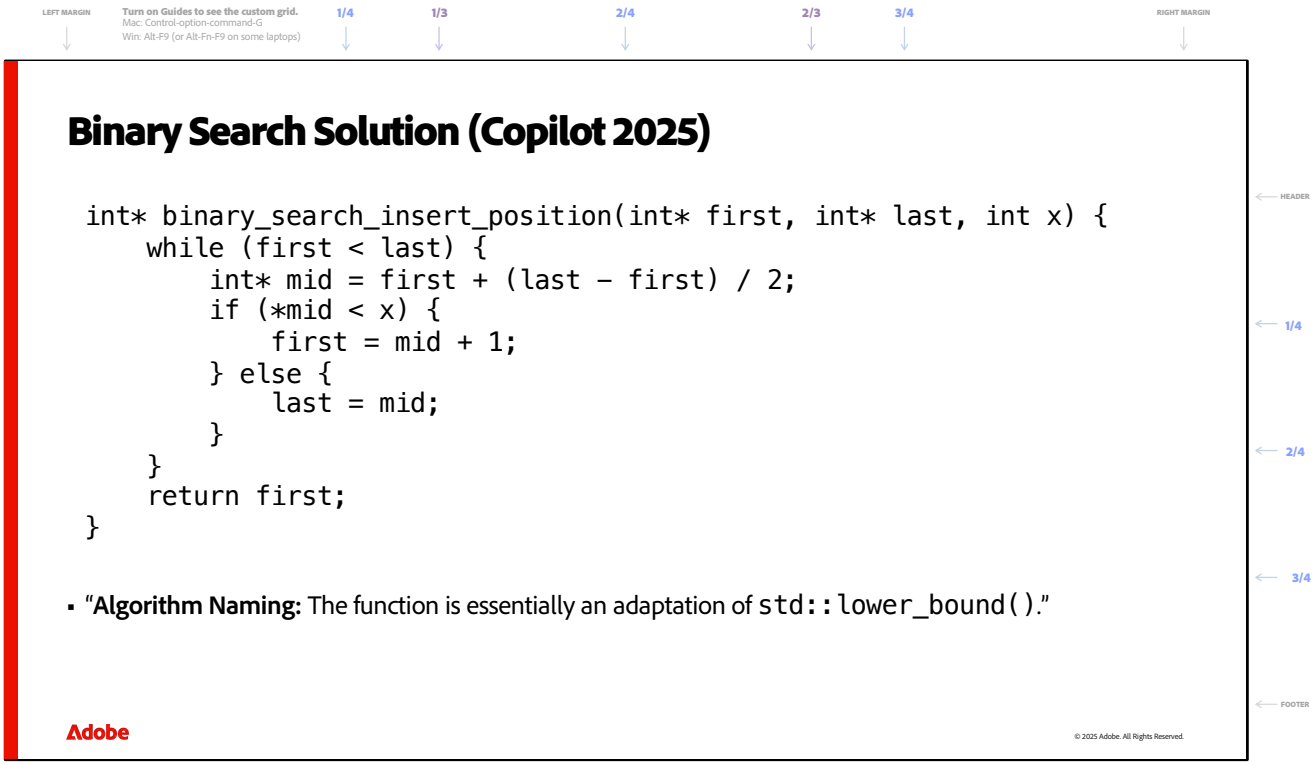


AI dominates the analysts' predictions – even things like secDevOps have minimal mentions, and no mention of memory safety.

Large Quote



I use generative AI often when coding, but I still find this static terrifying.



This code is almost identical to mine. I would write `while first does not equal last`, but this is fine. I've tried this experiment a few times, starting with GPT 2.5 (it failed). The code quality has gotten surprisingly good, surprisingly fast.

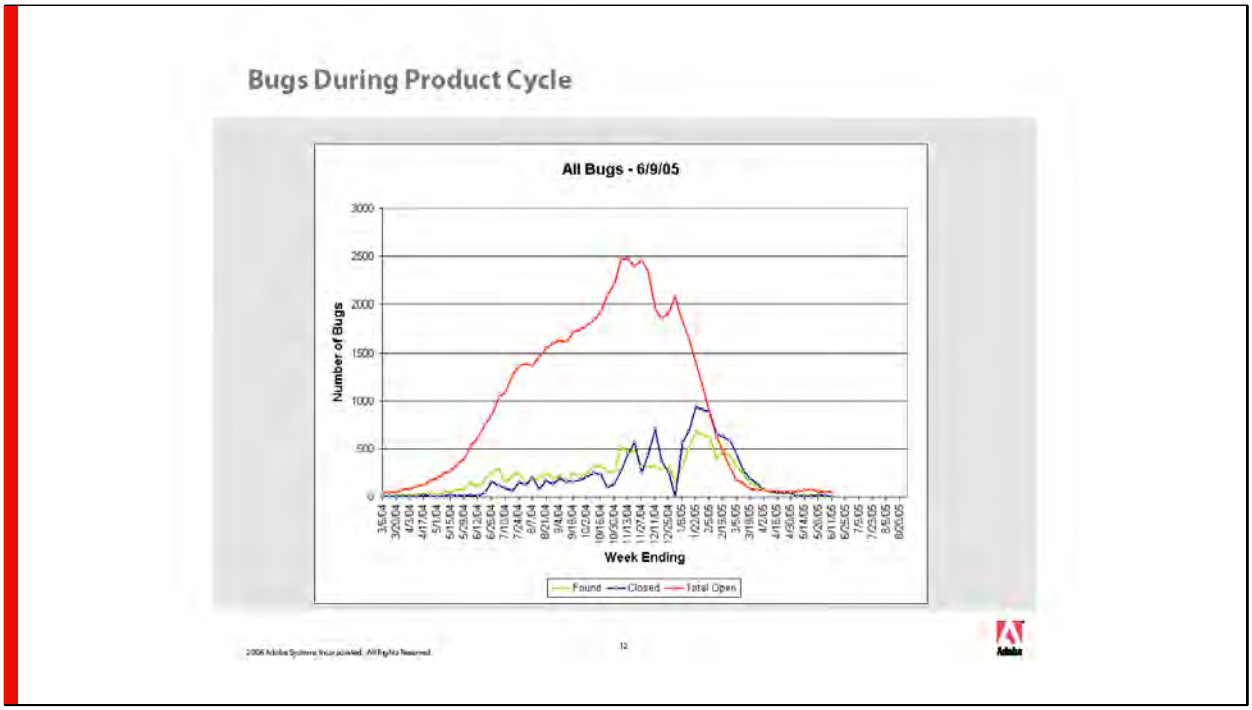
The name of the algorithm was basically pulled from the description of the problem. I asked how the code could be improved, and it pointed out the name, and that this is essential std::lower_bound() – I consider calling std::lower_bound() an

acceptable solution to the binary search problem.

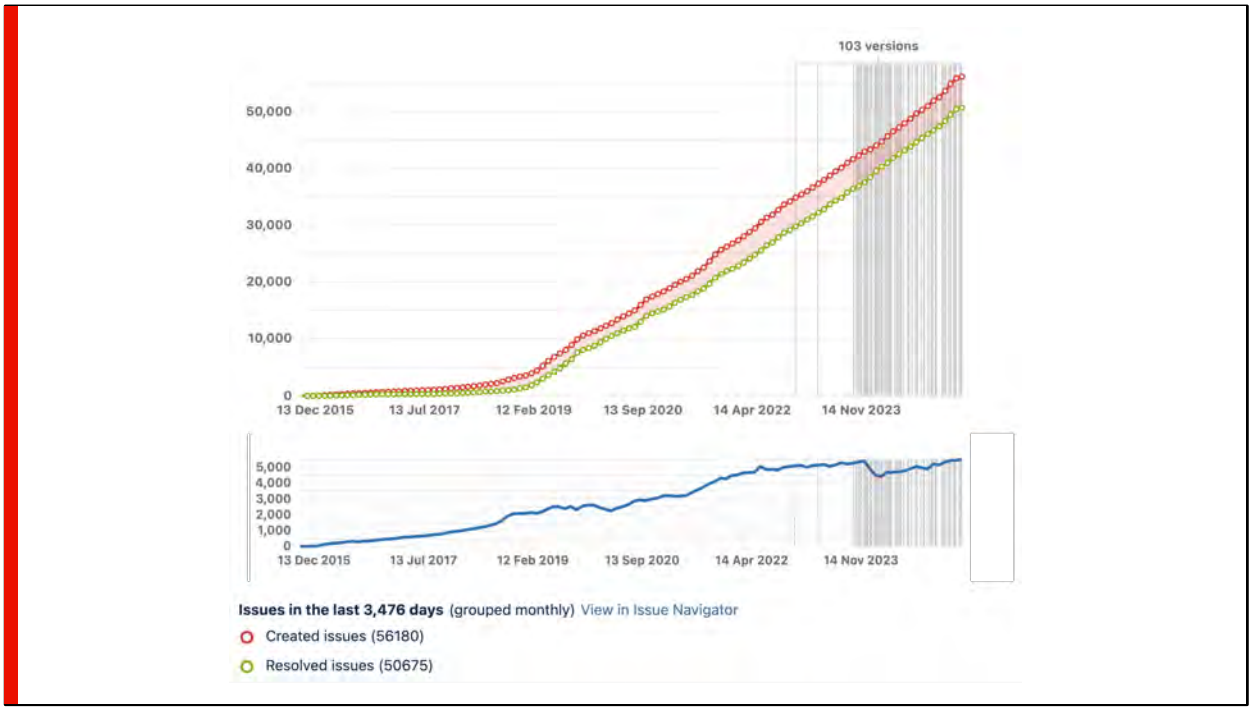
Closer Slide

[20:00 – 70:00]

The analysts are right, problem solved – that's the end of my talk… I think we have a little more time? Let's continue since we aren't all out of a job yet.



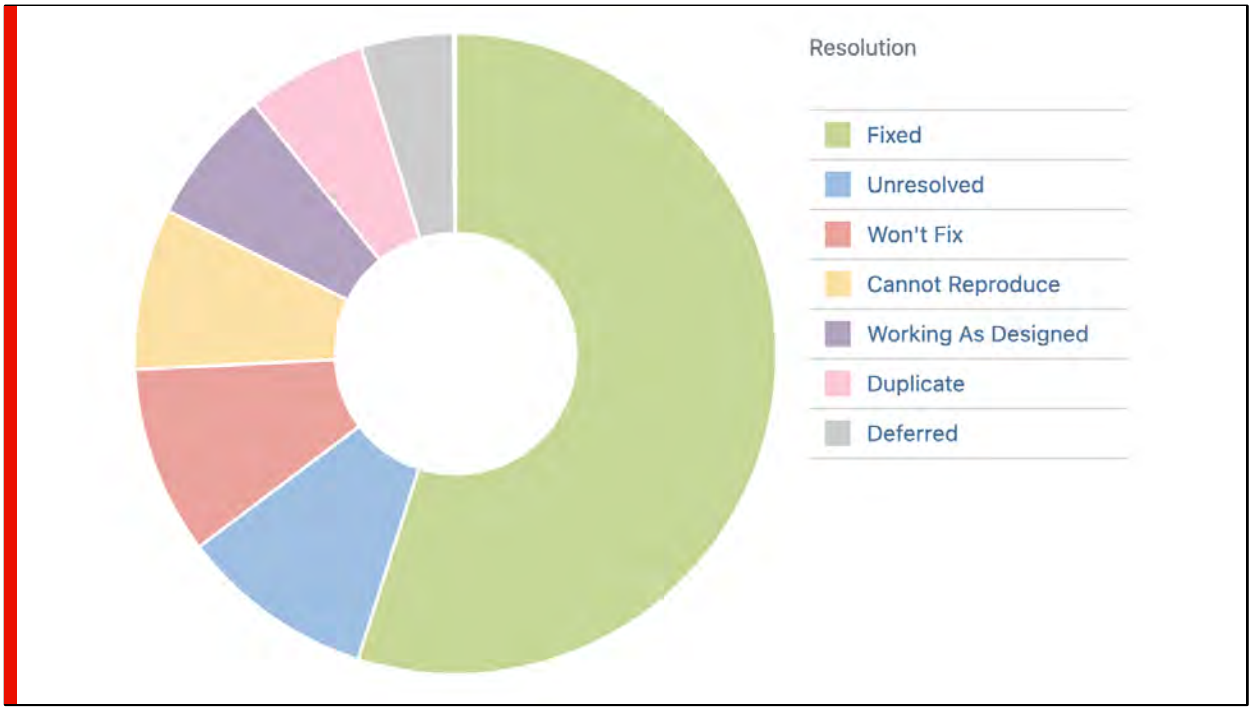
Fortunately (or unfortunately), I don't believe AI fixes everything, yet. Yet. Let's look a little more at how software is developed.



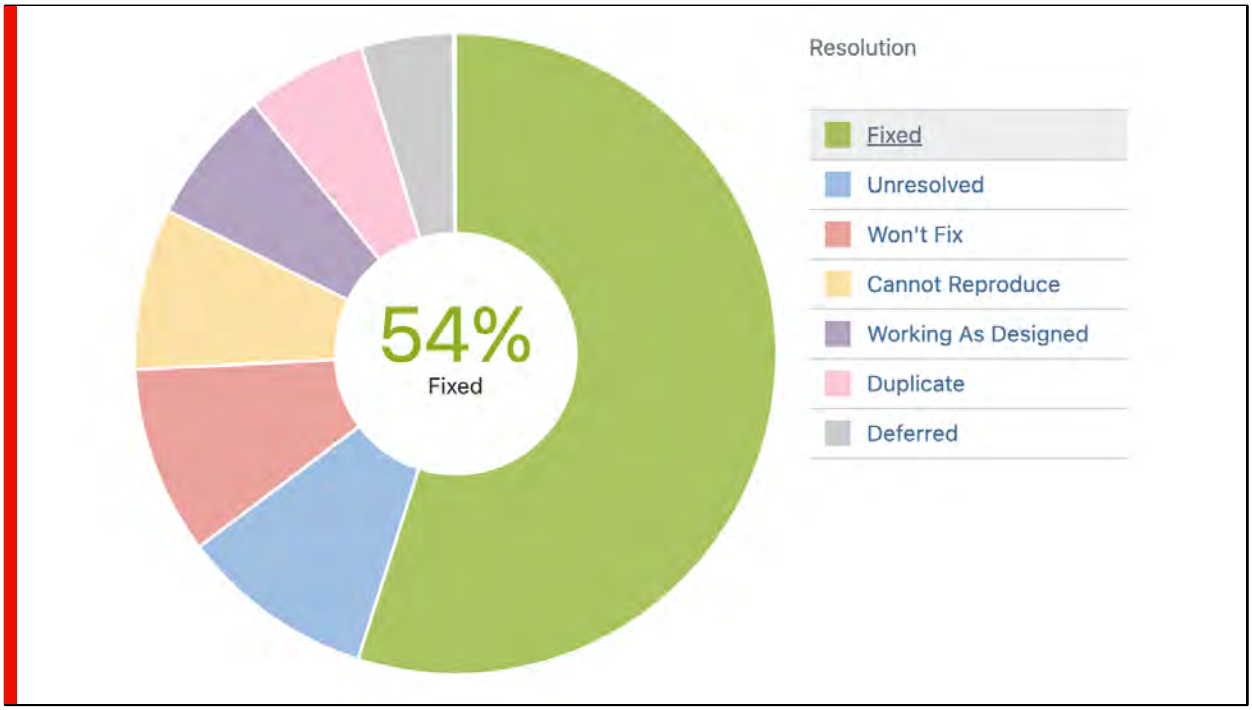
My ability to generate a Jira report that doesn't look like noise sucks.
This is a graph of issues (bugs) for a product for the last 10 years.
The top red line is the running total of issues created
The green line is the running total of issues resolvedThe version lines are a new field, but they give a sense of the release cadence in later releases.

The bend in 2019 was a significant scale-up of the team as the product became an “ecosystem” and extended to more platforms

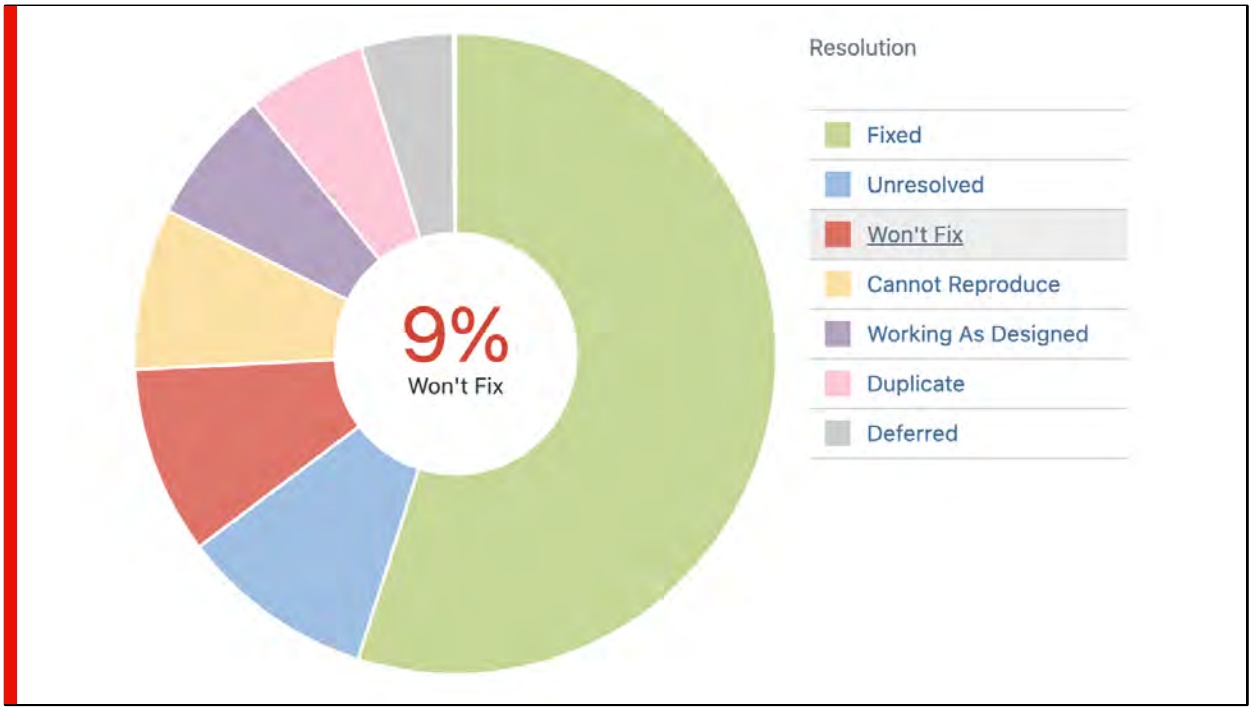
The blue line at the bottom shows the accumulation of unresolved issues. This is the more interesting bit – the bend in the curve on top caused a bump and jitter on the bottom, but it has normalized. For the last several years, there hasn’t been a noticeable increase in unresolved bugs.



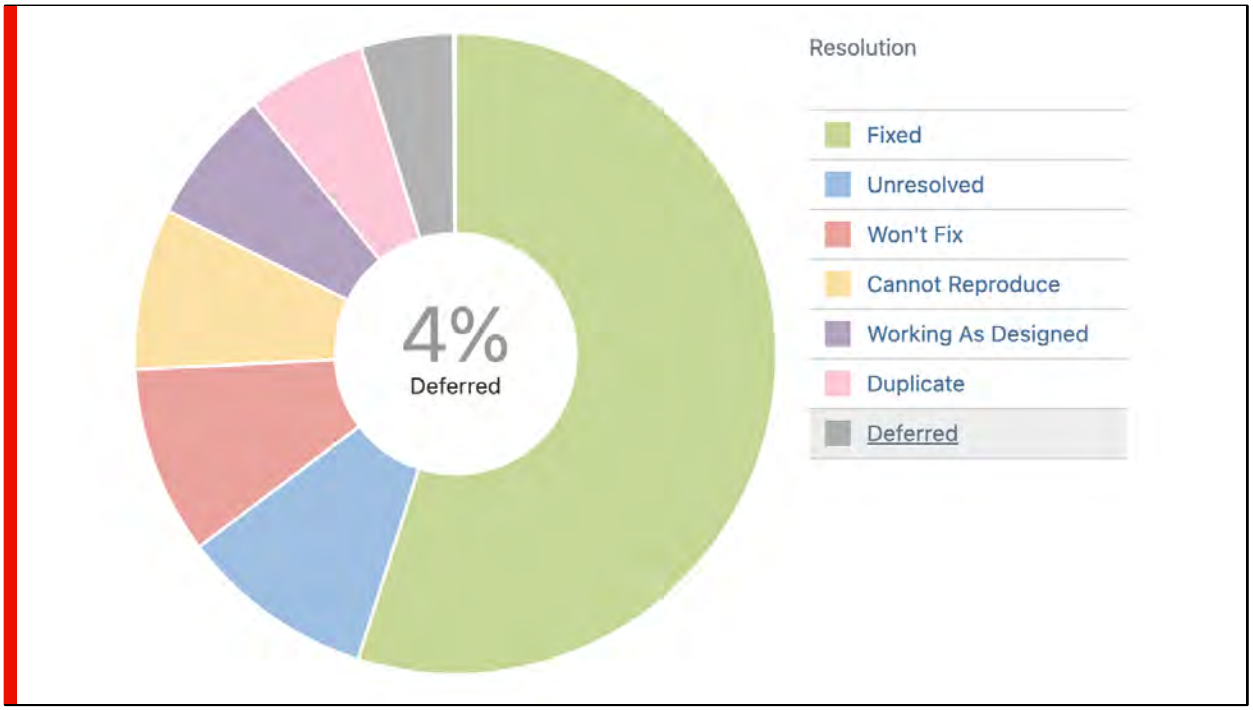
Here is the breakdown of how bugs are resolved over the same period.



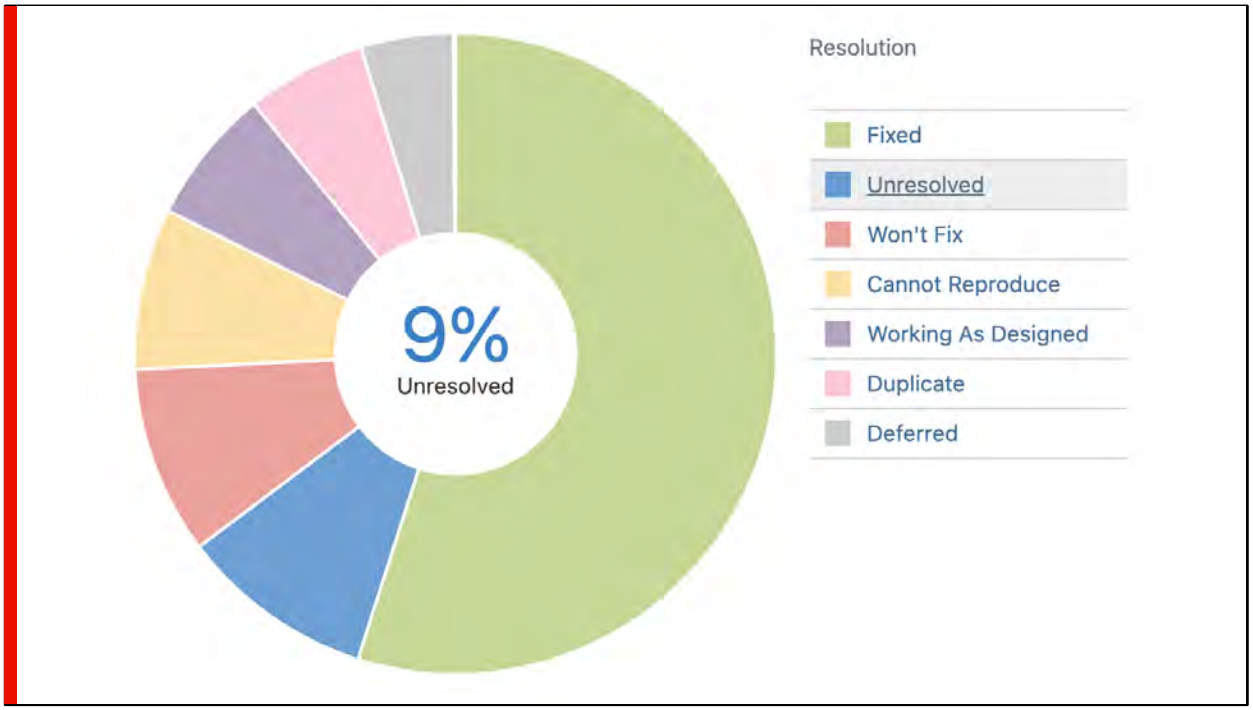
54% - At least more bugs are fixed than not?
In fairness, the % should be a little higher because "Working as Designed" and "Duplicate" are not bugs.
"Cannot Reproduce" is a little more problematic and could indicate a need for better test coverage.



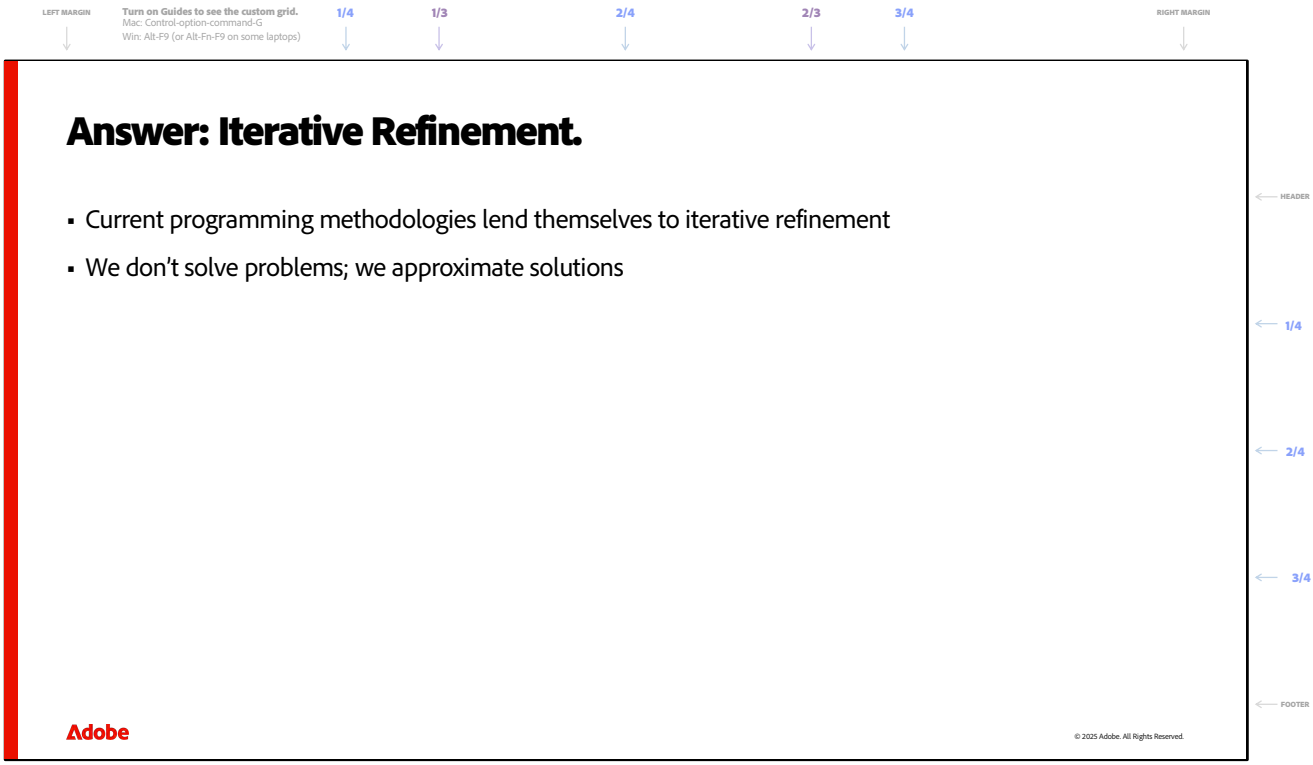
Won’t Fix: “Usually low priority bug, not worth the effort, or no need to fix for a situation that's not important anymore.”
Narrator voice: “At least they thought it was a low-priority bug.”



Deferred: “Will reconsider at a future date” (narrator voice: “They won’t”).

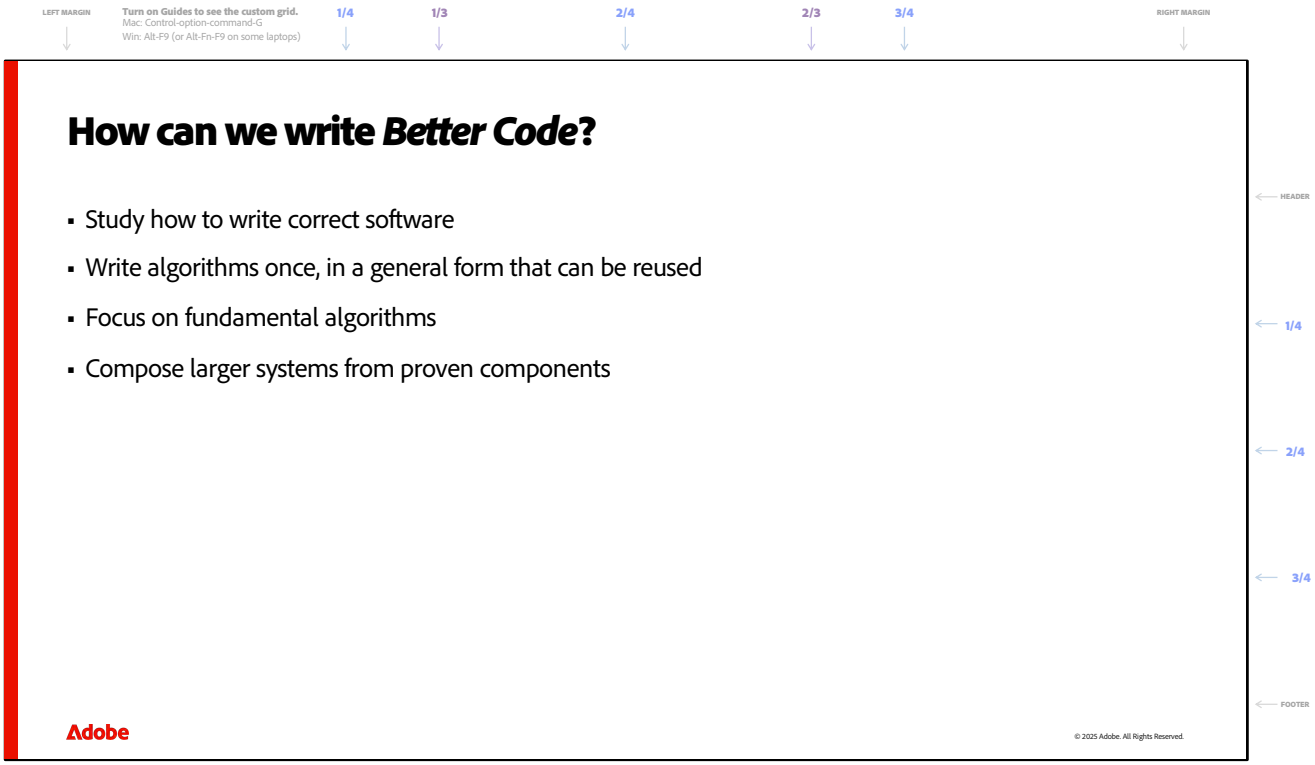


9% of all bugs reported remain unresolved.



Going back to the earlier question, we ship software through a process of iterative refinement.

Who here feels like a good portion of their career is recoding things others have written before, or things you have written before? Why is AI good at code? Because, as an industry, we repeat ourselves a lot.



[30:00 – 60:00]

All code is a liability. We need to focus on the problem of writing less code (not writing bad code faster).

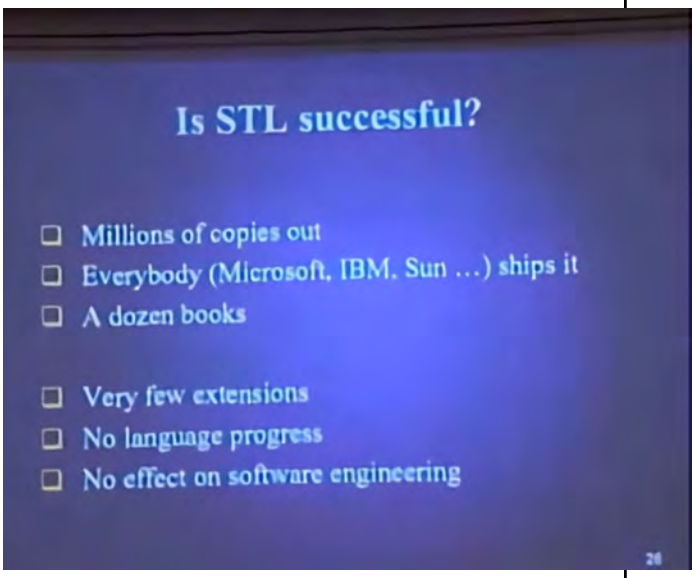
Generic Programming

Who here feels like they spend most of their time writing and rewriting either code you've written before or code someone has written?

LEFT MARGIN Turn on Guides to see the custom grid. Mac: Control-option-command-G Win: Alt-F9 (or Alt-Fn-F9 on some laptops) 1/4 1/3 2/4 2/3 3/4 RIGHT MARGIN

Generic Programming

- In 2006, the idea of generic programming was still young
- In 2002, Alex Stepanov gave a presentation at Adobe with this slide



FOOTER

Adobe

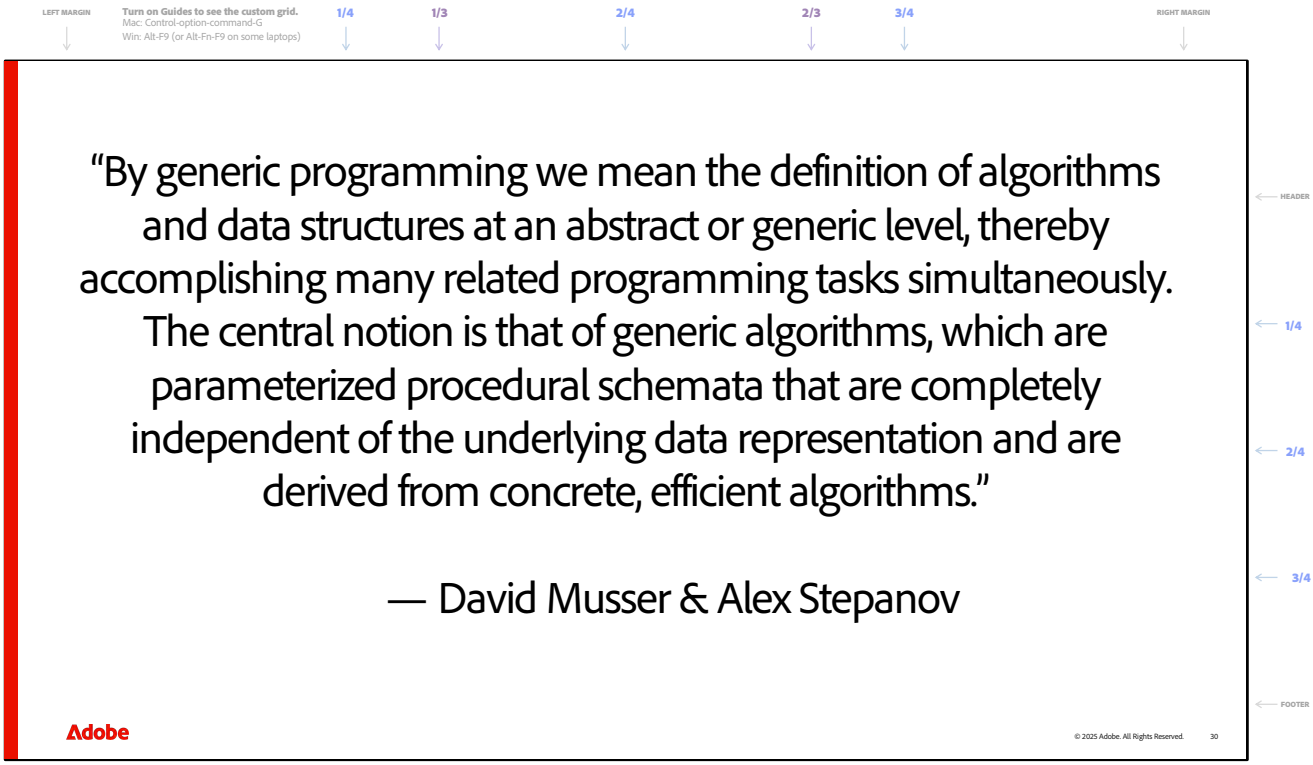
I'm in the right front corner of the audience for this talk – this was the day when I first met Alex in person.

By extension, Alex wasn't referring to standard library extensions, but things like the Boost Graph Library (shipped in 2000). There are many more extensions now, but certainly not as many as I would hope for.

No language progress – Alex's work had directly

influenced C++ Concepts, Rust Traits, and Swift Protocols. None of these do better than (named) duck conformance. The semantics are still relegated to comments.

I believe STL and Generic Programming have significantly impacted software engineering throughout the industry, although I know Alex still views it as a ”failure”.



I think it is worth taking a moment to look at the original definition of generic programming from 1988 (before STL).

"By **generic programming** we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms."

— David Musser & Alex Stepanov

Adobe

"By generic programming we mean the definition of **algorithms** and **data structures** at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms."

— David Musser & Alex Stepanov

Adobe

"By generic programming we mean the definition of algorithms and data structures at an **abstract** or **generic** level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms."

— David Musser & Alex Stepanov

Adobe

"By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby **accomplishing** many **related programming tasks simultaneously**. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms."

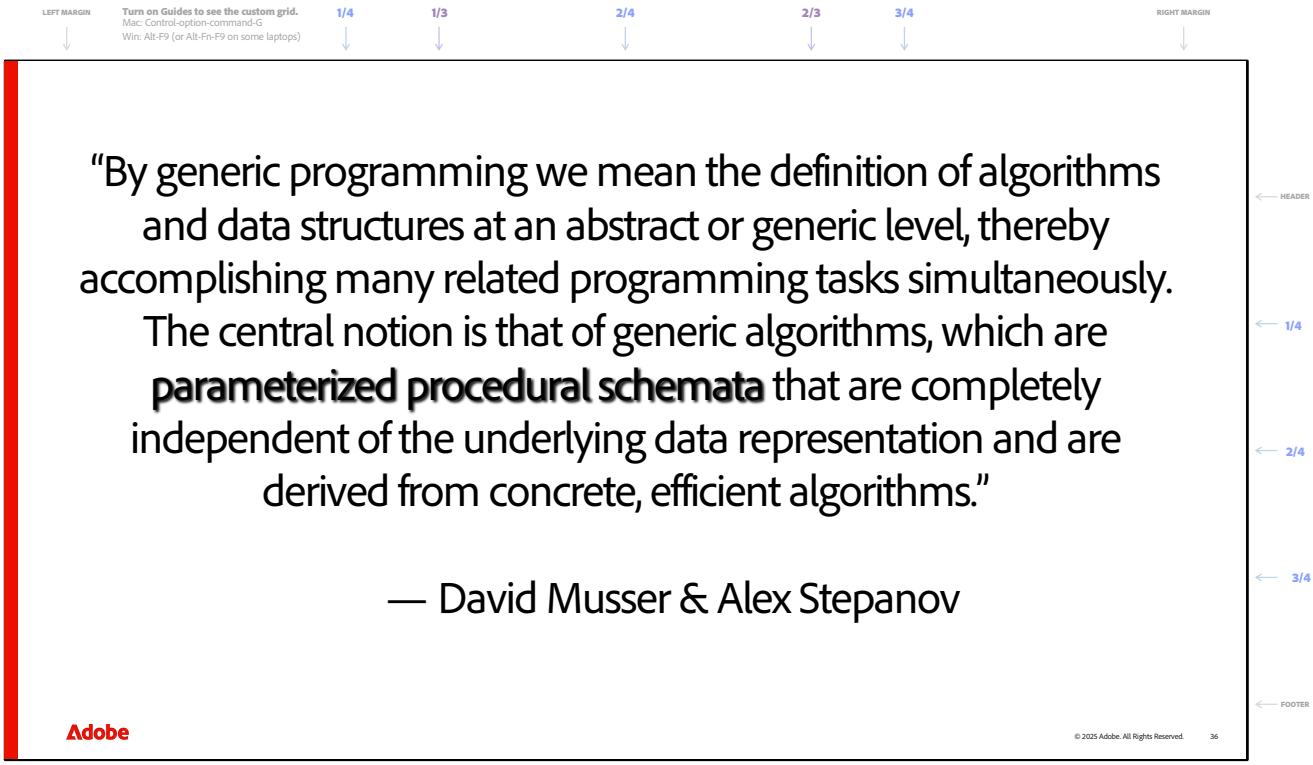
— David Musser & Alex Stepanov

Adobe

"By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The **central notion** is that of **generic algorithms**, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms."

— David Musser & Alex Stepanov

Adobe


LEFT MARGIN
Turn on Guides to see the custom grid.
Mac: Control-option-command-G
Win: Alt-F9 (or Alt-Fn-F9 on some laptops)
1/4
1/3
2/4
2/3
3/4
RIGHT MARGIN
"By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms."
— David Musser & Alex Stepanov
HEADER
1/4
2/4
3/4
FOOTER
Adobe
© 2025 Adobe. All Rights Reserved.
36

[40:00 – 50:00]

"By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely **independent** of the **underlying data representation** and are derived from concrete, efficient algorithms."

— David Musser & Alex Stepanov

"By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are **derived from concrete, efficient algorithms**."

— David Musser & Alex Stepanov

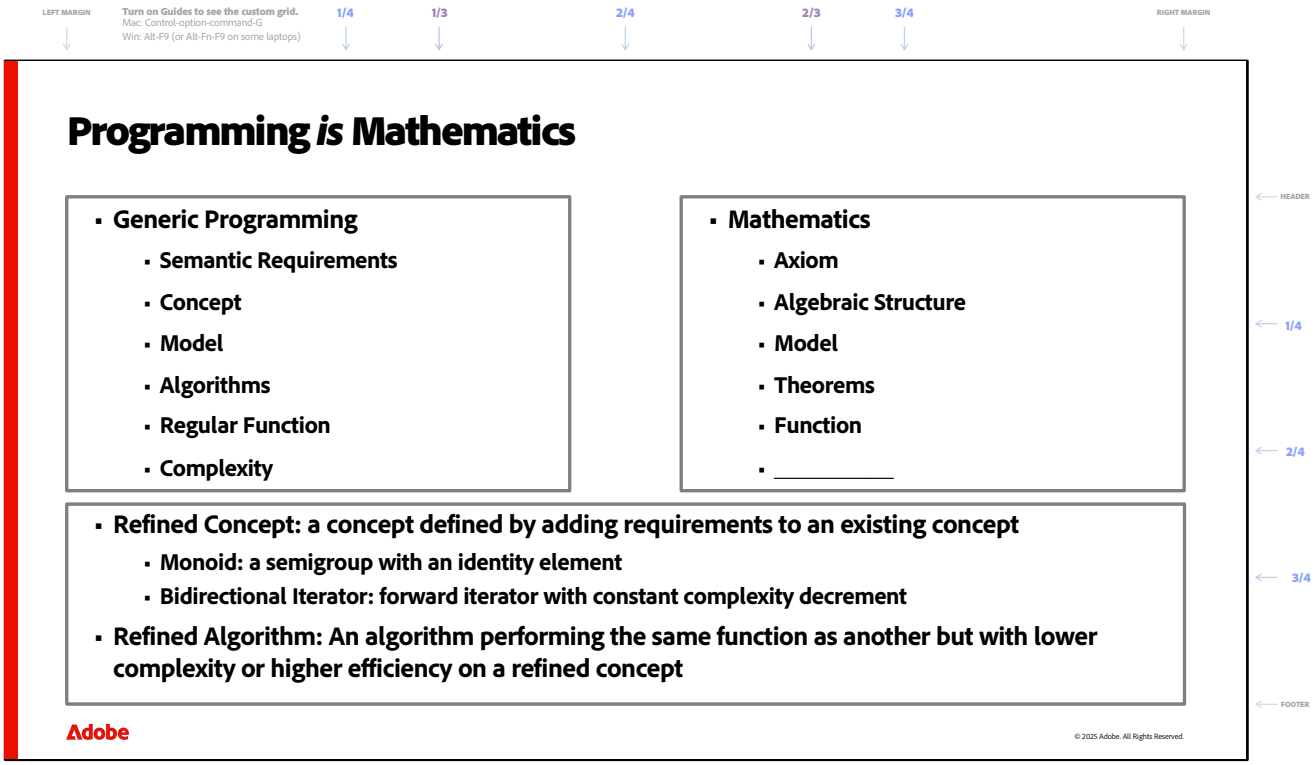
Adobe

Programming with Generics
≠
Generic Programming

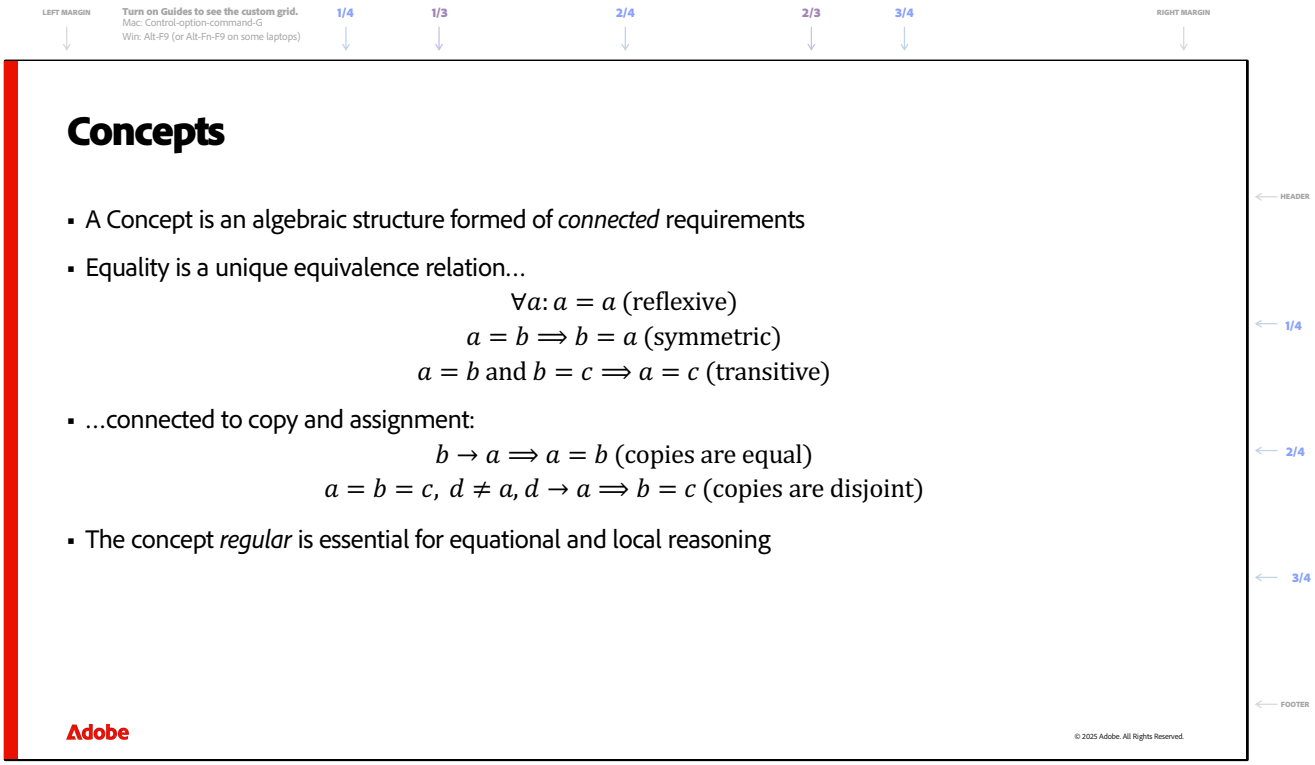
Generic programming has little to nothing to do with generics, or templates, or even traits, or protocols.

Metaprogramming
≠
Generic Programming

And generic programming certainly has nothing to do with metaprogramming. Metaprogramming and templates are mechanisms used to approximate generic programming.



See Alex's books, Elements of Programming and From Mathematics to Generic Programming. Alex Stepanov envisioned a national or international repository of proven, abstract, and efficient algorithms. There have even been some (mostly failed) attempts to start such a repository.

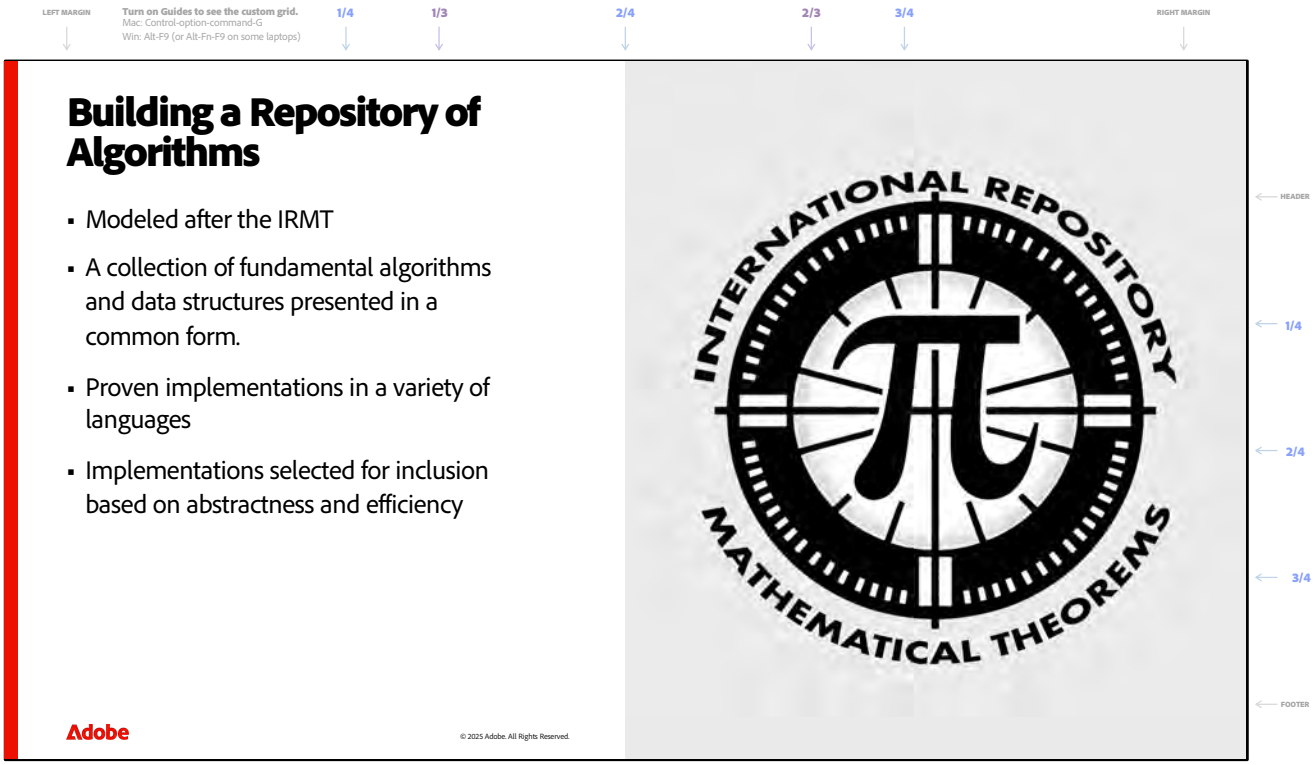


Concepts (not the C++ language mechanism) are central to generic programming. They are the substrate upon which algorithms (theorems) are defined.

The best we can do is require that an equality operation exists and require in the documentation that the semantics are upheld by the developer (or, in limited cases, assert the semantics in the code).

Alex Stepanov envisioned a national or

international repository of proven, abstract, and efficient algorithms. There have even been some (mostly failed) attempts to start such a repository.



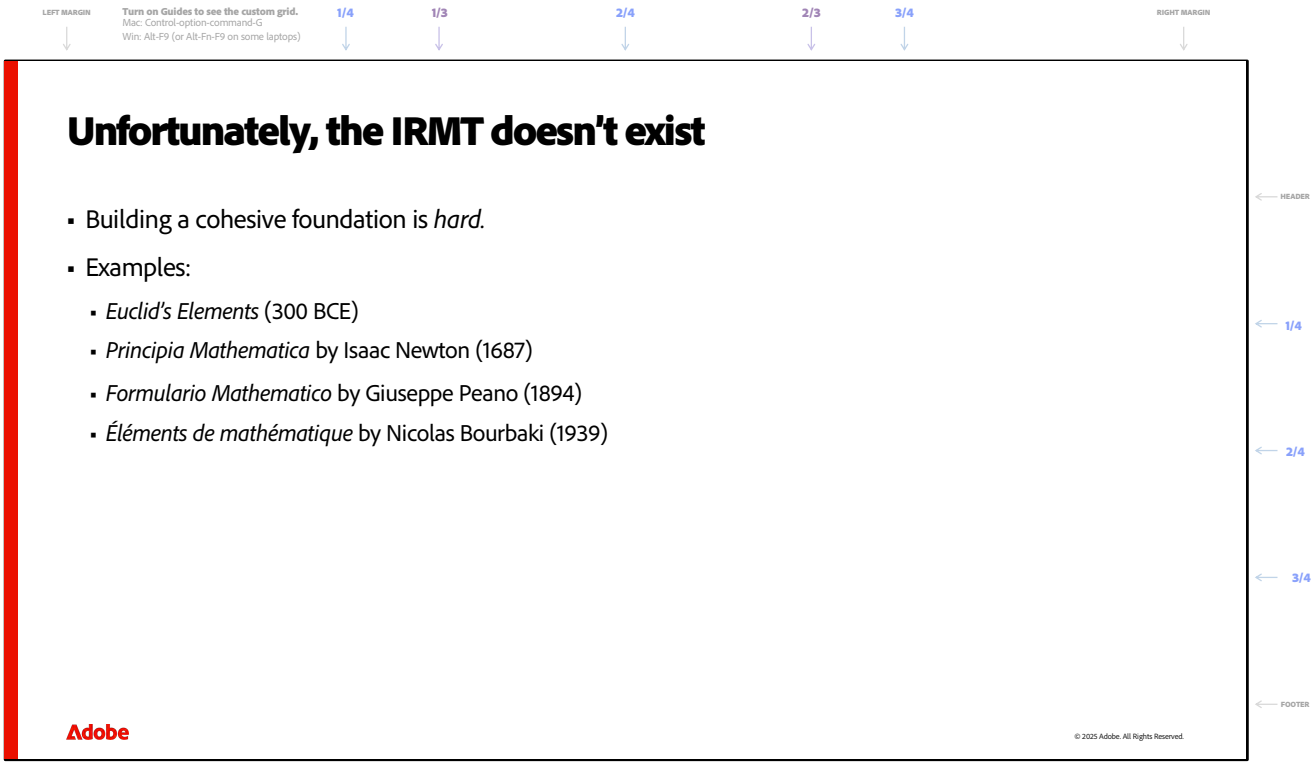
It should be easy to build such a repository – we only have to follow the IRMT model.

Just like the IRMT, it has all the fundamental theorems and formulas in standard form.

Just like the IRMT, it has both manual and automated proofs of all the theorems.

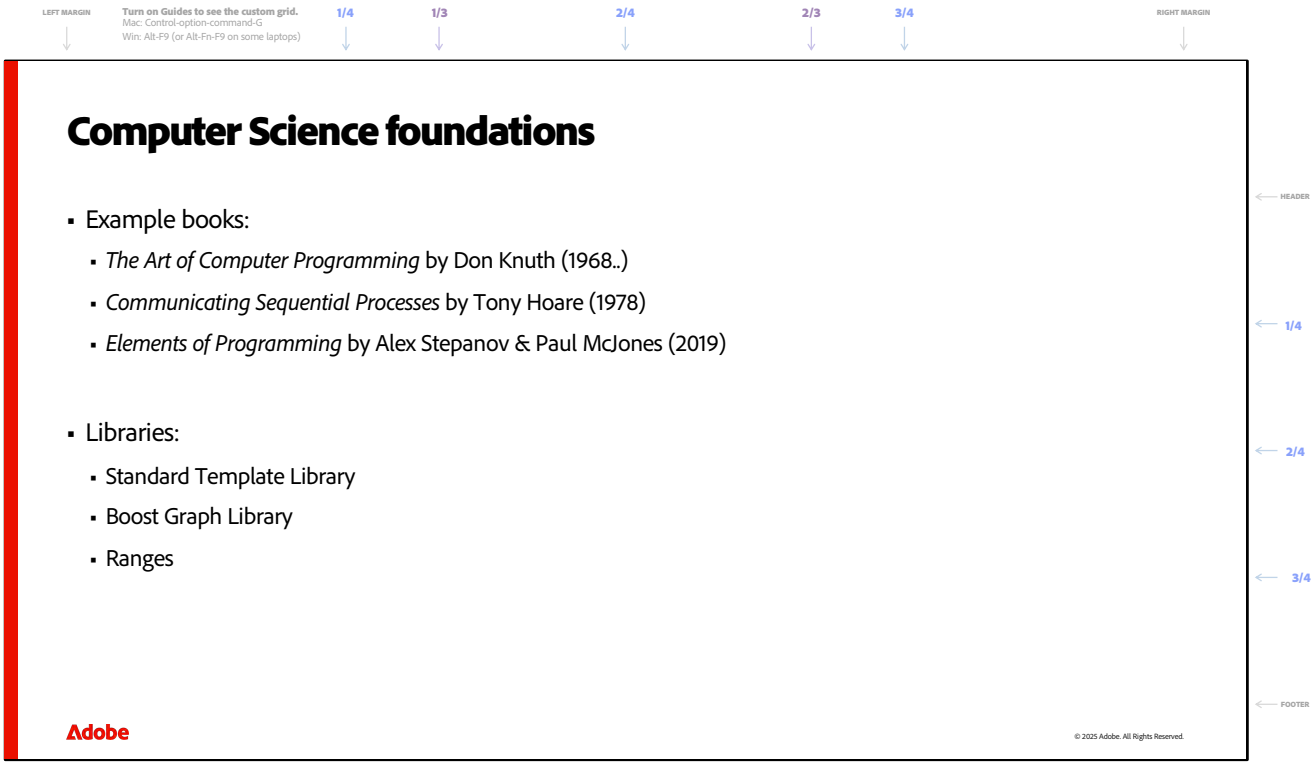
Just like the IRMT, we should select theorems for inclusion based on their general usefulness. These

are the theorems or algorithms upon which most others are based.



We have excellent tools like Wolfram Alpha, and you'll find many attempts online. But nothing that resembles a cohesive collection of theorems.

The closest we have are some of the great math texts. These are not all – but what you find is they amount to major works by single individuals or small groups.

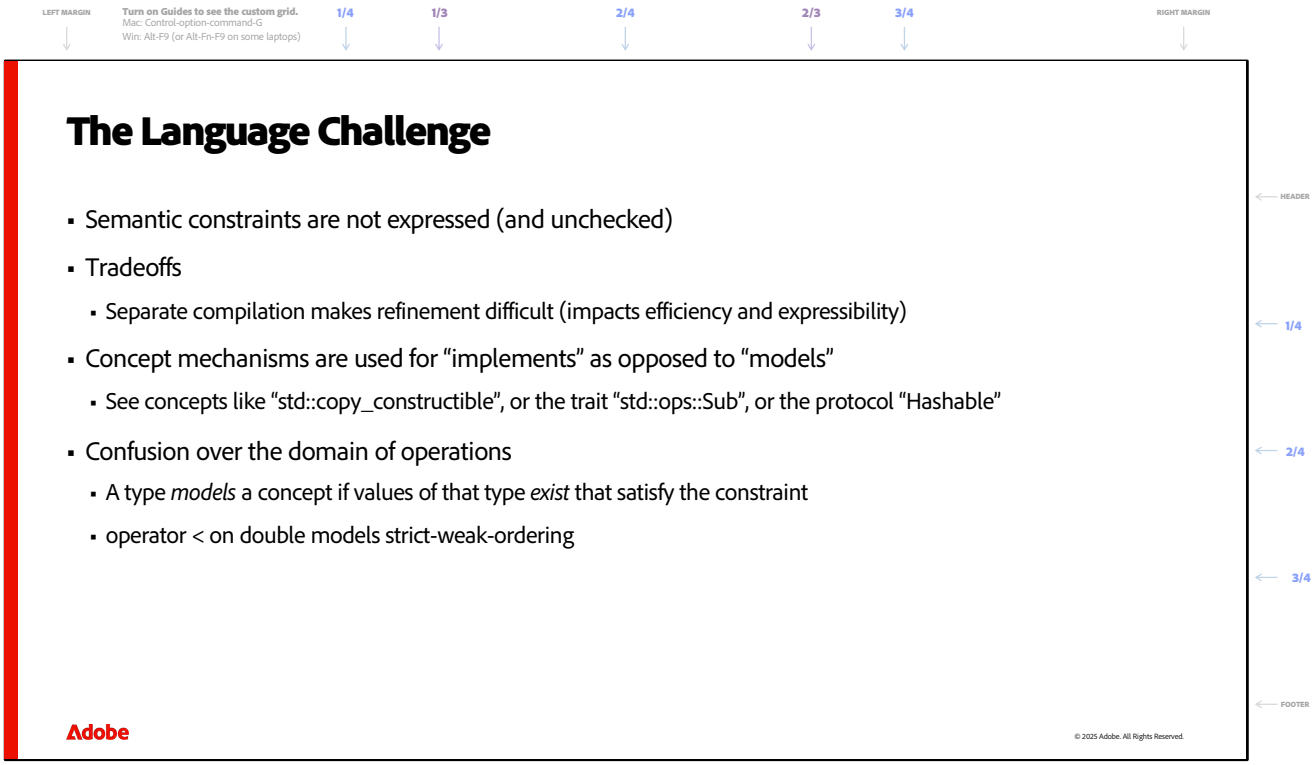


[40:00 – 50:00]

Discovering requirements of algorithms, the concepts they form, and the useful models of those concepts, then organizing those into a cohesive whole that can be built on is _hard_. Packaging those works into usable and proven code, and then maintaining that code in an evolving environment, is a significant effort.

As professional developers, we have a moral

obligation to contribute to standardization, to open source, and to publish. We have faith that someone will come along to collect and distill what proves to be of value into the cohesive whole. The algorithms in these works were not, in large part, the invention of the authors. They collected and categorized, built on the work of others, and sometimes were able to add some additional insight.

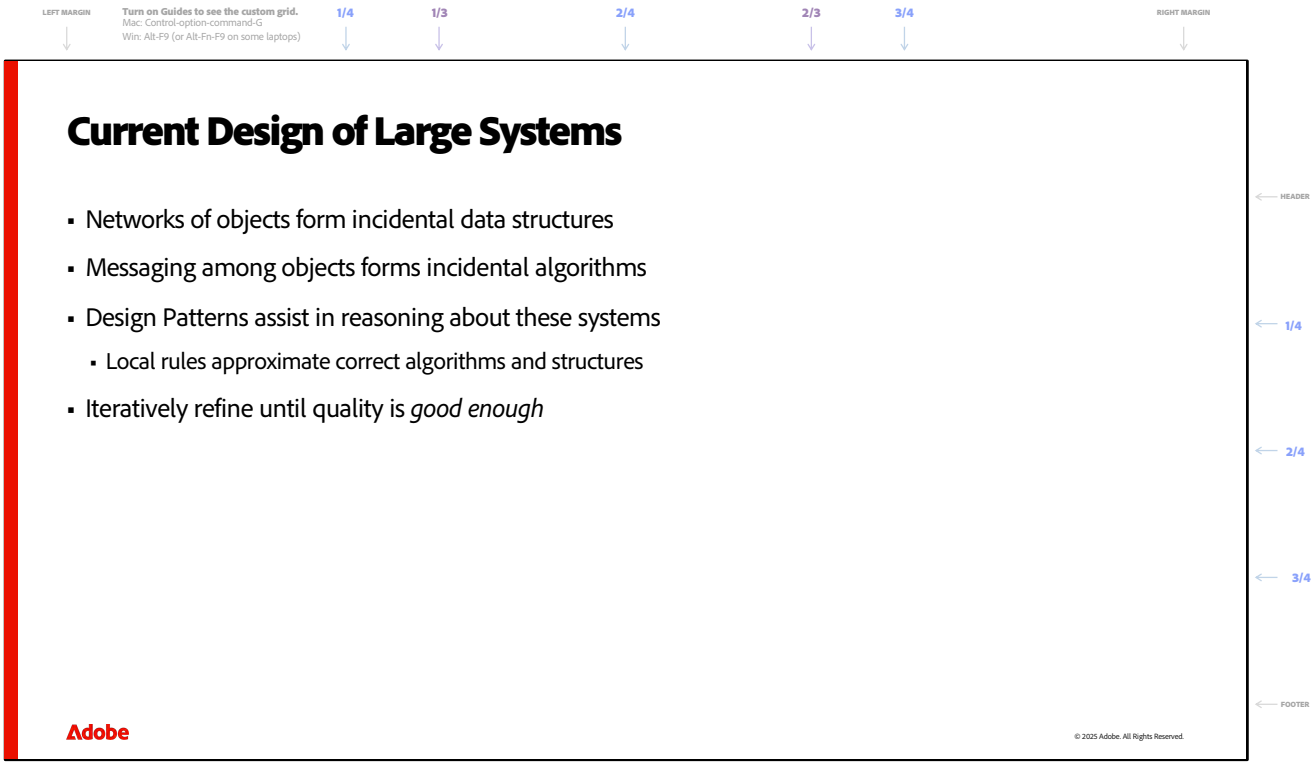


Since we cannot express semantic constraints, we associate semantics with the names of operations. And often overload the meaning of a given name for different types or different contexts.

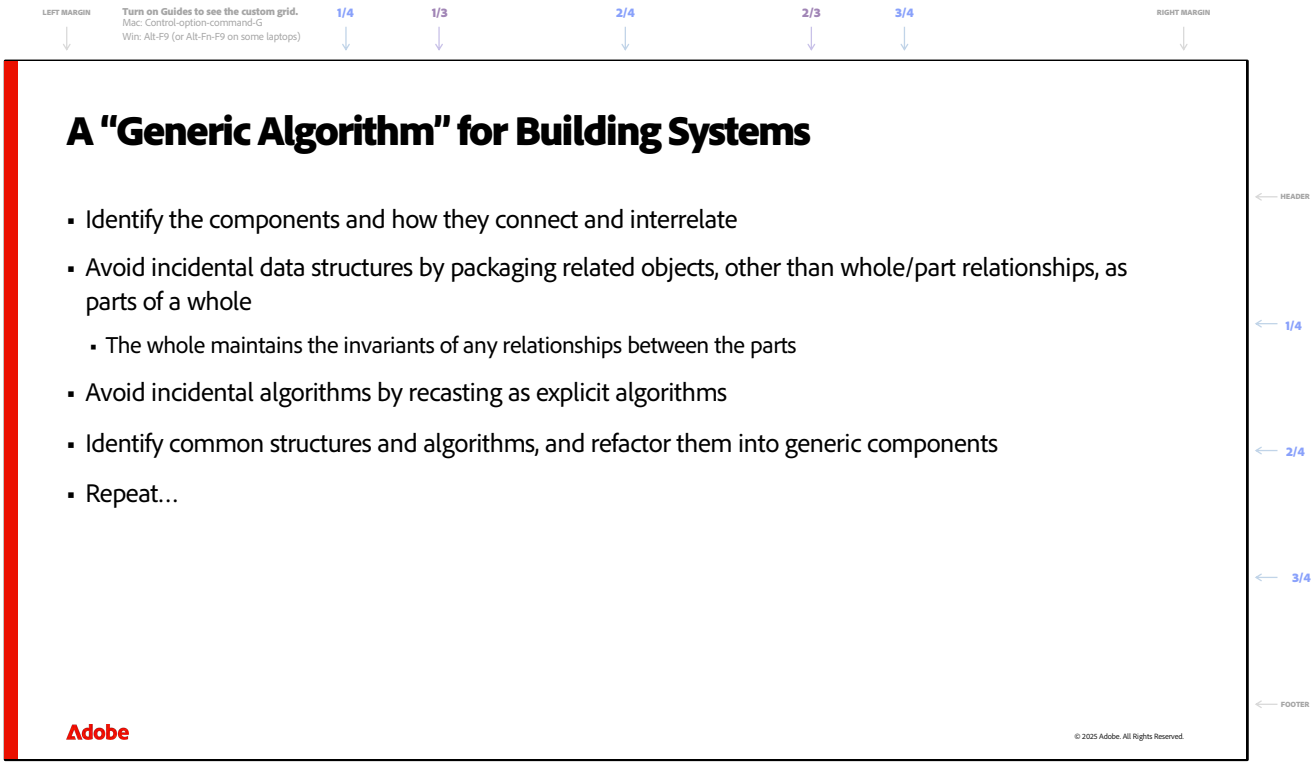
We cannot define the meaning of copy or hashable without equality. Both copy and hashable are inherent properties of types. We cannot define subtraction without addition and an additive identity. Of course maybe we are using minus to mean something other than subtraction, but then

the trait “std::ops::Sub” has no meaning other than “implements minus”.

A type may model a concept even if some of the operations for the concept are not implemented. It follows from the definition of a type, the representation of a set of values in memory, that all types are *regular*. Whether or not the operations on a regular type are implemented or implementable.



This is a slide from A Possible Future. Although I think the landscape has somewhat improved, especially in the web space (there is also a lot of garbage in the web space), unfortunately this is still the dominant way large systems are built.



An incidental data structure is formed by relationships outside of a containing class (usually due to shared-pointer or other forms of reference semantics). Pointers should not appear in class interfaces.

An incidental algorithm could be a raw loop or an algorithm constructed by messages sent through a structure – separate forms of iteration from the structure.

Why repeat? New relationships will happen. Code is promiscuous.

If you want to be a 10x developer, build libraries of reusable components. Consider STL - the work of Alex and Meng and they impact – they are 1Mx developers. But even at much smaller scale a good library can have significant impact on your company and, perhaps on the industry.

Question: Is generic programming sufficient to build software at scale?

Is this enough?

Conjecture: All problems of scale become a network problem

A “network problem” is the problem of establishing and enforcing a set of local rules between related components that guarantee a global behavior.
At scale, the relationships between parts of a complex system become increasingly difficult to reason about.

Dave once explained to be that bigger is different.

Scale may mean the number of components or the

number of relationships.
The number of possible computation paths through related elements is bounded by `choose` and grows exponentially.

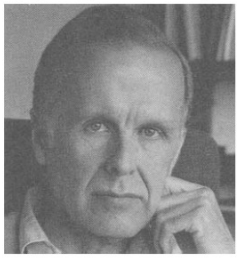
Declarative Forms

This brings us to what I call declarative forms.

Declarative Forms

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

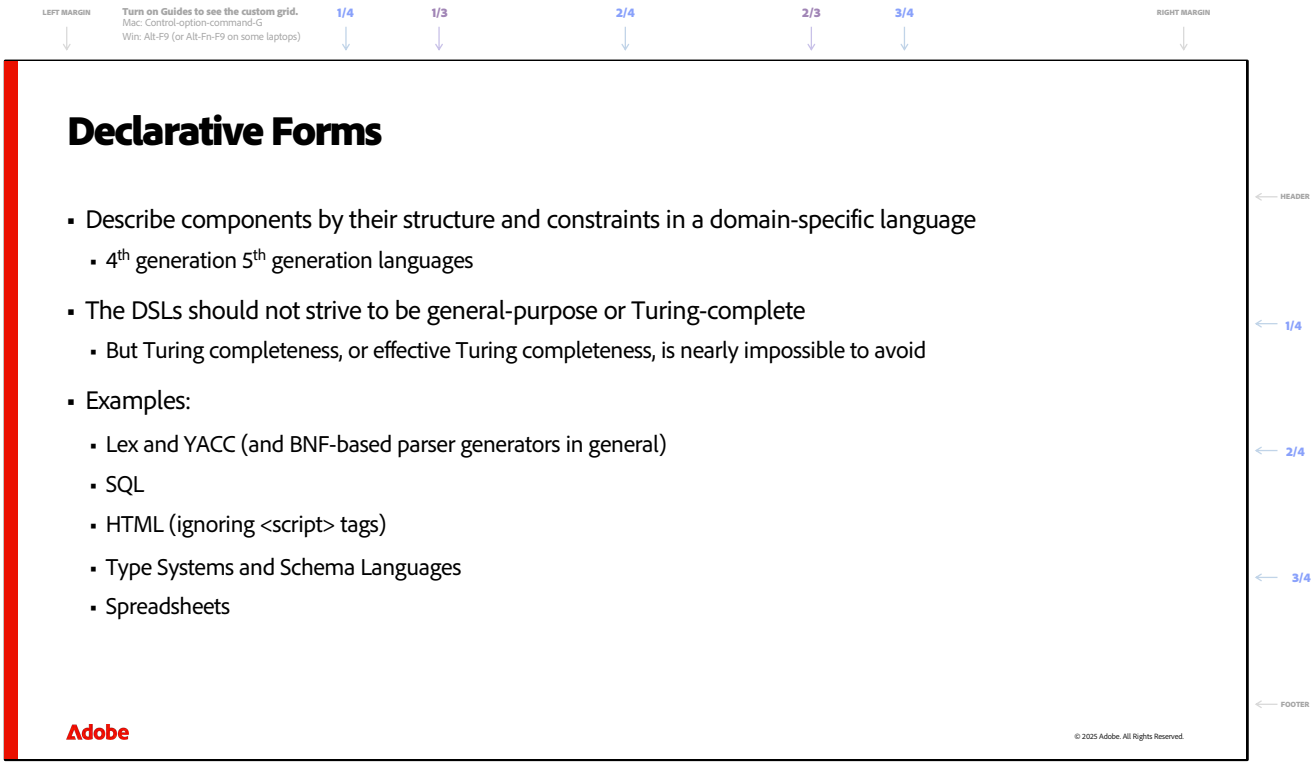
An alternative functional style of programming is

The idea of a “declarative form” is rooted in what John Backus referred to as “functional or combining forms” of which he identified 5, very simple ones:

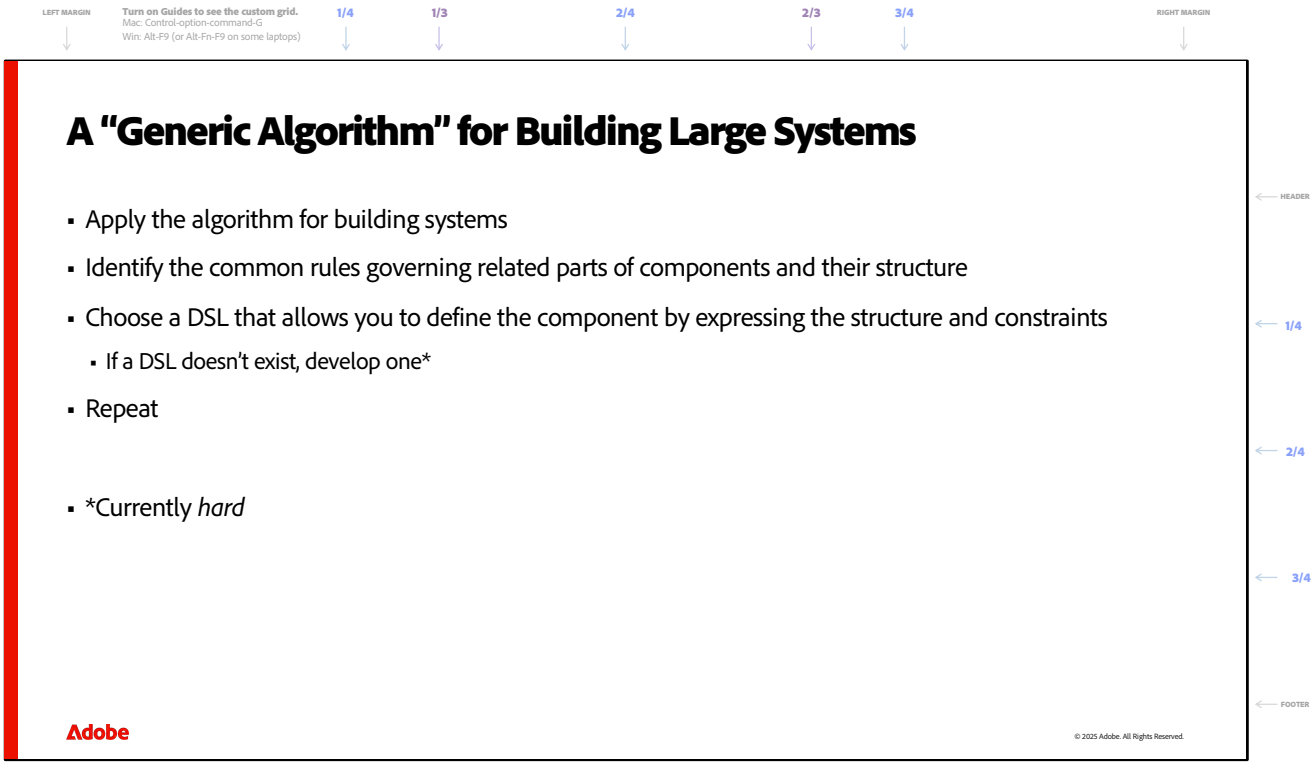
1. Composition
2. Construction
3. Conditioning
4. Iteration
5. Recursion

Alex Stepanov recognized these as theories and realized there are not few, but infinitely many.

Similarly, declarative forms are theories. They are patterns of structures and constraints that appear in many contexts. A few are fundamental (I believe a number of those are still undiscovered).



4th generation languages are domain-specific languages; 5th generation languages are languages where you operate by specifying the problem, not the algorithms to solve it. You specify the "what", not the "how". Prolog is the canonical 5th-generation language. However, general-purpose 5th-generation languages are difficult to use effectively.



[60:00 – 30:00]

We can extend our previous process to large system.

Rules Exist

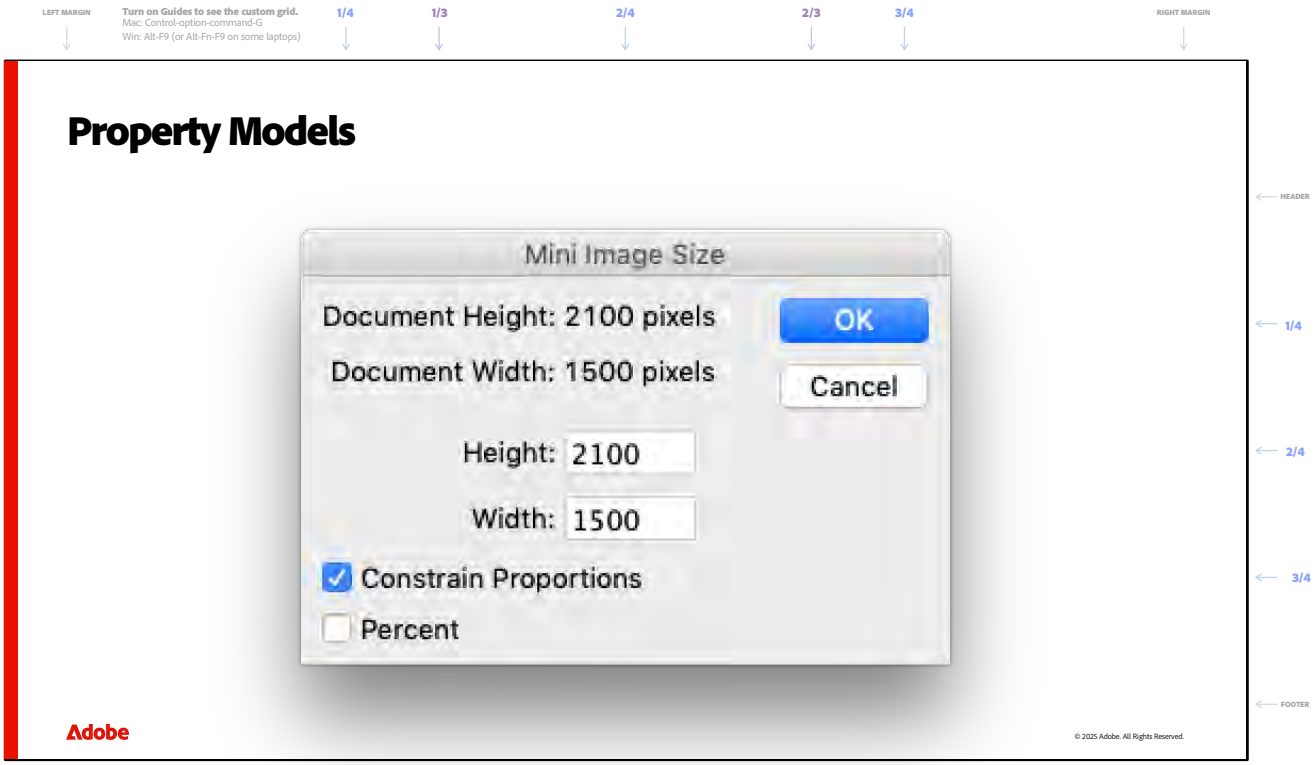
This algorithm works because rules exist.

Software development is a process of discovery, not invention.

The laws of mathematics and physics govern our systems. If a problem and solution appear unique, required for a specific case, they are probably wrong. Writing a “new algorithm”, or a “new structure”, or defining a “new concept” should set off red-flags. It is probably time to do some

research and read some papers.

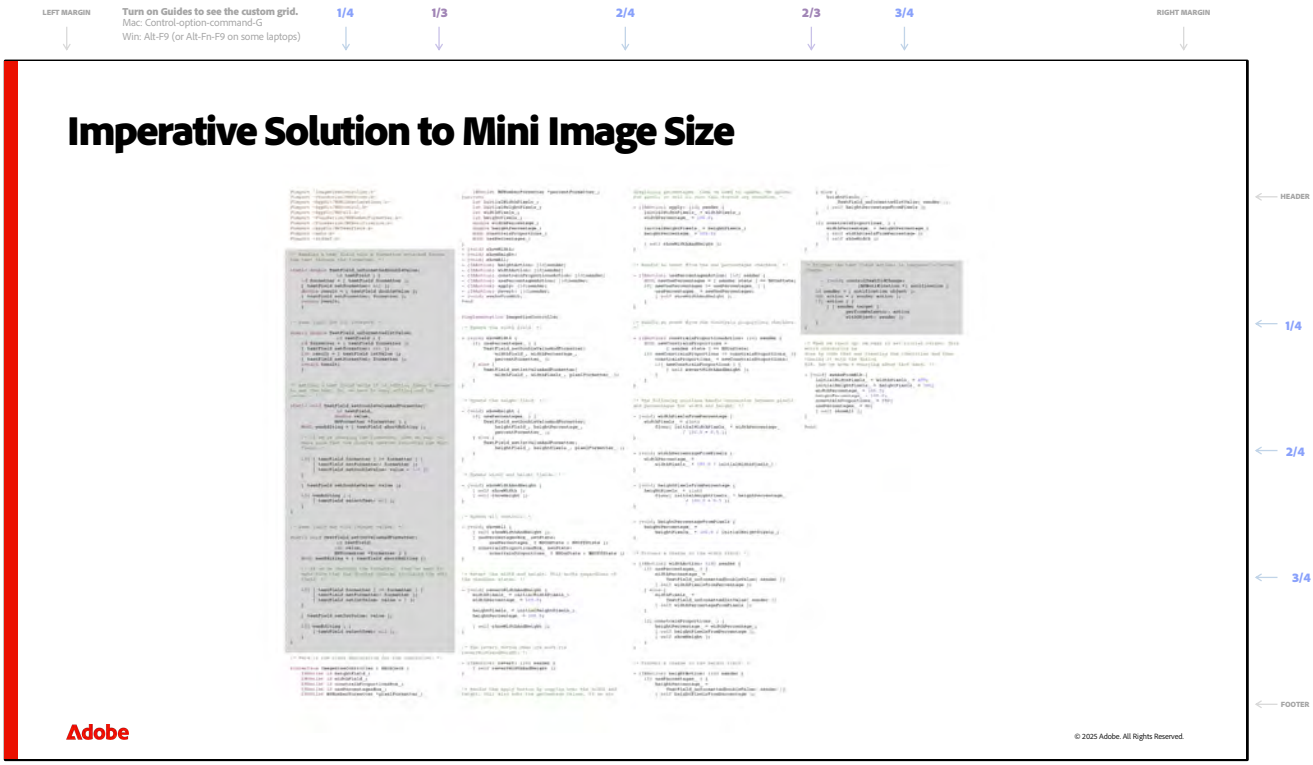
Engineering is about finding the best solution given a set of constraints, of which the laws are always a part of the constraint set. But we also add time to develop, the number of developers, the skills of the developers, the platform(s) upon which the solution must run, the required performance and features, etc…



Let’s take a look at an example. This is a greatly simplified version of a dialog that lets the user resize an image. They can specify the width or height in pixels or percent and toggle constraining proportions—pretty simple stuff.

I asked Mark Hamburg to code this dialog. Mark was the Photoshop lead in the early days and is the creator of Lightroom. He’s a brilliant engineer and Adobe Fellow.

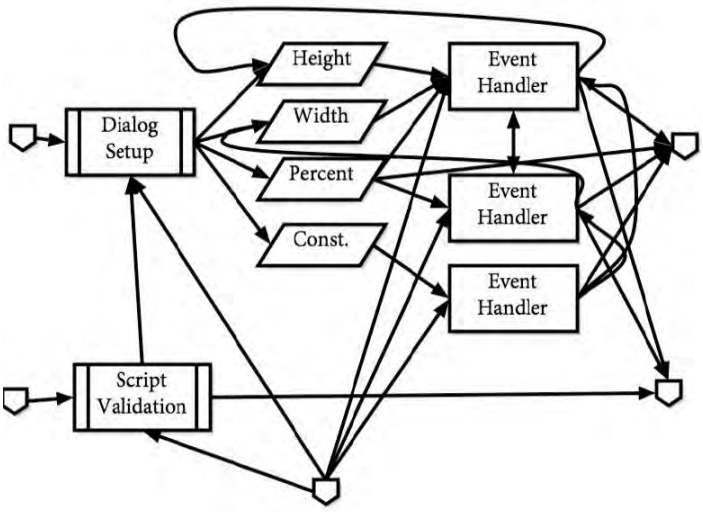
He could use the framework of his choice, and I wasn’t interested in the appearance, but the behavior.



This code is Obj-C using Apple's AppKit. This is _just_ the code from what Apple calls the "controller" logic. i.e., the behavior. The grey area loosely represents the code to enforce the constraints between the values in this dialog. The rest of the code is event-handling logic to drive the constraints.

Similar code is about 30% of the Adobe (client-side) codebases.

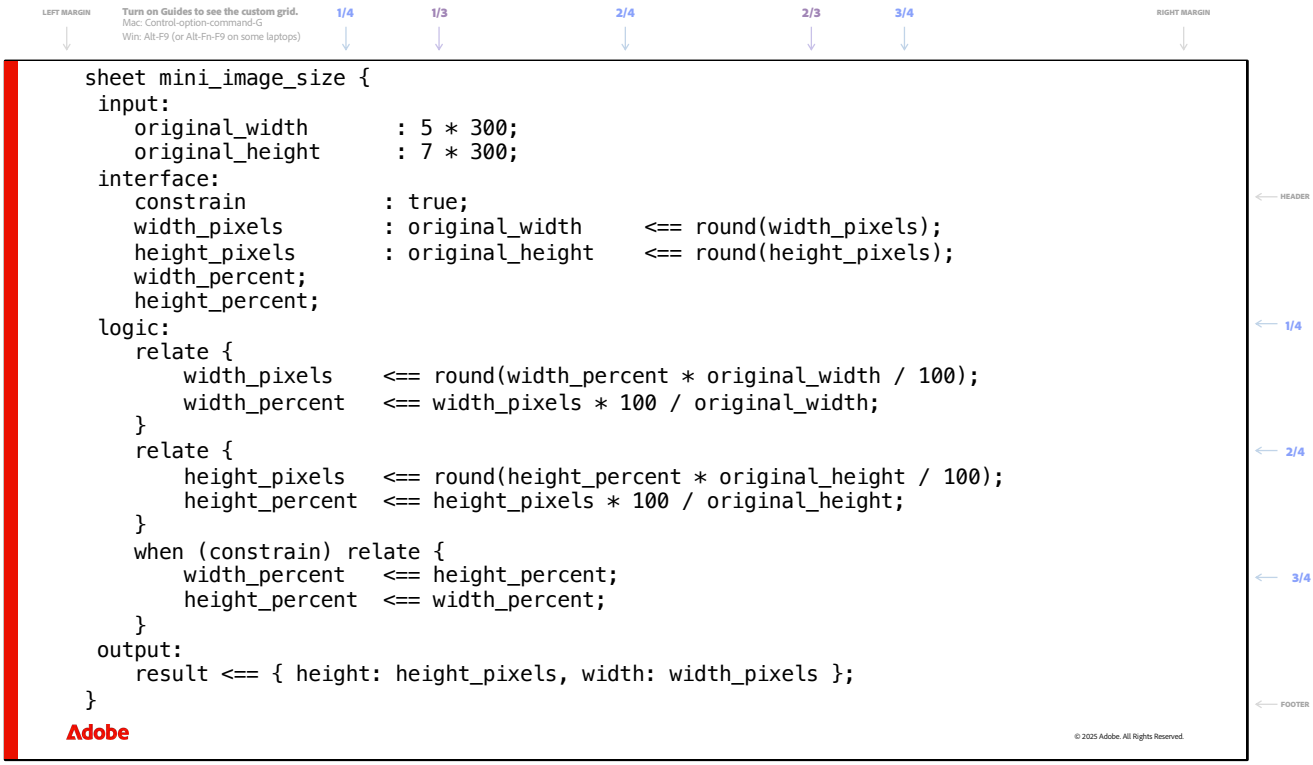
Event Flow



If you diagram the event flow, it would look something like this – this includes handling script playback, which I don't believe Mark implemented.

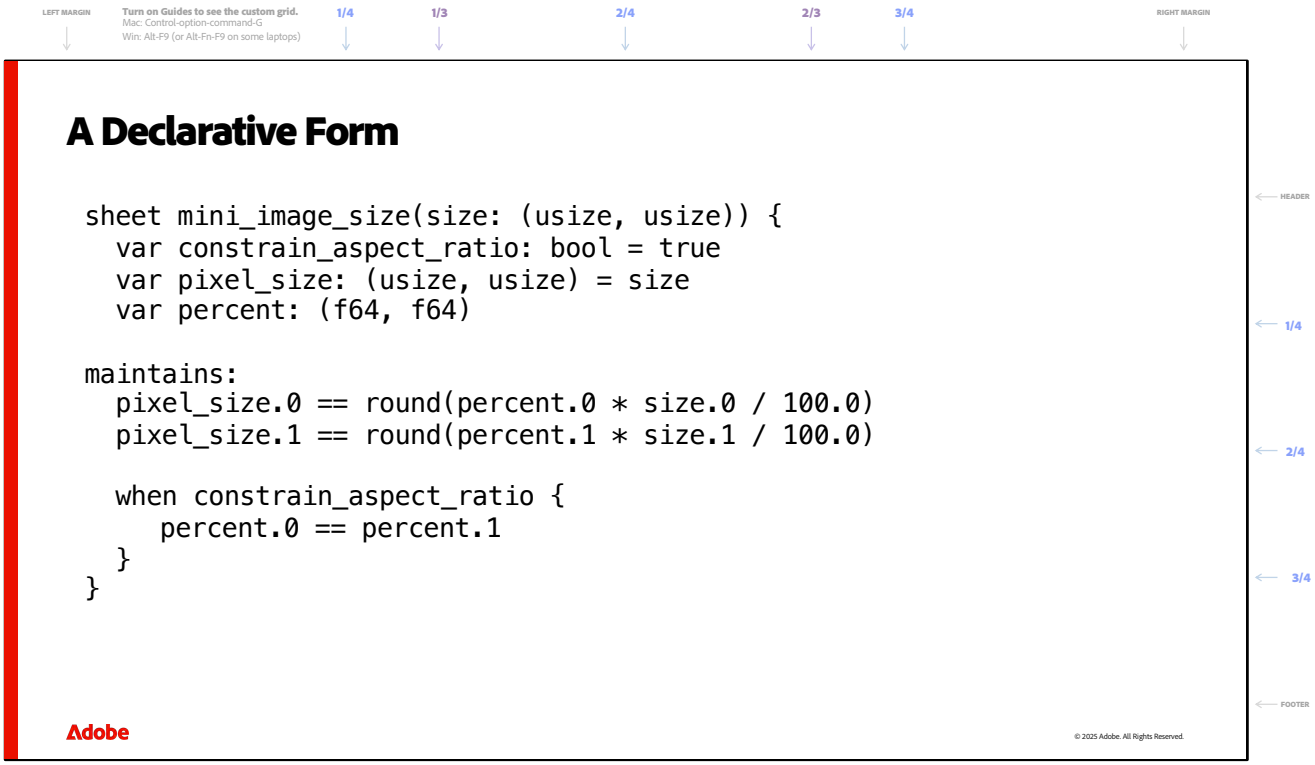
Demo

[65:00 – 20:00
Demo mini-image-size and also full image-size



[70:00 – 15:00]

This is the property model description for the mini-image size problem. Each `relate` clause is a constraint with a set of constraint satisfaction functions. The first two are two-way multiplicative relationships (the original value is pinned) and the last one is a conditional implication. Property models support any n-to-m relationship.



This is where we would like to get to, at least in a self-contained form. We want sheets to compose; dependent unit conversion is a common building block.

This is a single dependent unit conversion plus a single optional constraint. That's it.

About 85% of the code could be replaced by small declarative descriptions. Combined with generic programming, there is a potential two orders of

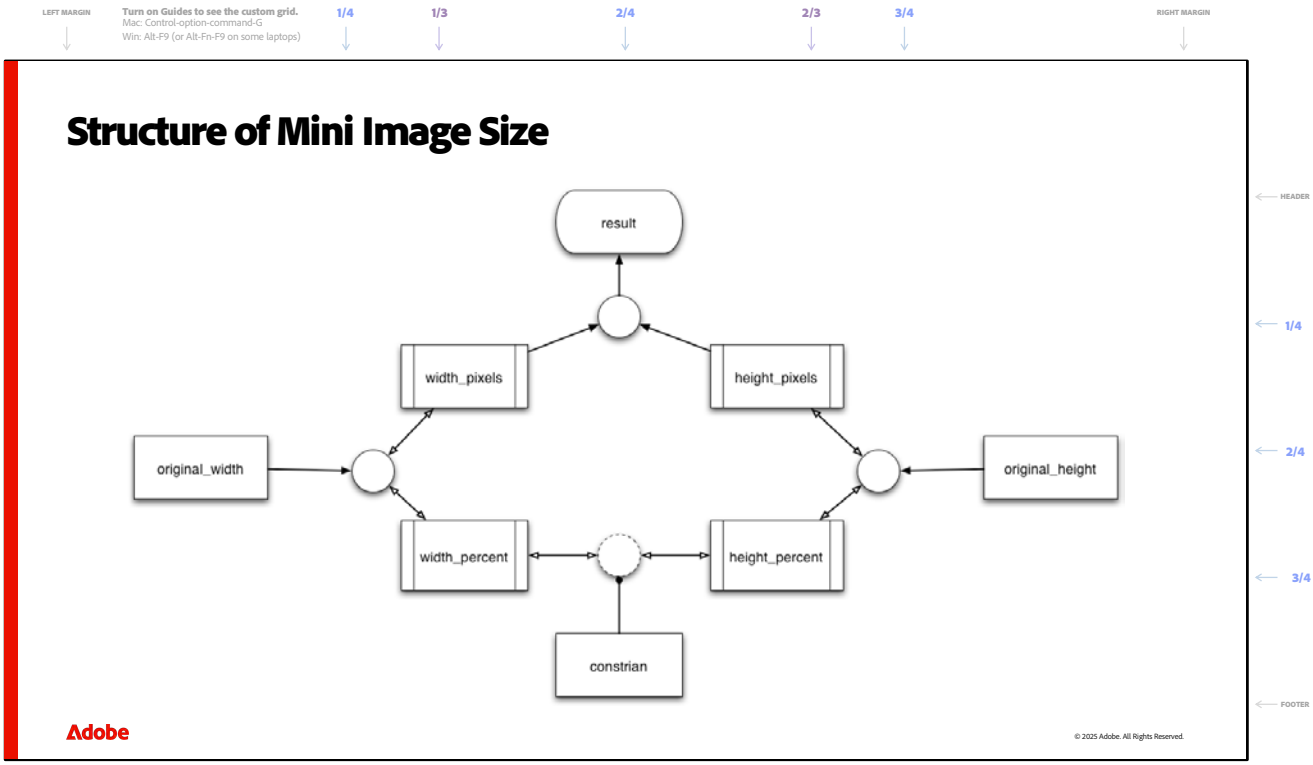
magnitude reduction in the code needed to describe our products and we would see a greater than two orders of magnitude reduction in defects.

[If a question arises about divide by zero…] If `size` is 0, `percent` becomes a “don’t care” and the control is disabled, showing the prior (empty) value. This secondary pass has not been implemented in the current work.

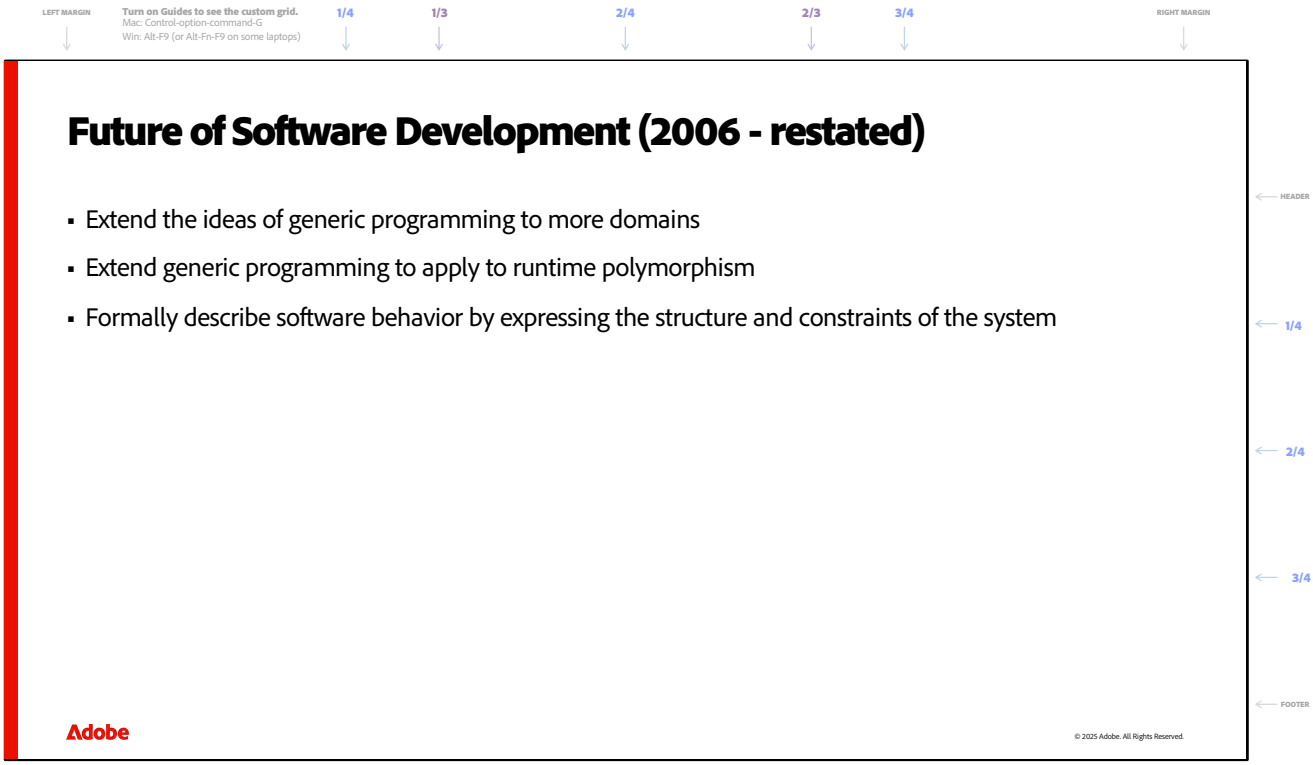
Which do you think is more likely to contain a bug…

Imperative Solution to Mini Image Size

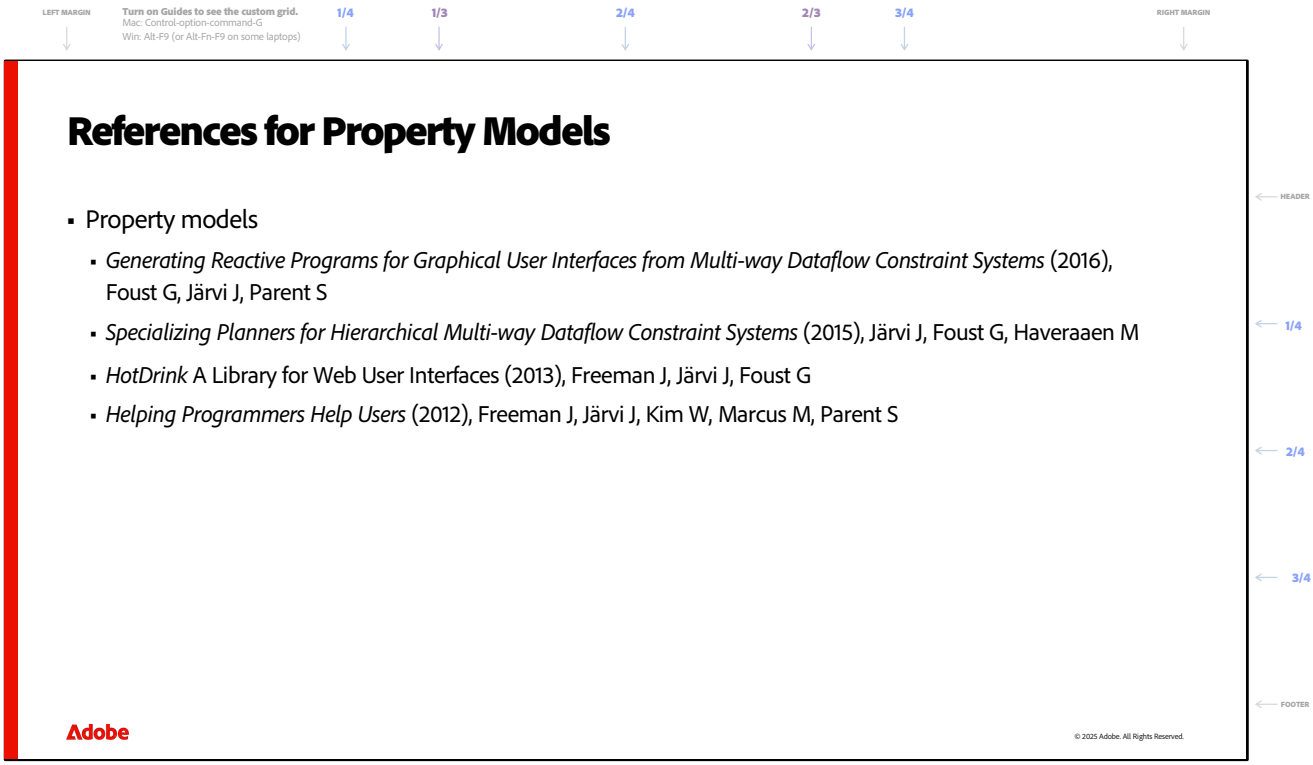
Adobe



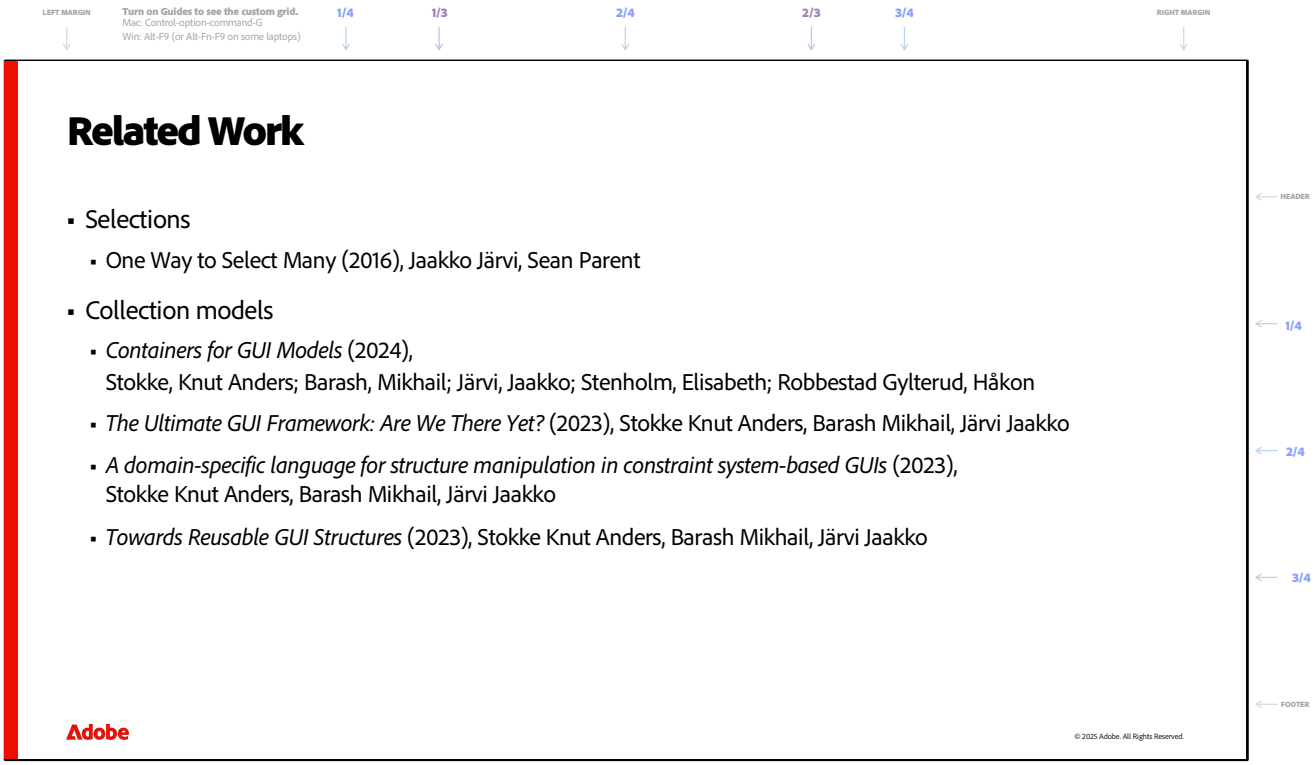
This is how the constraints are structured. I mentioned disabling controls in the demo. The rules for when a value-input in the UI is disabled is when it is a “don’t care” (it doesn’t contribute to the result under the current constraints) or when it’s value is implied and no contra-positive is expressed. All the information governing the UI behavior is in the relationships.



This was my final slide in the old talk.
There has been significant progress on the first two points
There has been some progress on the latter point, especially in web front-end development



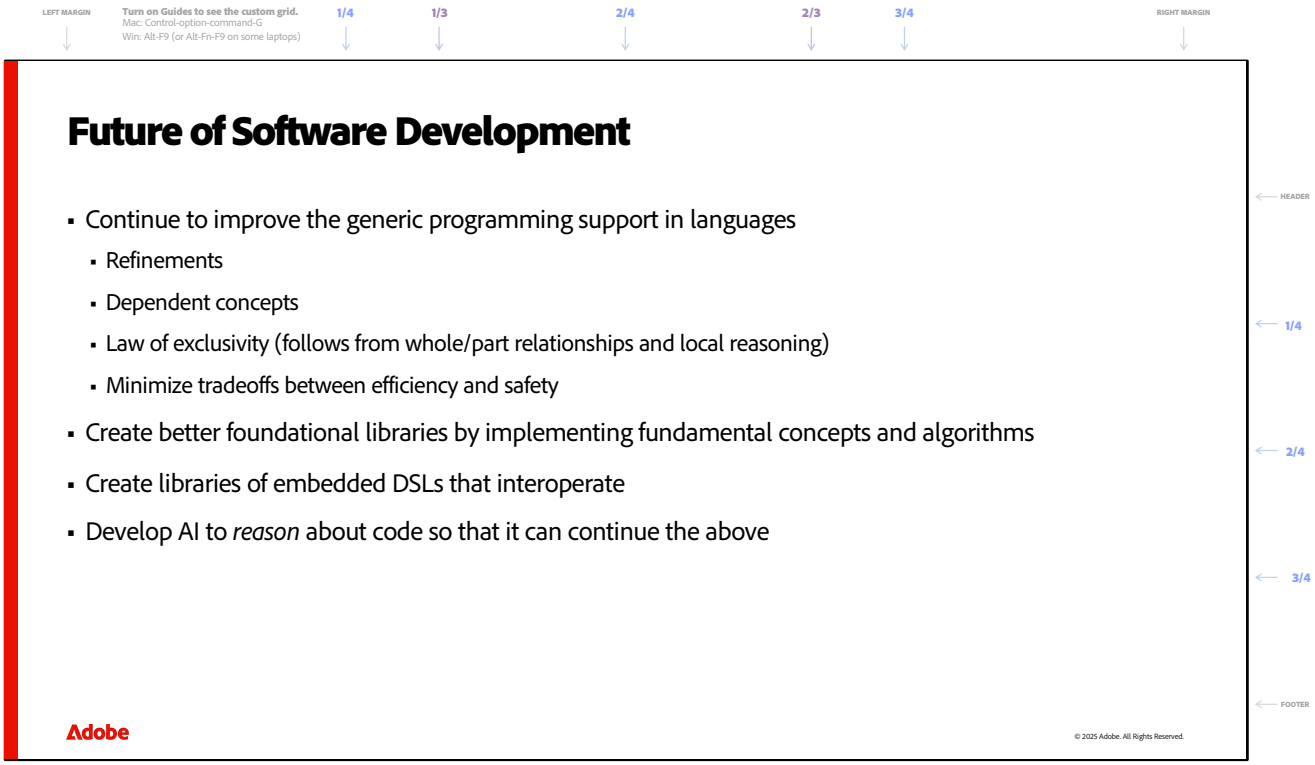
Although I haven't been working directly on property models some work has continued. Jaakko Järvi and I published some additional papers and he and his students pushed the tech forward developing additional solvers in JavaScript and Rust.



In 2016 Jaakko spent his sabbatical at Adobe, our plan was to work on extending the ideas of property models to collections. We get stuck on how to describe how to create a selection within a collection, and ended up developing a calculus for this and writing a paper. If you do anything with user interfaces it is worth reading.

Recently Jaakko has picked up the collection model work again and has published several papers. I didn’t know one of his recent ones is titled “are we

there yet?” before preparing this talk!



Foundational libraries should be proven correct. Interoperate with the imperative language and with each other

Demis Hassabis, the head of Google's DeepMind, estimates we will reach AGI in about 5 years. My back-of-the-envelope projection says we will match human intelligence capabilities (number of neurons, speed, power consumption) in 20-40 years. We

don’t yet know if intelligence scales or how to ensure our intelligent systems are “sane”. I wouldn’t worry about AI putting you out of a job; I would worry about AI putting *everyone* out of a job.

AI won’t code in any existing language when it takes our jobs. A reasoning AI will develop a better language that is likely not human-readable.

But if all you do is repeatedly rewrite some approximation of correct code, AI will take your job soon.

On that upbeat note – I hope you will all do your part to help create the future.

Closer Slide

[80:00 – 10:00]
Questions?