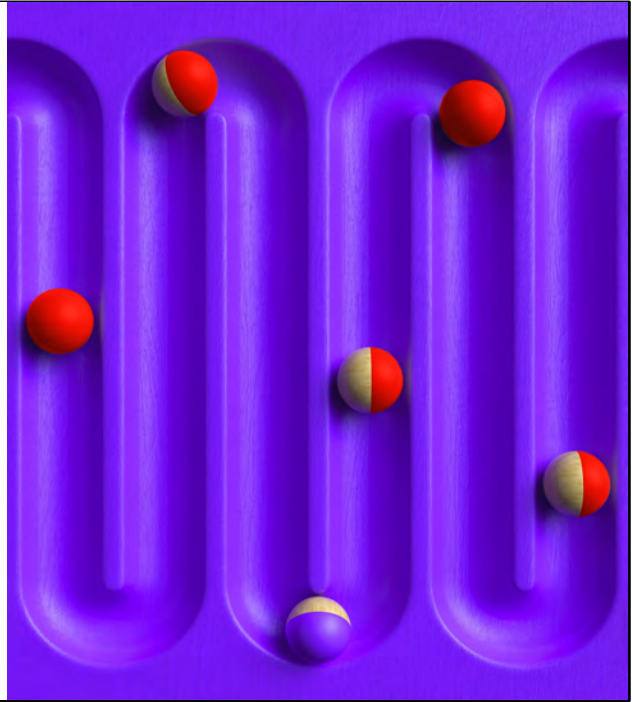


Are We There Yet?

Sean Parent | Sr. Principal Scientist
Software Technology Lab

Adobe

Image by Bruno Tornelli



I'm Sean Parent, Sr. Principal Scientist at Adobe and a member of Adobe's Software Technology Lab. I have the best job, plotting a course to improve software development practice in the industry.

In 2006, Bjarne invited me to speak at a Texas A&M industry affiliate event. I gave a talk titled "A Possible Future of Software Development." I later presented the talk as a keynote at the first BoostCon (now C++Now) in 2007, and gave it 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 attended that talk that it influenced the development of the Elm language, Prezi (the presentation software), and the UI software for Tesla and WebOS (originally developed for Palm, then HP, and later used in LG TVs).

Dave Abrahms suggested I give the talk this year, since 18 years later, few have heard it (this was long before "That's a Rotate" became a thing). How many here saw the original or the YouTube video?

This talk examines the changes that have occurred 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.

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

Adobe

2006

2025

Employees

6,000

30,000

Revenue

\$2.6B

\$22.0B

Adobe

© 2025 Adobe. All Rights Reserved.

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

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

Ps Photoshop

	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)

Adobe © 2025 Adobe. All Rights Reserved.

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

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

← HEADER

← 1/4

← 2/4

← 3/4

← FOOTER

“Organizations need to integrate security best practices, security testing tools and security-focused processes into their software development life cycle. Proper execution improves application security, reduces overall costs, increases customer satisfaction and yields a more-efficient SDLC.”

— Gartner Research, February 2006

Adobe

© 2025 Adobe. All Rights Reserved.

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 month for enterprise customers. Security, both on the client and server sides, remains a hot topic today.

Large Quote

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

← HEADER

← 1/4

← 2/4

← 3/4

← FOOTER

Adobe

© 2025 Adobe. All Rights Reserved.

“Microsoft has been slowly moving to a new development process that will affect how partners and customers evaluate and test its software... The new process should help Microsoft gain more feedback earlier in the development cycle, but it won't necessarily help the company ship its products on time or with fewer bugs.”

– Directions on Microsoft, March 2006

[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

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

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

© 2025 Adobe. All Rights Reserved.

← HEADER

← 1/4

← 2/4

← 3/4

← FOOTER

This is a quote from Jon Bentley that summarizes why our (then?) approach to software development was not scalable.

This quote led to the Photoshop team using “the binary search problem” as a take-home test for candidates. It remains a strong indicator of success.

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

← HEADER

← 1/4

← 2/4

← 3/4

← FOOTER

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

© 2025 Adobe. All Rights Reserved.

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

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

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

← HEADER

← 1/4

← 2/4

← 3/4

← FOOTER

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

© 2025 Adobe. All Rights Reserved.

This would be a better solution to the problem.

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

Question: If we can't write binary search...

- Jon Bentley's solution is considerably more complicated (and slower)
- Photoshop uses this problem as a take-home test for candidates
 - More than 90% of candidates fail
- Our experience teaching algorithms would indicate that more than 90% of engineers, regardless of experience, cannot write this simple code
- ...then how is it possible that Photoshop, Acrobat, and Microsoft Word exist?

Adobe

© 2025 Adobe. All Rights Reserved.

This slide is from my original talk and remains 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

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
↓

← HEADER

← 1/4

← 2/4

← 3/4

← FOOTER

“Almost all developers will increase their use of generative AI and machine learning as code assistants and low-code/no-code tools proliferate.”

– IDC Modern Software Development Survey, September 2024

Adobe

© 2025 Adobe. All Rights Reserved.

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

Large Quote

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
↓

←— HEADER

←— 1/4

←— 2/4

←— 3/4

←— FOOTER

“More than **55% of developers** automatically generate **40% or more of their code** in their IDEs, highlighting the widespread use of generative AI and machine learning in software development.”

— IDC Modern Software Development Survey, September 2024

Adobe

© 2025 Adobe. All Rights Reserved.

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

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

← HEADER

Binary Search Solution (Copilot 2025)

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

← 1/4

← 2/4

← 3/4

- "Algorithm Naming: The function is essentially an adaptation of `std::lower_bound()`."

← FOOTER

Adobe

© 2025 Adobe. All Rights Reserved.

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 improved surprisingly quickly.

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 essentially `std::lower_bound()` – I consider calling

`std::lower_bound()` an acceptable solution to the binary search problem.

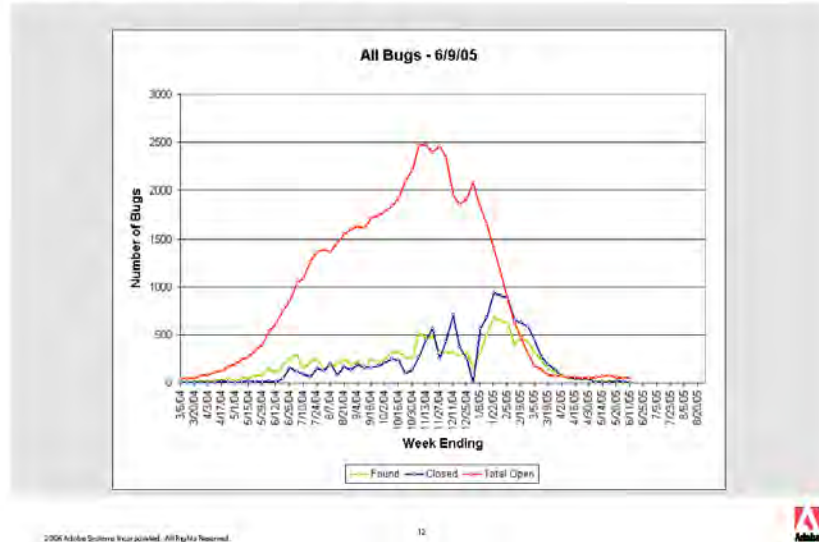
Closer Slide

The Adobe logo is centered on a solid red rectangular background. The word "Adobe" is written in a white, bold, sans-serif typeface.

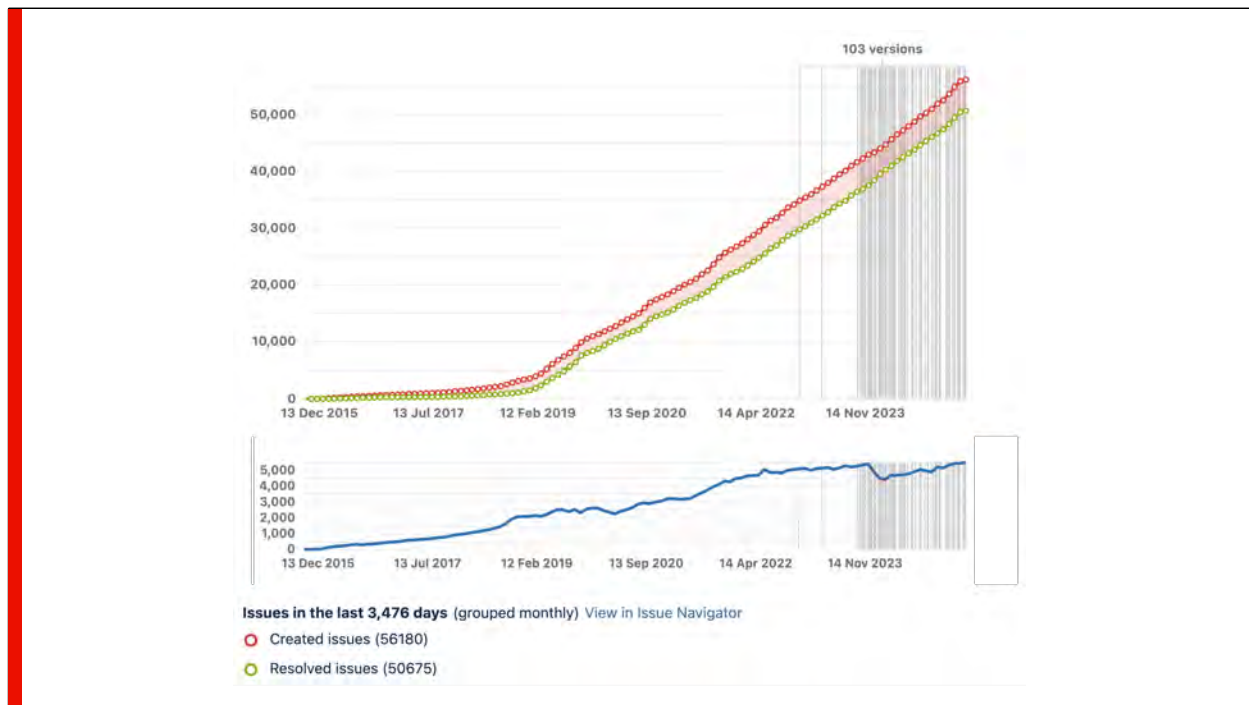
[20:00 – 70:00]

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.

Bugs During Product Cycle



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.

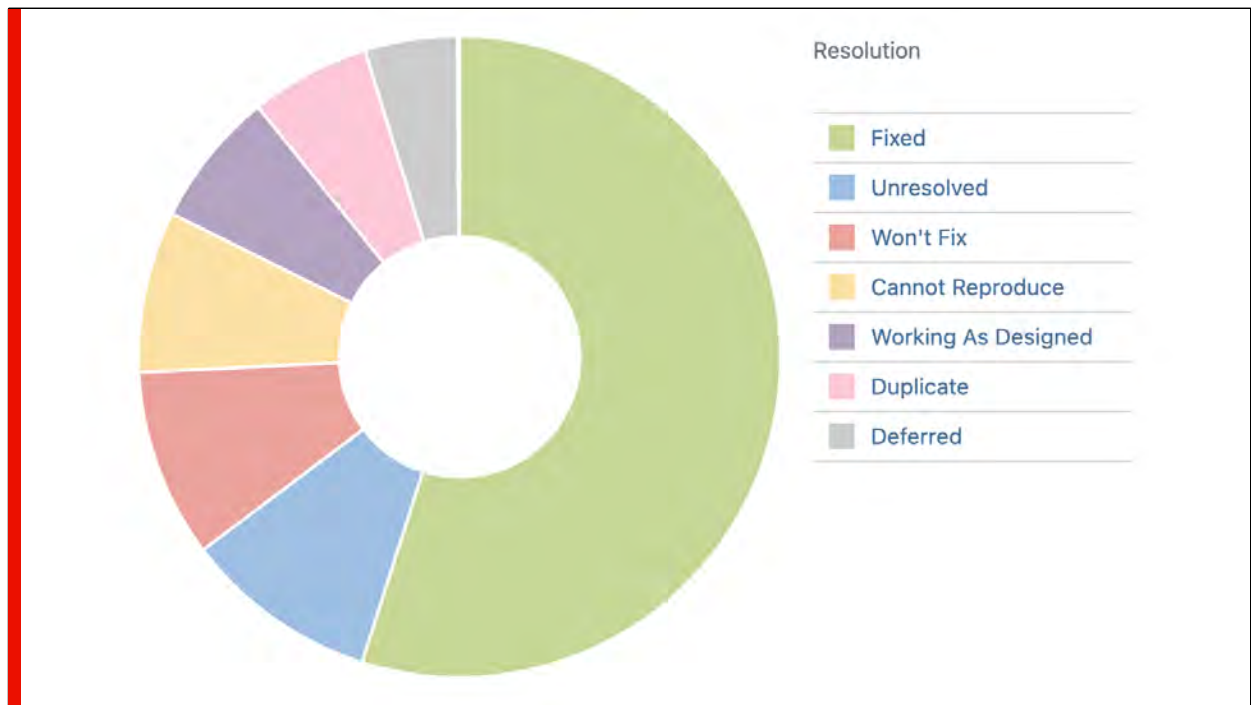
This is a graph of issues (bugs) for a product for the last 10 years.

The top red line is the running total of issues created

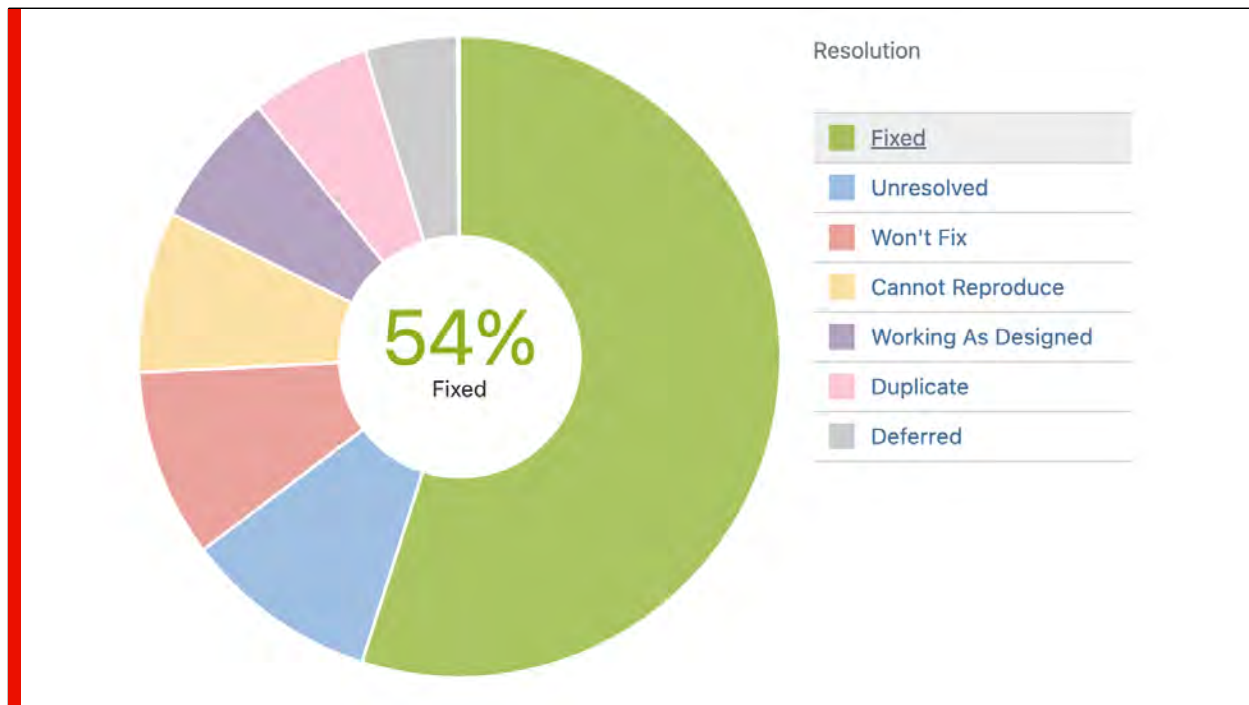
The green line is the running total of issues resolved. The version lines are a new field, but they give a sense of the release cadence in later releases.

The 2019 bend was a significant scale-up of the team as the product evolved into an “ecosystem” and expanded to more platforms, including mobile, web, and server-side.

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.

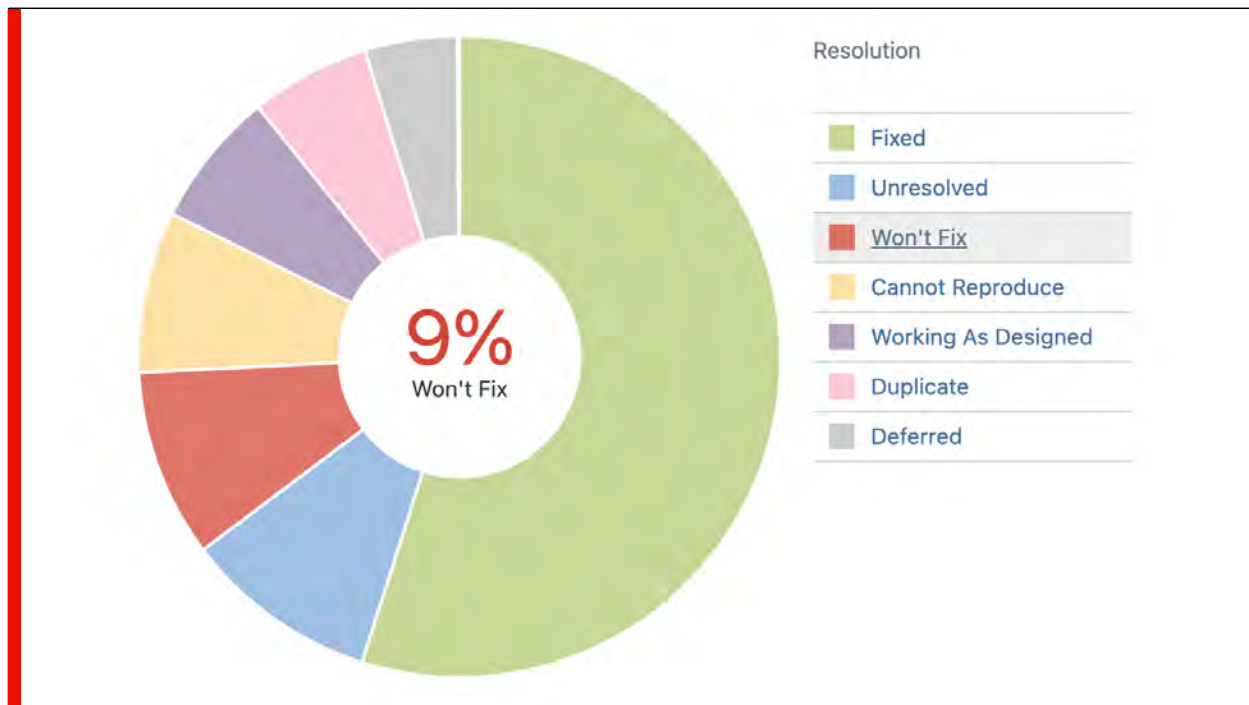


Here is the breakdown of how bugs are resolved over the same period.



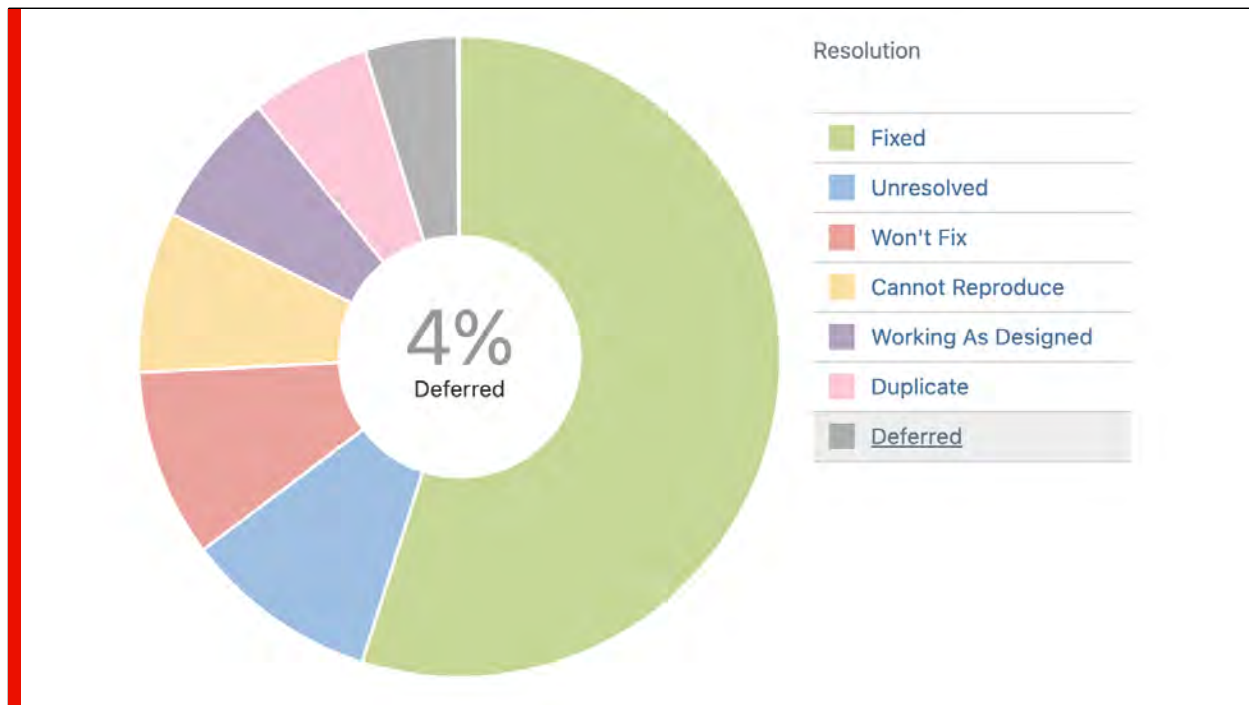
54% - At least more bugs are fixed than not?
In fairness, the % should be a little higher because
“Working as Designed” and “Duplicate” are not
bugs.

“Cannot Reproduce” is a little more problematic
and could indicate a need for better test coverage.

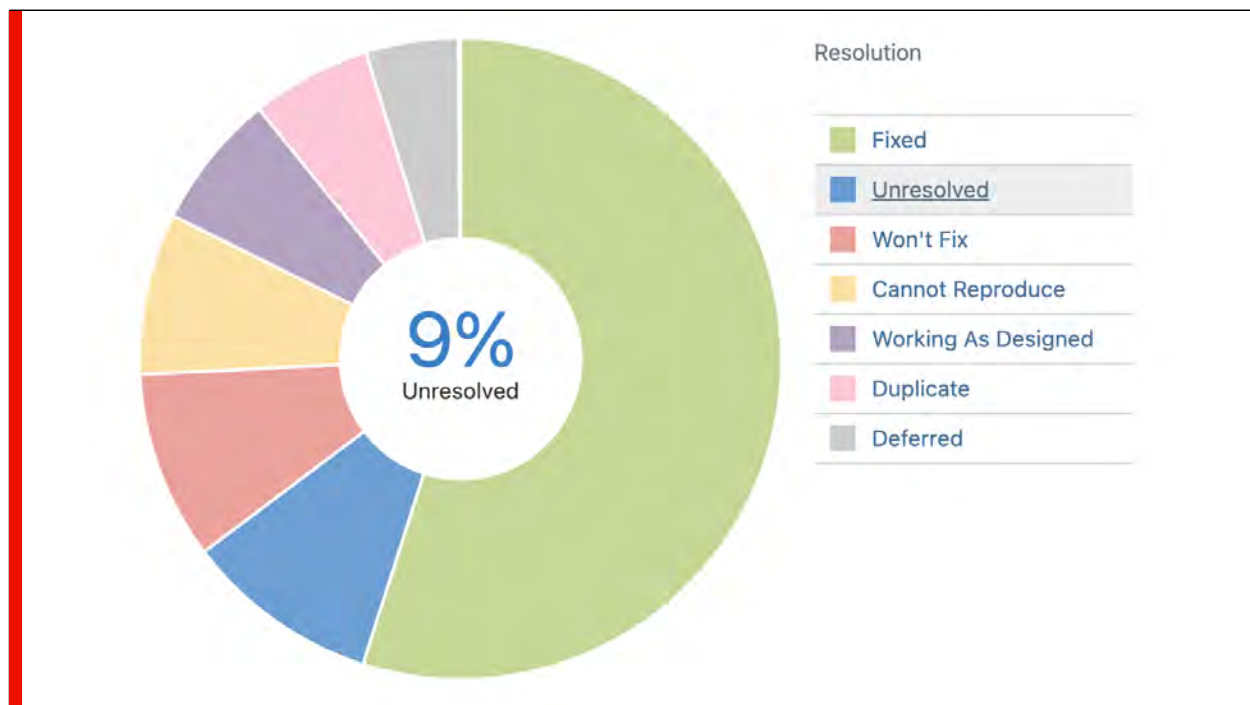


Won't Fix: "Usually low priority bug, not worth the effort, or no need to fix for a situation that's not important anymore."

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.

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

Answer: Iterative Refinement.

- Current programming methodologies lend themselves to iterative refinement
- We don't solve problems; we approximate solutions

Adobe

© 2025 Adobe. All Rights Reserved.

← HEADER
← 1/4
← 2/4
← 3/4
← FOOTER

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.

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

How can we write *Better Code*?

- Study how to write correct software
- Write algorithms once, in a general form that can be reused
- Focus on fundamental algorithms
- Compose larger systems from proven components

Adobe

© 2025 Adobe. All Rights Reserved.

← HEADER
← 1/4
← 2/4
← 3/4
← FOOTER

[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

And that brings us to generic programming

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

Is STL successful?

- ☐ Millions of copies out
- ☐ Everybody (Microsoft, IBM, Sun ...) ships it
- ☐ A dozen books
- ☐ Very few extensions
- ☐ No language progress
- ☐ No effect on software engineering

26

Adobe

© 2025 Adobe. All Rights Reserved.

FOOTER

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".

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

Adobe

© 2025 Adobe. All Rights Reserved. 29

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

“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

“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

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

Adobe

© 2025 Adobe. All Rights Reserved. 35

[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

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.

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

Programming is Mathematics

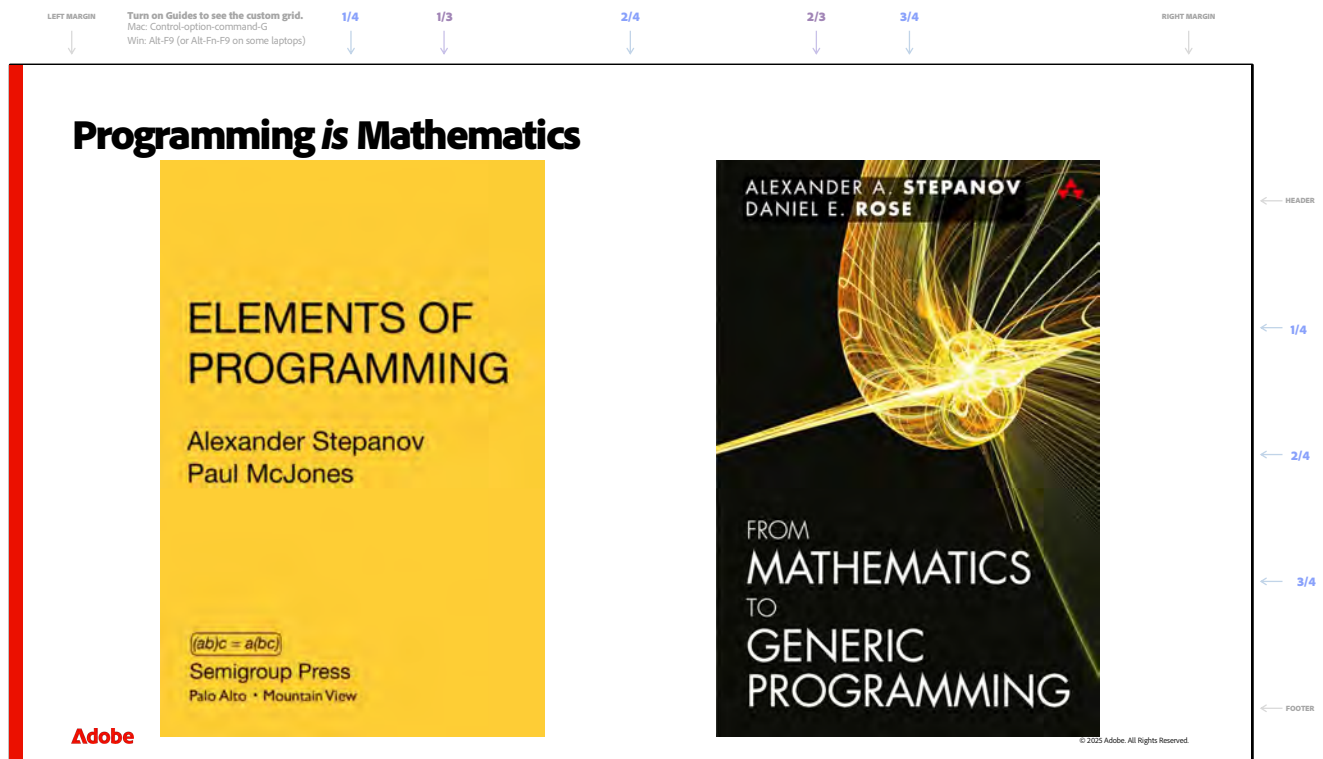
▪ Generic Programming ▪ Semantic Requirements ▪ Concept ▪ Model ▪ Algorithms ▪ Regular Function ▪ Complexity	▪ Mathematics ▪ Axiom ▪ Algebraic Structure ▪ Model ▪ Theorems ▪ Function ▪ _____
▪ Refined Concept: a concept defined by adding requirements to an existing concept ▪ Monoid: a semigroup with an identity element ▪ Bidirectional Iterator: forward iterator with constant complexity decrement ▪ Refined Algorithm: An algorithm performing the same function as another but with lower complexity or higher efficiency on a refined concept	

Adobe © 2025 Adobe. All Rights Reserved.

← HEADER
← 1/4
← 2/4
← 3/4
← FOOTER

The notion of a refined algorithm, or specialization, is what allows generic programming to be efficient. The more we know about the properties, the more we can specialize an algorithm, all the way to the point where there is only a single model. At this point, the algorithm is “concrete”.

Generic programming is not a paradigm, but a lens through which we evaluate the correctness and efficiency of any code.



Since I first presented that slide, Alex has published Elements of Programming and From Mathematics to Generic Programming. The first lays the mathematical foundations for Generic Programming, the latter traces algorithms through history from Pythagoras, showing how that evolution can be expressed as increasingly generic algorithms.

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

Concepts

- A Concept is an algebraic structure formed of *connected* requirements
- Equality is a unique equivalence relation...

$\forall a: a = a$ (reflexive)
 $a = b \implies b = a$ (symmetric)
 $a = b \text{ and } b = c \implies a = c$ (transitive)
- ...connected to copy and assignment:

$b \rightarrow a \implies a = b$ (copies are equal)
 $a = b = c, d \neq a, d \rightarrow a \implies b = c$ (copies are disjoint)
- The concept *regular* is essential for equational and local reasoning

Adobe
© 2025 Adobe. All Rights Reserved.

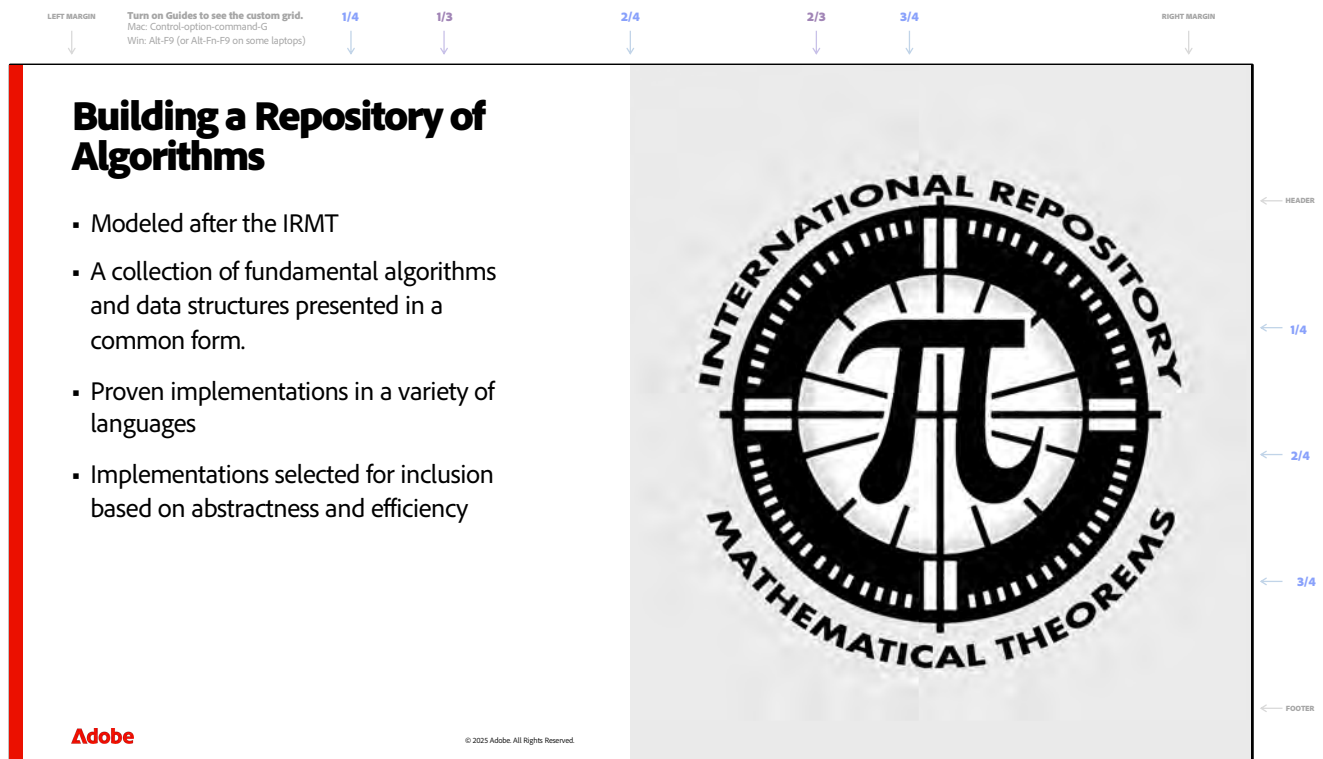
HEADER
1/4
2/4
3/4
FOOTER

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.

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

Unfortunately, the IRMT doesn't exist

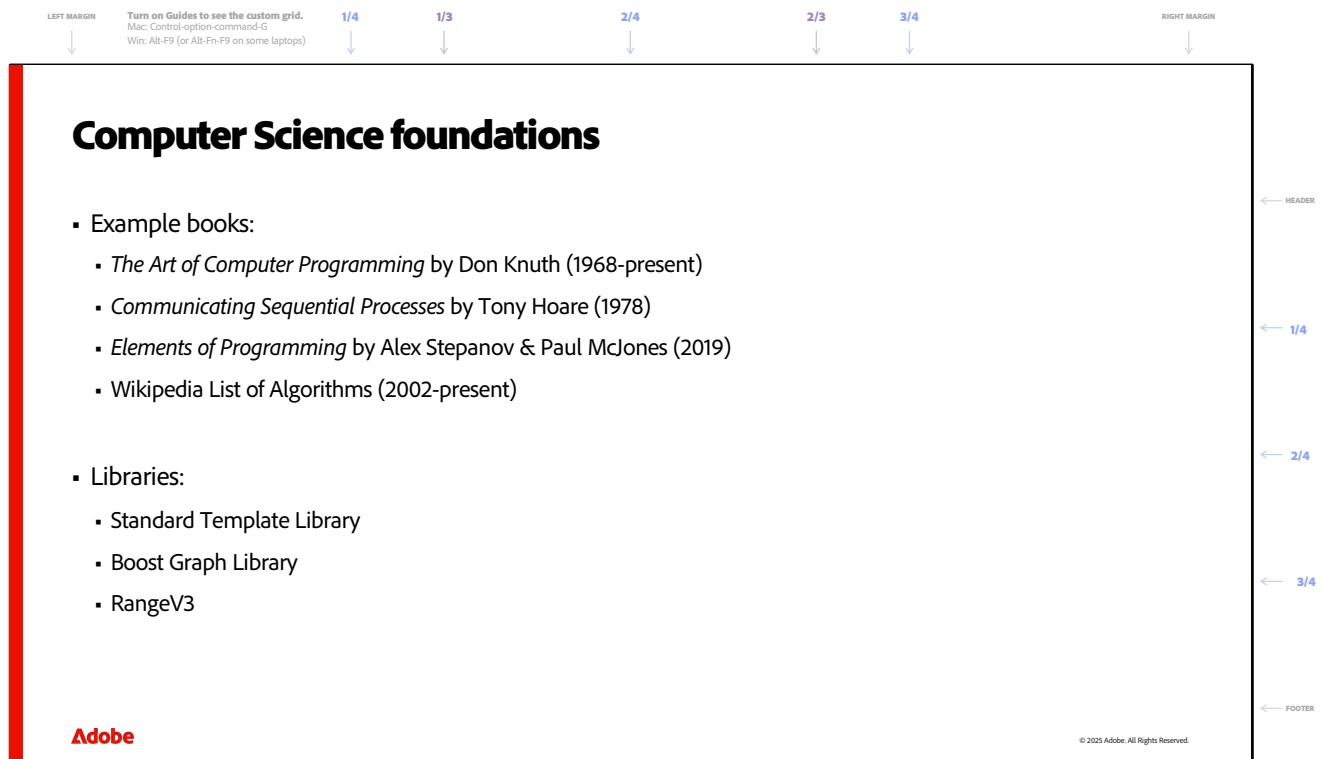
- Building a cohesive foundation is *hard*.
- Examples:
 - *Euclid's Elements* (300 BCE)
 - *Principia Mathematica* by Isaac Newton (1687)
 - *Formulario Mathematico* by Giuseppe Peano (1894)
 - *Éléments de mathématique* by Nicolas Bourbaki (1939)
 - *Wolfram MathWorld* by Eric Weisstein (2000-present)

Adobe

© 2025 Adobe. All Rights Reserved.

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.

We have excellent tools like Wolfram MathWorld, and you'll find many other attempts online; even Wikipedia has a decent list. But nothing that resembles a cohesive collection of theorems.



[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.

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

← HEADER

← 1/4

← 2/4

← 3/4

← FOOTER

The Language Challenge

- Semantic constraints are not expressed (and unchecked)
- Tradeoffs
 - Separate compilation makes specialization difficult (impacts efficiency and expressibility)
- Concept mechanisms are used for “implements” as opposed to “models”
 - See concepts like “std::copy_constructible”, or the trait “std::ops::Sub”, or the protocol “Hashable”
- Confusion over the domain of operations
 - A type *models* a concept if values of that type *exist* that satisfy the concept
 - operator < on double models strict-weak-ordering

Adobe

© 2025 Adobe. All Rights Reserved.

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.

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

Current Design of Large Systems

- Networks of objects form incidental data structures
- Messaging among objects forms incidental algorithms
- Design Patterns assist in reasoning about these systems
 - Local rules approximate correct algorithms and structures
- Iteratively refine until quality is *good enough*

Adobe

© 2025 Adobe. All Rights Reserved.

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.

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

A "Generic Algorithm" for Building Systems

- Identify the components and how they connect and interrelate
- Avoid incidental data structures by packaging related objects as parts of a whole
 - The whole maintains the invariants of any relationships between the parts
- Avoid incidental algorithms by recasting as explicit algorithms
- Identify common structures and algorithms, and refactor them into generic components
- Repeat...

Adobe

© 2025 Adobe. All Rights Reserved.

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 their 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.

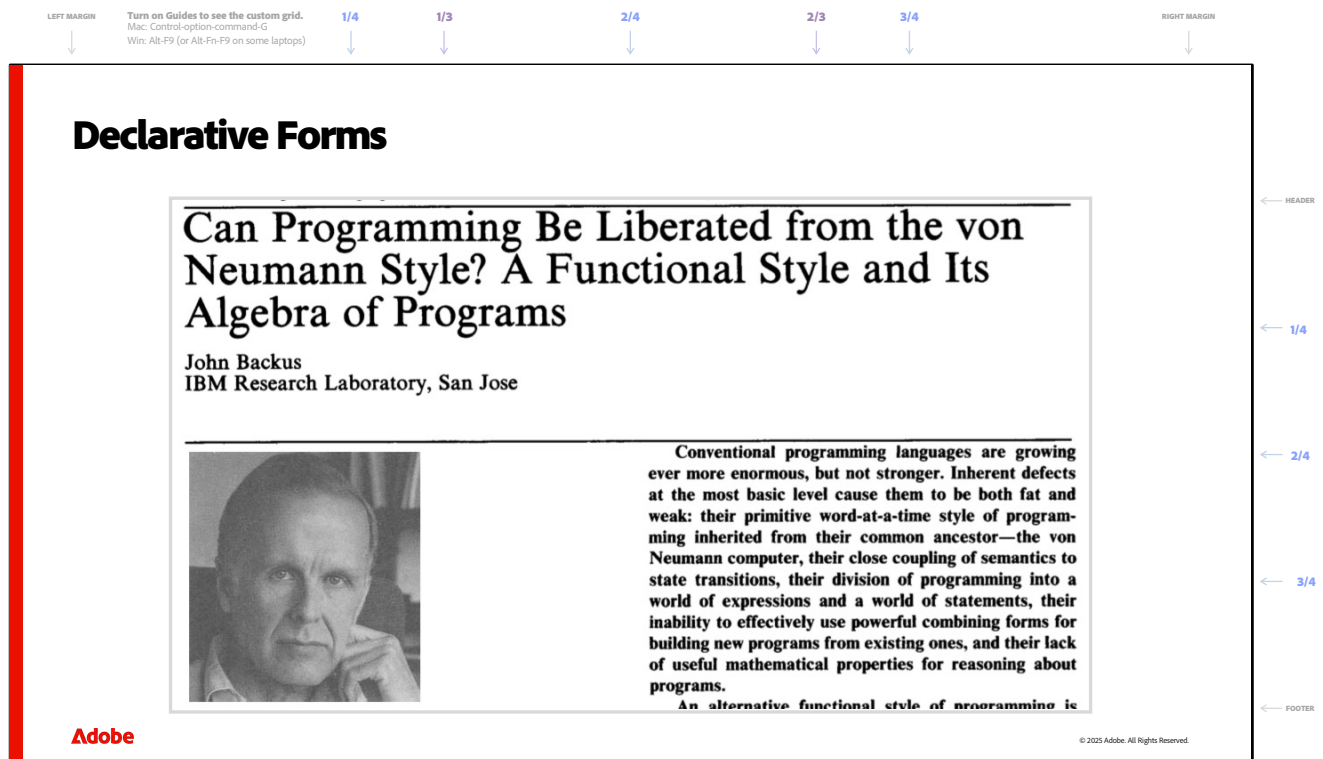
“More is different,” Philip W. Anderson – large systems have emergent behaviors.

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.



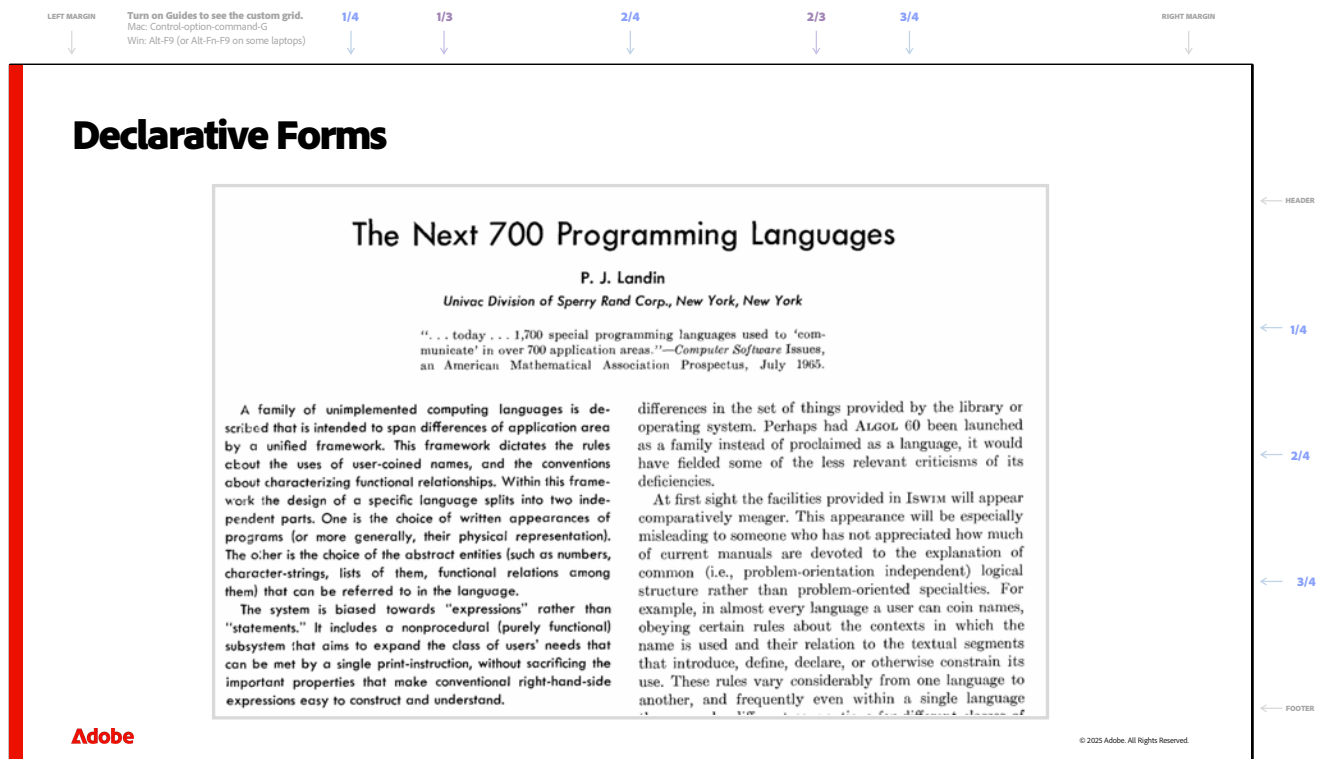
The idea of a “declarative form” is rooted in what John Backus referred to as “functional or combining forms” in his 1977 Turing Award lecture.

Backus identified identified 5, very simple forms:

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).



Declarative forms also take inspiration from Landin's 1966 paper.

The idea presented in this paper is to develop a language that facilitates the creation of domain-specific languages. This is done in a functional form, with a “sugared” lambda syntax (Landin coined the term “syntactic sugar” in a 1964 paper). This paper was a direct inspiration for modern functional languages, including Haskell and Scheme.

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

Declarative Forms

- Describe components by their structure and constraints in a domain-specific language
 - 4th generation 5th generation languages
- The DSLs should not strive to be general-purpose or Turing-complete
 - But Turing completeness, or effective Turing completeness, is nearly impossible to avoid
- Examples:
 - Lex and YACC (and BNF-based parser generators in general)
 - SQL
 - HTML (ignoring <script> tags)
 - Type Systems and Schema Languages
 - Spreadsheets

Adobe

© 2025 Adobe. All Rights Reserved.

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.

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

← HEADER

← 1/4

← 2/4

← 3/4

← FOOTER

A "Generic Algorithm" for Building Large Systems

- Apply the algorithm for building systems
- Identify the common rules governing related parts of components and their structure
- Choose a DSL that allows you to define the component by expressing the structure and constraints
 - If a DSL doesn't exist, develop one*
- Repeat

▪ *Currently *hard*

Adobe

© 2025 Adobe. All Rights Reserved.

[60:00 – 30:00]

We can extend our previous process to large system.

Rules Exist

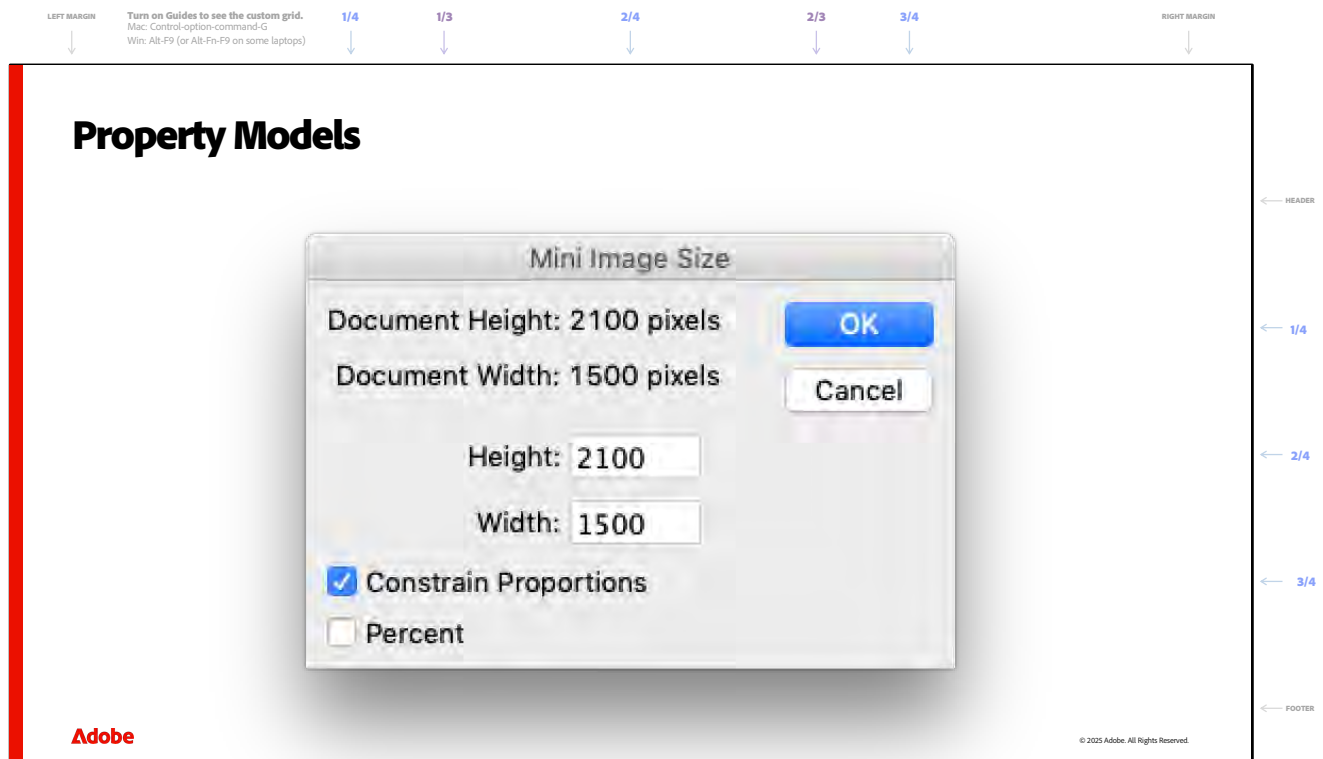
This algorithm works because rules exist.

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...

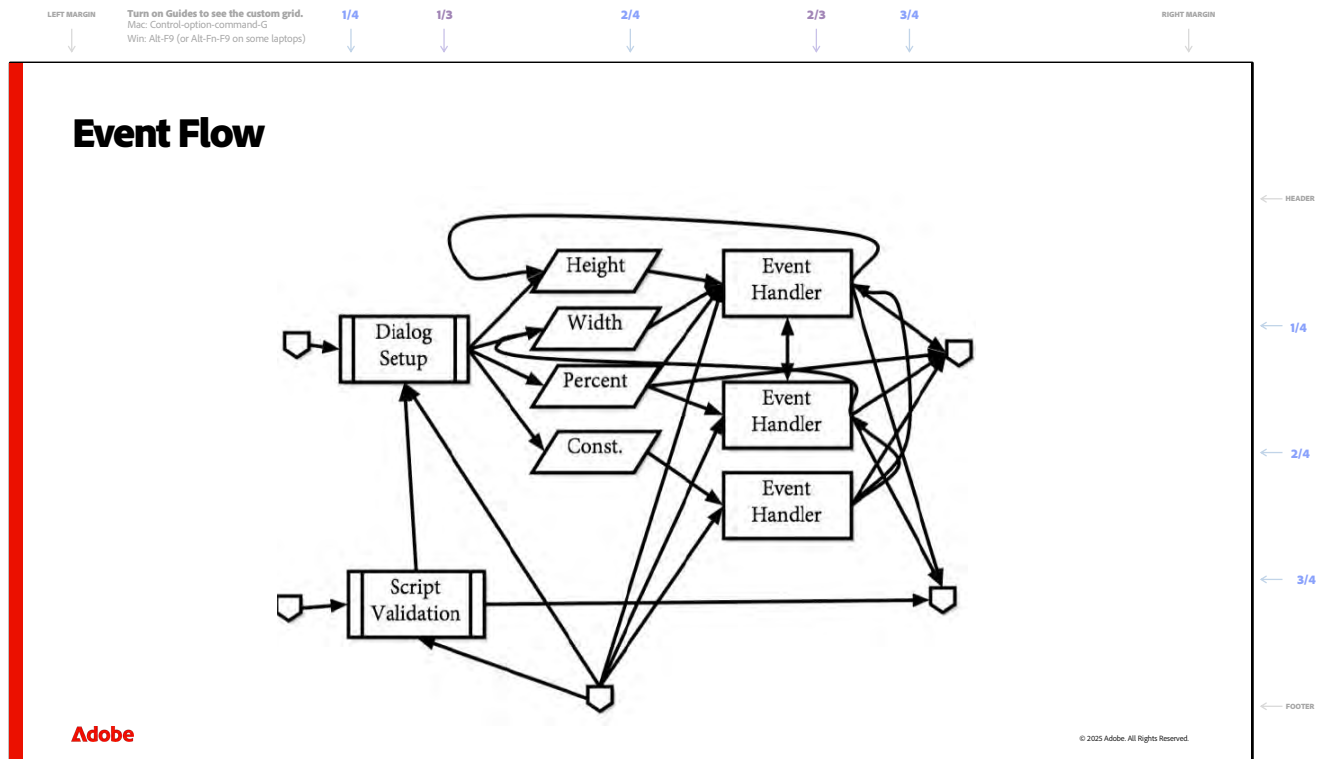


Let's take a look at an example. This is a greatly simplified version of a dialog that lets the user resize an image. They can specify the width or height in pixels or percent and toggle constraining proportions—pretty simple stuff.

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.



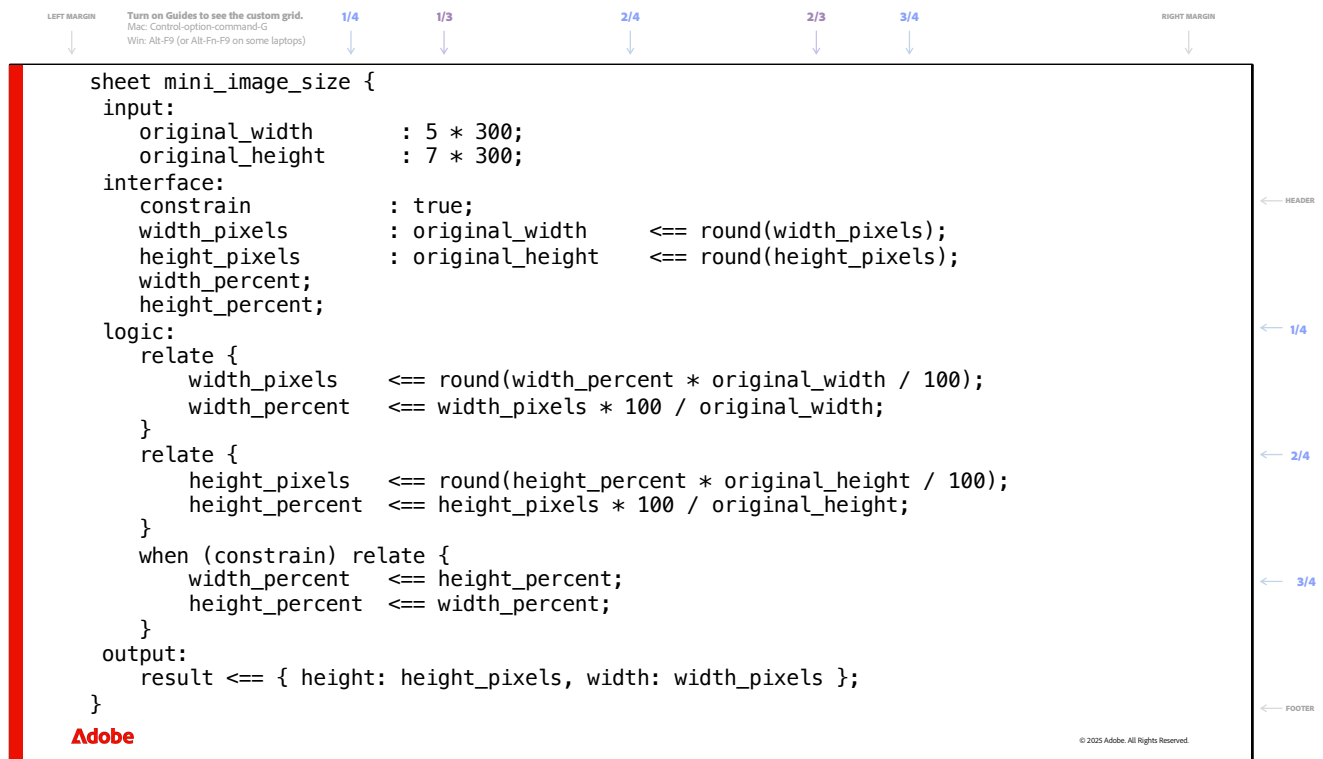
If you diagram the event flow, it would look something like this – this includes handling script playback, which I don't believe Mark implemented.



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.

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

A Declarative Form

```
sheet mini_image_size(size: (usize, usize)) {  
  var constrain_aspect_ratio: bool = true  
  var pixel_size: (usize, usize) = size  
  var percent: (f64, f64)  
  
  maintains:  
    pixel_size.0 == round(percent.0 * size.0 / 100.0)  
    pixel_size.1 == round(percent.1 * size.1 / 100.0)  
  
    when constrain_aspect_ratio {  
      percent.0 == percent.1  
    }  
}
```

Adobe

© 2025 Adobe. All Rights Reserved.

← HEADER

← 1/4

← 2/4

← 3/4

← FOOTER

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

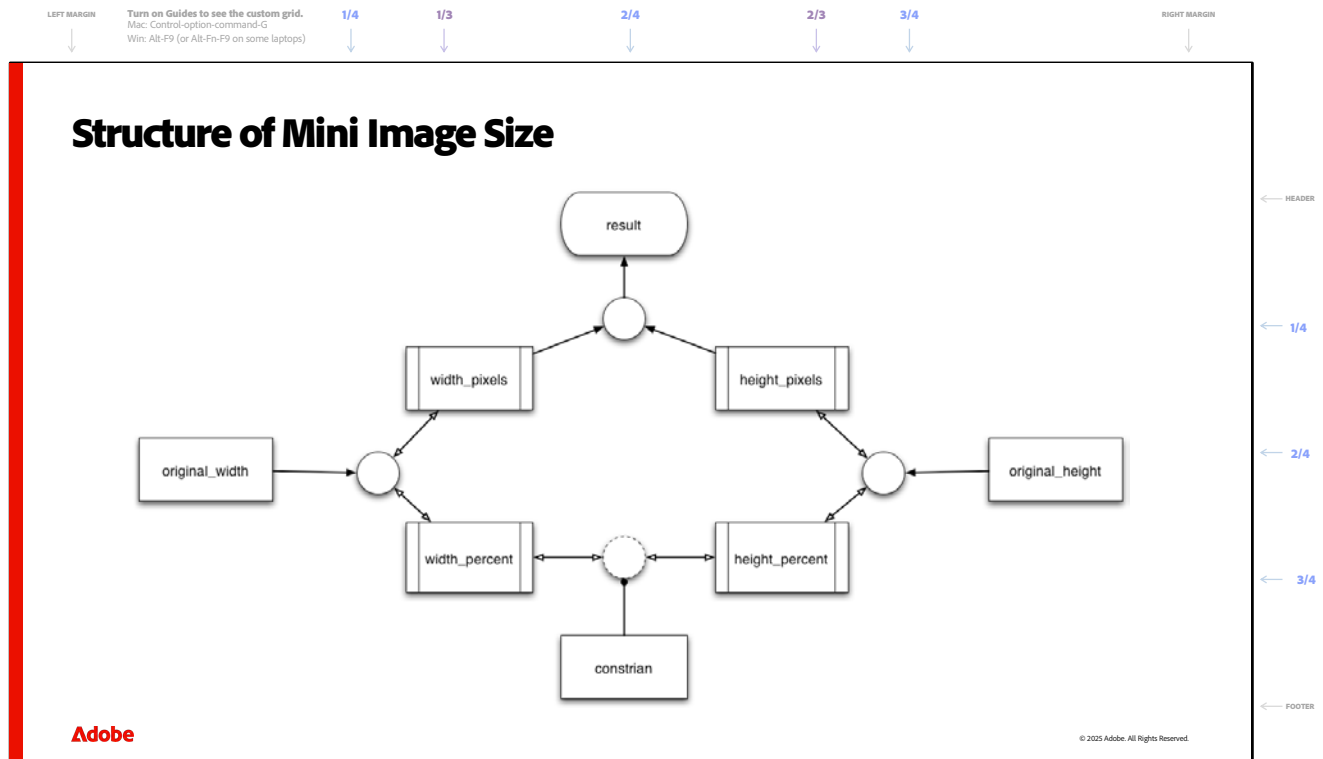
magnitude reduction in the code needed to describe our products and we would see a greater than two orders of magnitude reduction in defects.

[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

```
1. Create a new document (File > New) with the following settings:  
- Name: Mini Image Size  
- Width: 1000 pixels  
- Height: 1000 pixels  
- Color Mode: RGB Color  
- Resolution: 72 pixels/inch  
- Background Color: White  
2. Create a new layer (Layer > New > Layer) and name it "Background".  
3. Fill the background layer with a solid color (Edit > Fill > Color > #FFFFFF).  
4. Create a new layer (Layer > New > Layer) and name it "Image".  
5. Drag the image file into the "Image" layer.  
6. Resize the image to fit the canvas (Image > Crop).  
7. Save the document (File > Save).  
8. Export the image as a PNG file (File > Export > Export As > PNG).
```



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 are when it is a “don’t care” (it doesn’t contribute to the result under the current constraints) or when its value is implied and no contra-positive to the implied relationship is expressed. All the information governing the UI behavior is in the relationships.

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

Future of Software Development (2006 - restated)

- Extend the ideas of generic programming to more domains
- Extend generic programming to apply to runtime polymorphism
- Formally describe software behavior by expressing the structure and constraints of the system

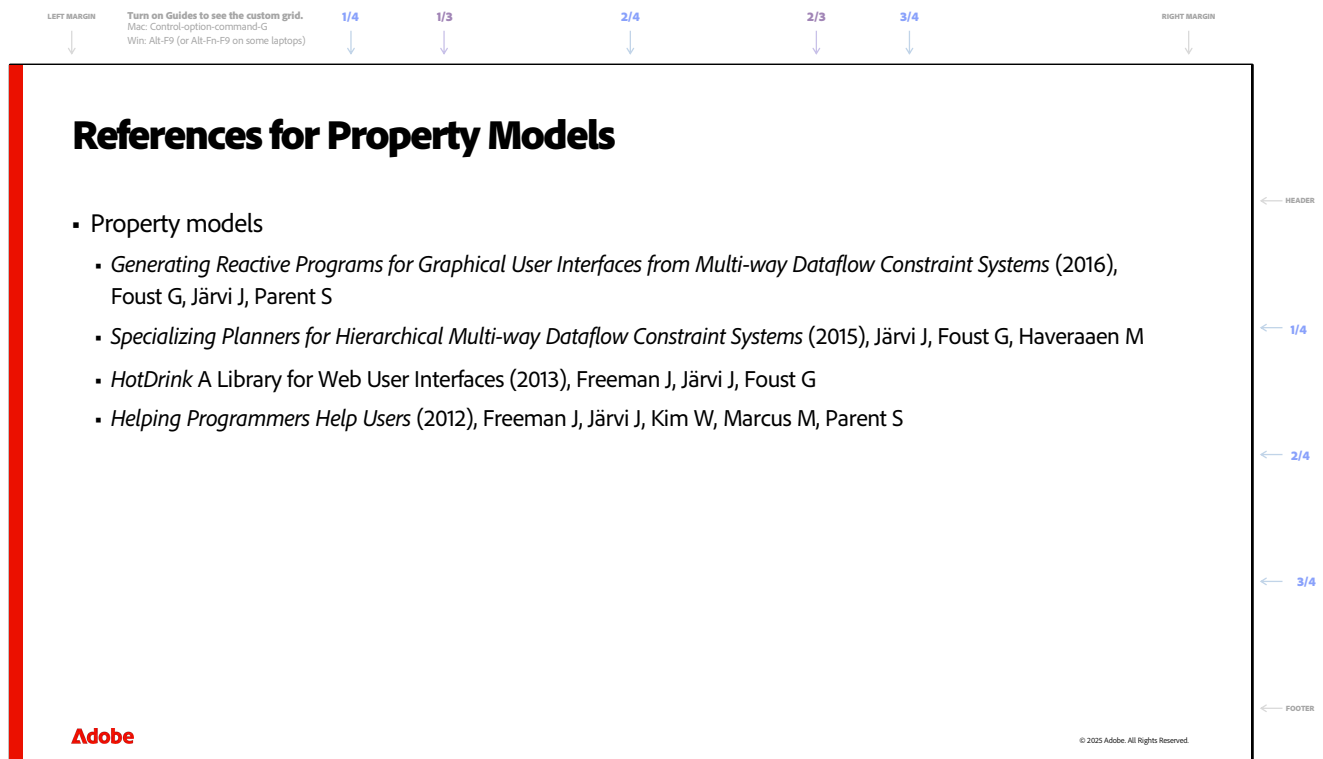
Adobe

© 2025 Adobe. All Rights Reserved.

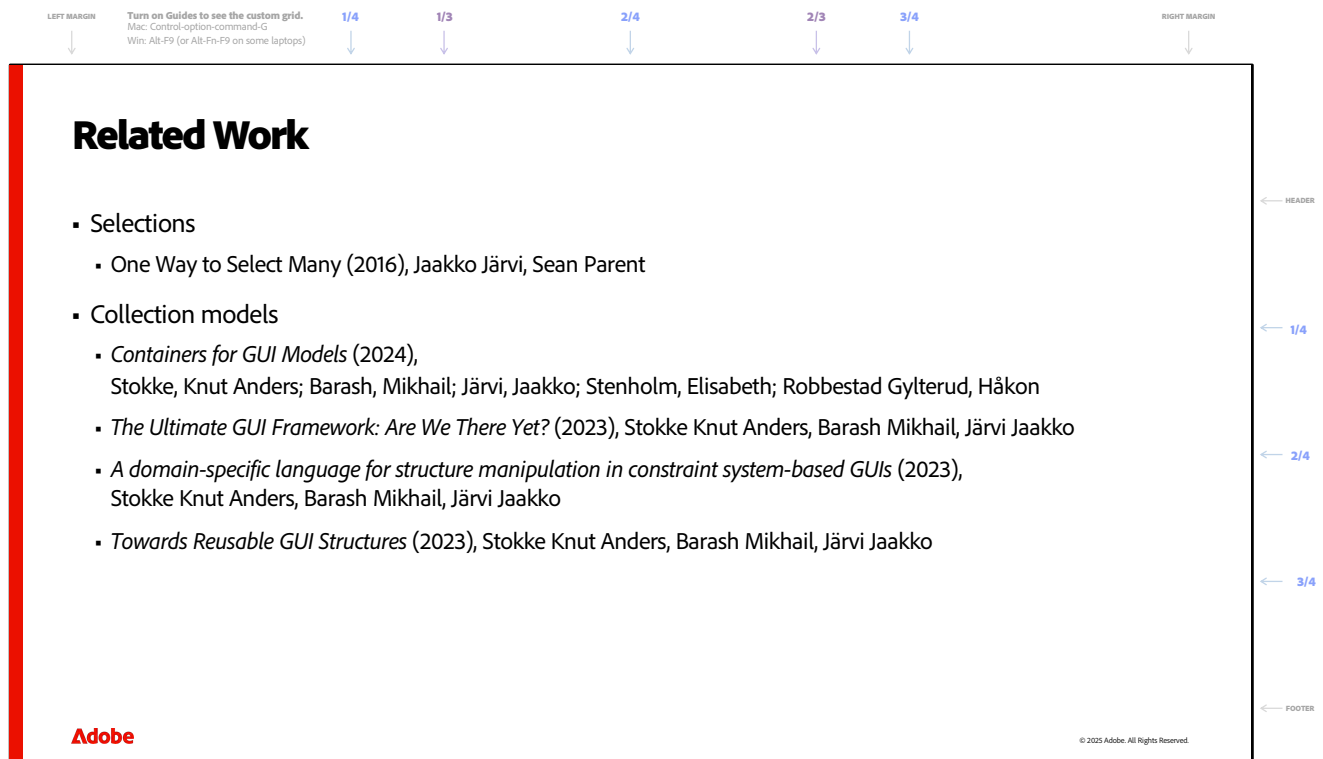
This was my final slide in the old talk.

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!

Since I’ve returned to this work, I was planning to have something more to present today – but I’m *not there yet*. Stay tuned.

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

Future of Software Development

- Continue to improve the generic programming support in languages
 - Refinements
 - Dependent concepts
 - Law of exclusivity (follows from whole/part relationships and local reasoning)
 - Minimize tradeoffs between efficiency and safety
- Create better foundational libraries by implementing fundamental concepts and algorithms
- Create libraries of embedded DSLs that interoperate
- Develop AI to *reason* about code so that it can continue the above

Adobe

© 2025 Adobe. All Rights Reserved.

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.



You can find this presentation, and all my presentations, at sean-parent.stlab.cc.

Closer Slide

A large red rectangle with the word "Adobe" in white, bold, sans-serif font centered within it.

Adobe

[80:00 – 10:00]

Questions?