ACRUMEN:

What IS Software Quality, Anyway?!

by Dave Aronson
T.Rex-2025@Codosaur.us

www.Codosaur.us

@davearonson

Current time: ~43 (slot is 45, NO Q&A, so want 40-44), okay, but watch the ad-libs

NOTE TO SELF: have biz card ready to show, and more as handouts!

ACRUMEN:

What IS Software Quality, Anyway?!

by Dave Aronson
T.Rex-2025@Codosaur.us

www.Codosaur.us

@davearonson

(Duplicate slide so I can flip to a new one to start my timer, ignore this.)

Bonjour, CERN!

(Hello, CERN!)

www.Codosaur.us

Image: standard emoji

@davearonson

Je m’appelle Dave Aronson,

(I'm Dave Aronson,)

www.Codosaur.us

Image: me speaking at JSConf Hawai'i 2020

@davearonson

je suis T-Rex chez Codosaurus,

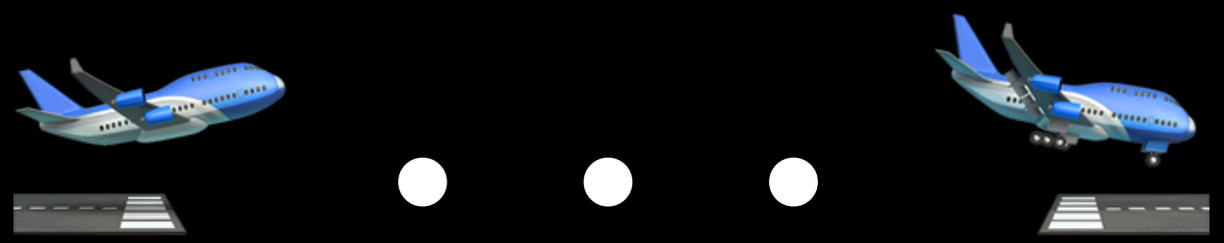
(the T. Rex of Codosaurus,)

www.Codosaur.us

Image: my company logo!

@davearonson

et j'ai volé jusqu'ici



(and I flew here)

www.Codosaur.us

Image: standard emoji

@davearonson

sur mon ptérodactyle de compagnie

(on my pet pterodactyl)

www.Codosaur.us

Images: https://pixabay.com/vectors/dinosaur-tyrannosaurus-t-rex-6273164/
and https://pixabay.com/vectors/bird-flying-wings-dinosaur-ancient-44859/

@davearonson

pour vous parler d’

(to tell you about)

www.Codosaur.us

Image: standard emoji

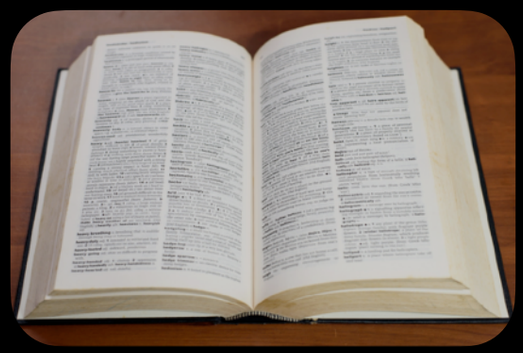
@davearonson

ACRUMEN

www.Codosaur.us

@davearonson

ma définition



(my definition)

www.Codosaur.us

Image: https://pixabay.com/vectors/turtle-tortoise-cartoon-animal-152079/ plus X's

@davearonson

de la qualité logicielle.

(of software quality.)

www.Codosaur.us

Image: https://pixabay.com/en/software-packaging-quality-500956/

@davearonson

Mais . . .

(But . . .)

www.Codosaur.us

Image: standard emoji

@davearonson

je le ferai en anglais.

(I will do it in English.)

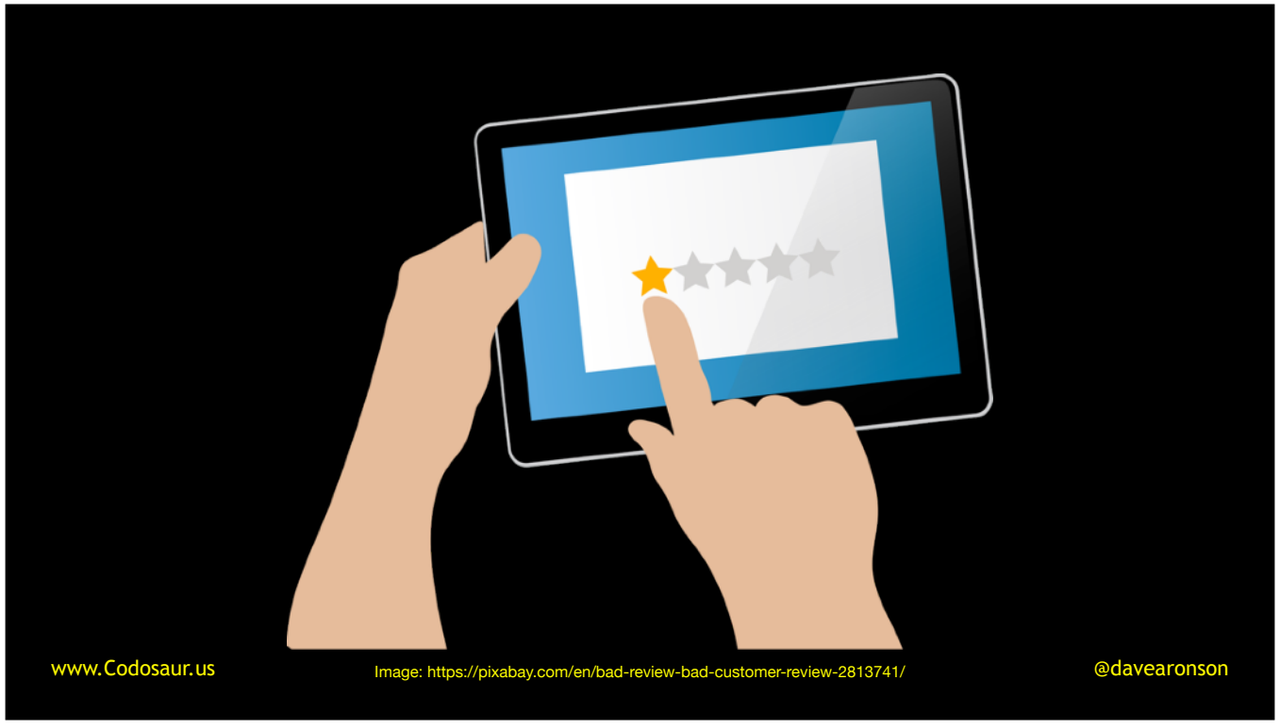
www.Codosaur.us

Image: standard emoji

@davearonson

(PAUSE!)

Let’s start things off with a question. Do you like . . .



. . . low quality software? (pause) Anybody? Nobody?!

Let’s try another question. Have you . . .

. . . *written* any low quality software? (raise hands) Yeah, I know I sure have!

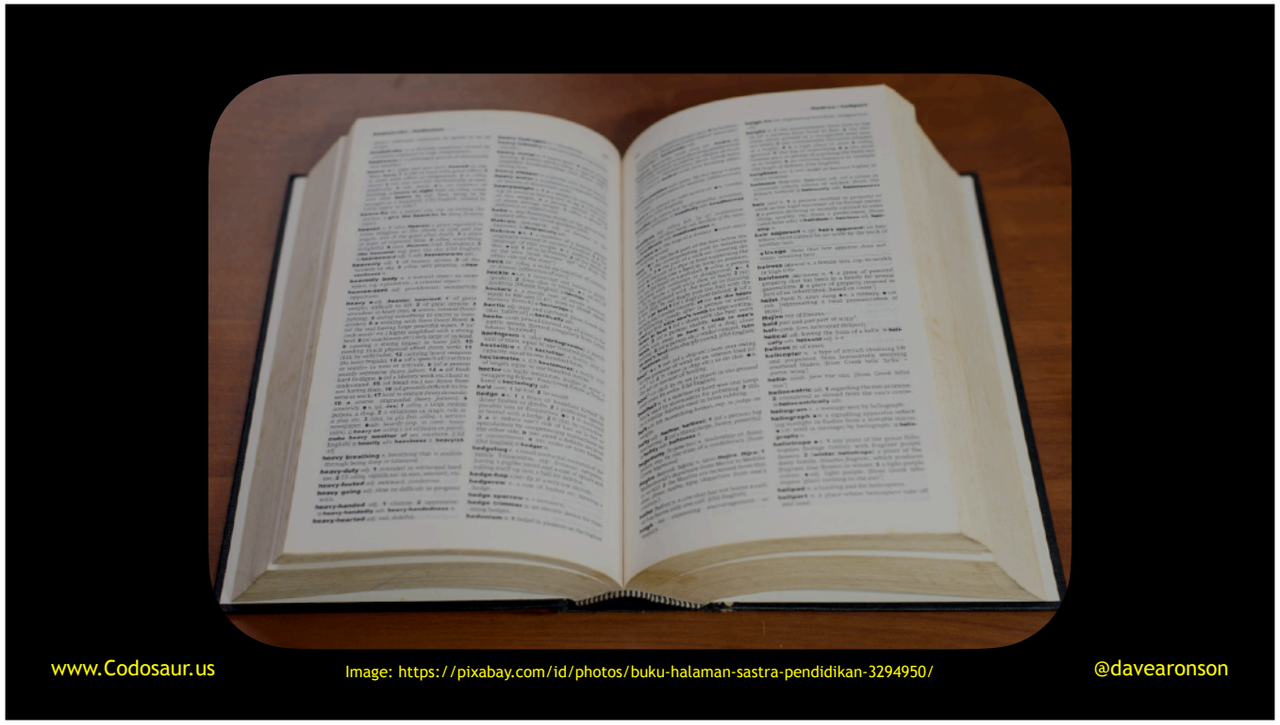
That's more like it! To those of you who said yes, congratulations! As the saying goes, Step One is to realize: you have a problem! For the rest of you: welcome to software development! I hope you enjoy this career you've obviously just started.

So we've got a lot of people writing (or at least having written) low-quality software, but we don't *like* it. It seems pretty clear, we need . . .

. . . more software quality!

But that leads us to one tiny little question: . . .

. . . What *is* it? Without a good . . .



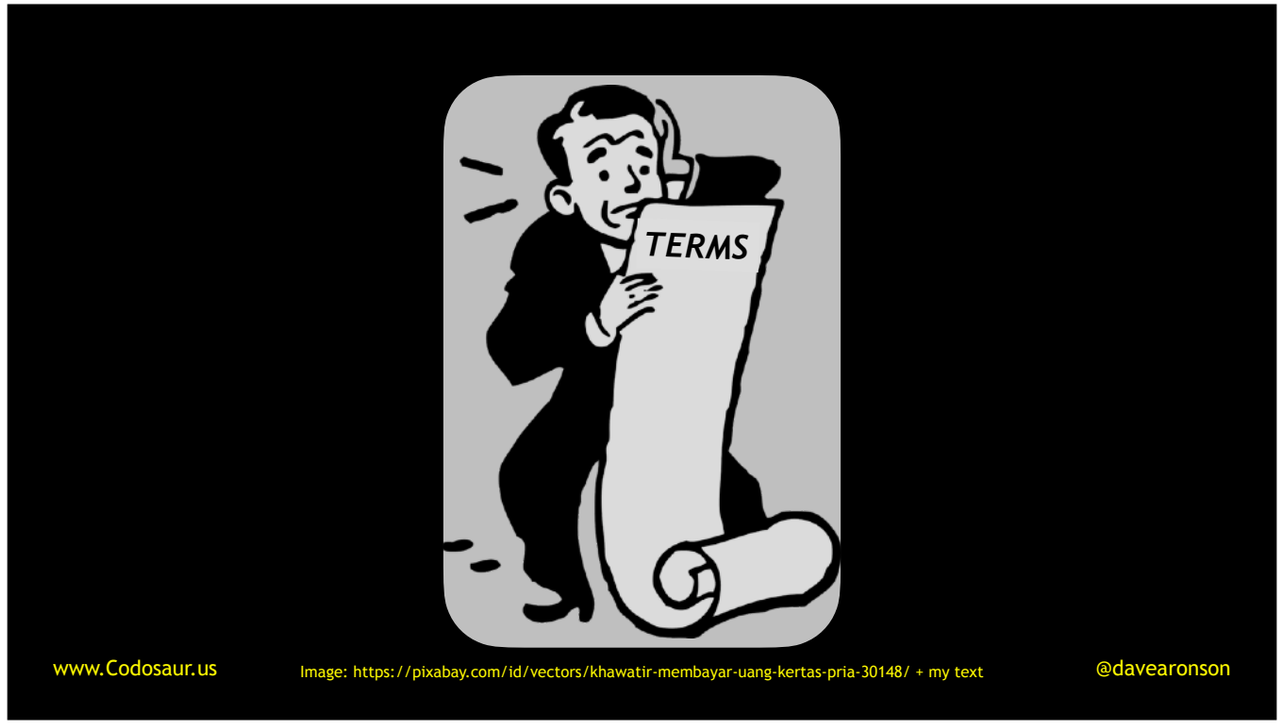
www.Codosaur.us

Image: https://pixabay.com/id/photos/buku-halaman-sastra-pendidikan-3294950/

@davearonson

. . . *definition*, it's very hard to achieve — we can't hit a nonexistent target! And if we don't *share* that definition with other people, it's very hard to get them to *acknowledge* when we *have* achieved it, or at least we *think* we have. So, I'm trying to get everybody on the same page. (Yeah, I know, good luck with that!)

There are *already* many definitions out there, so, why do we need a *new* one? Several years ago, I was *looking* for a good one, but everything I found had some serious problems, at least for this purpose. Most were . . .



. . . long lists of complicated terms, full of developer jargon. Now, jargon is fine for talking amongst *ourselves,* but I wanted a definition that *other* people would understand, *even non-technical people*, so they could understand our *challenges* better, and give us more precise feedback about *exactly how* our software . . . falls short, to put it politely. Also, I wanted something *short* and *simple* enough that people could easily learn it, remember it, and apply it, without having to consult lengthy documentation.

Some definitions were . . .

www.Codosaur.us

Image: https://www.flickr.com/photos/59937401@N07/5857412037

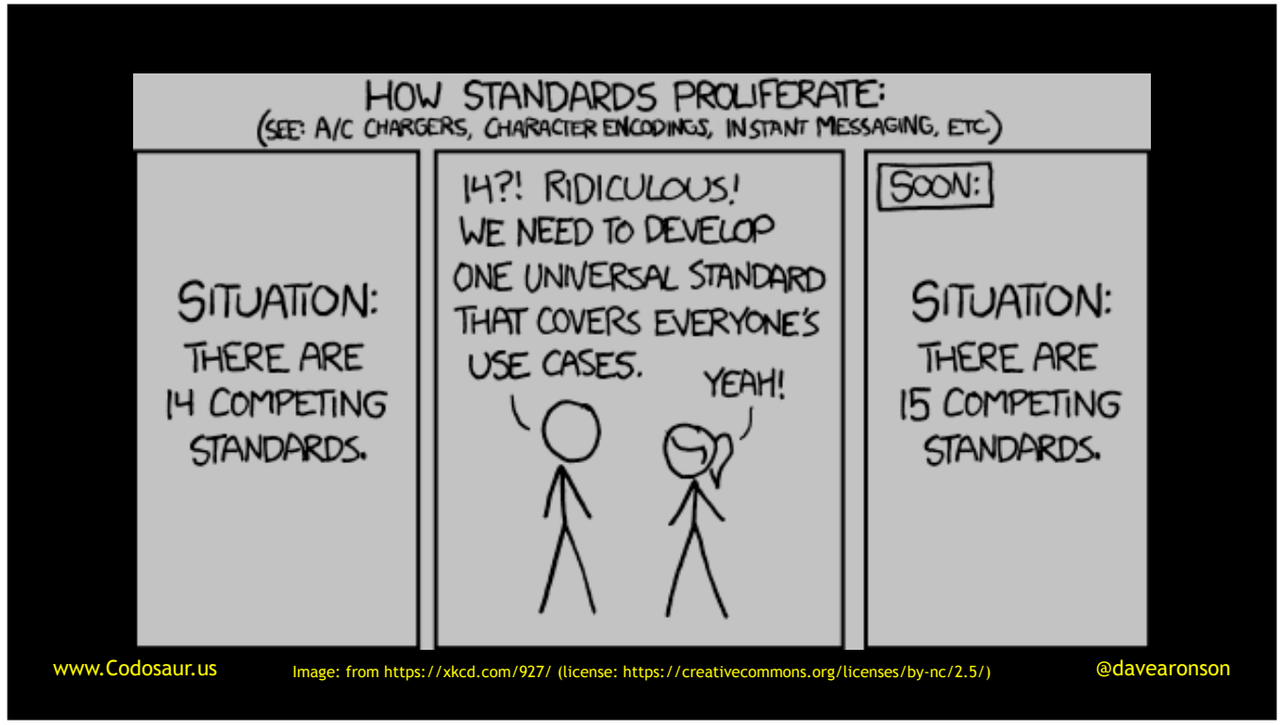
@davearonson

. . . proprietary, making us buy expensive software tools, or at least relatively expensive documentation — like I just said I didn't want people to have to use. Some were only applicable within the context of certain technologies, often also proprietary, or certain styles of programming. But I wanted something we could use with *all* software, for *free*.

Some weren't even really about the software, but all about the process or its byproducts, dictating that we must . . .

. . . hold these meetings or write those documents. Now, some of these meetings and documents may be helpful, but to make them the actual definition, I felt completely missed the point. I wanted something focused on the software itself, *de*scriptive rather than *pre*scriptive.

Long story short, I didn't see any that I liked, nor that was commonly accepted, so in the spirit of . . .



. . . XKCD (PAUSE!), I decided to make my own. But rather than the usual approach of taking the best parts of all the existing ones, I decided to whittle it down to just the bare essentials. That resulted in a list of just six aspects, with simple names and *relatively* simple explanations. It's so short, it literally fits on the back of a business card, without even using small print, and (HOLD UP BIZ CARD!) here's mine to prove it. See me later if you want one as a cheat-sheet.

I call this list of aspects . . .

ACRUMEN

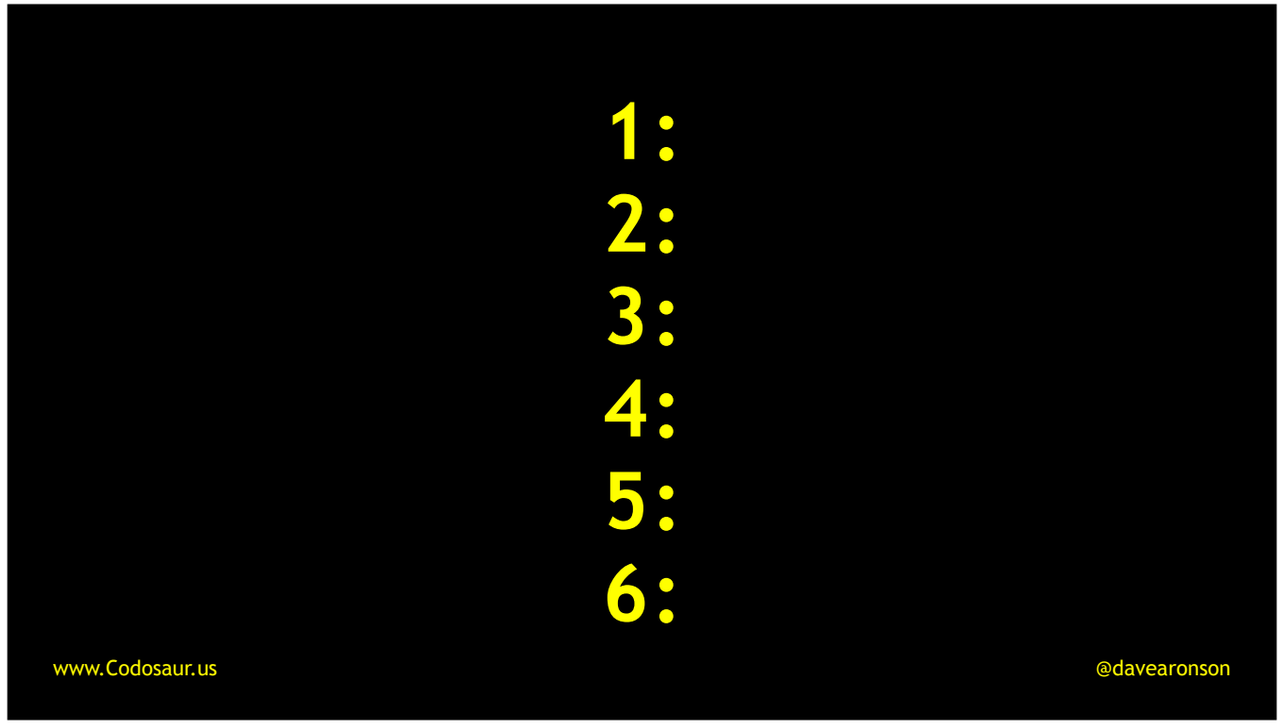
www.Codosaur.us

@davearonson

. . . ACRUMEN, but what does that mean? Originally, that was a Latin word, meaning sour fruit, like grapefruits, limes, and especially . . .

. . . lemons. That’s why you'll see lots of lemon yellow throughout these slides. It’s basically “The Official Color of ACRUMEN”.

But what is ACRUMEN *in the context* of software quality? The *acronym* ACRUMEN (try saying that ten times fast!), just takes those six aspects, and . . .



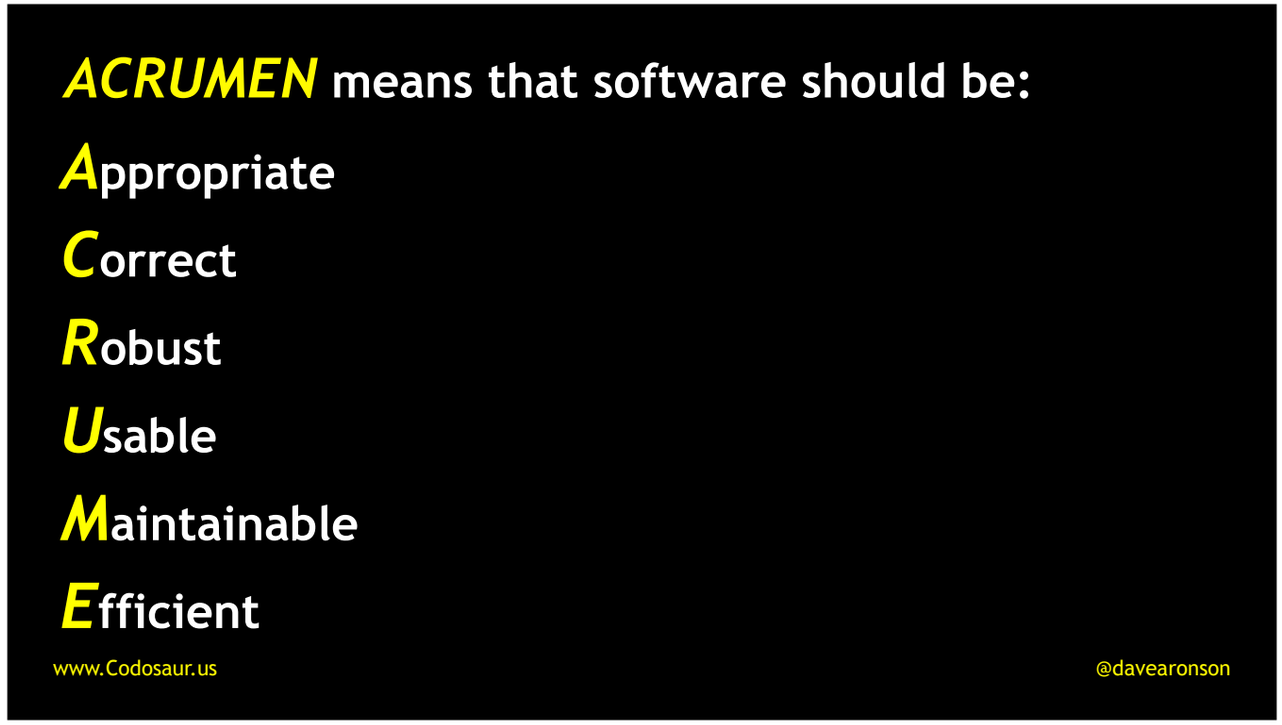
. . . puts them in priority order. And those aspects are, at long last, that . . .

ACRUMEN **means that software should be:**

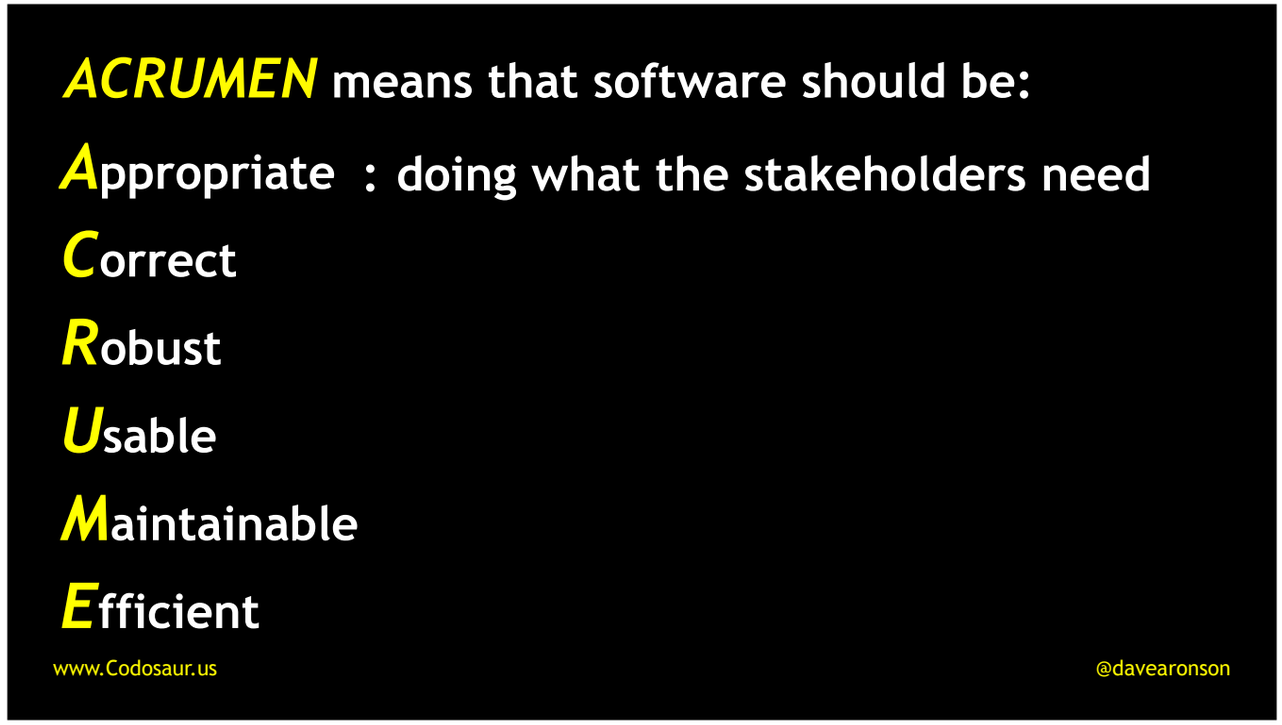
www.Codosaur.us

@davearonson

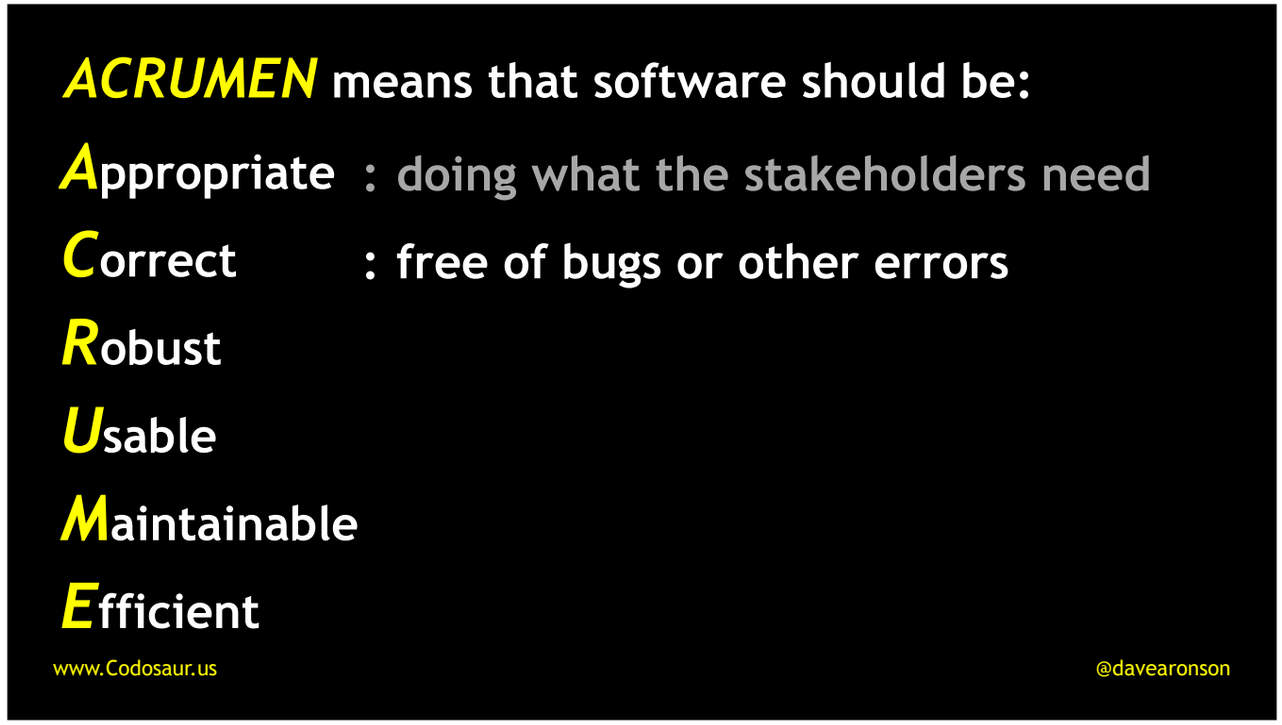
. . . software should be: (INHALE) . . .



. . . Appropriate, Correct, Robust, Usable, Maintainable, and Efficient. But what does all *that* mean? First and foremost, software needs to be . . .



. . . *doing what the stakeholders need* it to do, in other words, doing the *right job*, or maybe job*s*. And notice I say “stakeholders”, not “users”! The Project Management Institute defines “stakeholders” as everybody involved in, or affected by, the project, or in this case, the software. So that’s not just the users, but also the people involved in, for instance, the development (hey, that’s us!), deployment, monitoring, customer service, sales, marketing, and so on, *and* their *management*. Then it needs to be . . .



. . . *doing* those jobs, *without bugs or other errors,* or in other words, doing the right jobs *right*. This one is pretty much exactly what it sounds like, so, moving on, it should be . . .

ACRUMEN **means that software should be:**

A**ppropriate** : doing what the stakeholders need

C**orrect** : free of bugs or other errors

R**obust** : hard to make malfunction *or seem to*

U**sable**

M**aintainable**

E**fficient**

www.Codosaur.us @davearonson

. . . hard for anyone to make it crash, or otherwise malfunction, or even seem to, whether deliberately or accidentally. On the other claw, it should be . . .

ACRUMEN **means that software should be:**

A**ppropriate** : doing what the stakeholders need

C**orrect** : free of bugs or other errors

R**obust** : hard to make malfunction *or seem to*

U**sable** : easy for the stakeholders to use

M**aintainable**

E**fficient**

www.Codosaur.us

@davearonson

. . . easy for the stakeholders to use, whether they're interacting directly with the software or just with some data it produces, or whatever. It should also be easy . . .

ACRUMEN **means that software should be:**

Appropriate : doing what the stakeholders need

Correct : free of bugs or other errors

Robust : hard to make malfunction *or seem to*

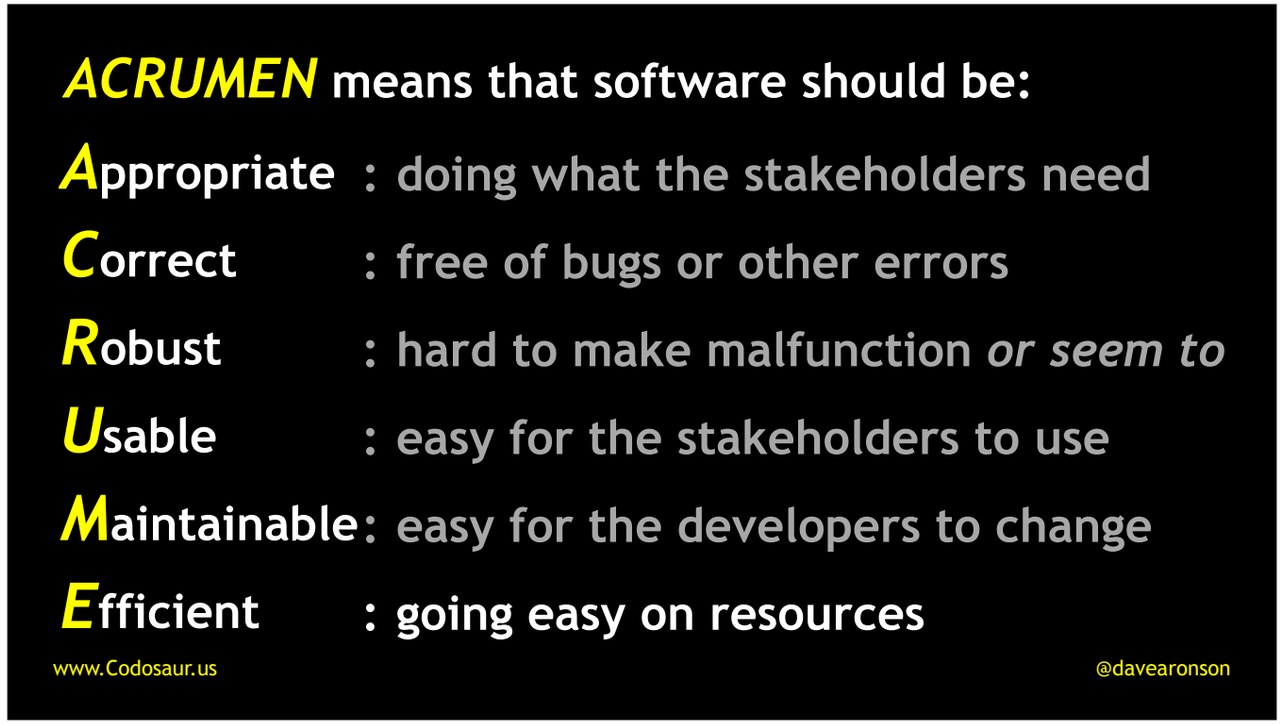
Usable : easy for the stakeholders to use

Maintainable: easy for the developers to change

Efficient

www.Codosaur.us @davearonson

. . . for the developers to change. One might consider that a special case of Usability, but I think it's important enough, especially to us, to stand on its own. And last, *dead last* despite how we developers tended to absolutely worship this just a few short decades ago, it should . . .

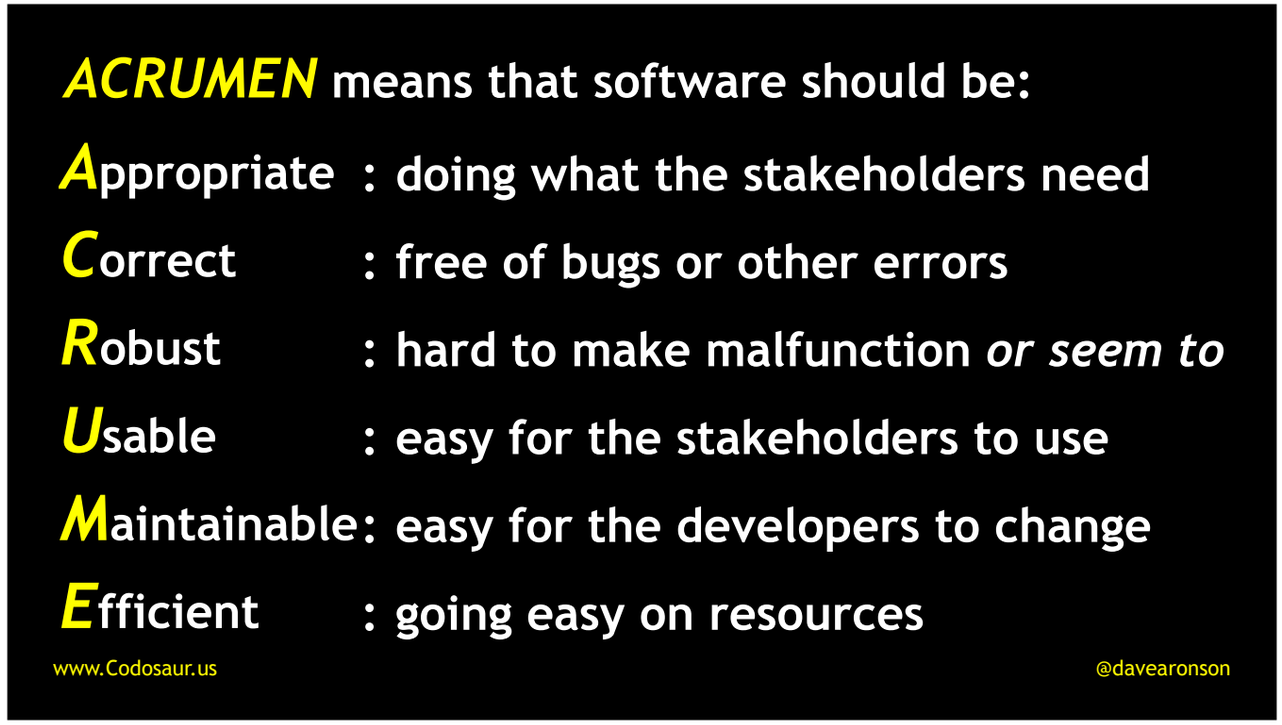


. . . go easy on resources, not only the technical ones that we usually think of, but *other* kinds as well.

Now, I've said that there are six aspects, and you see six listed up there, but ACRUMEN has seven letters! So what does the N stand for? Nnnnn . . .

. . . nothing! I just tacked it on to make a real word, even if an obsolete one.

Now, while . . .



. . . the basic definition is fresh in our minds, I'll address some frequently asked questions. First, aside from going into detail on the tips, how do we actually *use* ACRUMEN itself, the list of aspects?

Mainly, we can keep it in mind as a sort of . . .

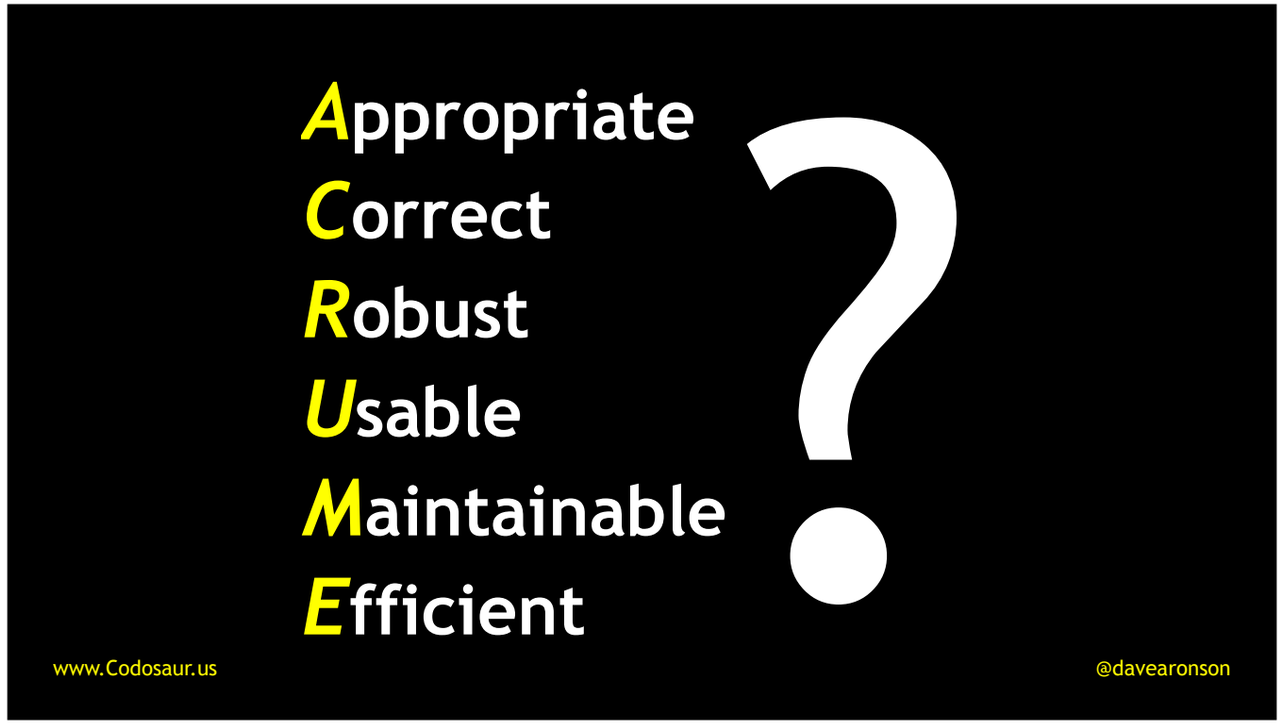
. . . *checklist*, when writing or evaluating software. We can ask, *is* it Appropriate, *is* it Correct, and so on, or *how* good is it in each aspect, whether that be on some scale, or by simple triage, or is it *good enough* for *our needs?* And if the answer is ever that it's not good enough, we can ask, what can be done to *. . .*

. . . *make* it so?

In the short term, we can ensure that our current *projects* are likely to *meet* these criteria, and in the long term, we can ensure that our *processes support* these criteria, by including various helpful activities and requirements, and maybe even an explicit evaluation against the ACRUMEN aspects. We can also set . . .

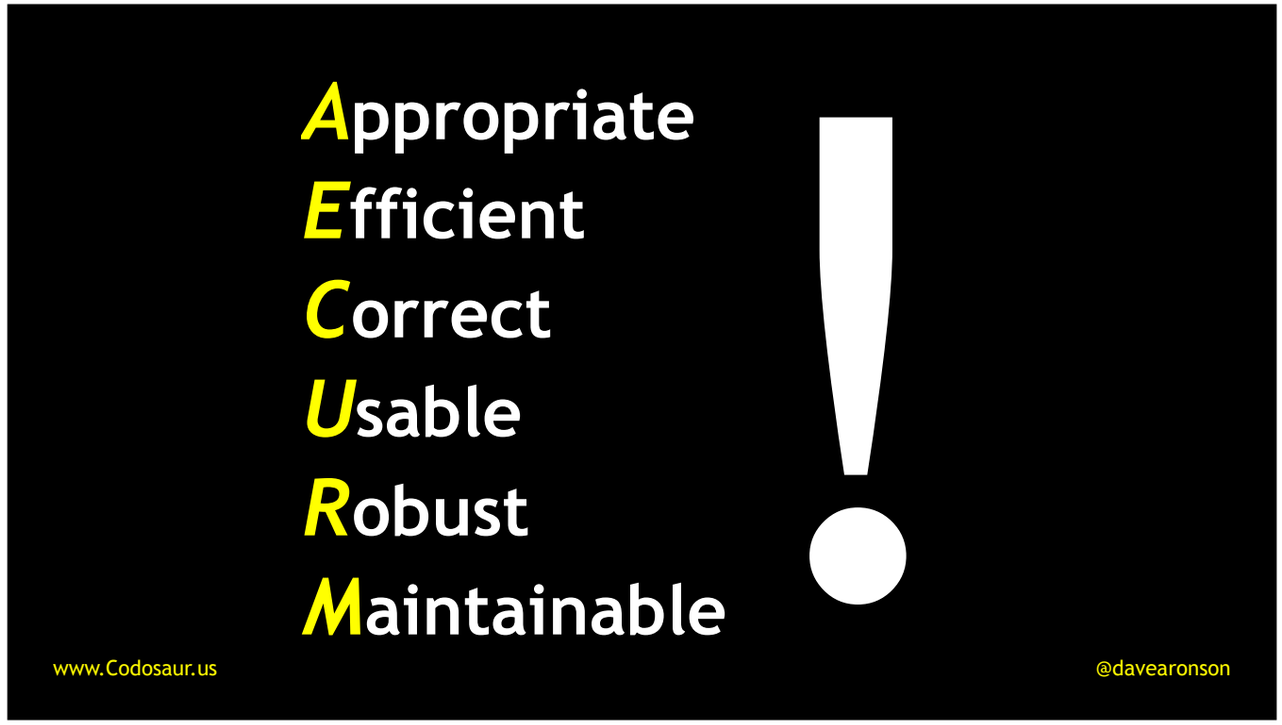
. . . targets, for how good we *need* some system to be in each aspect.

The second question is: . . .



. . . is ACRUMEN, or rather ACRUME, always the right ordering? Some projects seems a little different.

The answer is, no, ACRUMEN is just the *typical* case. Your mileage (or over here, maybe your kilometrage) may well vary. Consider, for instance, a company-internal command-line physics simulation tool, using a standard algorithm that will never change, finding a close-enough answer to something where precise calculation would take forever, like maybe weather forecasting. I'll spare you the consideration of each aspect, but its list may well look more like . . .



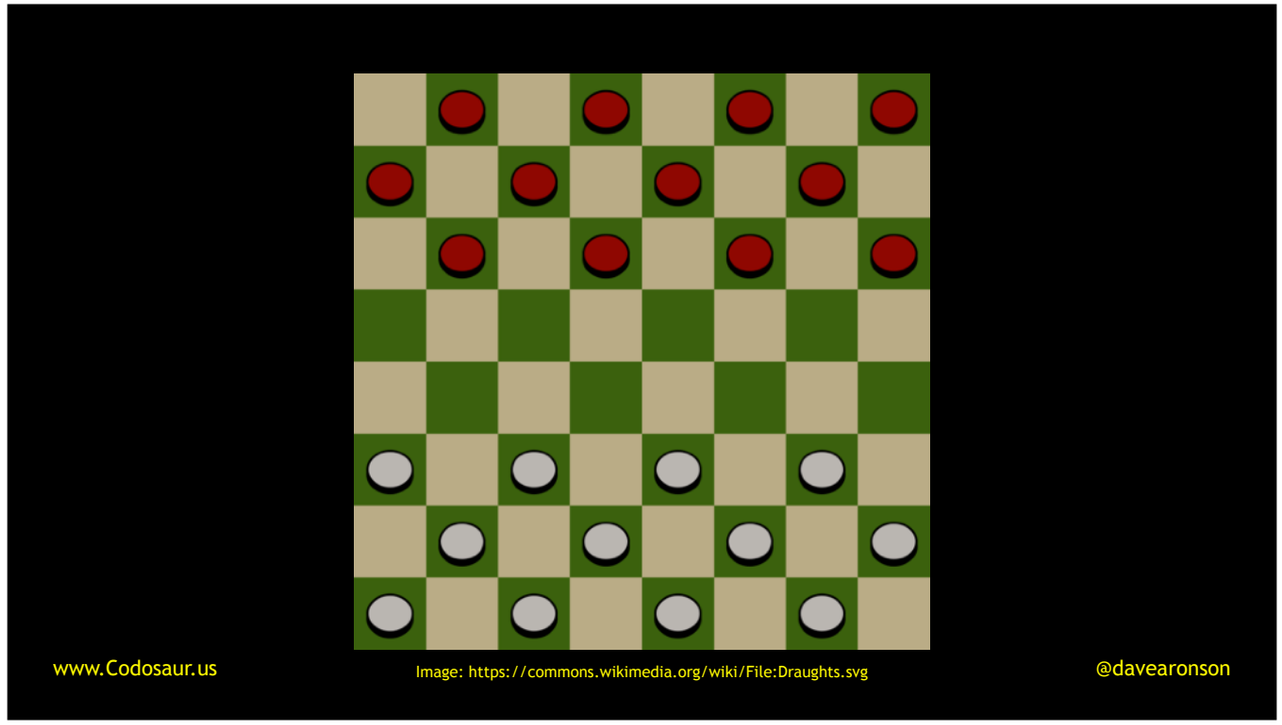
. . . this, AECURM, rather than ACRUME. The only real constant is that Appropriate will always be at the top. We'll see why very soon, because now we're going to look at each aspect in more detail, and up first is of course . . .

. . . Appropriateness.

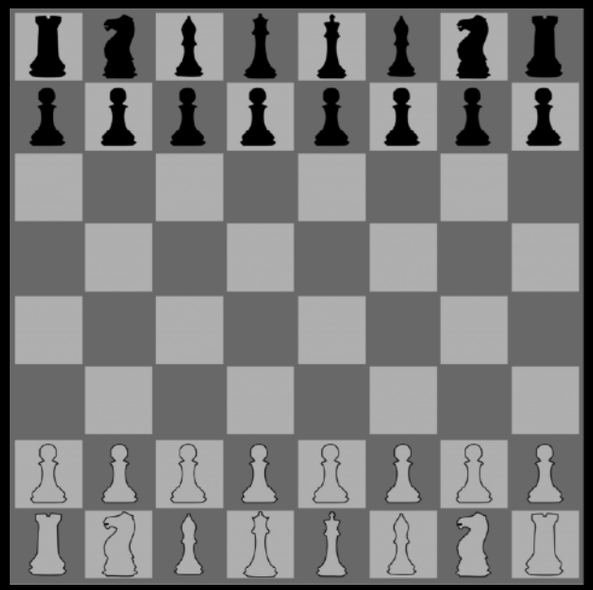
If our software doesn't have this, then Nothing. Else. Matters. (PAUSE!) If our software is doing the *wrong job*, then it *doesn't matter* how *well* it's doing *the wrong job!* So, appropriateness is not only more important than any other aspect, it's even more important than . . .

. . . *all* the others *put together*. And yet, we developers are generally not taught that this is even a thing, let alone one that *we* need to think about.

To *prove* this importance, let's try a little thought experiment. Suppose you want a program to play . . .



. . . checkers (or in British English, draughts), and I write for you the world’s greatest . . .

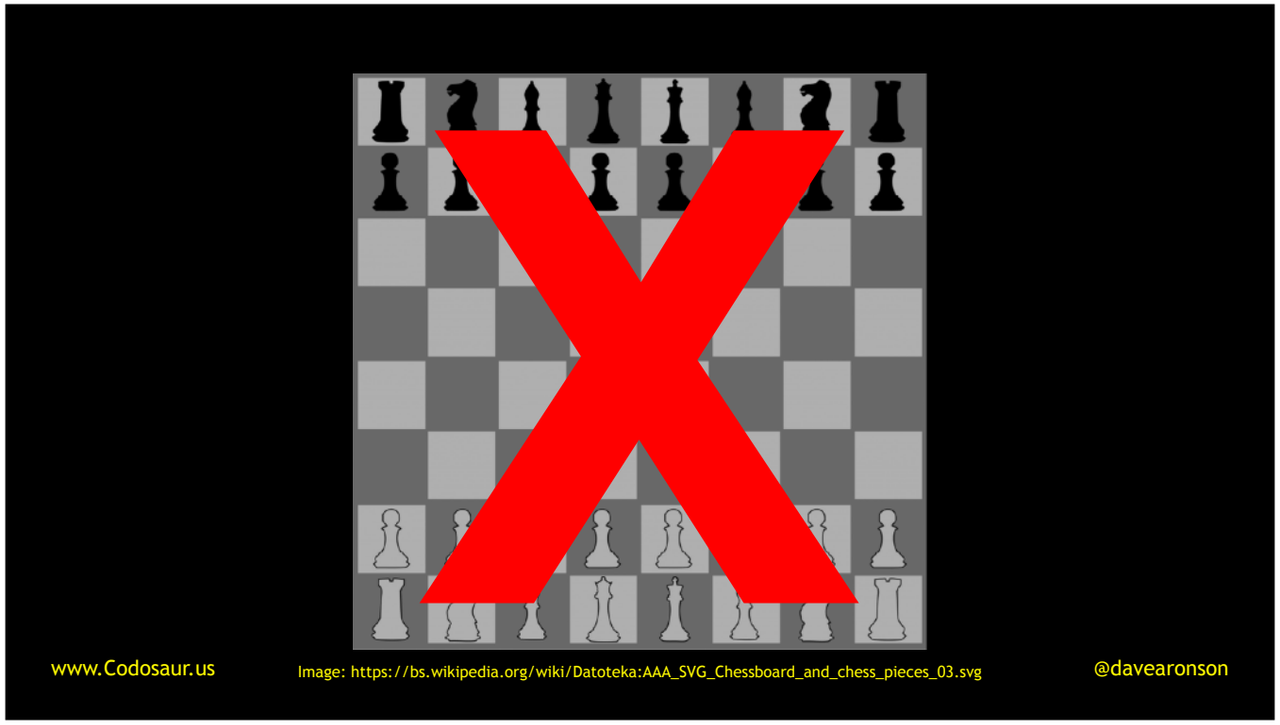


www.Codosaur.us

Image: https://bs.wikipedia.org/wiki/Datoteka:AAA_SVG_Chessboard_and_chess_pieces_03.svg

@davearonson

. . . *chess* playing program. It's as correct, robust, usable, maintainable, and efficient as anyone could ever want. But will you be happy with it? (PAUSE!)
Probably . . .



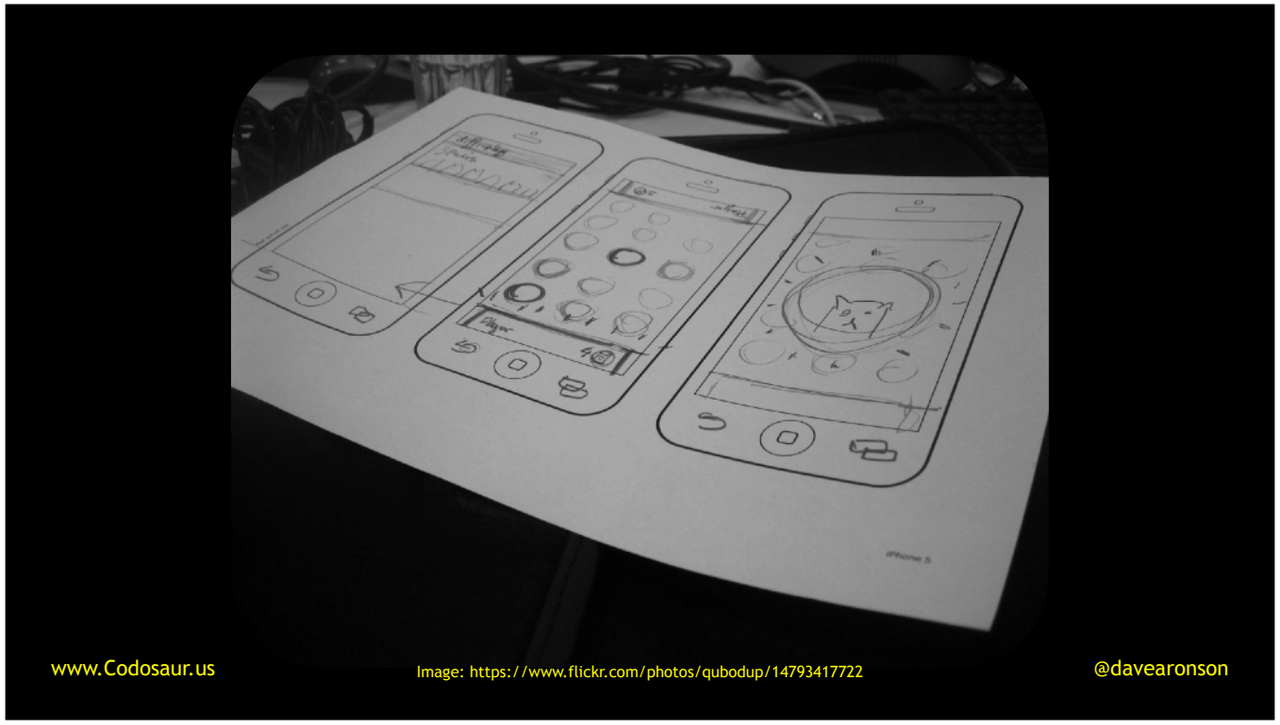
. . . not. But why not, if it's such a great program? (PAUSE!) Because it's not checkers. It's not what you asked for. It's not what you (presumably) need. Or in ACRUMEN terms, it's not *appropriate*.

So now that we know how important this is, how do we achieve it? In an ideal world, we would have . . .

. . . frequent direct contact with the stakeholders — of *all* kinds! Ideally face to face, or as close to that as practical. Unfortunately, we don't usually get that opportunity. Second best is to bring in the experts, which in this case would be Requirements Analysts. But on this . . .

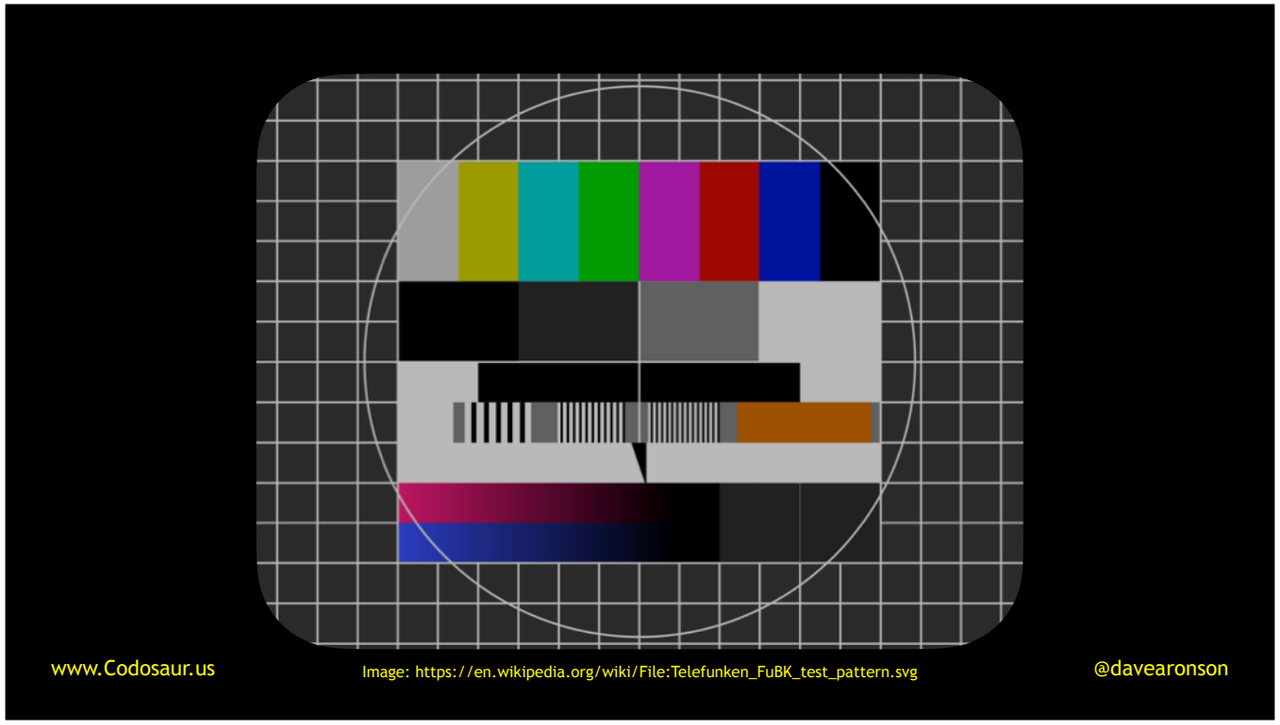
. . . planet, we usually don’t get those either, at least outside of huge companies. We might not even have *business* analysts available, the next step down. So we usually have to settle for occasional remote indirect contact with a representative of some stakeholders, like a Product Owner in Scrum. It doesn't work quite as well, but having *some* communication with *someone* with *some* clue, is *vital*.

Once we think we have a good grasp of their needs, we can show them . . .



. . . mockups and prototypes of what we *intend* to do, and demos of what we *have* done. This gives them a chance to *correct our wrong ideas* of their needs, before we go too far down the wrong rabbit-hole. Has anyone else been there, wasting time implementing the *wrong thing?* (PAUSE!) Looks like a lot of you, though fewer than I'd have thought. The rest of you, you probably did, but might not have known it. Anyway, ideally, show them these things *frequently,* as a sort of continuous course correction. Frequent feedback *from* the stakeholders is even *more* important than being able to ask them questions.

There's another thing, though, that I'll be returning to over and over in this talk. We can propose . . .



. . . *tests!* In particular, I recommend the Given/When/Then pattern:
given: these preconditions, such as data being in a certain state;
when: this happens, usually some kind of input;
then: this is the result, usually either something the user sees, or data being in a desired new state.
This makes a great link between the worlds of business and tech, because the business people can understand it, and we can turn it into a runnable test.

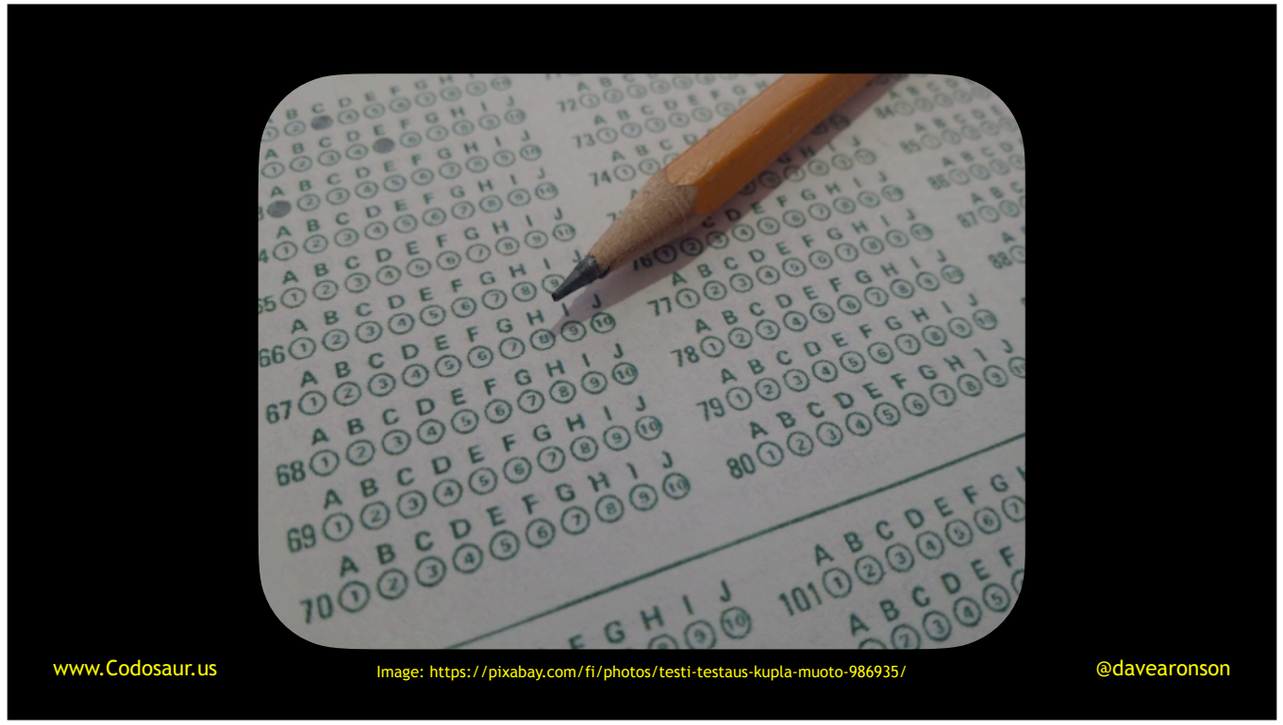
Our next aspect is . . .

. . . correctness. If our software doesn't have this, then, it has bugs, and nobody likes that! Not only do the users not like using buggy software, we don't like having our names on it.

Nothing can actually *stop* us from *writing* buggy code, at least with the reasonably effective tools we have today. So, the big question is: . . .

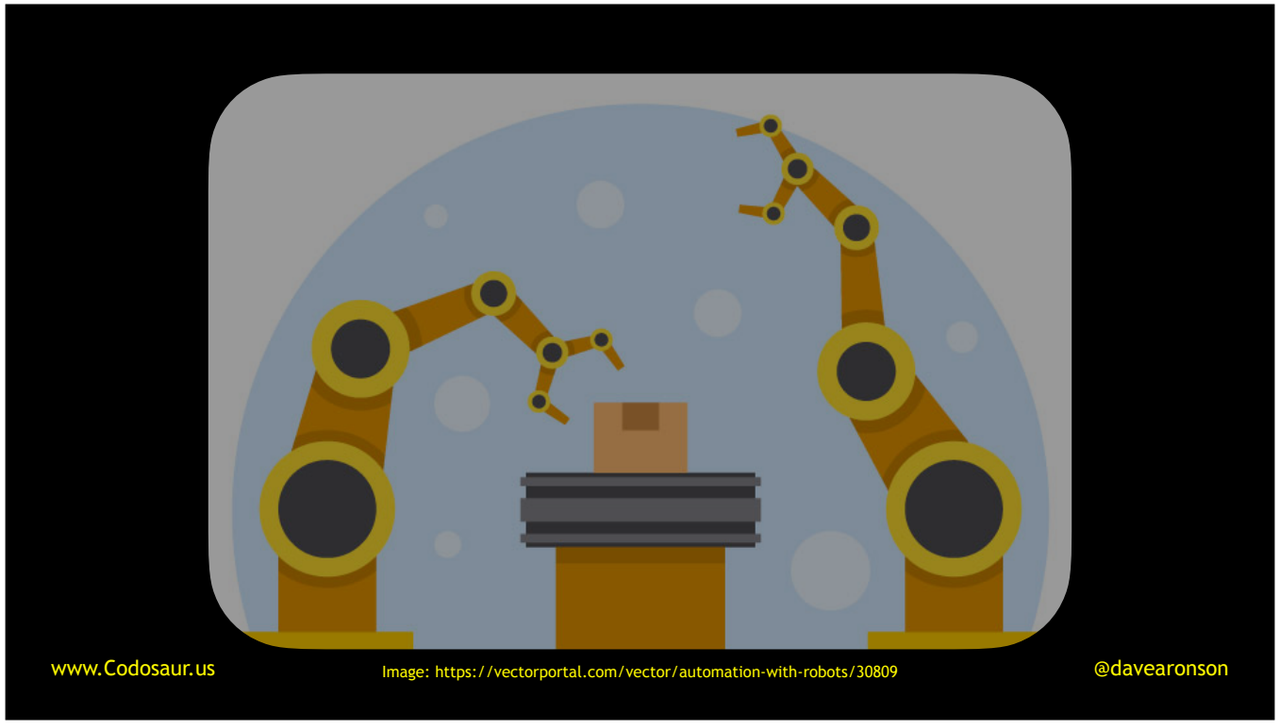
. . . how do we *know*? (PAUSE!)

I just mentioned it, and most of you probably know already, that . . .



. . . *tests* prove whether our code is correct . . . *assuming* that the tests *themselves* are correct. Actually, even then, it's not quite true, but I'll get to that in a moment.

Ideally our tests are . . .



. . . automated, which makes them more repeatable, and maintainable than manual tests, and *much* faster, so they get us vital feedback while the problem at hand is still fresh in our minds . . . again, *assuming* that our tests are reasonably *efficient*.

We can start with the tests that the stakeholders approved for the sake of Appropriate-ness. These are likely to be . . .

. . . *high*-level types like end-to-end system tests, feature tests, etc. But that's not all we need. As we implement the system, we should still add our own . . .

. . . *low*-level tests, like unit tests, integration tests, and so on.

However, normal tests like these can only prove the correctness of cases *we thought* to test. But there are some advanced techniques that can help find edges and other unusual cases we didn't think of. For instance . . .

. . . property-based testing tests whether some desired property, of the output or its relationship to the input, holds true for all valid inputs. A property testing tool makes up lots of random test data to try, somewhat like the security concept of "fuzzing", but staying *within* defined bounds of validity, rather than trying to find and exceed them to see what happens. If it finds a valid input that makes our property fail, that means that there is a case that we didn't consider.

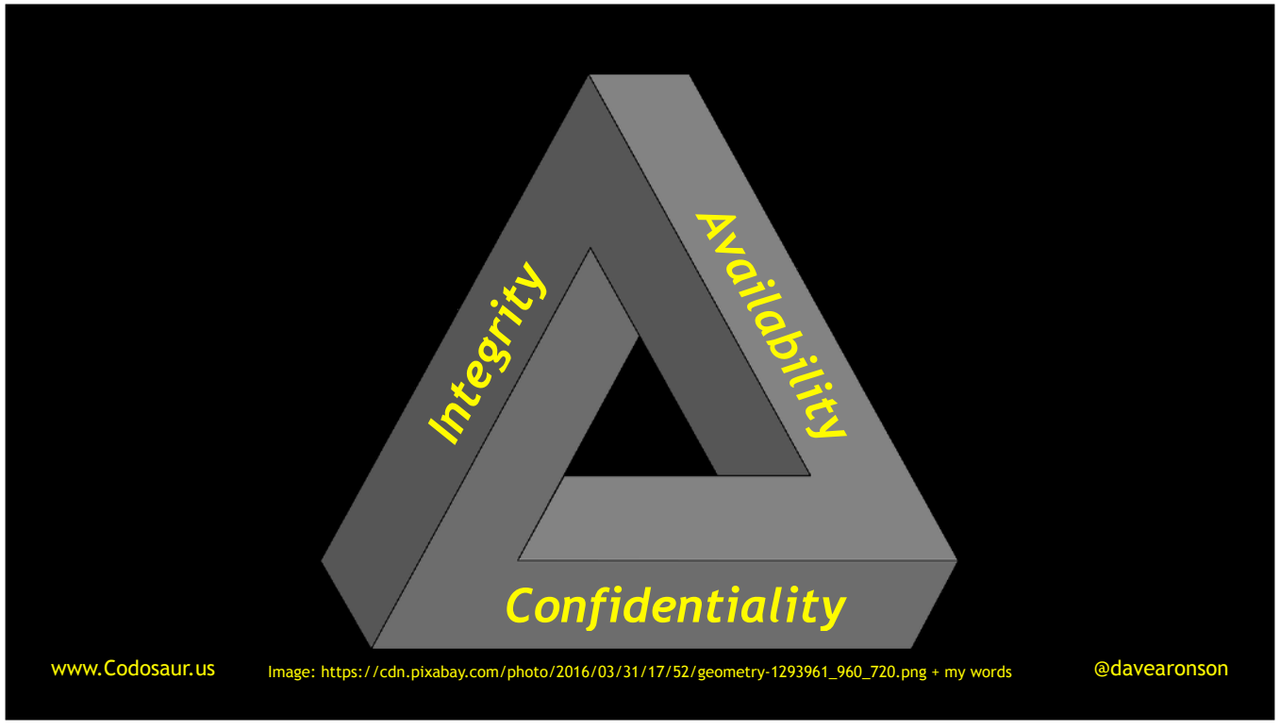
Mutation testing runs our tests against slightly altered versions of our code, called mutants. Each mutant should make at least one test fail. If not, that means that we probably have gaps in our test suite, redundant, unreachable, or otherwise ineffective code, or both, no matter what our so-called “coverage” statistics might say. For more information on that, I’ll be speaking on mutation testing right here this afternoon.

Anyway, we should have enough test coverage, of assorted kinds and levels, *and verified to actually be meaningful*, to have strong confidence in the correctness of our code.

Next up we have . . .

. . . robustness. If our software doesn't have this, then, *at best*, it may show a lot of error messages, and *seem* fragile and unreliable, or it may crash a lot and actually *be* fragile and unreliable, or it may even . . . Get Hacked, because Robustness includes Security.

The short explanation is that it's hard to make the software malfunction (or even seem to), but what does *that* mean?! There are a few other things, but most of what I mean is covered by a core concept of information security: . . .



. . . the CIA Triad. No, it's nothing to do with spies and gangsters, it's this triangle here of Confidentiality, Integrity, and Availability. So, robust software does *NOT:*

www.Codosaur.us

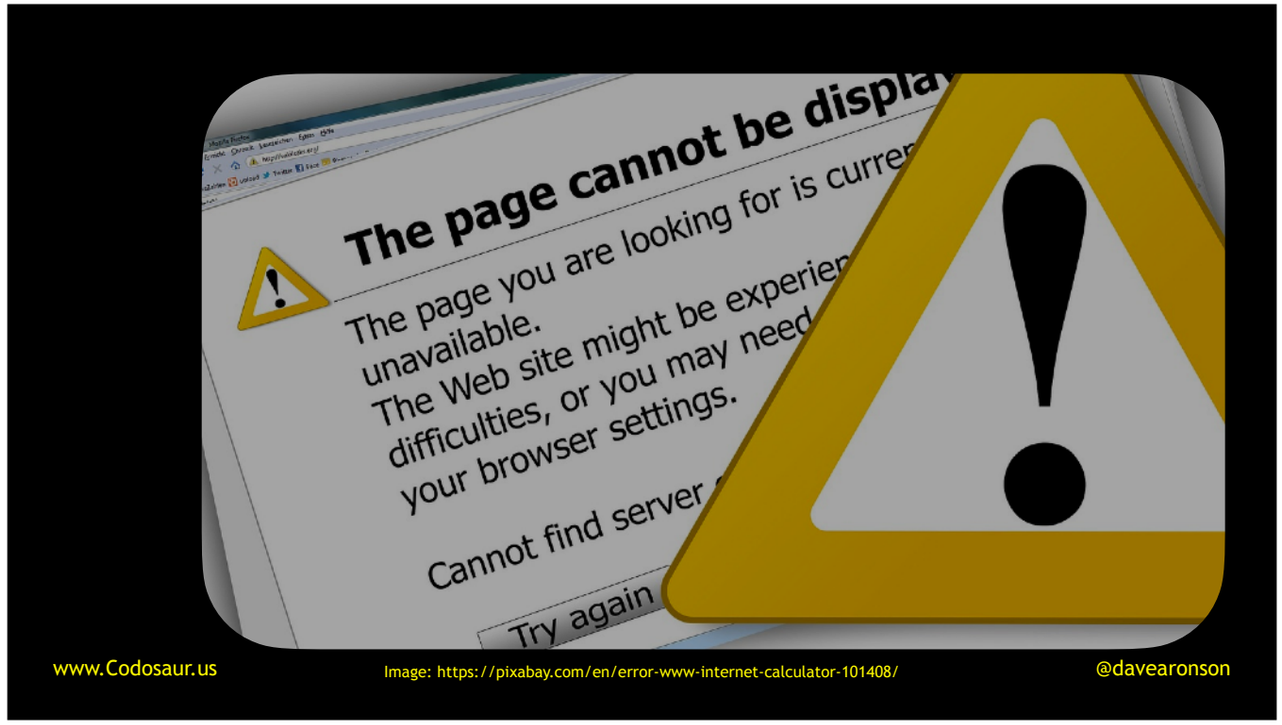
Image: https://commons.wikimedia.org/wiki/File:Somebody_blabbed.jpg

@davearonson

. . . reveal data when it’s not supposed to, . . .

www.Codosaur.us Img: http://www.barksdale.af.mil/News/Article/321176/military-clothing-sales-reopens-inside-base-exchange/ @davearonson

. . . alter data when it’s not supposed to, . . .

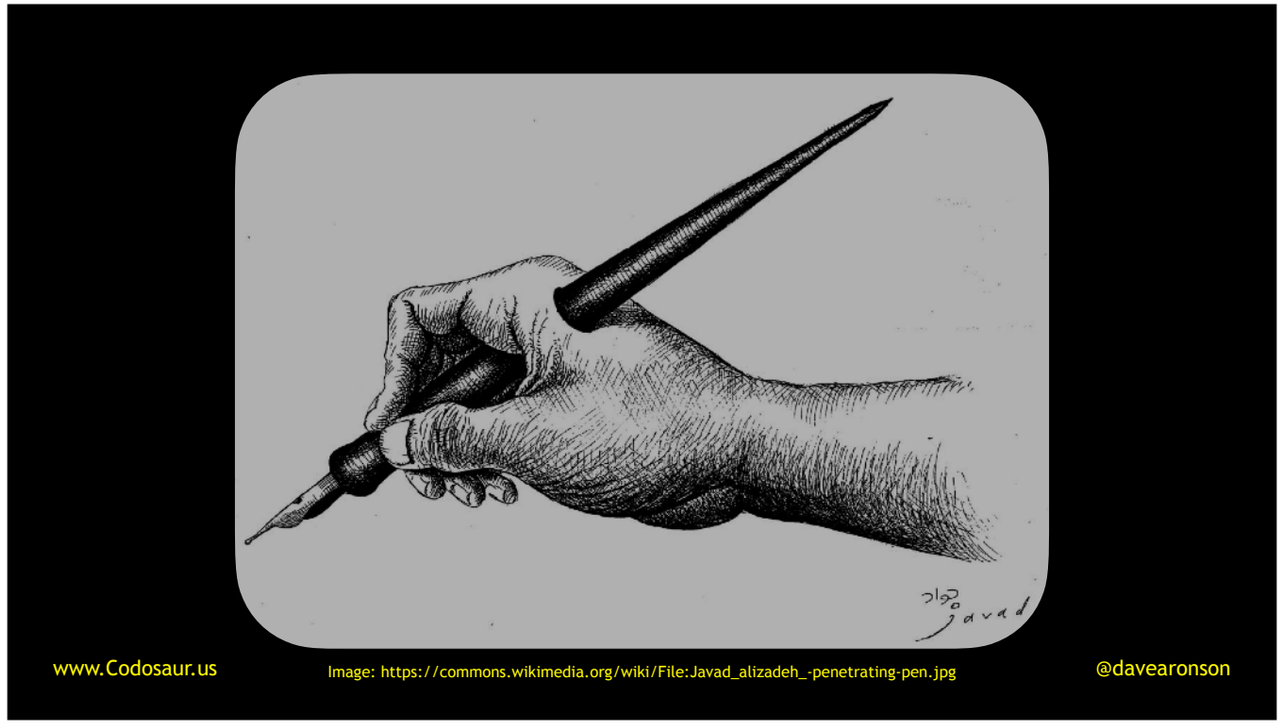


. . . or become unavailable when it’s not supposed to, even when under attack, by someone trying to . . .

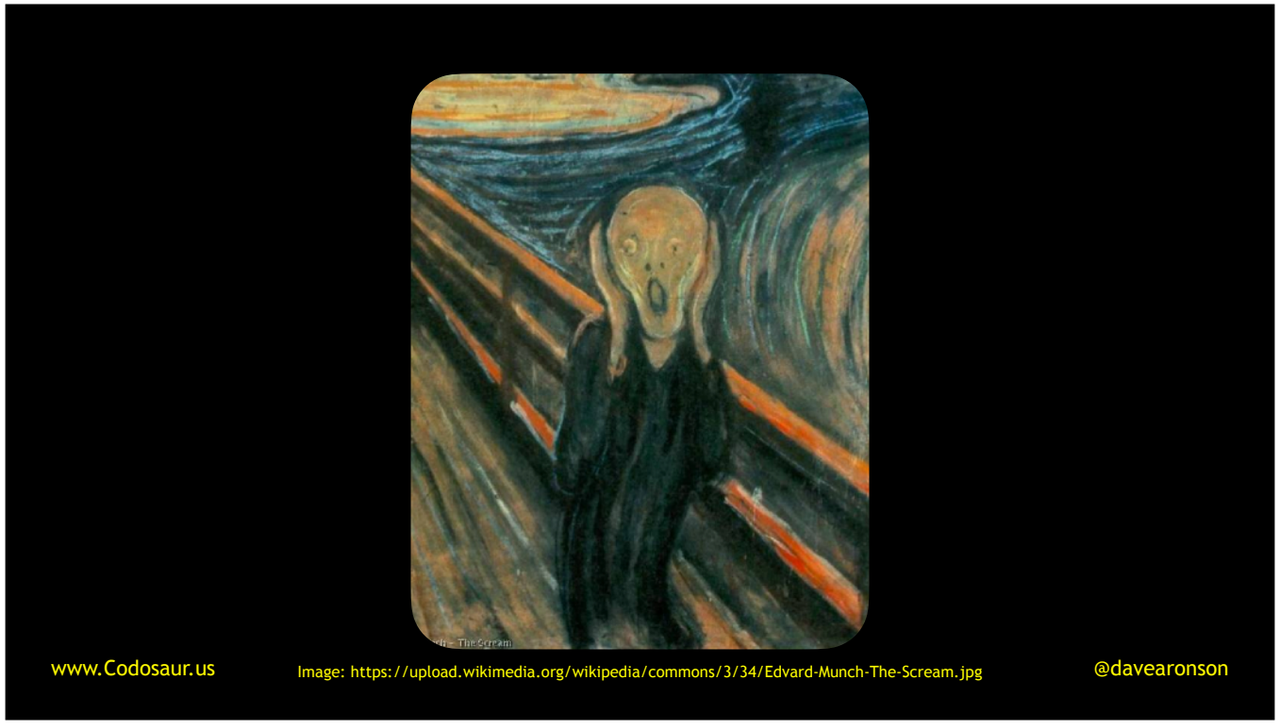
. . . *force* it to do so.

So how do we achieve all *that?*

Once again, we could bring in the experts, and in this case, that would be . . .



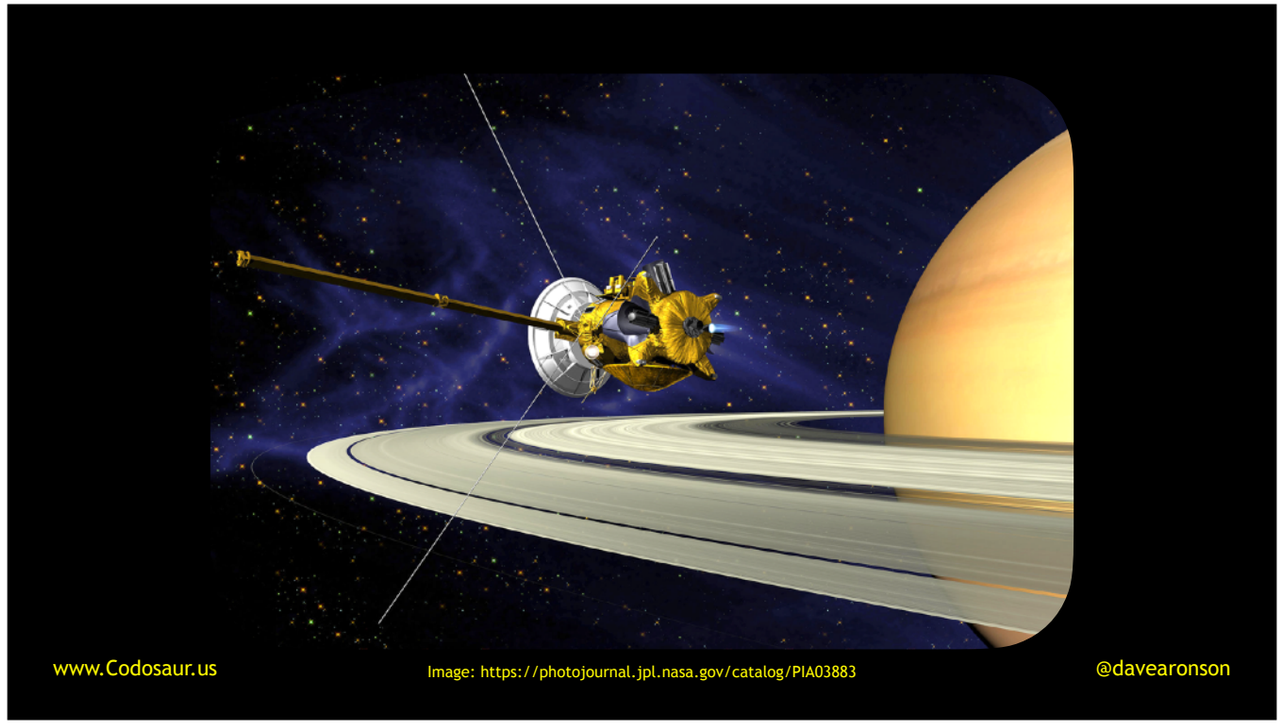
. . . penetration testers, or for short, pen testers. (You can see why I couldn't resist using that image!) The good news is that you don’t have to work for a huge company to use them. Many work for computer security companies you can hire on a short one-off contract. However, they are usually quite expensive, and disruptive, because they *need* to test the *production* system. (QUICK CUT TO NEXT SLIDE!)



So, once again, we'll usually have to do without them, *but*, we can use some of their tools, especially software such as . . .

. . . static analyzers, which simulate the execution of our program . . .

. . . fuzzers, which test our program’s reactions to various kinds of invalid inputs, in the "fuzzing" technique I mentioned earlier . . .



. . . and probes, which test our system for vulnerability to specific known attacks.

Many of these are available as open source, or at least free or cheap software.

But even without such software, which your workplace might be leery of allowing, we can still get a long way just by using the *mindset* of a penetration tester, or indeed pretty much any security professional. A large part of that is to ask ourselves . . .

. . . what could go wrong, either accidentally or thanks to an attacker. Here the tone of voice is very important, it's not "*What* could go *wrong?*", as though we think nothing could, but almost statement-like, "What could go wrong.", as if to say, "I know a lot *could* go wrong, I'm trying to list it, (LOOK UP) I don't need a demonstration thankyouverymuch!"

Once we've run out of answers, then for everything we've come up with, we must somehow . . .

. . . handle it. Yes, that's extremely vague, but how to handle something is going to vary immensely, depending on exactly what it is. Our software should handle *all* reasonably foreseeable types of problems, from simple user error like mistyping a filename, to system problems like a full disk, and even external problems like losing the network connection. It should handle these all as gracefully as practical, but *also* give as little useful information as possible to potential *attackers*. And then, whatever response we decide on, we must TEST it, as it is now an important part of our system.

Our next aspect is one often seen as a tradeoff with security: . . .

. . . usability, or ease of use. If our software doesn't have this, our users will become frustrated, and may stop using or recommending our software. That could be *disastrous* for a software vendor, or a software-as-a-service company, especially if there are viable alternatives!

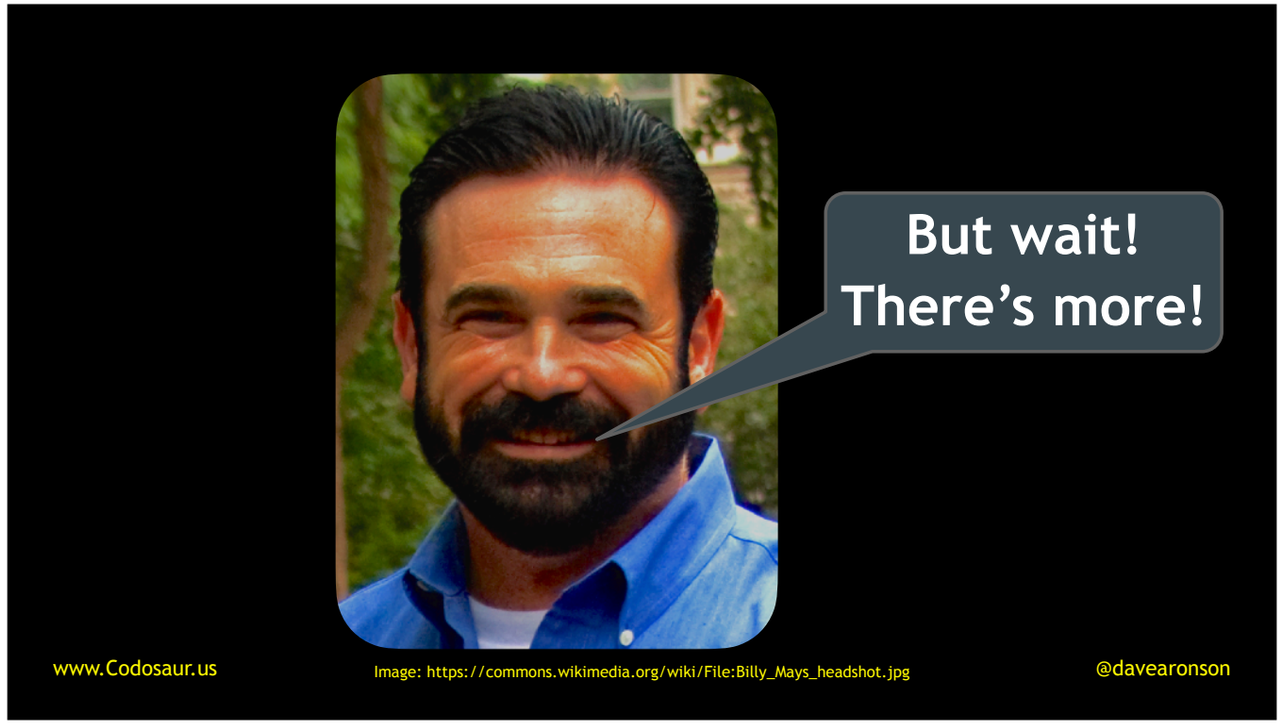
Also, hard-to-use software can lead the user to do the wrong thing. I could give some grimmer examples, but in mid-April 2024, out of the 150 people collecting endorsements to run for President of Iceland, almost half of them actually just meant to endorse some other candidate, not launch their own candidacy. Many of them only found out about the error when someone else endorsed *them!*

Now once again, the definition says "stakeholders" instead of "users". So it should *also* be easy for, for instance, the *customer service* people to *provide customer service* about the software. It should provide them with easy access to all necessary data, in an easily usable format — but *only with* proper *authorization*, because security. It's all interconnected. But anyway, from here on I'm just going to say "user", which will avoid some confusion because most of the other stakeholders don't interact *directly* with the software.

Unfortunately, if we Google software usability, we find mostly things about ensuring that users with various challenges can use our software about as well as the rest of us. In other words, accessibility. That's a good goal in itself, but I'm adding on that it should *easy for everyone* to use, not just *equally difficult!*

To start defining it in more depth: it should be . . .

. . . clear, at all times, what the user can do, should do, and *must* do, how they can do it, and . . .



. . . what *else* the software can do, especially any help facilities.

And all of that should be . . .

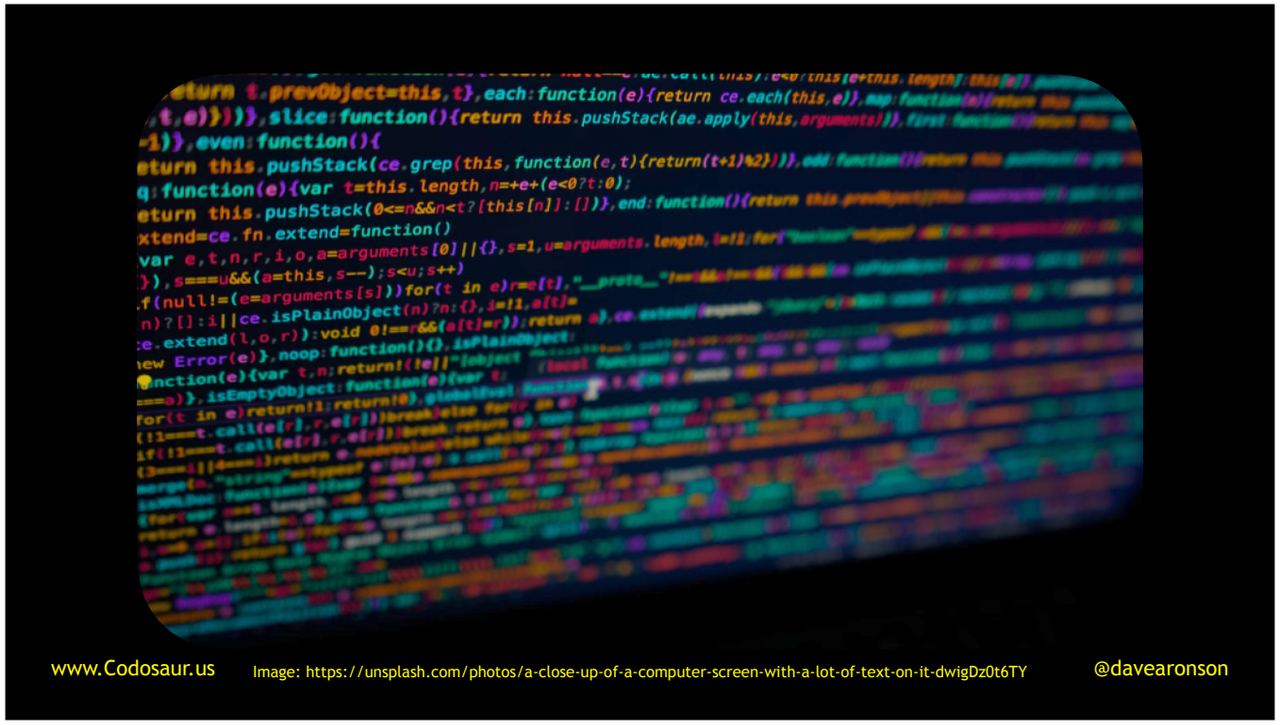
www.Codosaur.us

Image: https://www.flickr.com/photos/jeepersmedia/13943817487

@davearonson

. . . *easy to do*, despite any *challenges* the user may be facing. We can *start* with the things that *accessibility* usually addresses, like lack of vision, hearing, or fine motor control. But there are other whole *types* of challenges we should be aware of. Not everybody is comfortable with computers, knowing even which direction to scroll, or when to single or double click. The user may be illiterate, at least in our character set. They may lack certain other knowledge, especially pop culture references. They may even be of low *intelligence*. Yes, we may like to joke about stupid users, but statistically, about half of them *will* be below average.

Another often-overlooked part of usability is that ALL software should be usable, whether it's some kind of GUI app like web or mobile or desktop, a command-line app, or using an embedded display like a point-of-sale terminal, or . . .



. . . an API, or whatever!

So how do we achieve all this? Once again, ideally we can bring in . . .

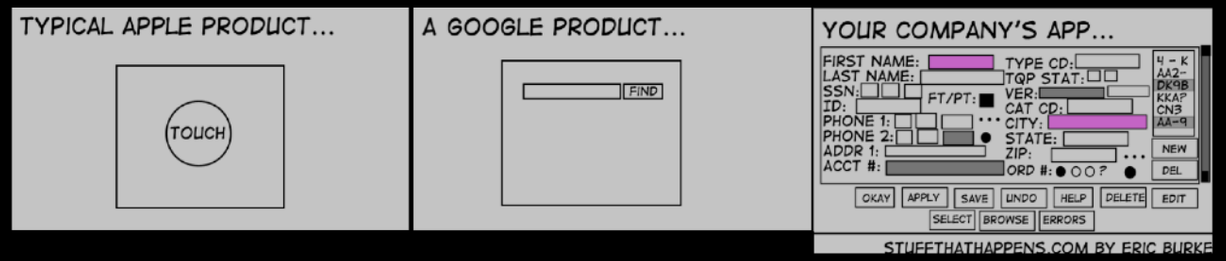
www.Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:The_Experts_MET_DT209332.jpg

@davearonson

. . . the experts. We have a wide range of professions we can get help from! We mainly want a User Experience expert, or maybe a User Interface expert. But a *designer*, even an old-fashioned *print* graphic designer, also has training in principles of practical visual design that can help us, at least in that aspect of usability.

However, as usual, we'll often have to do without any help, but we can go a long way by applying some of their principles. For instance . . .



www.Codosaur.us

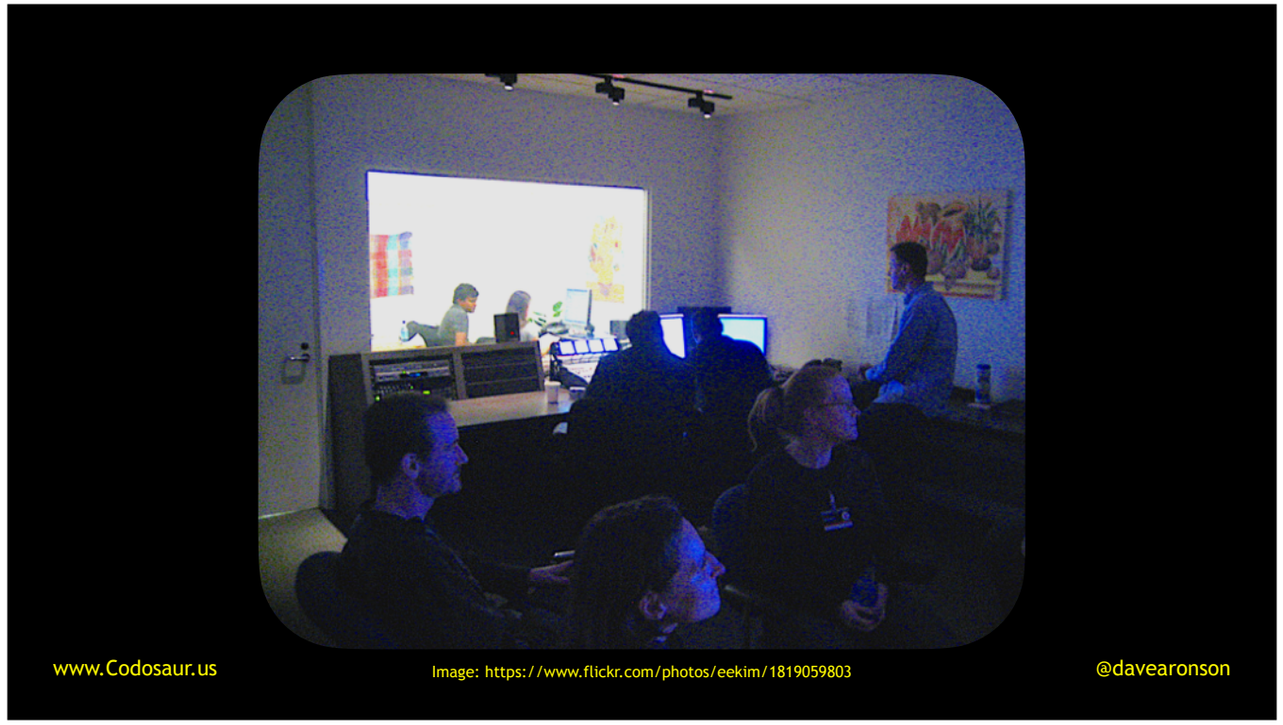
Image: http://stuffthathappens.com/blog/simplicity/
(awaiting permission, but Fair Use anyway due to educational purpose)

@davearonson

. . . here we see an illustration of the KISS Principle, meaning “Keep It Simple, Stupid”, or if we don’t want to be so negative, “Keep It Super-Simple”. Note the *simplicity* of these stereotypical apps from two highly successful companies with reputations for simple ease of use, versus the cluttered unusable mess from “your company”. I think many of us will recognize some of our own work in that, especially back-enders like myself.

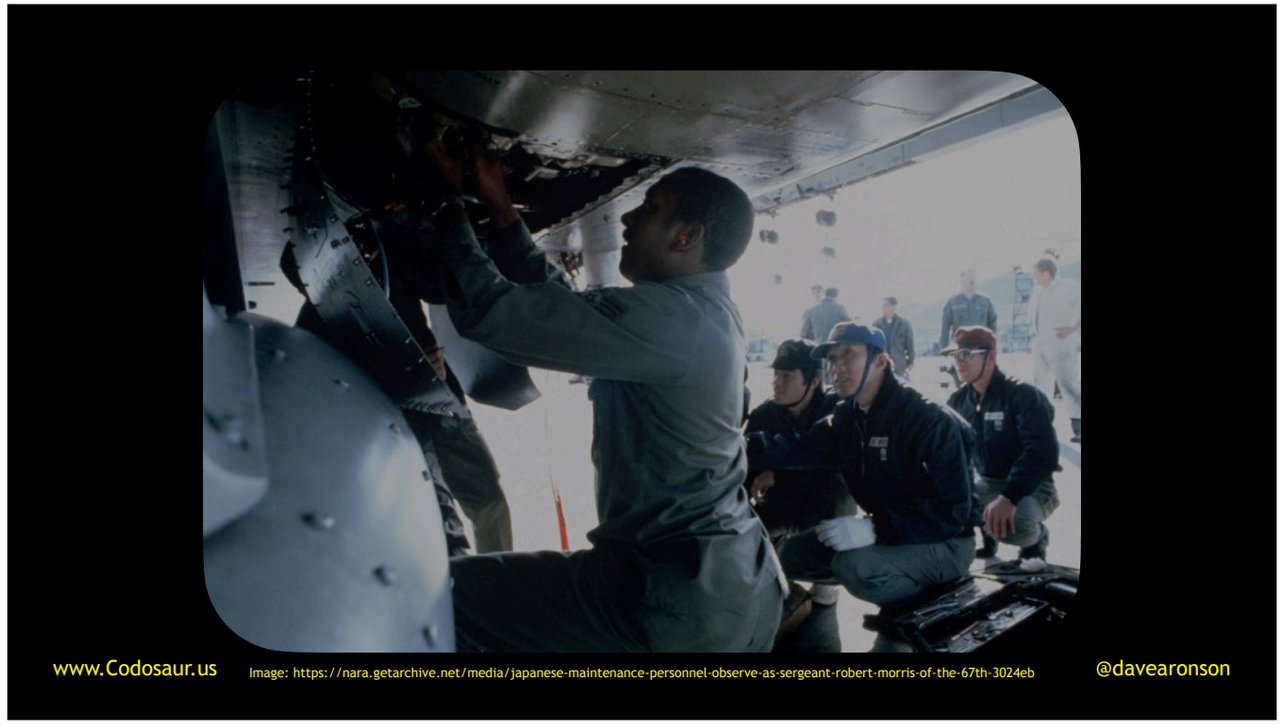
For the non-visual parts, great progress has been made in API design standards like REST, GraphQL, Swagger, and so on. This is also helped by test-driven development, TDD, letting us specify the API while we’re using it, to write tests, *before* we’ve implemented it, rather than finding out that it's hard to use *later*, *after* we’ve done all the work of writing the implementation, so it's harder to change. And similarly, BDD, *Behavior* Driven Development, which is pretty much the same thing at a user-visible level.

Lastly, a user interface may not be as definable and quantifiable as correctness, but it can still be . . .



. . . tested! We can bring in some of our typical users, especially ones that *don't* already know our system, and have them try to do common tasks. We can watch them use it (which is what's going on in this photo), and look for signs, on their faces and screens, of confusion or frustration, or if we're lucky, satisfaction or happiness. Afterward, *ask* them what they found hard or easy, unclear or obvious, fix their pain points, and do more of the good parts.

The next aspect is the one we usually think of most: . . .



. . . maintainability. If our software doesn't have this, then changes take longer, and are more likely to introduce bugs, and developer headaches.

We'd probably all agree that the basic concept is that "maintainable" software is easy to change. (Thank you, Captain Obvious! I think that's him, in the back there.) But I'm adding that it's easy to change, *with* . . .

. . . low *chance* of error (we don't want a *dicey* situation), and . . .

. . . low *fear* of error, even for . . .

. . . a novice programmer, who is also . . .

www.Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:New_Guy_(5895483627).jpg

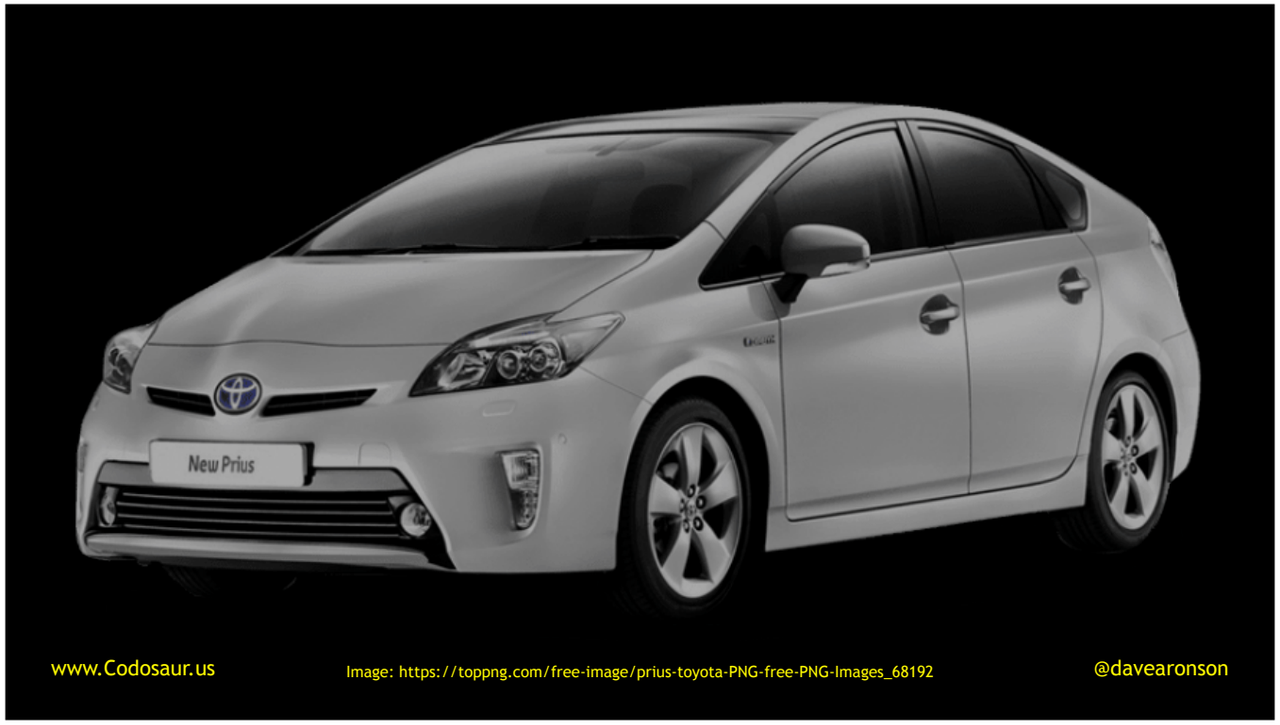
@davearonson

. . . new to our project.

Now how do we achieve all this? For better or worse, the vast majority of software engineering advice is aimed squarely at this. So, rather than expound on lots of generic principles like High Cohesion, Low Coupling, and so on, I'm going to stick to my theme and tell you how . . .

. . . testing can help with maintainability. All of the tests we wrote to verify any *prior* changes, like adding a feature or fixing a bug, form a *regression* test suite, to catch anything we break, that used to work. Just *knowing* that that is *there*, as a *safety net*, will reduce our *fear* of error. And *that* will allow us to progress at a quick and steady pace, with a clear and focused mind, rather than creeping along … slowly … and erratically … because we're terrified … of breaking something accidentally … and not discovering it until the users start complain. And *that* speedup is why I mentioned fear at all.

For the final aspect, software should . . .



. . . go easy on resources. If we don’t have this, then our programs may run slowly, or make the users buy more resources. They could clog the network, or even crash machines by running out of memory or disk space, or drain other resources. Mainly we know about techical resources, like CPU, RAM, and bandwidth, but there are other kinds, such as the user's *patience* and *brainpower*, and the company's *money*!

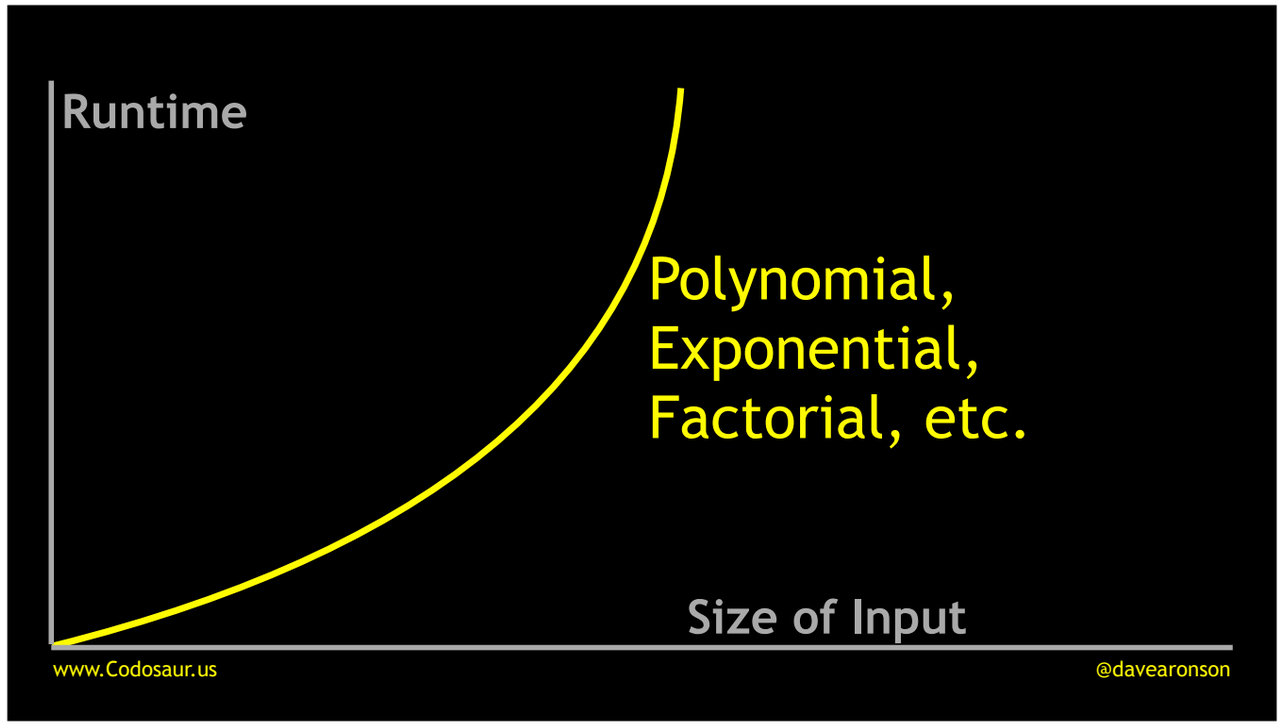
So how do we achieve efficiency? There are many kinds of resources, and many ways each can be abused, so there are many many different kinds of inefficiency, but for the sake of this discussion I’m going to focus on fixing the most obvious and common kind: slowness.

It’s common to have a program run slowly, then we stare at the code, spot where we think it’s inefficient, spend a long time optimizing that little piece, run the program again, and . . . it’s still slow! Right? Has anybody else been there? (PAUSE!) Don’t do that!

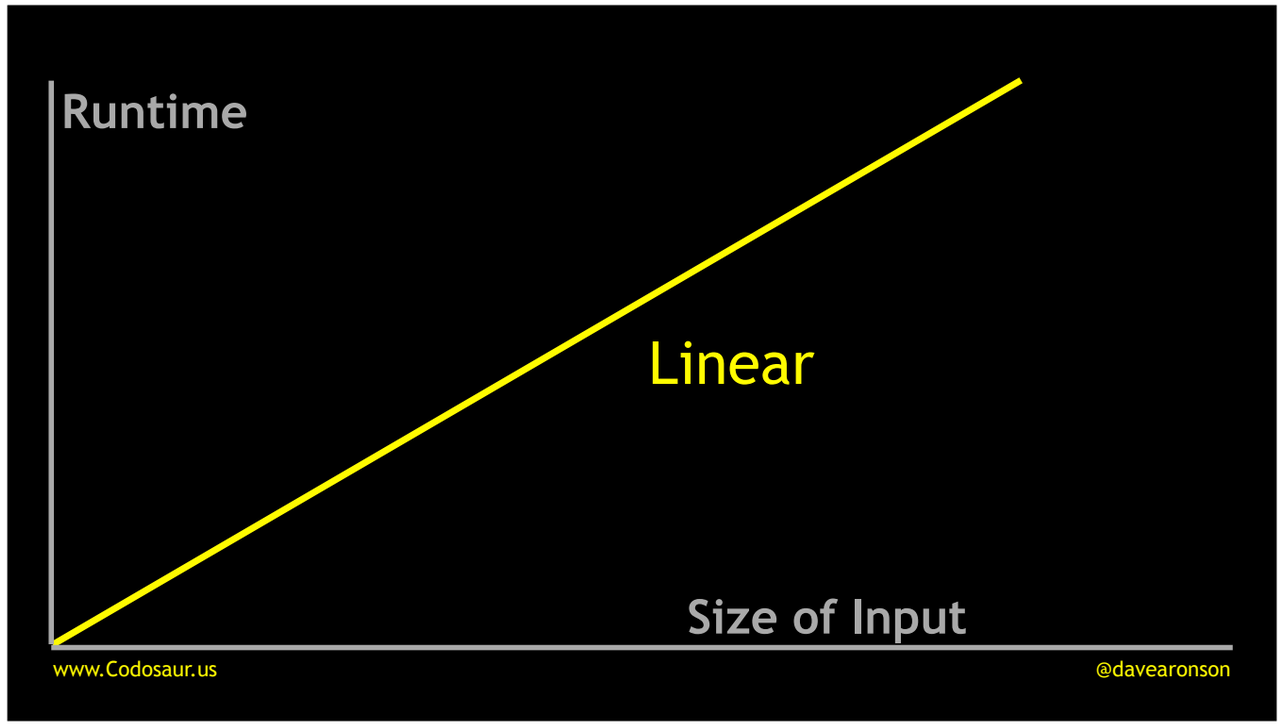
Measure it instead! Humans aren't really very good at spotting the inefficiencies, especially when buried in a dependency, but there are *profilers* and *packet capture programs* and such, that will tell us *exactly* where, or at least when, we're using too much of the *technical* resources.

Once we've found the slow spot, though, there's still the question . . .

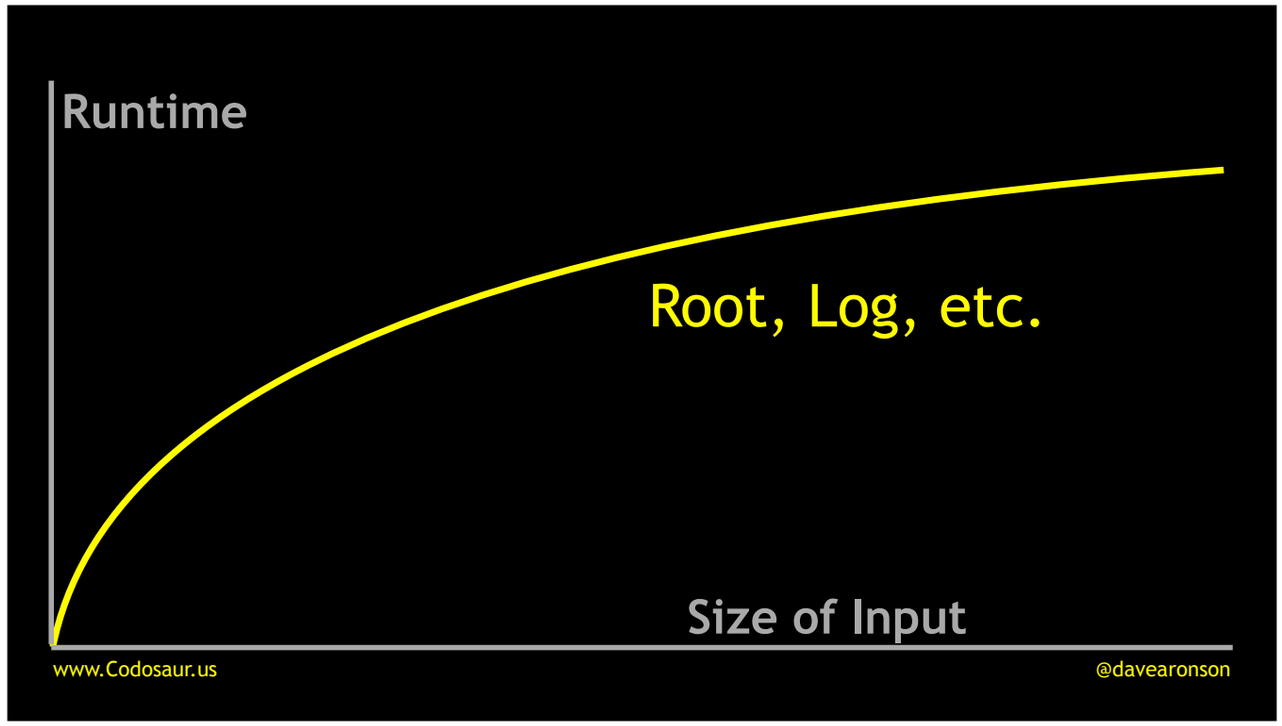
. . . *why* is it slow? Certain kinds of programs tend to have certain problems. For instance, distributed systems may be doing too much communication, or using a slow network. A database-driven system may have an inefficient query, either inherently inefficient or because it's being run against an inefficient underlying data model. But in the general case, usually the problem is either something architectural, which is more complex than I want to get into right now, or a *bad algorithm*. Maybe we're using something with a . . .



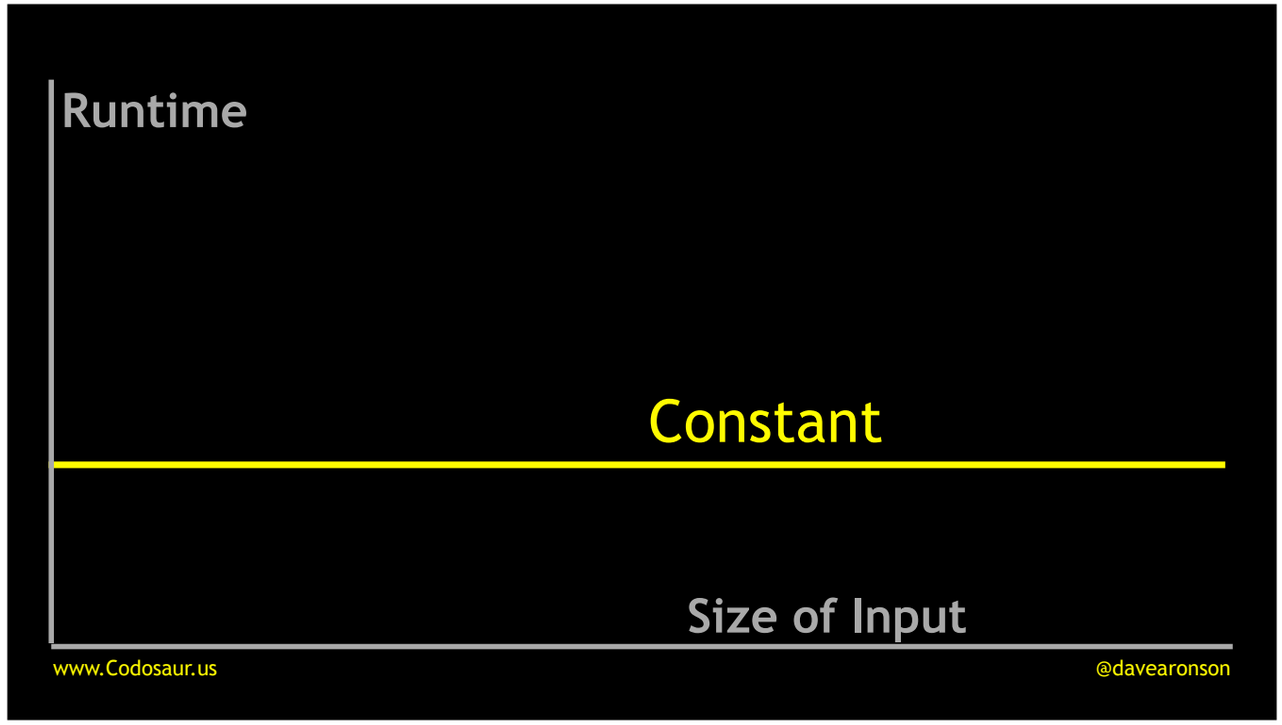
. . . polynomial, exponential, maybe even factorial runtime, when looking at the problem from a different angle, could let us use a better algorithm, such as one with . . .



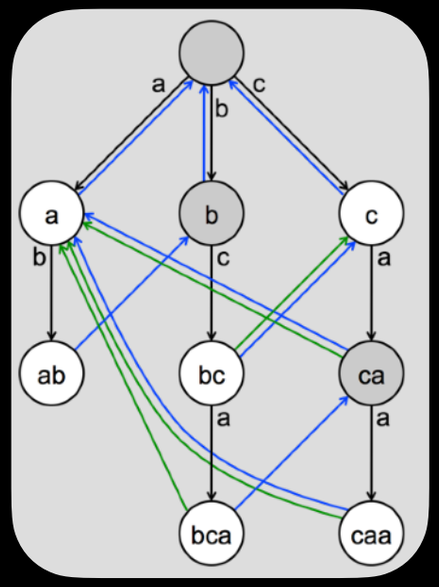
. . . linear or . . .



. . . *root* or *logarithmic* runtime, or maybe even . . .



. . . constant time. Perhaps we're using . . .

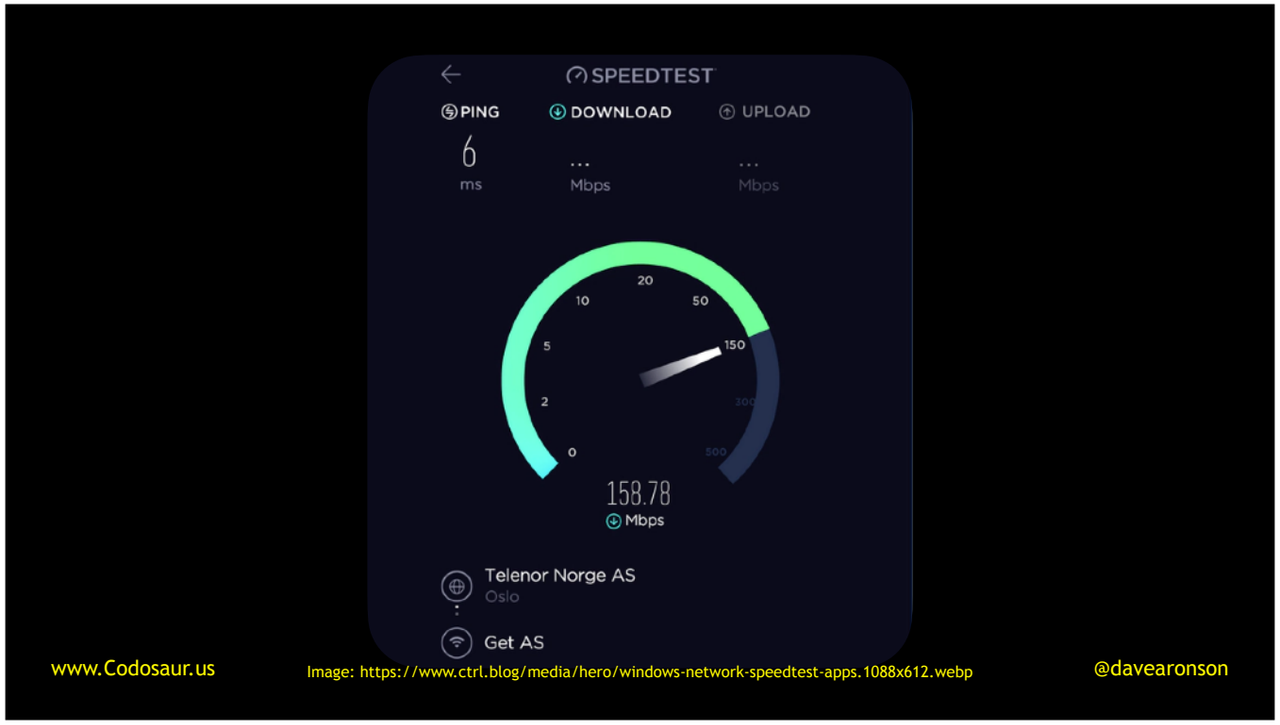


www.Codosaur.us Image: https://commons.wikimedia.org/wiki/File:A_diagram_of_the_Aho-Corasick_string_search_algorithm_extracted.png @davearonson

. . . a bad *data structure,* and *that* is *forcing* us to use a bad algorithm.

The upshot is that, no matter how sick we may be of hearing about data structures and algorithms this, and DSA that, with their abuse in Leetcode based hiring, we do *need* to be familiar with at least the common basic ones, so we can *recognize* them when we see them in real-world problems, *analyze* and *compare* their demands on our resources, and *choose* and *change* and *combine* them. That way, we can use solutions that have stood the test of time, sometimes with ready-made implementations that are well tested and maybe even optimized.

Once it's fast *enough*, we can slap a . . .



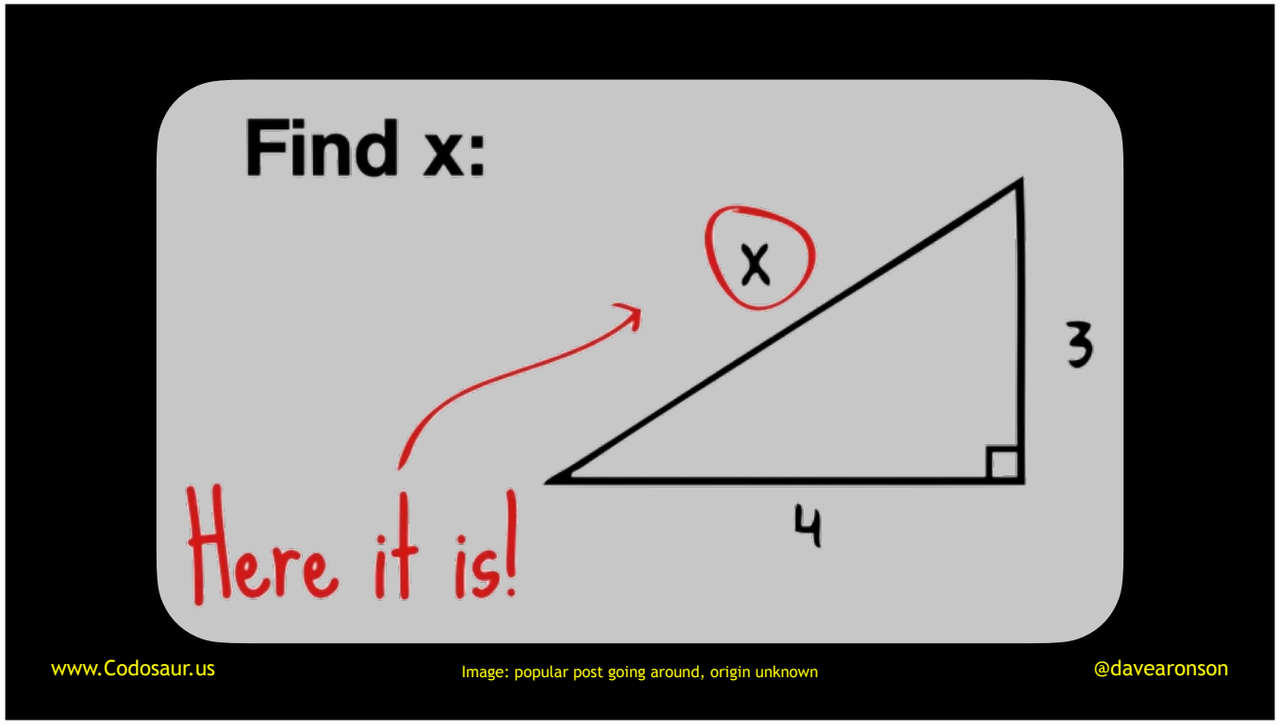
www.Codosaur.us

Image: https://www.ctrl.blog/media/hero/windows-network-speedtest-apps.1088x612.webp

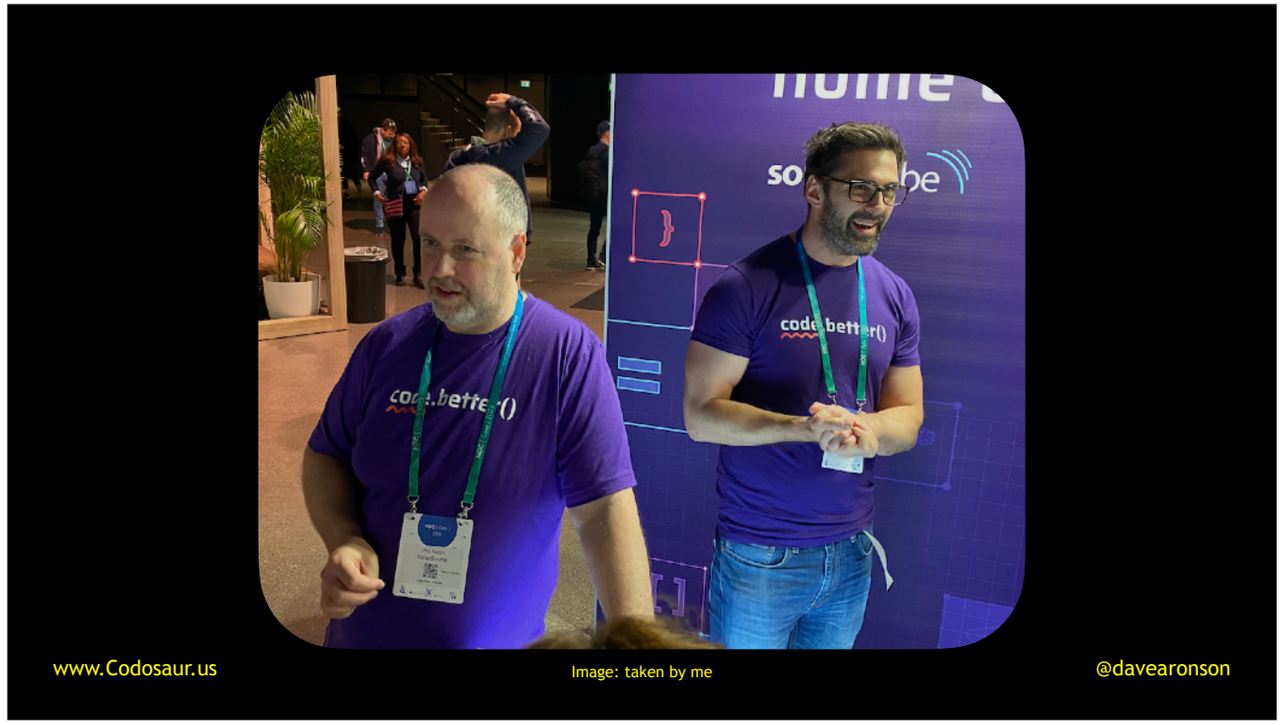
@davearonson

. . . *performance test* around it (you knew I had to advocate doing some kind of testing eventually), to prevent that kind of regression.

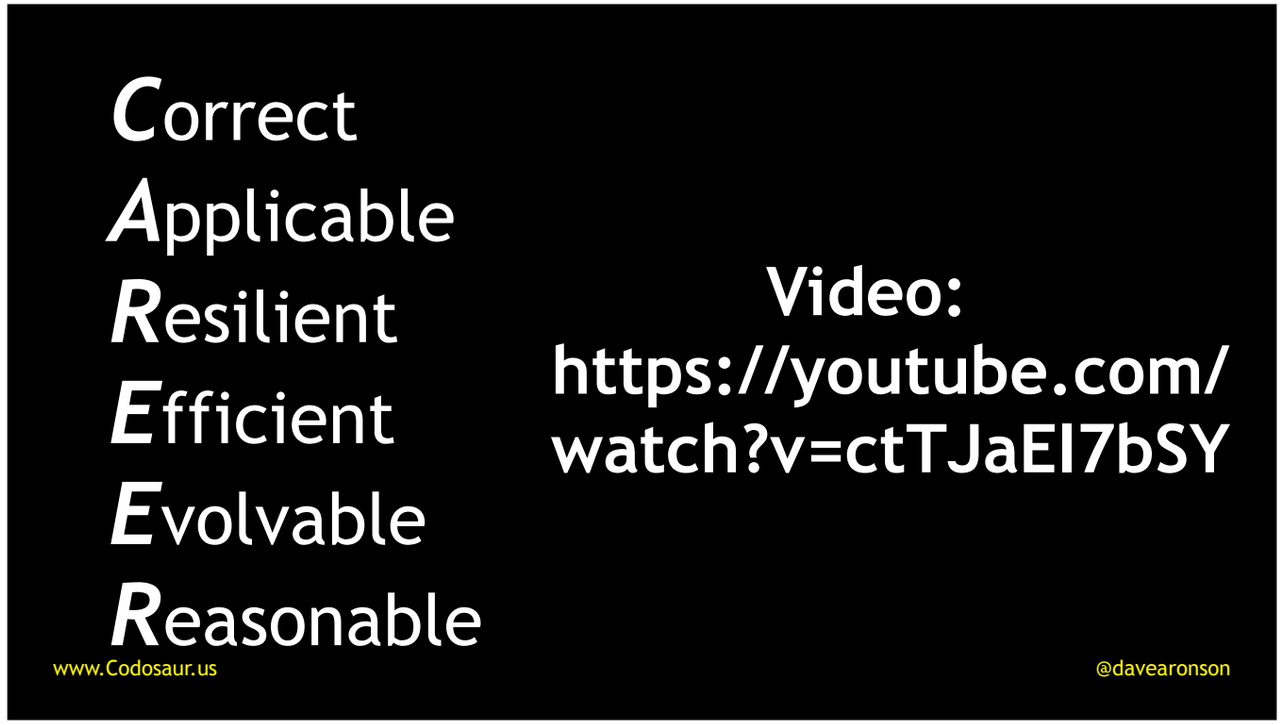
Now that we've explored ACRUMEN in more depth, I'll answer one last frequently asked question: didn't you . . .



. . . find *any* definitions worth using? Sort of. Not at first, while I was still formulating ACRUMEN. However, after I had already been writing and speaking about it, I finally found some I liked. Mainly, those of you who go to a lot of conferences might know . . .



. . . Phil Nash. These guys are *both* frequent international software development conference speakers named Phil Nash, and yes, hilarity often ensues. The on the left spoke at a conference a few years ago, where I also spoke. (Not doing this talk, but the mutation testing one I'll be doing this afternoon.) Before it, I was looking through the agenda, and saw that he was going to speak on *his* definition of software quality, which turned out to be a list of aspects, six of them, going by the acronym, no not ACRUMEN but . . .

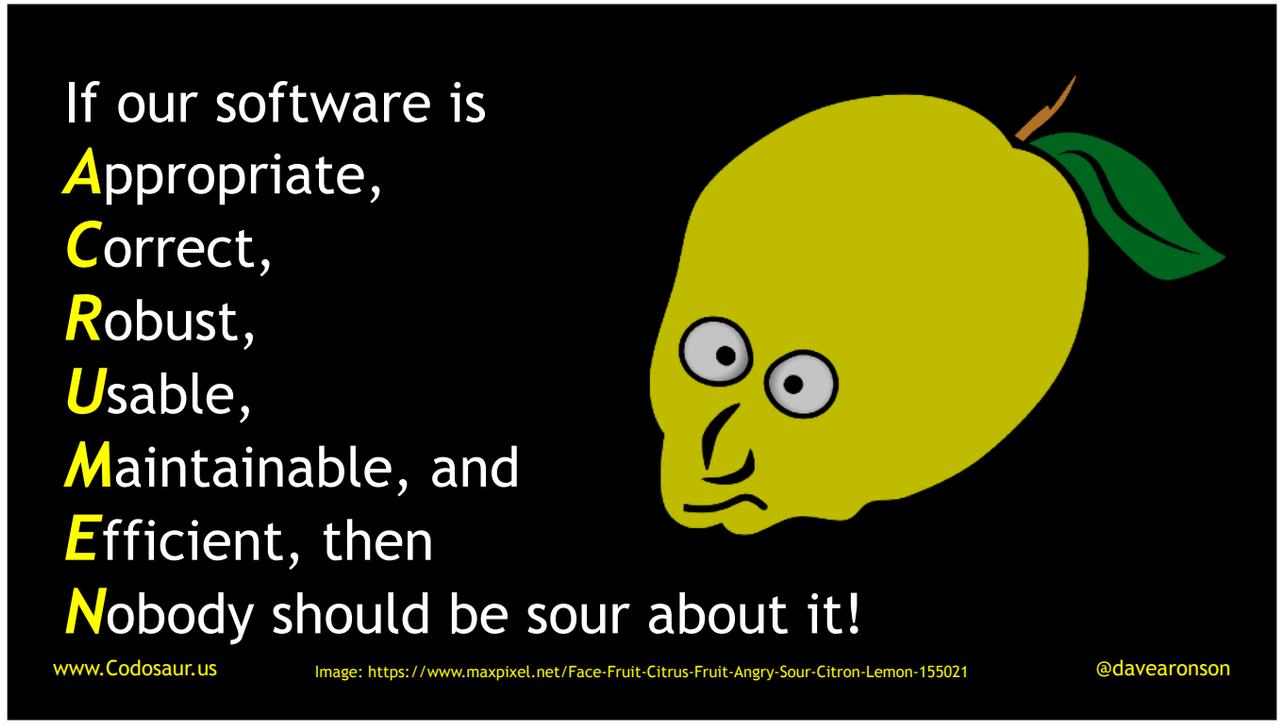


. . . CAREER. After watching some of his videos on it, it was clear that it said mostly the same thing as ACRUMEN, which is how I knew it was good. (PAUSE FOR LAUGHS) No, seriously, it was a bit of a relief to see someone who was already well-respected on the conference circuit, thinking along similar lines. It may have been a stupid idea, but at least I wasn't the only one to have it! The main difference is that he split a few things apart that I kept together, and vice-versa. I have the mapping in both directions on my website. Of course there's also the acronym. His has the advantage of being a common, modern, English word, with no excess letters, and even relevant to most people who would use the definition. However, I still prefer mine, because it doesn't repeat any letters, and prioritizes the aspects. Also, when he speaks about it, he tends to focus on the interplay between aspects, versus my deep dives on each one separately, so he makes some great additional points, even after you know all about ACRUMEN, so I would still recommend you watch some of his videos, and there's the link to one.

Also, Stu Crock has a very interesting and very *short* definition:

“the absence of unnecessary friction”. This wasn’t at all the *sort* of definition I was looking for, but it *is* a wonderful way to spark some very useful discussions about sources of friction in your system, and to what extent some of them may be *necessary*. Even the underlying idea that some friction may sometimes be necessary, is an interesting thought.

In conclusion . . .



. . . if we developers remember to make sure that our software is Appropriate, Correct, Robust, Usable, Maintainable, and Efficient, then nobody should have any cause to be SOUR about the FRUITS of our labors.

We’re not going to do Q&A on this, but I’ll be around for the whole conference, and if you think of something later . . .

. . . there's my contact info, plus the URLs for my web page about ACRUMEN, and for these slides, complete with a full script, which I've *mostly* stuck to. Please remember to vote about the talks, and if you don't give me the top rating, for whatever reason, I only ask that you *tell* me what should be improved, and ideally how. Now let's all get out there and *write better software*, now that we know what that even means!