Software Development Is About Learning

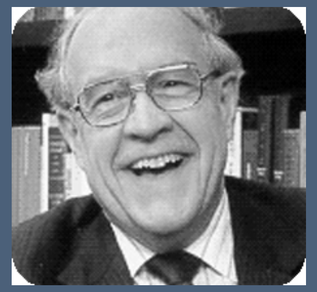
Lessons from Fred Brooks' "No Silver Bullet - Essence and Accidents of Software Engineering"

Jeff Gonzalez

About Me

- Jeff Gonzalez
- Lakewood, OH
- Programmer, Teacher
- Dad (two kids, two cats), husband.
- 39 Years as a professional software developer
- 29 Years of teaching

Fred Brooks



- Turing Award Winner, 1999
- The Mythical Man Month
- IBM System/360
- No Silver Bullet: Essence and Accidents of Software Development (1986)

We Are In A Crisis

Software Development Is Being Attacked!

In the 1990s, code will be generated by the click of a mouse or a tap of a key. With Matrix Layout 2.0 you can do that now. And the results will surprise you.

Preview the 1990s with Layout

In Layout, you create programs by designing an object-oriented flowchart, with all the options of traditional programming. It's a technology we call desktop programming.

Once you're done, simply choose the language you want for the finished program. There's Microsoft C, Lattice C, and Turbo C, as well as Turbo Pascal and Microsoft QuickBasic. You can even create a .EXE file that's ready-to-run on any IBM PC or compatible.

1990s Power without 1980s Pain

Because Layout works with today's standards, you can painlessly take advantage of the power behind Layout – object oriented programming, CASE (Computer Aided Software Engineering) technology, hypertext databases, and graphical user interfaces. All without giving up your favorite computer language.

An Architecture for the 1990s

Layout comes with objects that produce real code for everything traditional computer languages can do – math, branching, variable management, complex data structures – and it extends each language to include powerful user interface and hypertext database capabilities. But best of all, you can extend Layout past the 1990s by building your own objects – BlackBoxes – that can do anything you imagine. Added together, Layout cuts your development time by up to 70%.

Welcome to the 1990s

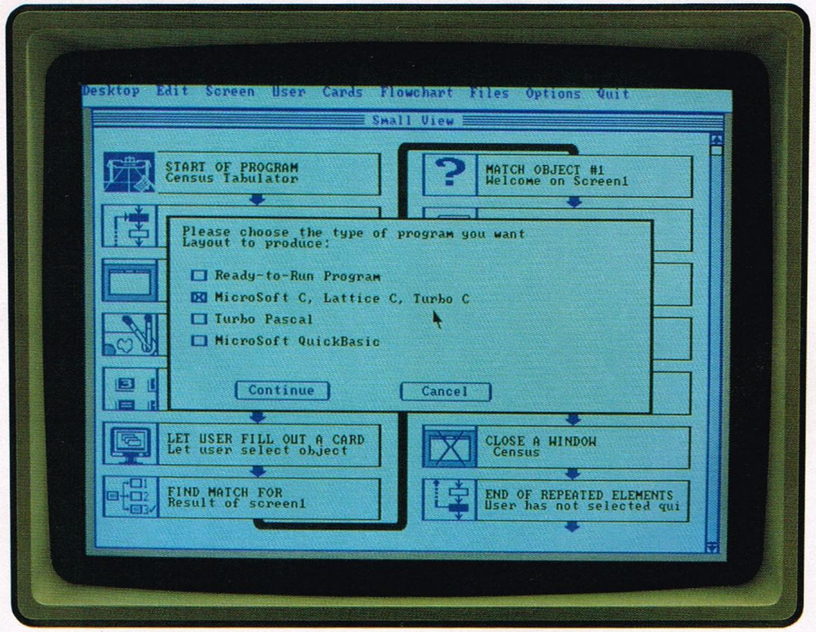
Ready to jump into the 90s? Get Layout today. It's available for just $199.95. Or for a glimpse of the 90s, see the Layout video tape for just $9.95. Give us a call at

1-800-533-5644

In Massachusetts call (617) 567-0037.

MATRIX
MATRIX SOFTWARE

In the 90s, this is how you'll write code.



Matrix Software Technology Corporation • One Massachusetts Technology Center • Harborside Drive • Boston, MA 02128

Matrix Software Technology Ltd. • Matrix House, Derriford Business Park • Derriford, Plymouth • Devon PL6 5QZ, England • 0752-796-363
Matrix Software Technology Europe N.V. • Geldenaaksebaan 476 • 3030 Leuven, Belgium • 016202064.
All trademarks and registered trademarks are of their respective companies.

Circle 182 on Reader Service Card (DEALERS: 183)

An Allegory

I hate allegories.

A great artist, a painter of world-renown, was once visited by the owner of the gallery at which she was employed. The gallery had increased profits nearly 20-fold since 1971 when the artist was hired. (See NASDAQ History)

They had been monitoring her work, and had been talking to a new vendor that told them that the artist was only actually painting about an hour or two a day. The rest of the time the artist was doing “tedious” tasks such as sketching, washing brushes, and just *staring* at various subjects.

Painting was about putting paint on the canvas, and this vendor had a new model that could do it *much* faster and more *efficiently*.

They have proven an increase of nearly 60% when measuring actual paint applied to the canvas.

Some Context

From an old guy

- Banks have been around for about 4000 years, followed by insurance.
- Healthcare has been around forever.
- Selling things?
- Software development is from the 1960s. (About 65 years)
- Companies made money and knew what they were doing before software.

Building Software Isn't Digging Holes

There is no "bottom" or "top"

- Improvements historically have only made what we have already done easier.
- What we've already done is already done.
- "Code a Netflix Clone"
  - https://www.youtube.com/results?search_query=code+a+netflix+clone

When we set the upper limit of PC-DOS at 640K, we thought nobody would ever need that much memory. — William Gates, chairman of Microsoft, Probably.

Things I Can Do

That I Don't Want To Do

- If I ***never*** have to do about 90% of what I do as a developer again, I'll be happy.
- Examples:
  - Any form validation. At all. Ever.
  - Writing regular expressions.
  - Implement a "CRUD" API.
  - What are yours?

There will always be code.

- You are screwed if you think software development is *just* about knowing how to code stuff.
- You are nuts if you think software development won't ever involve coding stuff.

What Is Our Job as Software Developers?

The Essential Tasks of a Software Developer

"The fashioning of the complex conceptual structures that compose the abstract software entity"

Fred Brooks, 1986 - No Silver Bullet

The Accidental Tasks of a Software Developer

"The representation of these entities in programming languages and mapping these onto machine languages within space and speed constraints."

Fred Brooks, 1986 - No Silver Bullet

The Tasks of a Software Developer

Essential: Learning over time.

Accidental: Coding.

Fred Brooks, 1986 - No Silver Bullet

The Essential Tasks of a Software Developer

"The fashioning of the complex conceptual structures that compose the abstract software entity."

Fashioning: A creative act.

Complex Conceptual Structures: We make thingies that do stuff.

Abstract Software Entity: A representation of our current thinking.

The Tasks of a Software Developer

1. Don't Write Software
2. Rapid Prototyping
3. Grow Software Organically
4. Develop your Team

Fred Brooks, 1986 - No Silver Bullet

1. Don't Write Software

We (probably) don't need to build another:

- Testing Tool
  Web Framework
  Database
  Etc.

- Learn what is available. Share that.

1. Rapid Prototyping

“Proto” = “Before”, “Typing” = “A category of things with commonality”.

Writing code at the lowest common denominator:

- How much do you understand the problem at hand?
- How much do you understand the tech?

1. Grow Software Organically

Software is discovered, not designed.

Get a nose for duplication.

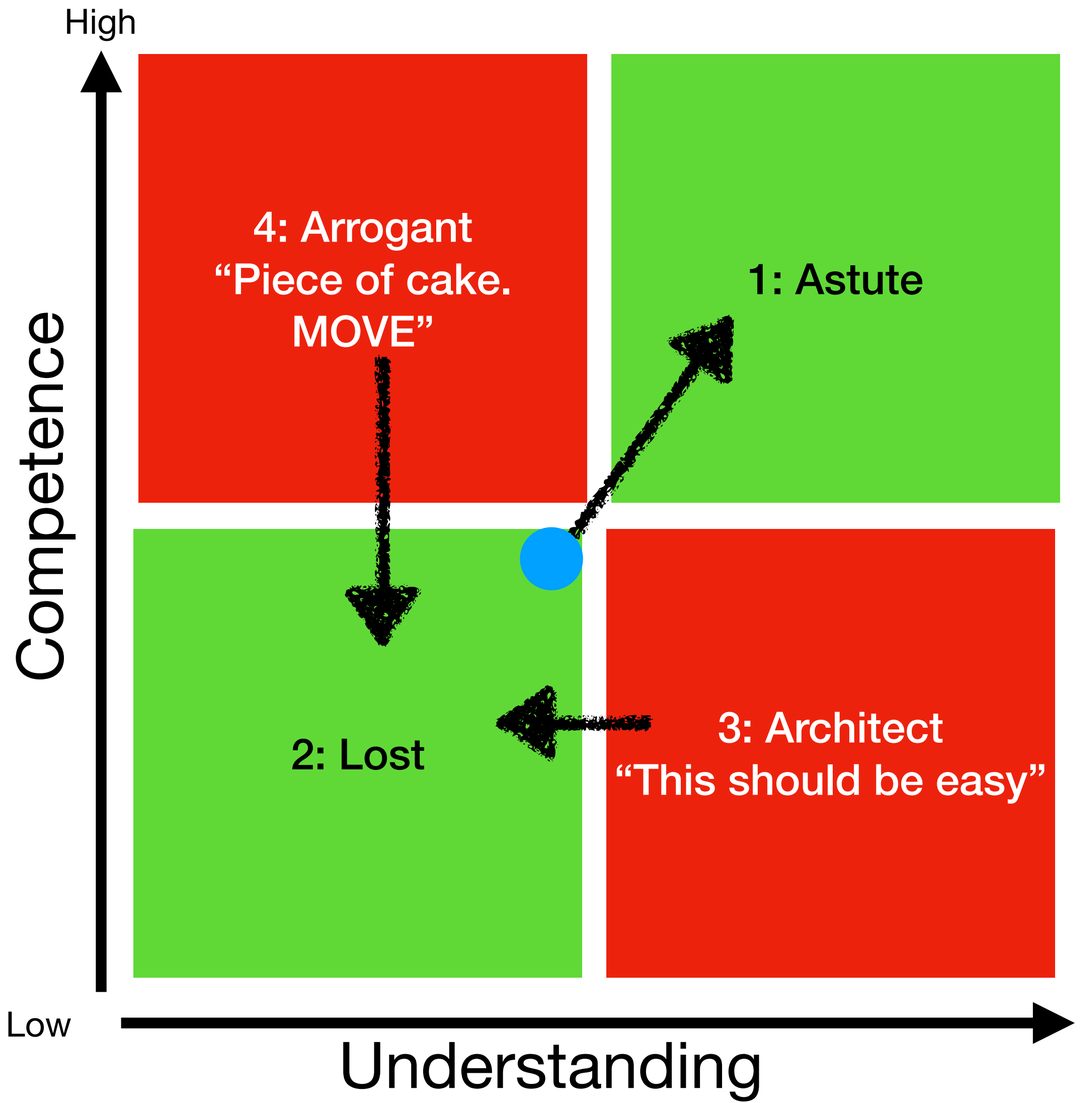
Three Virtues:

1. Laziness
2. Impatience
3. Hubris

Larry Wall

Your Confidence Zone

For Any Given Task, Find Your Zone And Work From There



The blue dot is the path to being astute.

Going from 4 - 1: Enterprise Fizzbuzz

Going from 3 - 1: ChatGPT Wrote This Mess

See: https://martinfowler.com/bliki/BeckDesignRules.html

See: “Make it Right Then Make It Good”

See: “Everyone who has said to you ‘I have this great idea for an app, just need someone to code it up’”

A Hard Truth

Software Development is About Learning

- LLMs are good at doing what has already been done a bunch of times.
- That means, in some way, “experience” (as in “I have the documentation memorized and can do leetcode while on LSD” is going to be less valuable.
- The “best practices” are *always* time-boxed.
- Don’t confuse “I have 20 years of experience” with “I have one year of experience I’ve sold for 19 more years”.

Dance in the Freakout

- “The bad news is you’re falling through the air, nothing to hang on to, no parachute. The good news is, there’s no ground”
  - Chogyäm Trűngpa, Rinpoche

The Tasks of a Software Developer

Fred Brooks, 1986 - No Silver Bullet

Essential and Accidental Properties

- Is Software an Essential Property of Business?
- If Yes, What is “Essential” About Software? The Code?
- If No, Explain Yourself.
- The “Essential Properties” of a Business only have one metric: Profit
  - “How much CEOing did you do today?”

Self-Victimization

Let's retire these phrases, please.

- "Imposter's Syndrome"
- "Technical Debt"
- "We need a refactoring sprint"
- "*They* (the 'business') won't let us"
- "Let's build a prototype"
- Hot take warning: "Architect" and "Architecture"
- Guidance / Reference Apps / Coding Standards

Imposter's Syndrome

Definitions

- Imposter: a person who pretends to be someone else in order to deceive others, especially for fraudulent gain.
- Syndrome: a group of symptoms or behaviors that occur together that indicate a specific condition. Clinical.
- Developer's claim "Imposter's Syndrome" because *someone* gave the impression that *any of us* knows what the hell we are doing. We just know how to F around and Find Out.

Imposter's Syndrome

a person who pretends to be someone else in order to deceive others, especially for fraudulent gain.

Technical Debt: The implied cost of future reworking because a solution prioritizes expedience over long-term design.

Technical Debt: Th plied cost of future rew a solution prioritizes ver long-term desig

Technical Debt: We always have “design” - it’s only “debt” when it gets in our way.

Some huge percentage of what I see as *actual* technical debt is just *bad* design.

Refactoring: Is just coding. Don't make it or call it something "else". It's what we do. As you learn more, about the "complex conceptual structures that make up the abstract software entity", you have *insights, breakthroughs*. One of the highlights of software development is when you discover something's *true name*.

Evolve your code over time to more accurately represent and model the *concrete*.

They Won’t Let Us:

“We wanted to do a rewrite in React 18 but…”

You *won’t* win every argument. But take the responsibility *as a team*.

You’ll find sometimes you dodged a bullet.

If you find you were right, and feel completely disempowered, you are probably arrogant or working at the wrong place.

Give “them” the benefit of the doubt, until you just can’t any more.

Let’s Build a Prototype:

What they hear is “let’s spend money to build code that won’t make us any money”.

Prototype in production.

Hypothesis driven development.

Feature flags/feature toggles.

Organically grow software through iterations.

Architects and Architecture:

What I mean by this is not establishing “wizards” or a “priest class”.

I think “architect” is a wrong borrowed term.

Architecture is Emersonian “Genius”

Architecture is the part of the software that is difficult or impossible to change in the future.

Guidance / Coding Standards / Reference Apps:

The sacred dogma of the priest class.

We will probably always need *some*, but not as much as I often see.

They *can* be used as a way to protect the priest class.

Guidance:

“Good practices” vs. “Best practices”.

“Best” sounds too *final*. Make it a discussion. Document the discussion. (More on this later)

Coding Standards:

Formalize them *even more*. Linting rules, automated tests.

If there is an *invariant* can you generalize it in a way that gives the developer feedback as they write/test the code? Prevents it from happening in the future as opposed to a sort of Latin mass to be beat them with in code reviews?

Coding Standards - Continued:

Don’t be so binary. Let the code evolve.

This is how we “identify and develop the great conceptual designers of the future”.

People have a hard time understanding the solution for a problem they’ve never had.

Reference Apps:

Any application worth it's salt is the representation of all the things that were learned through building the application.

"Oh you want to write a novel, edit this copy of 'War and Peace'."

Reference apps are you saying "Here's what worked for us, and I am confident it will work for you, too."

Note: This, unfortunately, includes most tutorials and training. Saying this as a trainer. Forgive me.

Reference Apps - Continued:

What are some other ways that are less lazy to share your ideas?

All I know is step 1 is drop the “imposter’s syndrome” crap. You have valuable insight, and some of the best insight is “wrong”.

You learn more from having “bad” code that you communally make better than from adoring someone else’s idea of “good code”.

Team Technical Discussion Records

- Based on the idea of “Architectural Decision Records”, but you can probably guess how I feel about that.
  - “Architecture is the parts of you application that are hard to change later” - Source: ??
- Can decisions made by the team:
  - Be validated through *some* discussion?
  - Be open to review/modification over time?
  - Be helpful?

Technical Discussion Records

Suggestions

- Format: Decided on by the team.
- Tools: None. I prefer markdown files in the repository. Use Git to track revisions.
- Some things to consider:
  - Context for any decision.
  - Alternatives considered.
  - Is there a “trigger” for reconsideration?

Example TDR

Summary

- We “fashion the complex conceptual structures that compose the abstract software entity” needs some marketing help.
- How about: “I am the creative mind behind computer programs”?