Software Talmud #00: Anatomy of Automation

A Software Article by Efron Amber Licht

Introduction

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

> **Amber & Amber, Anatomy of Automation**: Automation is not a specific discipline but a philosophy of doing things.
>
> > **Norbert Weiner, *GOD and GOLEM, INC***: For us, machine is a device for converting incoming messages into outgoing messages.

Software, signal processing & industrial automation are at their core the same field: the task of making a system that works, despite hundreds, thousands, or millions of potentially unreliable components. These skills of *engineering*, *optimization*, and *logistics* - which in modern business-speak we might call “technical leadership” - are core to software and computers but do not originate there. All the great leaders of history share these abilities to rationalize, simplify, mechanize, scale up. It is not a coincidence that Caesar, Napoleon, and Eisenhower’s armies shared a commitment to rationalization, simplification, engineering prowess. We do not find it strange when a general searches the campaigns of Napoleon or Hannibal for wisdom. The technological distance between a drone strike and a cavalry charge does not change older commanders’ fundamental insights. In fact, the distance between now and then provides the perspective to help highlight which advice is contextual, limited to it’s time and place

(such as, say, the proper mixture of oats and hay), and what is essential and universal (good supply lines, proper planning, a minimum of internal backbiting).

In such a way, the great engineers, mathematicians, and programmers have much to teach us today. We'd be well served by taking another look at the industrial and electrical engineers of the the 40s and 50s - who we might call the generation of the 'computer builders'- and their successors, the 'mainframe programmers' of the 60s, 70s, and early 80s. These generations - the ones who first developed and mastered digital computers but did not yet rely upon them or take them for granted - have much to teach us about how to build systems that work.

Two of those great engineers of the 'Computer Builder' generation were my grandfather, Paul S. Amber, and his brother, George H. Amber. Identical twins, partners in childhood, war, and business, Amber & Amber were wrote **Anatomy of Automation** in the early 1960s, a brilliant look at the *fundamental philosophy* of automated, reliable, reproducible, mass-production systems on the edge of the computer age.

I can think of no better place to start trying to transmit this knowledge than **A&A**.

cover of anatomy of automation, with dust jacket

Why Read This? (“Talmud?”)

In Jewish tradition, Talmud (lit. ‘study’ or ‘learning’) was the codification of thousands of teachers and scholars in the Jewish tradition, an attempt to solidify and reify generations cultural and institutional knowledge. I’m trying to do something along the same lines. This is an attempt to communicate wisdom, which is much harder than technique or raw knowledge. I don’t know if it succeeds. It is meant to, at best, to give you a way of seeing things, or at least implant a little demon in the back of your head that says ‘this system could be simpler’.

If nothing else, I hope it will get you to read old books and write new code.

Note on Structure

The remainder of this work will work as follows: I’ll quote **Anatomy of Automation** by **Amber and Amber**, usually called “A&A” from here on. I’ll usually comment on them, and may add relevant quotes from other figures. I’ll try to provide where the quotes are from and/or link to them where I can, but no promises.

Quotes may be shortened or slightly shuffled. I may *remove words or sentences*, edit punctuation, or shuffle the order of entire sentences. I will never intentionally add or edit words or change the meaning of a sentence. I transcribe this manually and have no formal editor, so mistakes may sneak in.

Commentary on Anatomy of Automation (1)

anatomy of automation cover, no dust jacket

Dedication / Why Quotes?

> **A&A:** To mathematician Norbert Wiener, whose cybernetic interpretations of the roles of men and machines stimulated the authors’ work in automation.

To the memory of Paul Amber and Gerald Licht, my grandfathers, on whose shoulders I stand.

> **Norbert Weiner, *The Human Use of Human Beings***: Messages are themselves a form of pattern and organization. Just as entropy is a measure of disorganization, the information carried by a set of messages is a measure of organization. In fact, it is possible to interpret the information carried by a message as essentially the negative of its entropy, and the negative logarithm of its probability. That is, the more probable the message, the less

> information it gives. **Cliches, for example, are less illuminating than great poems.**
>
> **Samuel Johnson**: The excellence of aphorisms consists in the comprehension of some useful truth in a few words.
>
> **Wittgenstein**: When we can't think for ourselves, we can always quote.

Preface

> **A&A:** The engineers and executives of tomorrow have commonly been taught how to develop yesterday's machines. Education must be redirected to basic principles if engineering and management students will be able to meet the problems of the future.

Software changes too quickly to bother teaching 'modern' software. By the time a trend filters through to the Academy, it is dead or dying. The sides of the roads are littered with the corpses of clever languages, frameworks, and build systems. An engineer with a strong grounding in programming fundamentals will be able to build the tools they need to solve new problems: one with an encyclopedic knowledge of Spring Boot and Ruby on Rails decorators will be bewildered by simple problems.

> **A&A**: We favor the use of many examples.
>
> Niklaus Wirth: Programming is usually taught by examples.

Paul & George *really* wanted you to understand what they meant. The text is illustrated with dozens of tables which provide many, many examples of any term they bring up. Putting in too many would bloat what has already become a fairly long article, so I'll just scatter them throughout the text to give you a taste.

A&A Chart 2-1: Data Processing Activities: many examples of office work that can be automated, such as payroll, insurance records, and many, many more.

> **A&A:** Information seems to obey the same basic laws as thermodynamics. Information tends to flow from a higher state of organization to a lower state of organization. A natural tendency exists for all organized systems to continually break down, to increase their entropy. In an information system, a man or computer can raise the potential by increasing the state of organization of information.
>
> **A&A:** Efficiency is a basic goal of all our activities. Efficiency is the ratio of the output of a system compared to the input. The designer is concerned with the thermodynamic efficiency pertaining to the transfer and conversion of energy. However, few concern themselves with *cybernetic efficiency*. That is, few consider the ratio of information used to accomplish some useful work, compared to the theoretical minimum information necessary to do the

> job. Where thermodynamic efficiency requires the reduction of energy losses, cybernetic efficiency requires the reduction of information losses.

One measure for cybernetic efficiency is the length of the program (in words). Another is the size of the resulting binary.

Levels of Automation

Automating a system isn't an all or nothing thing. Tool use and automation have a variety of forms and "levels". There's no way I can present this better than A&A did:

Areas of Automation

inside cover art: many kinds and levels of automation

> **A&A:** Manufacture by processing is much more amenable to automatic methods than manufacture by fabrication.

This is the old chestnut: 'batch vs stream'.

Compare:

- (Domestic) Household Electricity, Water, Gas vs. weekly garbage collection
- (Consumer) Bottled Soda vs Fountain Soda.
- (Programming): Input/Output streams vs discrete requests and files.

`stdio` pipelines in shell scripts (`./myprog | grep | sort | ...`) are still the gold standard for processing pipelines in programming.

> **A&A**: Tooling is a sensitive spot of industrial automation. The better_tools are those that increase productivity, when a machine uses less power to use a job or uses control information more efficiently. The tool is the most vulnerable part of the entire process.

To build a machine, first you have to have the right tools. Software 'tools' include traditional UNIX programs, your code editor, logging and debugging libraries, and most importantly, your build system and test suite. Good software tools are both *physically* efficient (low power & memory costs, low latency) and *cybernetically* efficient - they require little context or setup. While software tools cannot literally dull or break, the changing software environment around them means your 200ms test suite can quickly become a twenty minute ordeal without constant vigilance.

I wrote articles about speeding up your docker builds and test suites that might be handy.

Office Automation

> **A&A**: Any information that can be handled in a routine way can be handled by an automatic data-processing system; however, **the work must first be re-organized to facilitate automatic data processing. Mere assignment of man tasks to machines does not succeed.** It is necessary to program the work, that is, to break the task down into small elements that can be performed in sequence. Before this can be done, the work must be carefully studied and all redundant operations culled out… **the entire office procedure must first be simplified.**
>
> > **Brian Kernighan**: Controlling complexity is the essence of computer programming.

First reduce the amount of work you need to do as much as possible: *then* automate it. Tooling and build systems tend to accumulate zillions of layers, usually in the form of cursed YAML files representing multiple 'automation' systems such as Kubernetes, CI, helm, terraform, make, etc. etc. Any one of these can be fine: the combination is always wrong. People ask "how can I do this process faster". This is the wrong question: the question should first be "what is the simplest process possible?", and *then* "what is the minimum added complexity to hit my goals? (hopefully, none)".

> **A&A**: If the procedures of the office are studied and reprogrammed for most efficient action and then automation is *not* used, a great deal of benefits will still result… one cannot help inferring that many of the advantages of office automation can be obtained without actually making use of electronic equipment. The alert business manager does not immediately rush out and buy big scale office automation equipment.

Or billions of dollars of GPUs or SAAS subscriptions or cloud computing. Prefer a small checklist to a large program, then write a small program to automate the checklist.

> **Adam Drake**, Command-line tools can be 235x faster than your Hadoop cluster: people use Hadoop and other so-called Big Data ™ tools for real-world processing and analysis jobs that can be done faster with simpler tools and different techniques.

The size of the solution should be proportional to the size of the problem.

Systems Engineering & The Military

> **A&A**: **The earliest engineers were military engineers.** Military has led the way and continues to lead the way in engineering. No one else has had such fabulous success or such costly failures… (we) can learn much from the military control, both successes and failures.

The US Military provides plenty of examples, both good and bad, during WW2. The design of the M2 Sherman Tank and Navy Logistical Systems are triumphs of clear-headed design. The computational advances at Bletchley Park, including Tommy Flowers' staggering Colossus Computer are great examples of incredibly complex technical tasks done well and quickly. On the other hand, the Mark 14 Torpedo is a fantastic example of multiple layers of engineering boondoggle.

> Military History Online: "Mark 14 Mayhem": "The Bureau of Ordnance (BuOrd) convinced itself that at $10,000 a piece, the Mark 14 was just too expensive to blow up during testing"
>
> Submarine Operational History: "The war began with an entire generation of submarine personnel none of whom had ever seen or heard the detonation of a submarine torpedo."

The longer you wait before beginning to test, the less likely the system will ever work.

> **A&A**: Military systems developers have learned at great cost that a system is much more than a mere collection of components, no matter how well described and specified... the system must consider the complex of all parts working together, not merely the summation of individual units.
>
> > **Dijkstra**: "Testing can show the presence of bugs but not the absence".

Bit of information theory here. We can think of a machine 'succeeding' or 'failing' as it's chance to produce the correct output: i.e, to produce the correct message on a noisy channel. If we compose many machines with independent success rate `M_0`, `M_1`,... `M_N`, where `0 < M_I < 1` then our chance of success is **at best** `(1-M_0)*(1-_M1)*...(1-M_N)`... - this gets real bad, real fast with even a handful of slightly flaky components. This is the **best** case when everything has been designed correctly. They usually aren't. Jimmy Bogard's talk Avoiding Microservices Mega-Disasters is a good illustration of how quickly this can go wrong. It turns out the composition of dozens of ~50ms services does not take ~150ms - it takes seconds *at best*.

> **John Gall:** A complex system that works is invariably found to have evolved from a simple system that worked. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work.

Quality, Testing, and Automation

> **A&A:** The quality of the finished product depends on each part attaining the proper measure values of specific characteristics.
>
> > **John Von Neumann:** There's no sense being exact about something if you don't even know what you're talking about.

> **Daniel Yanklevoitch, "The New Odds", 1971**: It is a short, fatal step from the statement, "There are many intangibles and imponderables that we can't put on our computers," to the statement, "Let's measure what we can and forget about the intangibles."

If you don't have tests for quality, you must assume your product is garbage. On the other hand, what are the right characteristics to test? "correctness" is obvious: performance and portability are less so, but critically important. You must measure both in artificial conditions ("unit" tests) and in combination with other parts ("integration"), as a whole in simlulated conditions ("end-to-end" tests), and in the field with substantial monitoring ("smoke tests").

The best possible test is a pre-existing implementation that is known to be correct.

> **Norbert Weiner**: Control of a machine on the basis of its *actual* performance rather than its *expected* performance is known as *feedback*. It is important that the release for opening the door be dependent on the fact that the elevator is actually at the door.

The other way around is called 'mocking', and it doesn't work very well.

> **A&A:** It is characteristic of all machines that they do not produce *exactly* the same results on each part but work within a range. The narrower the *spread* of the output, the better is the *capability* of the machine. (For low-capability-machines), the operator of the machine is it's control system. He is the agency providing feedback, for he adjusts the machine to minimized the error between the machine output, the dimension he desires. This manual method of measurements and control is good enough when a man can stand by the machine and continually check on it's operation. Otherwise, a large number of defective pieces will be produced.

By this definition, large-language model tools are "low capability" machines. These can be useful but need constant supervision by a person - they *cannot* be used as an intermediate step in a fully automated process. (They *can* be used as part of semi-automated processes, w/ mixed success.)

Mass Production & Materials Handling

> **A&A:** Mass production, using powered machines, succeeded to the point where further improvements in machining operations resulted in negligible time savings because the tasks of *moving and positioning the workpiece between operational stations consumed most of the time.* This led to concentrated efforts to reduce the inter-operation time… the operations are programmed in a fixed sequence and arranged physically to accomplish a smooth *flow* of the work from one operation to the succeeding operation.

This is even more true in software. Most programs are not slow because they can't do operations fast enough, but because they're waiting on data transfer, whether this is waiting for data to go from main memory to cache, loading from disk or network, repeated serialization and deserialization (str to int, int to str, JSON to struct, etc.). Be extremely careful of proposed designs that proliferate APIs, services, and processes: every extra layer adds latency for moving and positioning.

The composition of many good ideas can be a bad one, **even if each idea is good and even if each *pair* of ideas is good**. C++ is probably the canonical example here, where most of it's features are good ideas in isolation but the combination of them is overwhelming and in some cases literally formally undecidable. No offense to Stroustrup and his contributors: they did trailblazing work we can constantly learn from and great software continues to come out in C++ (somehow). We benefit both from our predecessor's successes and their failures.

> **A&A**: >Every improvement in materials handling directly improves automation. This is because the actual machining of a manufacturing process is usually only a small part of the production time. **The major amount of time is devoted to handling materials rather than working on them. Materials handling should be designed with the machine, not improvised later.**

Physical layout is critically important for good throughput in any discipline. When preparing pasta for three dozen, I don't build and store 36 portions separately. Instead, I chop all the onions at once, brown all the meat together, etc. The proper arrangement of fork, knife, spoon, glass, and bowl at the dinner table is one of each per place setting, as "arrays of objects" - it's but that's not how you arrange them in the dishwasher or in the cabinets, where all of one type are stored together ("struct of arrays"). 'Best' is contextual: the use pattern dictates the organization.

> **Nils-Peter Nelson, Bell Labs:** The fastest I/O is no I/O.

Consider that (nearly) every operation in a dynamic or boxed-by-default language involves 'materials handling' in the case of type lookup, even there's no actual dereference, and so does any call to a file or network. It is extremely difficult if not impossible to match the use patterns of your processor in Java, Python, or JavaScript.

This is true in computers too in multiple senses. Internet and disk traffic and context switches are big sources of 'materials handling' delays: cache misses and function calls are smaller ones.

A brief interrupt: materials handling in software

Let's consider some of the sources of "materials handling time" in software overall.

- fundamental complexity: too many "systems"
    - processes

    - programs
    - services
- in the small
    - too many
    - functions calls
    - copies instead of reuse
- (de)compression - failure to compress large communication - unnecessary compression of already-compressed data - unnecessary de-and-re-compression between forwarding stations
- serialization
    - unnecessary in-process serialization
    - inefficient serialization formats
    - slow (de)serialization libraries
    - unnecessary de-and-re-serialization (go: `json.RawMessage` is your friend for forwarding materials with few or minor changes)
- communication overhead
    - machine-level (endianness)
    - protocol-level (TCP frames & handshakes, HTTP frames)
    - application-level (JSON de-and-encoding, string processing)
    - OS-level: context switches,
    - terminating connections rather than re-using them
- memory
    - cache misses
    - fragmented allocation
    - allocations forced on to the caller: see Dave Cheney: Don't Force Allocations on the Callers of your API
    - garbage collection (not always a bad thing!)

Commentary on Anatomy of Automation (2)

a european publishing of anatomy of automation. 'anatomija' is the word for anatomy in a variety of european languages, so I'm not sure which one this is.

Designing for Automatic Production

> **A&A**: The ideal solution to automatic assembly problems is to redesign the product radically, to simplify manufacturing and assembly operations. Trying to assemble automatically products which were originally intended to be put together by hand results in needlessly complex and costly machines.
> **A&A**:It is mandatory for automation that the design of the product and its manufacturing process be coordinated. The goal… is to get the best product from the simplest machinery.

This is best illustrated by one of A&A's wonderful diagrams. I've somewhat spoiled the diagrams by writing on them - I was counting the parts, and so should you.

diagram of 'traditional design', 'design for mechanization', and 'design for automation', using the examples of ice and chairs. each step radically reduces the number of parts

> **A&A**: **The design or redesign of the product for automation is often even more important than the automation machines themselves.**
>
> > Gordon Bell: The cheapest, fastest, and most reliable components are those that aren't there.

The best way to fix most software projects is to delete most of them. Look for overlaps and redundancies. Eliminate middlemen and translation layers. There are too many projects which use six different message queues for a problem that could be solved with zero.

> [Richard Gabriel, "The Rise of Worse is Better"](): It is more important for the implementation to be simple than the interface. **Simplicity is the most important consideration in a design.** It is slightly better to be simple than correct.
>
> **Brian Kernighan & Rob Pike, "The Unix Programming Environment"**: (On representing text files as 'raw bytes'): For most purposes, this simple scheme is exactly what is wanted. When a more complicated structure is needed, it can easily be built on top of this; the converse, creating simplicity from complexity, is harder to achieve.
>
> **Brian Kernighan & Rob Pike, "The Practice of Programming**: No matter what, there is a limit to how well we can do in designing an interface. Even the best interfaces of today can become the problems of tomorrow, but good design can push tomorrow off a while longer.

[Production Fundamentals and Folly]()

> **A&A**: The armed forces engineers learned, after spending vast sums of money, that a complex system will not work successfully when made up of independently designed components however carefully the individual units are built… experience has proved that unanticipated interactions degraded the expected effectiveness.

This lesson is learned over and over again by companies attracted by the siren songs of outsourcing and microservices. The prize of doing something cheaper, faster, and in parallel is so tempting that no amount of experience or advice seems to be proof against it. Very carefully designed interfaces and specifications can help somewhat - "design to interface" - but even there you should expect testing failures and significant hurdles during integration. It is better to remove a feature, or simplify it to the point where you can handle it 'in-house', then to outsource components.

> **A&A**: A *new model* product requires extensive retooling of an automatic production system. That is, much of the basic machinery can often be reused, but all cutters, tool holders, work holders and dies must be changed to accommodate the new product. Manual work with simple power tools doesn't have this restriction to model change. **This points out that high productivity is commonly the antithesis of flexibility.**

High 'productivity' can sometimes be actively harmful. Consider the war machines of France and Germany prior to World War 1. From one point of view, these are some of the most 'rational' and effective organizational and logistic systems ever made by human hands: they made war on an unimaginable scale on time not just possible but, due to that power coming at the expense of flexibility, nearly inevitable. Be careful what machines you build.

> **Stafford Beer**: The manager wants information, not facts, and facts become information only when something is changed… again we are concentrating on slick ways of doing things rather than on *what* we do. What is the user of the ever-faster, ever-slicker, more nearly perfect implementation of rotten plans?
>
> Daniel Yankelovich, "The New Odds", 1971: The first step is to measure whatever can be easily measured. This is okay as far as it goes. The second step is to disregard that which can't be easily measured or give it an arbitrary quantitative value. This is artificial and misleading. The third step is to presume that what can't be measured easily really isn't very important. This is blindness. The fourth step is to say that what can't be easily measured really doesn't exist. This is suicide.

If you don't know what you want yet, don't automate it.

[Productivity vs. Flexibility in Computer Sceince]

- interpreted vs compiled
- statically typed vs. dynamically typed
- 'plain text' vs JSON vs structured binary data
- local vs cloud deployment
- familiar language / "best" language

> **A&A:** Any procedure which permits a product to be manufactured in fewer operations is of definite benefit. In other words, it is frequently desirable from the standpoint of both cost and process simplicity to make use of materials that have already undergone substantial pre-working.

Software has the unique advantage that a product once manufactured can be re-used forever with no wear and tear. Don't just reuse the final artifact - cache early, cache

often. This doesn't need to involve fancy things like redis or databases: just find common subexpressions and save them in a variable.

View Points

> **A&A**: When planning a machine, first consider solely the functions of *what* the machine is to do. By considering function only at first and by making use of block diagrams and other models, complex manufacturing processes and control systems can be worked out to an appreciable degree without any need to take specialized engineering details into account. Functional thinking is an attitude of mind which avoids pre-judgement and makes possible novel solutions to attain the desired results. The functional approach is less costly too, for the further a development can proceed before detailed designs are prepared and machines built, the less is the risk of wasted effort.
>
> > **Clausewitz, "On War"**: "No one starts a war—or rather, no one in his sense ought to do so—without first being clear in his mind what he intends to achieve by that war and how he intends to conduct it.". You should know - in general if not in detail - both the means and the ends of your task before you actually do it.

You have to know what you want. Too often the technology is chosen for reasons of popularity or perceived effectiveness before *what the technology is supposed to do* is known.

> **A&A**: Machines must be designed concurrently with their measuring, actuating, and control systems.

You have to know the expected *outputs* ahead of time, but the correct inputs - that is, the API or control systems - are usually discovered experimentally. These changes often 'feed back' and require further tweaks and iteration. Good APIs are the result of experimentation rather than brilliance. Ruthelessly iterate until you have the simplest API that solves your problem (but no simpler).

> **A&A**: The adoption of automation is one way to improve production, but it certainly is not the only way nor is it the only one that should be considered. Other production improving methods must be investigated and adopted first because they not only aid production directly but pave the way for maximum automation benefits. A redesign of the product for the sake of simplifying production is always advisable, as deficiencies and *temporary* measures have a way of being perpetuated.
>
> > **National Aeronautics and Space Administration (NASA):** Apollo 11 Source Code, 1969:
> >
> > ```
> > # <- this is a comment
> > # everything following is from the original source code that got u
> > ```

```
# Page 801
CAF     TWO             # WCHPHASE = 2 ---> VERTICAL: P65,P66,P67
TS      WCHPHOLD
TS      WCHPHASE
TC      BANKCALL# TEMPORARY, I HOPE HOPE HOPE
CADR    STOPRATE# TEMPORARY, I HOPE HOPE HOPE
TC      DOWNFLAG# PERMIT X-AXIS OVERRIDE
ADRES   XOVINFLG
TC      DOWNFLAG
ADRES   REDFLAG
TCF     VERTGUID
```

Rob Pike & Brian Kernighan, The UNIX Programming Environment: Ken Thompson was once asked what he would do differently if he were redesigning the UNIX system. His reply: "I'd spell `creat` with an e."

Human Engineering

> **A&A**: The policy of recognizing man's limitations and assuring that his abilities are not exceeded by the job requirements is termed *human engineering*. Any easing of man's tasks, even if within his range of capability, usually results in better work. Even simple refinements of the arrangement and form of control levers can be a big help. The easier it is for the operator to physically adjust the machine mechanisms, the more precise and faster will be his use of the machine. It is not always practical or desirable to go all the way to full man-replacing automation, but it is always desirable to help him do his job better. Lesser forms of automaticity are a big help too.

Reducing latency in any subtask tends to make those subtasks more reliable even when it does not speed them up.

Be on the lookout for minor automation activities. A regular expression or small program can often save you a lot of typing, as can the column-edit mode in your editor. The choice of equipment, both hardware (CPU, ram, mouse and keyboard, monitor, etc) and software (keyboard *layout*, text editor, operating system, etc) matters significantly. For example, typing speed and accuracy are greatly improved by lower-latency keyboards and monitors.

> **A&A**: For man to work most effectively, he must be considered part of the complete system in early design systems, his tasks made as simple as possible. He must not be burdened with unnecessary details. Information must be easy to understand and to act upon. The operator must have immediate knowledge of the effects of his actions.

Latency and ambiguity are death to quality.

> **A&A**: Man seems to work best as a simple amplifier. This means that if all the necessary information is easy for him to read, the control devices he

must manipulate are of good form, and he is given power assists, he can do the best job. If he must actively seek out the necessary information, his work is certain to be compromised.

A&A: Helping a human operator of a complex system to be aware of what is occurring at all times, instead of forcing him to seek out the information, makes him much more effective.

I can think of no better way to illustrate this than error messages.

Compare three error messages from `go`, `rust`, and `postgresql` on encountering an unexpected comma during compilation:

```
//go compiler: error message
failmult.go:21:52: syntax error: unexpected comma, expected expression
```

```
// rust compiler: error message
src/main.rs:3:16 error: expected one of '.', ';', '?', '}' or an operator, found ','
```

```
-- psql: error message
sql: procmon_insert.sql:27:4: syntax error at or near ")"
```

What is the syntax error? Any professional programmer who's written much SQL spends a dozen hours a year tracking down unnecessarily cryptic errors.

A&A: As much as possible, the controls for each section or station should be localized at that location. Simplified trouble-shooting diagrams, sequence of operation lists, fault detectors, test measurement points, and tables of normal readings should be provided at each material or section.

An error message should tell you not only what has gone wrong but what 'right' is. Most good diagnostics look like this:

```
# good error message
big task: small task (#3/12): expected a greeble ("b"), but got a grobble ("bb")
```

But in practice they usually look like this:

```
# bad error message
ValueError: nogreeble
```

Automatic Measurements

A&A: Measurements are absolutely necessary if any process is to be corrected. The machine which is to be automatic and capable of self-correction must have the measurement information.

At a bare minimum, this means keeping track of your error rates and performance (p95, p99, p99.99 latency, throughput, memory usage, CPU usage) in action. The competent programmer should know how to use a profiler in their language of choice, like go's `net/http/pprof` or python's cProfile, and a system-level event profiler to track system calls, network access, and other likely slow spots. For Linux, `strace` and `perf` are your friends, as is Window's`procmon`. A sophisticated engineer will measure not only the end product but each step of tooling and production. (I cover how to use `pprof` in Starting Systems Programming 3: Performance Counts: that's as good of a place to start as any).

Programming

> **A&A**: Programming ordinarily means devising the step-by-step operations necessary for automatic control - the *instructions to the machine*. Programming non-automatic jobs is valuable too - machine time is wasted while (people) make decisions, and the decisions must be remade every time. A program of preplanned instructions eliminates decisions and delays on the factory floor. Tell the machinist what to do - which machine, tool, speed, and cut to use, step by step. Make his decisions for him, and he will be able to do more and better work.

There are more kinds of programming than computer code. Any kind of large operation - whether that's a military maneuver or a Hollywood wedding - must be extensively programmed in order to go off smoothly. Any kind of decision point is, from the point of view of efficiency, 'wasted' time. On the other hand, the wrong result arrived at quicker and faster is still wrong.

Existing Machines vs. New Machines

> It appears to be good sense to use existing machines as much and as long as possible. First cost, though, is not final cost. The only valid cost of a machine is its cost per item produced, including all maintenance: suggestions: maintenance, production labor, and indirect costs. In this light, old machines may be too expensive. Only a thorough cost analysis can tell the true story.

Some tools and machines are *so bad* that they immediately contaminate everything they touch. Anything on Azure springs to mind, as does any one of the tens of thousands of "shell scripts written in YAML" orchestration tools.

Planning

> **A&A**: *Planning* is the key to all successful automation systems. Many firms buy machines as needed but select each machine separately on the basis of operational requirements (for that individual machine) alone. It is better to make up a long range plan or procedure so that all machines bought will be

compatible as much as possible. This can only be done if the long view is taken.

> **Eisenhower**: Plans are useless, but planning is indispensable.

It's not really planning if you don't *write things down*. Not because you intend to use those plans as written in the future - that's too rigid - but because it forces you to agree on what the plan means in the present. That is, it *should* force you to agree: be extremely wary of quick agreement when making important decisions: it generally signals that one or more of you don't understand each other or the decision.

When I talk about planning, I mean **operational** or **tactical** planning, not strategic - working out an implementation for an already defined strategic goal.

> **A&A**: Many jobs that appear non-routine on the surface can be *routinized*, making them amenable to automatic methods.
>
> > **Norbert Weiner**: The chief criterion as to whether a line of human effort can be embodied in a game is whether or not there is some objectively recognizable criterion of the merit of the performance of this effort.
> >
> > **Niklaus Wirth**: Good engineering is characterized by gradual, stepwise refinement of products that yields increased performance under given constraints and with given resources.
> >
> > **Rob Pike & Brian Kernighan, "The Practice of Programming"**: One way to write bug-free code is to generate it by program. If some programming task is understood so well that writing the code seems mechanical, then it should be mechanized.

If it can be measured, it can be routinized. If it can be routinized, it can be automated. Be careful that you measure the right thing: correlation is not causation.

The office environment

Executives

> **A&A**: Contrary to public opinion, executives do get ideas sometimes. Occasionally, something results form their contacts on the golf course, from their attendance at conferences, and from their membership in trade associations, that bears on the potential automation of their plant. No matter who contributes the key concepts to the proposed automation process, the executive must make the final decisions. Hence, he must keep informed as to the advantages and shortcomings of all proposed actions.

The boss is not always right, but he is always the boss. Executives live in a very different world from engineers and must manage relationships and make highly important decisions in short amount of times, based off short bursts of conflicting information mixed with nonsense. This is a 'hostile environment' and given the circumstance, it's not too surprising that most executives don't do a very good job at it.

As an engineer you need to try and help your executives: do that by protecting your engineers from their own overly clever ideas and from the execs' meddling. 'Managing up' is one of the subtlest and most difficult tasks of the engineer. I wish I were better at it and I wish I had better advice on how to do it.

Workmen

> **A&A**: In most cases, workmen will not suggest big scale or radical changes. Their greatest potential contributions are for spot improvement of existing processes… their aid is just as valuable after the new equipment has been installed and is undergoing shake-down. Some pointed comments from workmen on how to eliminate "bugs" can be of great value during this critical period.

Even better than asking the workmen how they use something or how they'd improve it - and you should do both - is *watching them use it without commentary*. People do not always know how to express what they think, and even when they do, they may know better than to say it where you can hear it. (Most people have had enough jobs to realize the boss doesn't want to hear the truth, especially when he looks you in the eye and says he does.) People's hands and eyes are simpler and more honest than their mouths.

When to automate?

> **A&A**:The basic technique for making machines automatic consists of setting up a program for them to follow.
>
> > Norbert Weiner, **GOD & GOLEM, INC**: In general, a game-playing machine may be used to secure the automatic performance of any function if the performance of this function is subject to a clear-cut, objective criterion of merit… there may be great doubt as to how to win a game, but not doubt whatsoever as to whether it has been won or lost.
>
> **A&A**: The best work man can do with his intellect is to raise the information potential of work tasks by routinizing them. He can then give the repetitive work to machine.
>
> **A&A:** If big repairs or improvements are needed, it may be a better move to make a big change to automation. It is seldom practical to replace a machine

with an operationally similar one. Upgrading is usually a a better course to follow.

If and when you *do* upgrade, follow the practice of memory allocators and scale up by orders of magnitude. If your new solution isn't *at least* twice as good, it's probably not good enough to switch.

> **A&A**: Disasters, such as a fire or flood, may also constitute a natural break in production when automation can be introduced with little additional inconvenience.
>
> > **Nitze, Weaver & Dickerson: Crisis Engineering**: A ripe crisis will exhibit failure of sensemaking - perceptions break down, existing maps and models don't work.

If you know there is a deeply important change to make and you can't get buy-in in the normal ways to make it, prepare the skeleton of your solution and wait for the crisis. When their models break down, yours will be ready. Emergencies are quite expensive: it's a shame to waste one.

> **A&A**: Probably the best criterion for evaluating man versus machine is dollars. Demand a big pay-off. Small potential savings by the use of automatic devices or machines mean that the change is not justified. Only if the estimated yearly saving is great (a quarter of the cost of the improvement) should the proposal be further considered
>
> > **Stafford Beer**: Too many managers have been dazzled by the 'more and quicker argument', with the result that little fresh thought has been given to the purposes which the information duly handled is supposed to serve.

The cost is as least as important as the benefit and probably significantly more so. The benefit may occur: the cost will definitely occur and will usually be more expensive than you expected.

> **A&A** Full automaticity has its place, but partial automatic methods are also valuable in their own right.
>
> **A&A: Being aware of cybernetic information efficiency and choosing to ignore it if unimportant is quite different from not being aware of it.**
>
> > **Ernest Hemingway**: If a writer of prose knows enough of what he is writing about he may omit things that he knows… A writer who omits things because he does not know them only makes hollow places in his writing.

We should not be ashamed by leaving out good work. We should be ashamed of putting in things that serve no purpose even if they are correct.

> **A&A**: For the most part, this text cites advantages obtainable from the use of automatic methods. It could therefore be inferred that this constitutes complete endorsement of automaticity as a general rule, and that automation is practically always desirable. This is not so. Objectivity must be retained. **First decide what a system is to do, then decide whether full automation, semi-automation, or a human operator is best.** The advantages of mechanization methods must be weighed against their shortcomings. The flexibility and judgement of a human operator are generally superior to a machine. Properly incorporated into a system, a man can be an efficient *control element*, often a real bargain compared to equivalent mechanisms. The cost of controls to replace man may be far greater than the cost of training an operator. Unless the cost can be amortized over a great volume of production, man may be the better choice.
>
> **A&A**: The decision whether or not to automate, and to what extent, must be based on a knowledge of the facts. This requires active efforts but is unavoidable.

If you can't write a test that confirms your solution, you don't know what your problem is. If you don't have a measurement of your current costs, you can't estimate savings from potential savings. The promise of automation is not enough: have real numbers, and resist the temptation to overinterpret noisy or ambiguous data.

A Brief Interruption: `thumbtoe` and the automation within this website

Let's break out of the Talmudic mode here for a second and write some code to demonstrate what grandpa means. This article contains a bunch of high-resolution photos which balloon the file size. I want my website to load **immediately**, so we can't have any large photos eagerly initialized. One way to solve this problem is by making **thumbnail images** - small, low-quality versions that load fast - that are themselves links to full-size images.

A manual algorithm for making thumbnails might look something like this:

- look for 'large' photos (say, over 320kib)
- load up `paint.net` for each photo
- click the resize button
- save as `imagename.thumb.png`
- change `![image text](link/to/image.png)` to `[![thumbnail text](link/to/image.thumb.png)](link/to/image.png)` (that is, change an image to an image *that is also a link*.)

This is a tedious manual process I'd have to do for article after article. It takes about ~45s/image, and I hate doing it.

Let's propose some automations for this process. First, what are our goals?

- We want every LARGE image linked in my markdown article to instead be an inline SMALL image (the 'thumbnail').
- Let's define 'LARGE' as >32KiB.

This means a full solution will involve all of

- finding 'large' images
- creating the thumbnails for images that don't already have them
- processing the markdown to replace those inline LARGE with inline SMALL thumbnails that link to the large ones.

Let's rephrase this in industrial engineering terms, like grandpa would.

1. automated selection: write a tool to find images that need thumbnails.

2. semi-automation: build a tool that automates making thumbnails from source images (while I was at it, i could combine it with another simple image-size-reduction techniques at the same time: dithering. **Tanner Helland's Image Dithering: Eleven Algorithms and Source Code** will teach you everything you want to know about that)

3. automated manufacturing of selected parts: write a shell script to create thumbnails for all my articles

4. automated assembly of finished products: make sure the finished HTML has thumbnails that link to the full-size images. This one sounds a bit trickier, since it will involve either text substitution (which can break) or Abstract Syntax Tree manipulation of the HTML or Markdown layers (often tricky, might need libraries).

It seems to me as though automating everything *up to but not included assembly* is the sweet spot here. Let's try that and we can move on to the tricky bit if we need it.

(Non-programming readers may want to skim this part, but I think it's pretty important!)

Automated Selection with `largeimages`

We could use `fd` or `find` or something for this, but it's easy enough to write a program. Let's do that.

```
1   // print large images. an image is a file that ends
    with .png, .jpg, or .jpeg and is larger than the size flag
    (default 320k).
2   // example: find images > 1MiB in the current directory and
    print their paths:
3   //
4   //  go run ./cmd/largeimages -size 1000000 .
```

```
 5  package main
 6
 7  import (
 8      "flag"
 9      "fmt"
10      "io/fs"
11      "log"
12      "os"
13      "path/filepath"
14      "runtime"
15  )
16
17  const usage = `Usage: %s [-size=320] DIR`
18
19  var size int
20
21  func init() {
22      log.SetFlags(0)
23      flag.IntVar(&size, "size", 320*1024,
    "minimum file size in bytes to consider an image 'large',
    mandatory")
24      flag.Parse()
25  }
26  func main() {
27      if size <= 0 {
28          fatalf("invalid -size flag: must be a positive
    integer")
29      }
30      if flag.NArg() != 1 {
31          fatalf("invalid number of arguments: expected 1,
    got %d", flag.NArg())
32      }
33
34      err := filepath.WalkDir(flag.Arg(0), printLarge)
35      if err != nil {
36          fatalf("walk directory: %v", err)
37      }
38
39  }
40
41  // print large images. an image is a file that ends
    with .png, .jpg, or .jpeg and is larger than the size flag
    (default 320k).
42  func printLarge(path string, d fs.DirEntry, err error)
    error {
43      if err != nil {
44          return err
45      }
46      if d.IsDir() {
47          return nil
48      }
49      switch filepath.Ext(d.Name()) {
50      default:
51          return nil
```

```
52          case ".png", ".jpg", ".jpeg":
53              // ok, it's an image. how big?
54              stat, err := d.Info()
55              if err != nil {
56                  return err
57              }
58              if stat.Size() >= int64(size) {
59                  fmt.Println(path)
60              }
61              return nil
62          }
63
64      }
65      func fatalf(format string, args ...any) {
66          log.Printf(usage, os.Args[0])
67          _, file, line, _ := runtime.Caller(1)
68
69          log.Fatalf("%s:%d %s", filepath.Base(file), line,
        fmt.Sprintf(format, args...))
70      }
71
```

Let's test it:
IN:

```
1   #!/usr/bin/env bash
2   # print images larger than 100kib, ignoring the directory
    server/static and get their sizes. then sort by size,
    largest first.
3   du -h $(go run ./cmd/largeimages -size 102400 . | grep --
    invert-match server/static) | sort -hr
```

OUT:

```
 1   12M    articles/swtalmud/aa_designing_for_automation.png
 2   11M    articles/swtalmud/office_automation_chart.png
 3   8.1M    articles/swtalmud/aa_cover.png
 4   660K    articles/startingsystems/waititsall.png
 5   576K    articles/librarydesign/ginbad/
     ginbad-httpresponse-statusline.jpg
 6   532K    articles/librarydesign/ginbad/
     ginbad-httpresponse-structure.jpg
 7   460K    articles/librarydesign/ginbad/
     ginbad-blackboard-requestline-example.jpg
 8   456K    articles/swtalmud/aa_crisis_engineering.png
 9   448K    articles/librarydesign/ginbad/
     ginbad-httprequest-structure.jpg
10   448K    articles/librarydesign/ginbad/
     ginbad-httprequest-example.jpg
11   416K    articles/librarydesign/ginbad/
     ginbad-json-structure.jpg
12   352K    articles/startingsystems/unix_geneaology.png
13   264K    articles/startingsystems/microsoft_geneaology.png
14   184K    articles/librarydesign/kudzu/gin_architecture.png
```

```
15   184K    articles/librarydesign/ginbad/gin_architecture.png
16   172K    articles/console/tt_concept_art.png
17   144K    articles/librarydesign/kudzu/
     http_stdlib_overview.png
18   144K    articles/librarydesign/ginbad/
     http_stdlib_overview.png
19   124K    articles/swtalmud/office_automation_chart_thumb.png
20   124K    articles/swtalmud/
     aa_designing_for_automation_thumb.png
21   116K    articles/swtalmud/aa_cover_thumb.png
```

This grabs our images. Let's see if we can automate creation of thumbnails with a small program , too.

semi-automation with `thumbtoe`

`thumbtoe` is a go program that makes thumbnails from images.

usage

```
thumbtoe -src:string -dst:string -size:int [-force] [-dither]
```

help

IN:

```
1   #!/usr/bin/env bash
2   thumbtoe --help
```

OUT (STDERR)

```
 1   Usage of thumbtoe:
 2     -dither
 3           dither the thumbnail using floyd-steinberg
     dithering to reduce color banding (default true)
 4     -dst string
 5           the path to the destination thumbnail, mandatory
 6     -force
 7           force overwrite of the destination file if it
     already exists
 8     -size int
 9           maximum allowed size of the thumbnail's longest
     dimension, mandatory, must be >= 40
10     -src string
11           the path to the source image, mandatory
```

Why `thumbtoe`?

JR said it wasn't a real word. She's right: It *wasn't*.

Source Code

```
// you can skip this if you want. it's not relevant to the rest of the article, but I do think it is very good code.

func main() {

    var (
        size   = flag.Int("size", 0, "maximum allowed size of the thumbnail's longest dimension, mandatory, must be >= 40")
        src    = flag.String("src", "", "the path to the source image, mandatory")
        dst    = flag.String("dst", "", "the path to the destination thumbnail, mandatory")
        force  = flag.Bool("force", false, "force overwrite of the destination file if it already exists")
        dither = flag.Bool("dither", true, "dither the thumbnail using floyd-steinberg dithering to reduce color banding")
    )
    flag.Parse()
    log.SetFlags(log.Lshortfile)
    switch {
    case *src == "":
        log.Fatal("missing required -src flag")
    case *dst == "":
        log.Fatal("missing required -dst flag")
    case *size < 40:
        log.Fatal("invalid -size flag: must be >= 40")
    }

    if err := run(*src, *dst, *size, *force, *dither); err != nil {
        log.Fatal(err)
    }
}

func run(src string, dst string, size int, force, dither bool) error {
    r, err := os.Open(src)
    if err != nil {
        return fmt.Errorf("open source file: %v", err)
    }
    defer r.Close()

    _, err = os.Stat(dst)
    if err != nil && !os.IsNotExist(err) {
```

```
37          return fmt.Errorf("check destination file: %v",
    err)
38      }
39      if !force && err == nil {
40          log.Printf("skip: -dst %s already exists; use
    -force to overwrite", dst)
41          return nil
42      }
43      w, err := os.Create(dst)
44      if err != nil {
45          return fmt.Errorf("create destination file: %v",
    err)
46      }
47      defer w.Close()
48
49      // decode source file and create thumbnail
50
51      img, _, err := image.Decode(r)
52      if err != nil {
53          return fmt.Errorf("decode source file: %v", err)
54      }
55
56      thumb, err := shrink(img, size)
57      if err != nil {
58          return fmt.Errorf("make thumbnail: %v", err)
59      }
60
61      b := thumb.Bounds()
62      X, Y := b.Dx(), b.Dy()
63
64      // draw a border around the thumbnail to make it clear
    that it's a thumbnail and not the original image.
65      // we just walk the pixels on the border and set them
    to a light gray color. This is pretty fast and looks fine
    for a thumbnail.
66      const BORDER_SIZE = 5
67      gray := color.Gray{Y: 0xC0}
68      for x := range X {
69          for y := range BORDER_SIZE {
70              thumb.Set(x, y, gray)
71              thumb.Set(x, Y-y-1, gray)
72          }
73      }
74      for y := range Y {
75          for x := range BORDER_SIZE {
76              thumb.Set(x, y, gray)
77              thumb.Set(X-x-1, y, gray)
78          }
79      }
80
81      if dither {
82          draw.FloydSteinberg.Draw(thumb, thumb.Bounds(),
    thumb, image.Point{})
83      }
```

```
 84
 85      return png.Encode(w, thumb)
 86  }
 87  func shrink(src image.Image, size int /*opts Options*/)
     (draw.Image, error) {
 88      if size <= 0 {
 89          return nil,
     fmt.Errorf("invalid size %d: must be > 0", size)
 90      }
 91      b := src.Bounds()
 92
 93      // scale thumbnail
 94
 95      x, y := b.Max.X, b.Max.Y
 96
 97      d := float64(max(x, y)) / float64(size)        //
     scale factor to get max(x, y) <= size
 98      X := int(float64(x) / d)                       // max
     x of scaled thumbnail
 99      Y := int(float64(y) / d)                       // max
     y of scaled thumbnail
100      thumb := image.NewRGBA(image.Rect(0, 0, X,
     Y)) // draw the thumbnail
101
102      // scale it down using approximate bilinear
     interpolation, which is pretty fast and fine for a
     thumbnail.
103      draw.ApproxBiLinear.Scale(thumb, thumb.Rect, src,
     src.Bounds(), draw.Over, nil)
104
105      return thumb, nil
106  }
```

Let's see if we can combine our two previous programs to automate the creation of thumbnails for all the large images in my articles.

3. Automated Manufacturing of Selected Parts

```
 1  #!/usr/bin/env bash
 2  # makethumb.bash - make thumbnails for all png and jpeg
    images in my articles that are larger than 32k
 3
 4  if [ "$#" -ne 2 ]; then
 5      echo "Usage: $0 <src> <dst>" 2>&1
 6      echo "Example: $0 articles server/static/thumb" 2>&1
 7      exit 1
 8  fi
 9
10
11  images=$(go run ./cmd/largeimages $1)
12  for img in $images; do \
13      src="$img"
```

```
14      dst="$2/$(basename "$img")"
15      # don't spin up the go runtime for thumbnails that
    already exist, tha'ts wasteful.
16      if [ -f "$dst" ]; then
17          echo "SKIP $src -> $dst: already exists" 2>&1
18          continue
19      fi
20      # run the thumbnailer as a separate background job for
    each image so we aren't blocked on IO. this makes this
    about 18x as fast on my machine: not bad for a '&' and
    'wait'!
21      go run ./cmd/thumbtoe -src "$src" -dst "$dst" -size
    32768 &
22  done
23  wait # wait for all background jobs to finish
24
25
26
```

I stuck the new code in my makefile as part of my asset generation step and it 'just works' and runs in less than a tenth of a second if there's no new images.

Makefile diff

```
1   diff --git a/Makefile b/Makefile
2   index 20afc5c..3e3f68f 100755
3   --- a/Makefile
4   +++ b/Makefile
5   @@ -9,11 +9,14 @@ clean:
6    deps:
7       # make deps
8       which flyctl || ./deps/install-fly.sh
9   +
10   generate: clean deps
11      # --- make generate ---
12      git rev-parse HEAD > server/commit.txt || true # add
    current commit to server logs
13  -   go run ./cmd/rendermd . ./server/static # generate
    static html from markdown
14  -   go run ./cmd/prezip ./server/static > ./server/static/
    assets.zip # zip up all of the assets
15  +   time ./makethumb.bash ./articles ./server/static/thumb
    # make thumbnails for all large images in articles
16  +   # TODO: automated process to change markdown image
    links to be thumbnails that point to the original image, so
    that the server can serve the original image when the
    thumbnail is clicked
17  +   time go run ./cmd/rendermd . ./server/static # generate
    static html from markdown
18  +   time go run ./cmd/prezip ./server/static > ./server/
    static/assets.zip # zip up all of the assets
19
```

```
20
21   deps:
22  @@ -22,8 +25,8 @@ deps:
23       go mod download
24
25       # ---- pandoc & weasyprint are used for generating pdfs
    from markdown
26  -    command -v weasyprint || apt-get install -y weasyprint
    || brew install weasyprint
27  -    command -v pandoc || apt-get install -y pandoc || brew
    install pandoc
28  +    @command -v weasyprint || apt-get install -y weasyprint
    || brew install weasyprint
29  +    @command -v pandoc || apt-get install -y pandoc || brew
    install pandoc
30
31   testrun: generate
32       # --- make testrun ---
33  @@ -71,3 +74,4 @@ deploy: clean generate build quickdeploy
34       # --- make deploy ---
35       @ echo "Deployed to https://eblog-test.fly.dev"
36
37  +
38
```

4. Automated Assembly of Finished Products

Let's examine what this problem would look like before we decide whether to do it. I can think of three ways to do it, each with their own trade-offs: textual replacement in the markdown, manipulation of the markdown AST, or manipulation of the HTML AST after rendering the markdown to HTML.

textual replacement:

- look for markdown that looks like `![image description](path/to/large/image.png)`
- replace it with `![image description](path/to/large/thumb/image.png)`
- and then turn that into a link: `[![image description](path/to/large/thumb/image.png)](path/to/large/image.png)`

This is not too hard to do but is likely to be fragile and break in unexpected ways.

markdown AST manipulation:

- Parse the markdown into an AST
- Find all image nodes

- Replace them with link nodes that contain image nodes.

This is a bit fiddly, but probably what I'd go for, since I'm pretty familiar with markdown.

HTML AST manipulation:

- Render the markdown to HTML
- Re-parse the HTML
- Do the above replacement, either textually or as an AST transformation.

This might be the best option, since the **end product is always HTML**. Rather than fiddling with the intermediate product, we can just make sure the final product is correct.

None of the above: Lazy Loading

The framing I presented above showed thumbnails as The Solution to a problem (large images). But the problem we have is not large images, it's **eagerly loaded large images**. If we can prevent eager loading, this also fixes the problem. Modern browsers support lazy loading of images, which only starts fetching it when the user's browser gets close.

We are now left with an important dilemma: **should we add thumbnails to eblog**, **lazy-loading**, **both**, or **neither?**

How much solution should I have for my problem, and which one?

Commentary on Anatomy of Automation (3): Features: When to stop?

a japanese printing of 'Anatomy of Automation'

> **A&A**: Excessive model variations, inadequate reliability, lack of standardization are typical indications that value analysis could help to "get the most for the money". Simplifications of the product and its parts are not an automation activity, but simplification is always desirable. Conventional production deserves it.
>
> **A&A**: There has always been a conflict between the production man's preference for the unvarying product, and the customers' demand for *specials* and for custom treatment.

Bjarne Stroustrup, "How Can you Be So Certain"?: There are two fundamental ways of approaching the unavoidable uncertainty in a design

- add "improvements" until everybody feels well served
- cut until there is nothing left to cut and all there is left is principled and fundamental

Rob Pike & Brian Kernighan: "Program Design in the Unix Environment, or `cat -v` Considered Harmful": Such a modification (`cat -v`) confuses what cat's job is - concatenating files - with what it happens to do in a common special case: showing a file on the terminal. A UNIX program should do one thing well, and leave unrelated tasks to other programs. Cat's job is to collect the data in files. Programs that collect data shouldn't change the data; cat therefore shouldn't transform its input. **The guiding principle for making the choice should be that each program does one thing.**

Niklaus Wirth: A primary cause of complexity is that software vendors uncritically adopt almost any feature that users want.

Gordon Ramsay: Hundreds of items on a Menu means the restaurant specializes in nothing.

Kernighan & Pike, The UNIX Programming Environment: (re: the design of a calendar program): 'If it's now December, and you ask for `cal jan`, should you get this year's January or next year's? **When should you have stopped adding features to `cal`?**

Back to `thumbtoe`

On that note, when should we have stopped adding features to `thumbtoe`? The answer is **right now**. Despite all that work, let's omit thumbnails and stick with lazy-loading. **Happily, we built a semi-automated solution first** - `thumbtoe`, which is totally freestanding. If we want or need to use it later, we can - and we can use it for thumbnails for completely different projects if we want to. It is better to omit good code than to ship needless code.

Commentary on Anatomy of Automation (4): The Engineer's puporse

> **Antoine De Saint Expry**: In anything at all, perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.
>
> **Strunk & White, Elements of Style, 1959**: A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts.
> **A&A**: The systems engineer must be in the employ of the **user**, either on a permanent or assignment basis. In any event, he must not have any pecuniary ties to the vendors and manufacturers of machinery and equipment used for the system… just as the architect may not deal in building materials, so the persons with over-all systems responsibility may not ethically be sponsored by the manufacturers of equipment.
>
> **Frank G Woolard, as quoted by Amber & Amber:**
>
> - **Mass production products must be simplified, both in general and in detail.**
> - Operations must be based on motion analysis and time study.
> - `...omitted...`
> - The mass production system employed, be it manual, mechanical, or automatic, **must benefit everyone - customers, workers, and owners.**
>
> **Stafford Beer, "The Brain of The System":** Anything which inflames the appetite for empire building not only becomes a vice, but detracts from the issues which ought to be discussed.

Far too many corporate software decisions are made to satisfy the whims of the investors than the user. This may be profitable, but it makes for bad software. The second any software places a hurdle in the way of ordinary usage - whether that's a pop-up ad, the removal of something that was previously available for free, or forced internet connectivity - the system is no longer being engineered for the user and the quality plummets.

> **A&A**: We have the ironic condition that man is not predictable when he does what he alone can do, acting extemporaneously. Therefore, if extreme dependability is needed and no human lapses can be tolerated (such as piloting a transcontinental jet aircraft), then as little as possible should be left to man.

> **Norbert Weiner, GOD AND GOLEM, INC**: (on the *wrong* reason to automate): There is one motive which it is harder to establish in any concrete case, but which must play a very considerable role nevertheless. It is the desire to avoid the personal responsibility for a dangerous or disastrous decision by placing the responsibility elsewhere: on chance, on human superiors and their policies which one cannot question, or on a mechanical device which one cannot fully understand but which has a presumed objectivity.

This is a stronger temptation than of us would like to admit. "There's nothing I can do" usually means there's nothing you *will* do.

Commentary on Anatomy of Automation: Conclusion

> **A&A**: Neither men nor machines are well utilized at present. Man can work with less physical effort than he is now doing, and do far more. Most machine work is also performed at less than the peak capability.

> > **Dan Luu, "Everything is Broken"**: If I had to guess, I'd say I probably work around hundreds of bugs in an average week, and thousands in a bad week. It's not unusual for me to run into a hundred new bugs in a single week.

> > **Norbert Weiner, GOD AND GOLEM, Inc:** Render unto man the things which are man's and unto the computer the things which are the computer's.

> > **Norbert Weiner, "The Human Use of Human Beings"**: We live on this earth… in a very real sense we are shipwrecked passengers on a doomed planet. Yet even in a shipwreck, human decencies and human values do not necessarily vanish, and we must make the most of them. We shall go down, but let it be in a manner to which we may look forward as worthy of our dignity.

If we're going to live in a world dominated by software, it's the least we can do to build it well. The software around us is largely half-baked if not actively hostile, but this is not an inevitable fate. After all, good engineering is out there - most of our lives happen beneath the quiet hum of reliable, efficient, and safe engineering on a massive scale, the power plants, factories and water treatment facilities ran by the casually competent. While all these institutions are far from perfect, they are fundamentally reliable in a way contemporary software is not. The average person does dozens of workarounds daily for the software in their daily life that they simply would not tolerate in, say, a flush toilet or lightbulb. If we wrote **simpler, less buggy software** we could use these wildly powerful computers to make our lives easier instead of harder.

Maybe this article will help. Thank you for reading.

back inside cover art: precursors of modern technology leading, presumably, to A&A

> **Frank Herbert, "Dune"**: Arrakis teaches the attitude of the knife - chopping off what's incomplete and saying: "Now it's complete because it's ended here".

Bonus Material

A Personal Note about A&A

paul & george amber, A&A themselves

It's not a coincidence that "Anatomy of Automation" would show up at the front of any index or publisher's catalog. Paul & George chose the name of the book - and the name of the authors. They weren't born Paul and George Amber: they were Haim and Sholem Boorshtein, the identical twin sons of yiddish-speaking immigrants. I don't know how much of their desire to Americanize themselves was positive identification with the country that had given them so many opportunities - and how much of it was fear of the world that mocked and feared jews, a world with death camps overseas and "no jews, blacks, or dogs" signs at home. I think it was a little of both. It was safer to be more "American", and it's hard to get more "American" than Paul and George. My grandfather deliberately did *not* teach my mother Yiddish, because he didn't want his children to be marked as 'outsiders'. I cannot fault Grandpa's logic or his decision, one of thousands of similar ones that marked the death of yiddish-speaking Ashekanzi Jewish culture, the culture that produced people like Norbert Weiner, like John Von Neumann, like Grandpa. I occasionally feel it as a kind of phantom pain, this disconnection from the culture of my grandparents and their grandparents.

I didn't know that kind of pragmatism was part of Grandpa; I was too too young when he passed away to really know him. I saw the old man with the spectacles who was always reading, who knew how to make a dozen different kinds of hats out of newspaper, the man who (to Bubbe's disgust) boiled his coffee in the same water he boiled his egg and washed his hair with bar soap like he'd learned in the army, leaving it full and white and stiff like a broom or a boar. A strange and kind and (to a child) omniscient oracle, a magical man whose house had a swingset in it's basement.

I didn't really know Grandpa. He's almost completely gone, just memories, some articles - and this book, Anatomy of Automation, through which - over months - I've tried to get to know him. I think that Grandpa wasn't just the realist with an eye to the market, or the kind and playful grandparent. I think Grandpa married that hard-nosed realism and that childlike sense of play with some utopian vision. Grandpa saw the possibility of technology to ease the burdens of man, to put machines to the use of machines and finally allow, as Norbert Weiner put it, **The Human Use of Human Beings**. As software at large continues wasting resources in order to make machines awkwardly substitute for people and people more like machines, I can't help but think we're heading in the

wrong direction. Still, I believe in the same vision he did. We can build real solutions to real problems and use automation for the benefit of society.

It just takes careful thought and hard work - and we have the knowledge of the past to help us. All we have to do is choose to use it.

Biblography

The majority of the work for this article was reading, including but not limited to the following titles.
a table covered with the programming books mentioned in the next section

Norbert Weiner: "Cybernetics"

Foundational text in information theory and computer science. Start with Human Use of Human Beings and the Mathematical Theory of Communication, then work your way back here.

Norbert Weiner: "Human Use of Human Beings"

The book that inspired my Grandpa to write A&A, and I can see why. So far ahead of it's time it's a little painful to consider.

Norbert Weiner: "GOD & GOLEM, INC"

There might be more great quotes in per page in this book than any other I've read, and I read **books of quotes**. Not quite as wide-ranging or relevant as "Human Use of Human Beings", but more poetic and beautiful. There's a profound morality here rare in programmers.

Stafford Beer, "Designing Freedom"

Inspiring and occasionally infuriating with it's lack of examples.

Strunk & White, "Elements of Style"

There is still no better guide to written English. My *other* grandpa, Gerald Licht, used to give a copy to every junior executive he worked with. (Grandpa Jerry was a computing pioneer in the pre-transistor days, something I'd love to write about once I learn more about hardware.)

Douglas Hofstadter, “Gödel, Escher, Bach”:

The book that made me think of myself as a programmer rather than just a mathematician. Wonderful exploration of the meaning of patterns and the patterns of meaning. Philosophy with substance.

Vadim Ponomarenko, “Mathematical Maturity via Discrete Mathematics”

Dr. Vadim Ponomarenko was my (unofficial) advisor and mathematics professor at SDSU. This is the shortest (and best!) book on learning the mathematician’s mindset I know. It’s extremely terse but that’s part of the point: **to a mathematician, every word should matter.** Be prepared for many, many homework problems.

Kernighan and Ritchie: “The C Programming Language”:

The best programming language book ever written and it’s not particularly close. The code in here is awe-inspiring in it’s simplicity.
Some of it is better admired from afar than actually copied, though.

```
1   /* an elegant and cryptic way to copy a string*/
2   while ((*s++ = *t++) != '\0'); /* load-bearing semicolon*/
```

Kernighan and Donovan: “The Go Programming Language”:

Still the best resource on Go more than a decade later and it’s not particularly close. Doomed to live in the shadow of K&R C, but while not as beautiful, it’s just as useful. If you write go and haven’t read this, now’s a good time to start.

Kernighan and Pike: “UNIX Programming Environment”

Wish I’d read UNIX Programming Environment a decade ago: it might STILL be the best introduction to working with an operating system, the shell, or the “programmer’s mindset” there is. Incredibly highly recommended. Brilliant, fast-paced, appropriate for everyone from the journeyman to expert (but not for apprentices).

Kernighan and Pike: “Practice of Programming”.

Incredibly valuable and shows you how to build everything from a Just-In-Time compiler to a regular expression engine in only a few hundred pages. The section on notation is worth it on it’s own.

Kernighan and Plauger: “Software Tools”, “Elements of Programming Style”

These books are from the Late Antiquitity of programming, the era of punchcards and paper terminals. They can be pretty hard to get into if you aren’t comfortable looking at, say, FORTRAN.

Kernighan and Plauger: “Elements of Programming Style”

There’s some great stuff in here, but you have to work pretty hard to get it. If you’re not comfortable with ancient FORTRAN there are probably better places to look.

Aho & Ulman, “Principles of Compiler Design”

Brilliant but very difficult and ‘math-y’ for a modern reader. Aho & Ulman’s *second* dragon book, Compilers: Principles, Techniques, and Tools 2nd Edition, is probably a better place to start. Through a quirk of fate, the copy I have is signed by one “Robert Pike”.

Roberto Ierusalimschy: “Programming in Lua (4th ed)”,

Lua’s emphasis on simplicity, composition, and re-usability makes it stand out in the bloated world of modern software. This book contains few quotable gems but is an absolute treasure trove of practical programming examples. While I often find myself on the opposite side of small divides with LUA (compiled vs. interpreted, 1-index vs. 0-index, etc), LUA as a language - and a philosophyu - match simplicity & humility with formidable ruthlessness & clarity of purpose. Big props to those guys from Brazil.

Shannon & Weaver: “Mathematical Theory of Communication”,

Perhaps the single most important book to modern telecommunications and the computerized world we live in. Surprisingly readable even if your mathematics isn’t that strong.

Von Neumann & Morgernstern: Mathematical Theory of Games and Economic Behavior

An extremely difficult read heavy on mathematics. Eye-opening in it’s ruthlessness and generality.

MORE ARTICLES