Current time: ~32:40 (slot is 40, reserve 5-10 for Q&A, leaving 30-35), speaking a little bit slowly

NOTE TO SELF: HAVE BIZ CARD READY AS CHEAT-SHEET!

ACRUMEN:

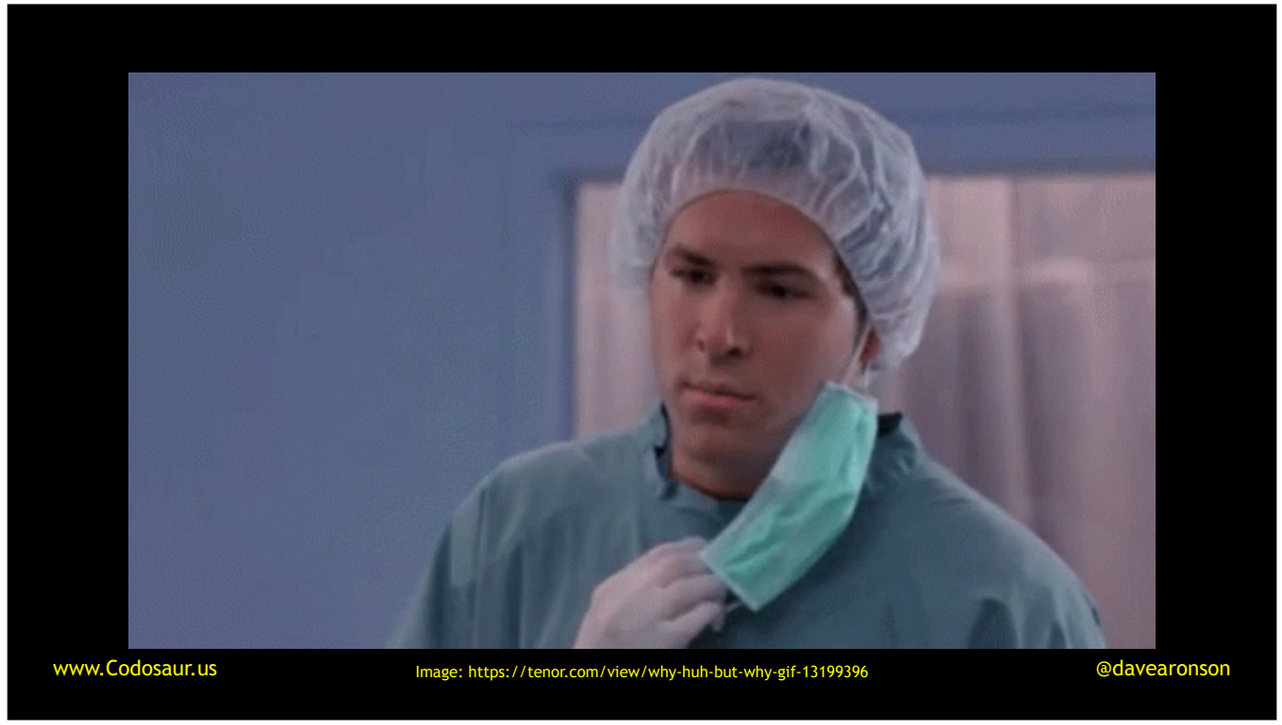
What IS Software Quality, Anyway?

by Dave Aronson
T.Rex-2024@Codosaur.us

www.Codosaur.us

@davearonson

Hi everybody, I’m Dave Aronson, the T. Rex of Codosaurus, and I’m here today to teach you about ACRUMEN, which is my definition of software quality, plus some tips to achieve it. But . . .



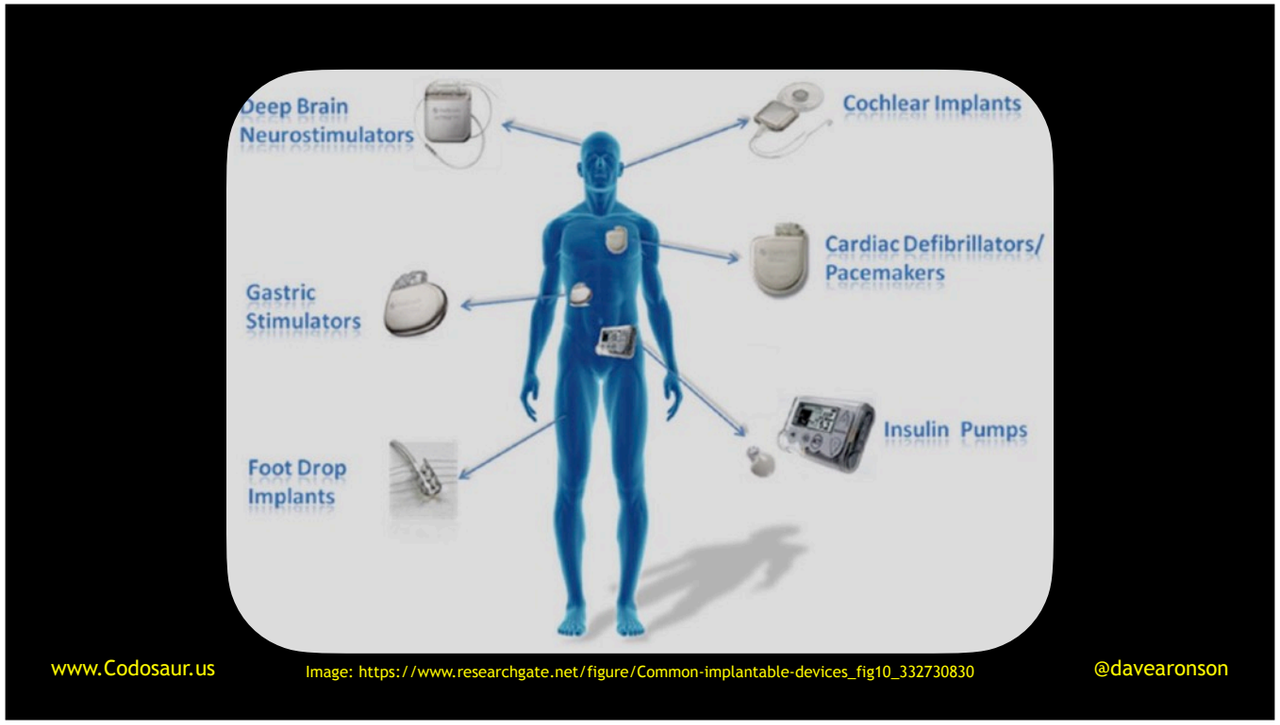
. . . why? I think we'd all agree that the state of software quality is pretty bad, but again, why? Without a *definition*, we have nothing to aim for, so it's very hard to hit the nonexistent target. Furthermore, if we don't *share* that definition, it's very hard to get anyone else to acknowledge that we *have* achieved it. So, I'm trying to get everybody on the same page. (Yeah, good luck with that!)

Before we delve into it, though, I'd like to . . .

. . . level-set some expectations. This isn't meant for the extreme levels of quality usually associated with the software used in . . .


www.Codosaur.us
Image: https://www.flickr.com/photos/vintage_illustration/44546915360
@davearonson

. . . avionics, . . .



. . . implanted medical devices, . . .

www.Codosaur.us

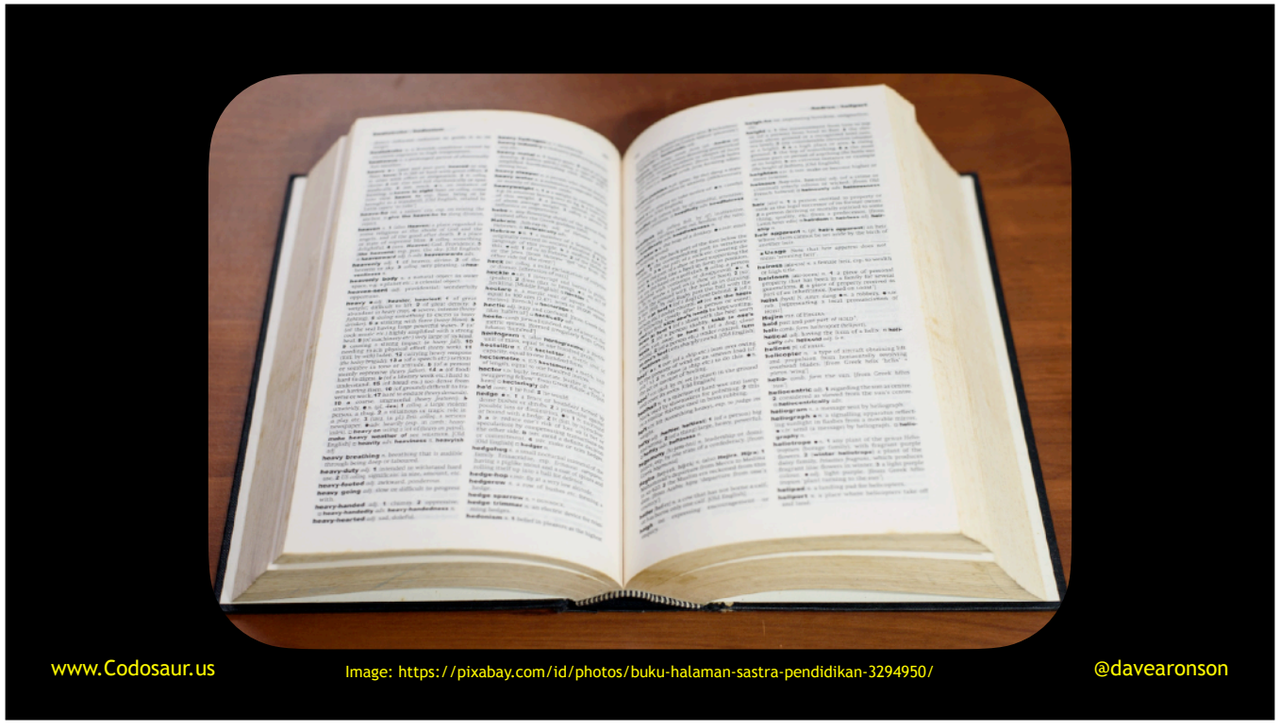
Image: https://commons.wikimedia.org/wiki/File:Kozloduy_Nuclear_Power_Plant_-_Control_Room_of_Units_3_and_4.jpg

@davearonson

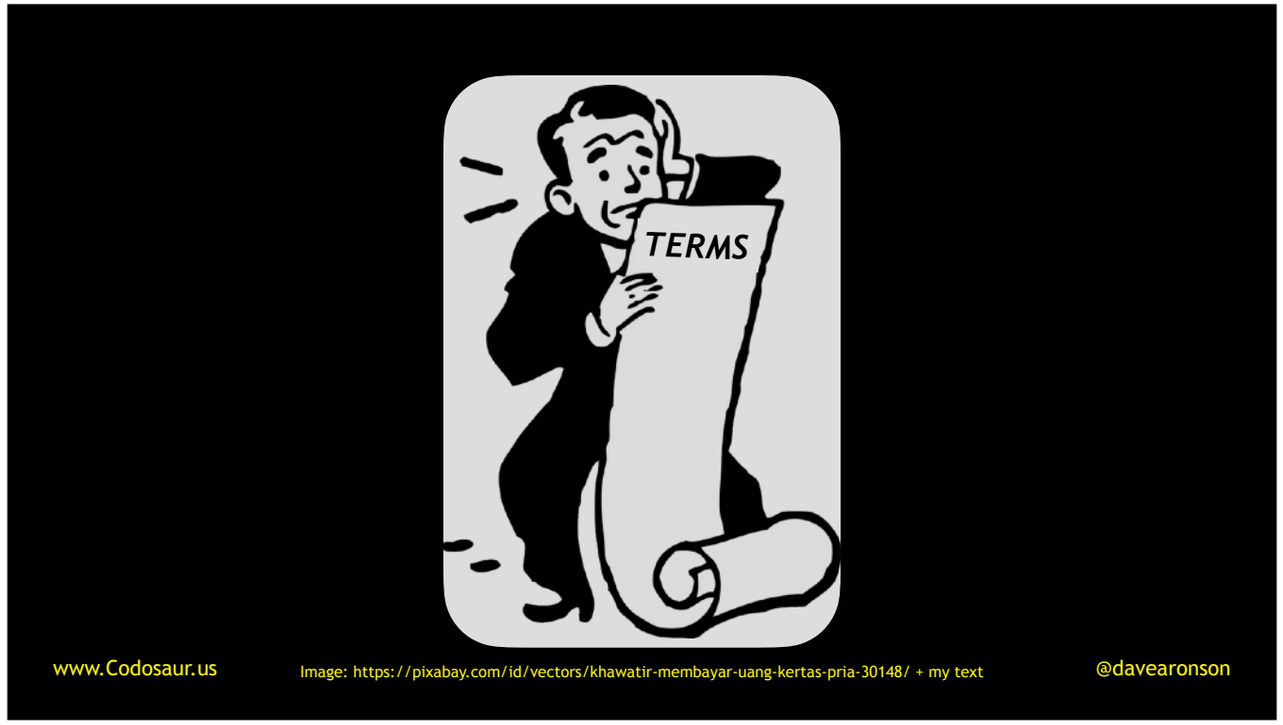
. . . nuclear powerplants, and so on. It's meant for the other five-nines of us, writing consumer-grade systems like web or mobile apps, where, if something goes wrong, there may be frustration on the users' part and embarrassment on ours, but nobody's going to *die*. Those other kinds of industries already have their own approaches, often including regulations and *much* . . .

. . . closer *inspection* than *our* software will ever get.

So with all these examples around, why do we need a *new* definition? A few years ago, I was *looking* for a good *more general* . . .



. . . definition, but the ones I found all had serious problems. Most were . . .



. . . long lists of complicated terms, full of developer jargon. 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 sucks. (And you know most of it does.)

Some definitions were . . .

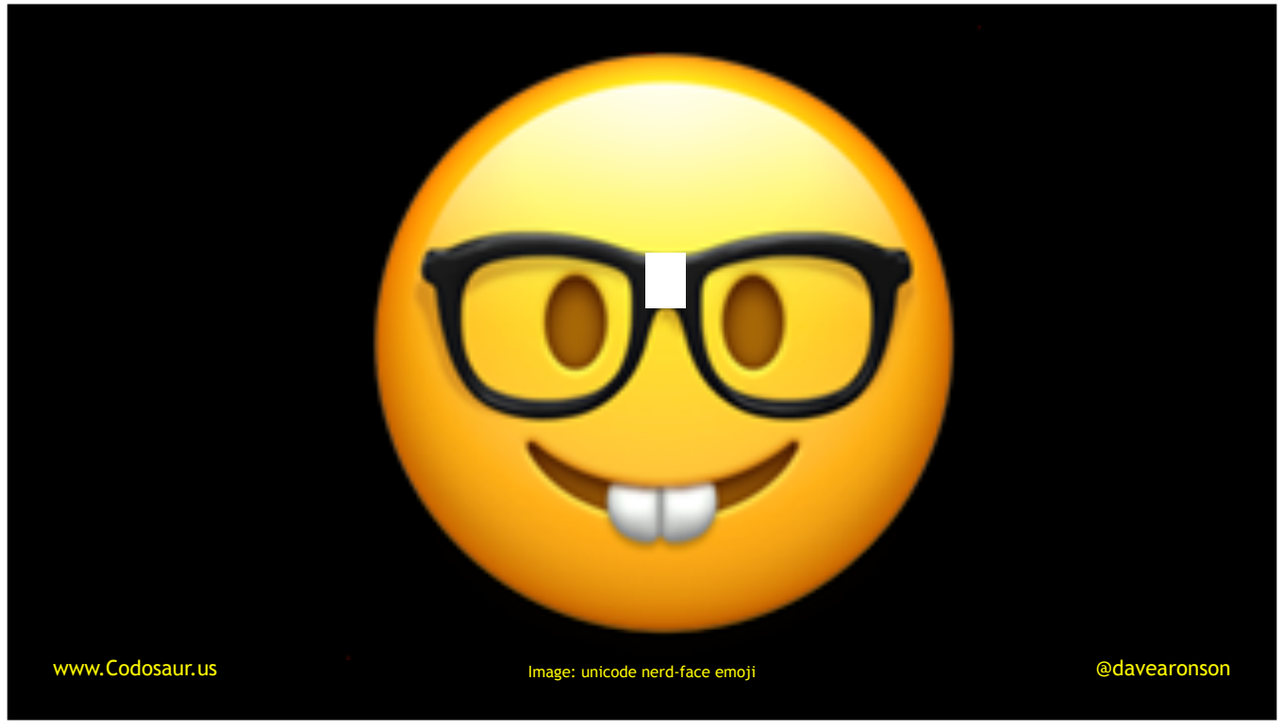
www.Codosaur.us

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

@davearonson

. . . proprietary, requiring us to buy expensive software tools, or at least relatively expensive documents. Some were only applicable within the context of certain technologies, usually also proprietary. I felt that all of that was just plain wrong. I wanted something that *everybody* could use, for *free*.

Some definitions focused exclusively on issues of interest to . . .

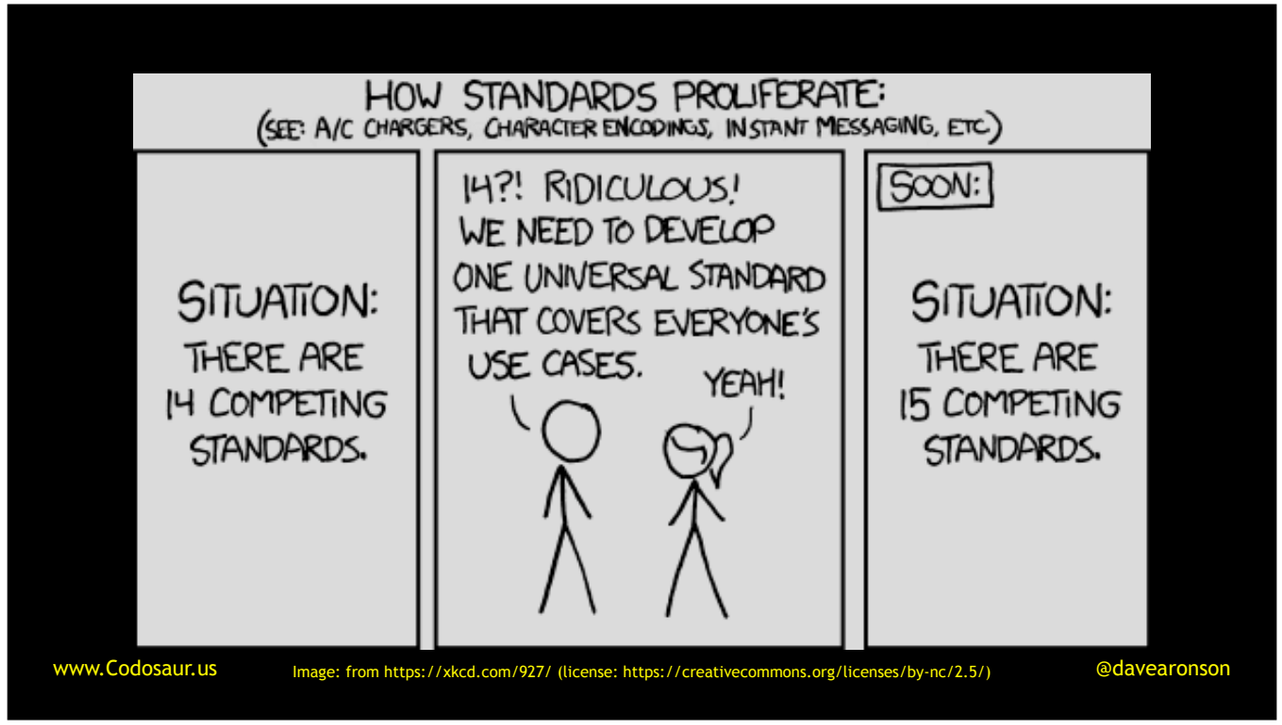


. . . us developers, ignoring the needs of the users and other stakeholders.

Some definitions weren’t even about the software at all, but all about the . . .

. . . process, or the byproducts, dictating that you must hold these meetings or produce those documents. Some of these meetings and documents may be helpful, but to make them the definition misses the whole point. I wanted something more flexible, *de*scriptive rather than *pre*scriptive, and more focused on the software itself.

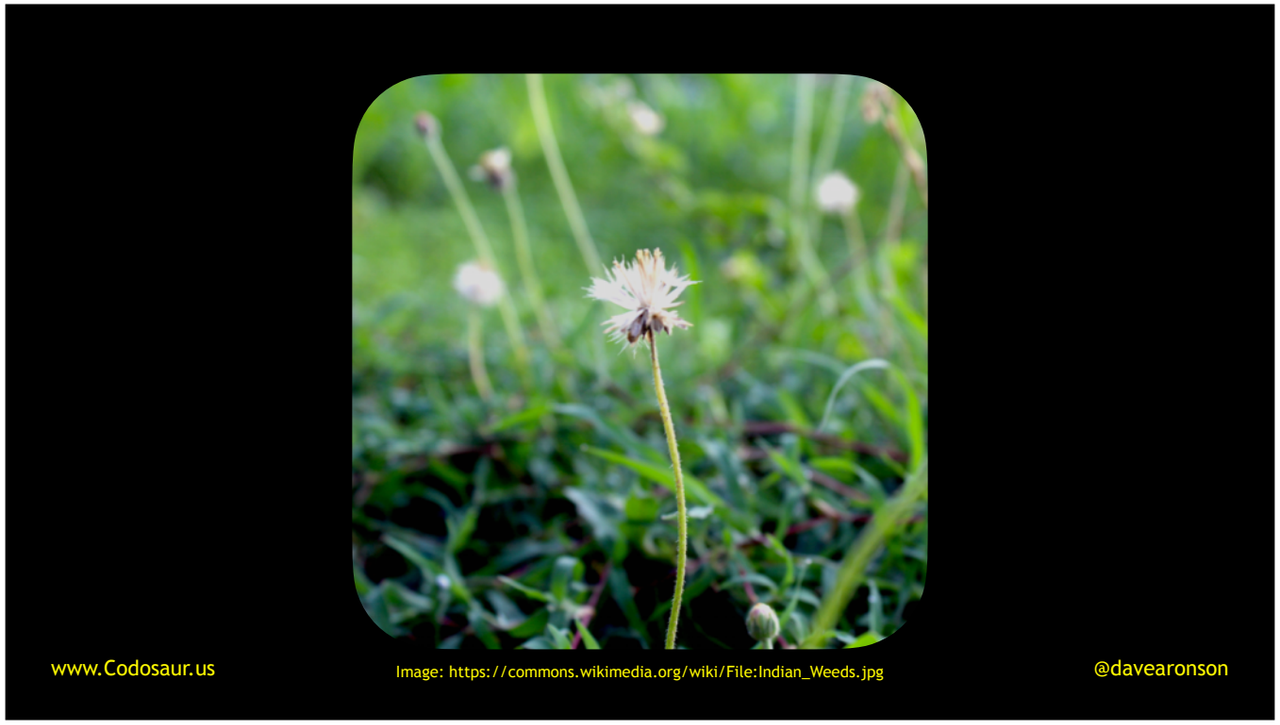
Long story short, I didn't see any that I liked, nor that was commonly accepted, so in the spirit of . . .



www.Codosaur.us Image: from https://xkcd.com/927/ (license: https://creativecommons.org/licenses/by-nc/2.5/) @davearonson

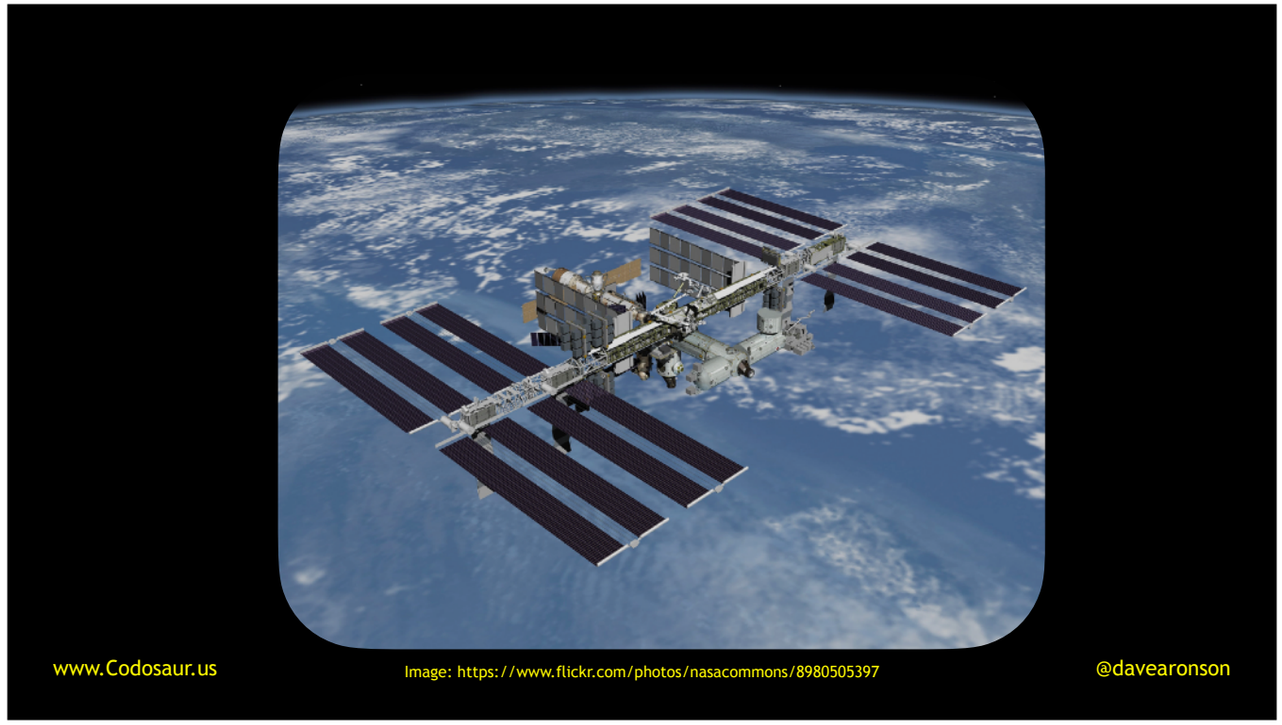
. . . XKCD (PAUSE!), I decided to make my own. But rather than the usual approach of taking the best parts of all the existing approaches, I tried to pare it down to the bare essentials.

To keep it simple, I (step back) zoomed out from . . .



. . . down in the weeds, where we developers tend to live, past the . . .

. . . 10,000 meter view, up to about . . .



. . . low earth orbit, so I could look at continents, not pebbles. That let me trim it down to just six aspects, with simple names and *relatively* simple explanations. The result is a list so short, it literally fits on the back of a business card, and (HOLD UP BIZ CARD!) here's mine to prove it. When I do this talk in person, I give them out as cheat-sheets.

I call this list of aspects . . .

ACRUMEN

www.Codosaur.us

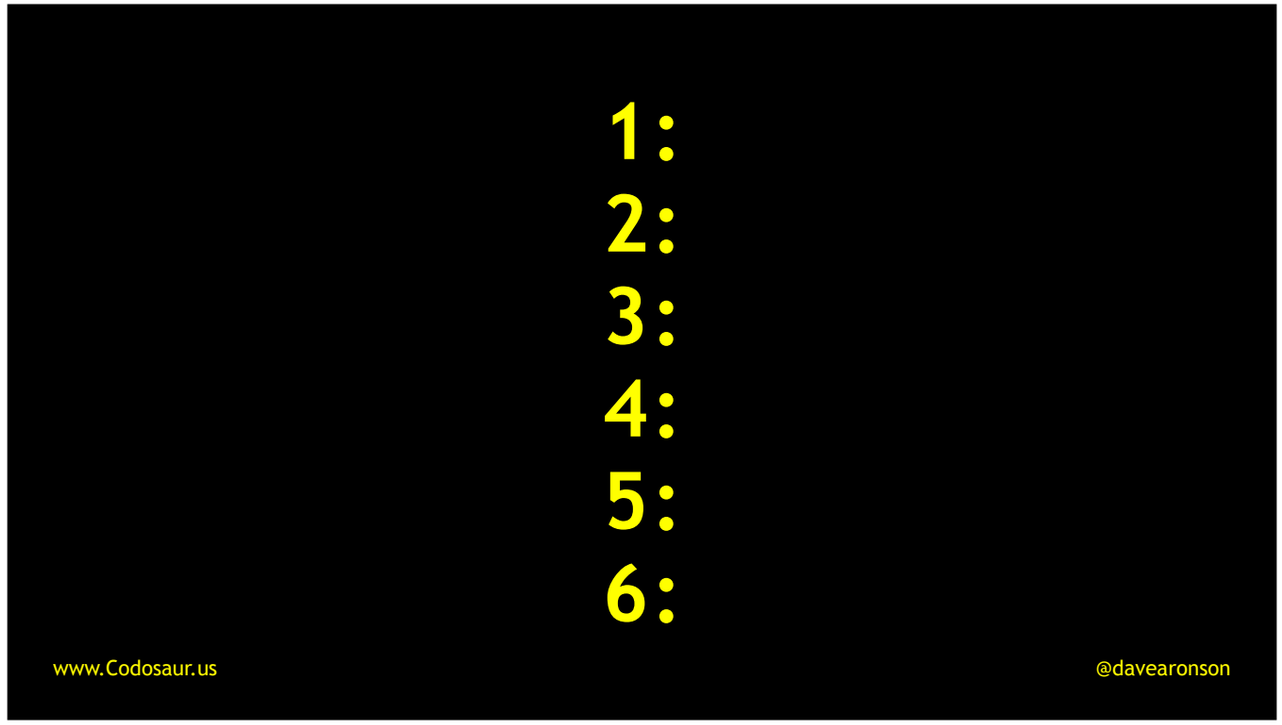
@davearonson

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

. . . lemons. That’s why you'll see lots of bright lemon yellow throughout these slides, it's basically the Official Color of ACRUMEN.

So, I’m taking those sour lemons that life hands us, often by the bushelful, sometimes in the form of low-quality software, and making what I hope will be delicious lemonade, in the form of the ACRUMEN software quality definition. Which finally brings us to the question, what is *that*?

The *acronym* ACRUMEN (try saying that ten times fast!), just takes those six aspects, and . . .



. . . puts them in priority order.

By now you're probably wondering, SO WHAT ARE THE BLANKETY-BLANK ASPECTS ALREADY?! They are that . . .

ACRUMEN means that software should be:

www.Codosaur.us

@davearonson

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

ACRUMEN means that software should be:

Appropriate

Correct

Robust

Usable

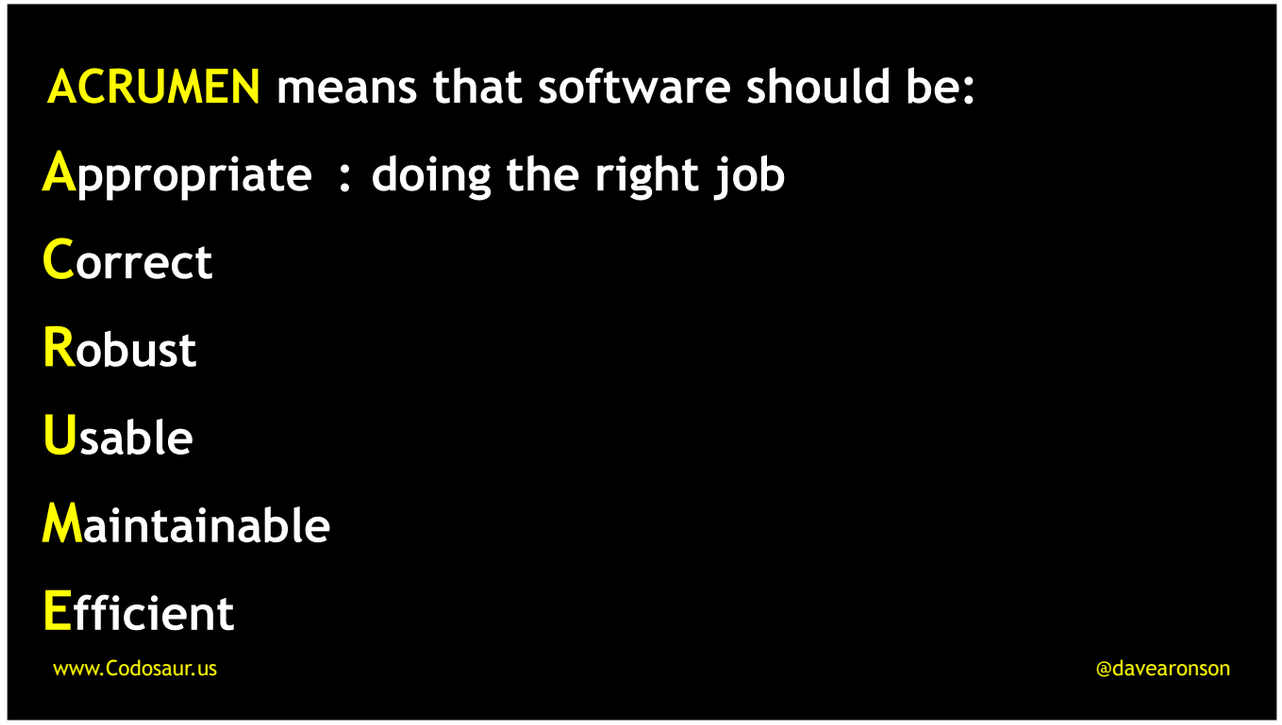
Maintainable

Efficient

www.Codosaur.us

@davearonson

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



. . . *doing what the stakeholders need* it to do, in other words, doing the *right job*. Then it needs to be . . .

ACRUMEN means that software should be:

Appropriate : doing the right job

Correct : doing the job right

Robust

Usable

Maintainable

Efficient

www.Codosaur.us @davearonson

. . . *doing* that job *correctly*, or in other words, doing the *job right*. It should be . . .

ACRUMEN means that software should be:

Appropriate : doing the right job

Correct : doing the job right

Robust : hard to make malfunction *or seem to*

Usable

Maintainable

Efficient

www.Codosaur.us

@davearonson

. . . hard for anyone to make it malfunction, or even seem to, but it should be . . .

ACRUMEN means that software should be:

Appropriate : doing the right job

Correct : doing the job right

Robust : hard to make malfunction *or seem to*

Usable : easy for users to use

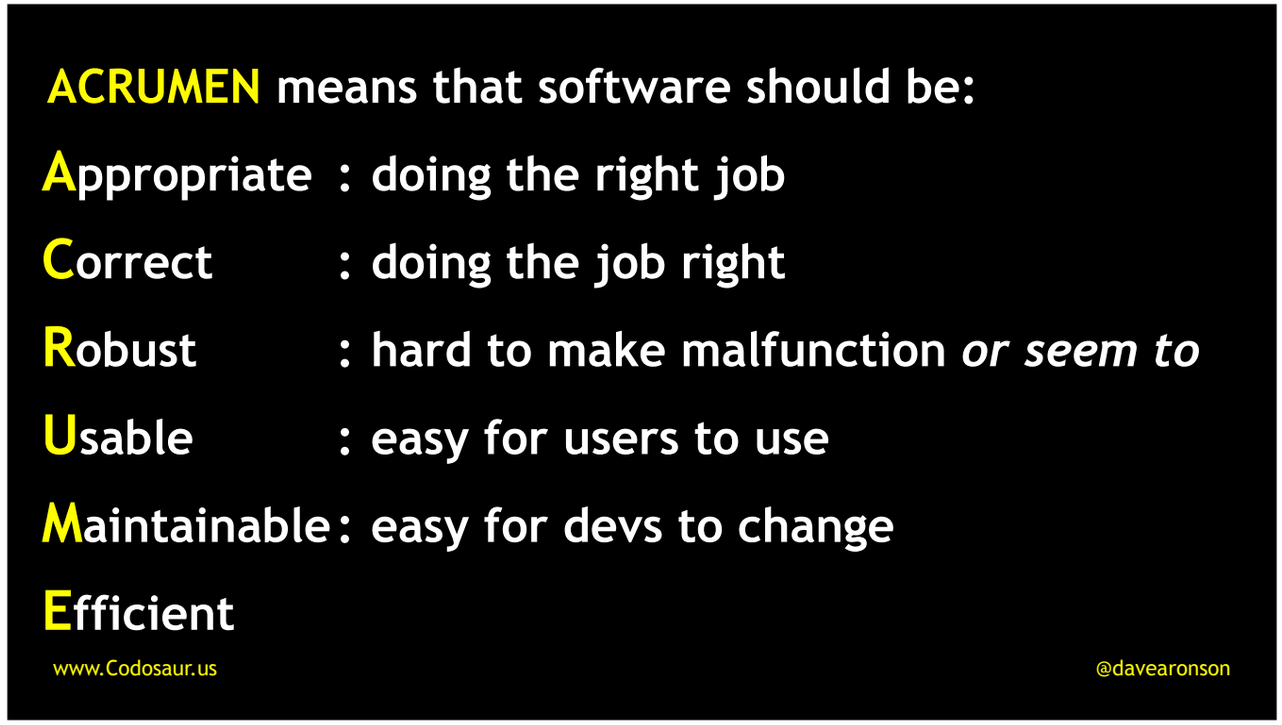
Maintainable

Efficient

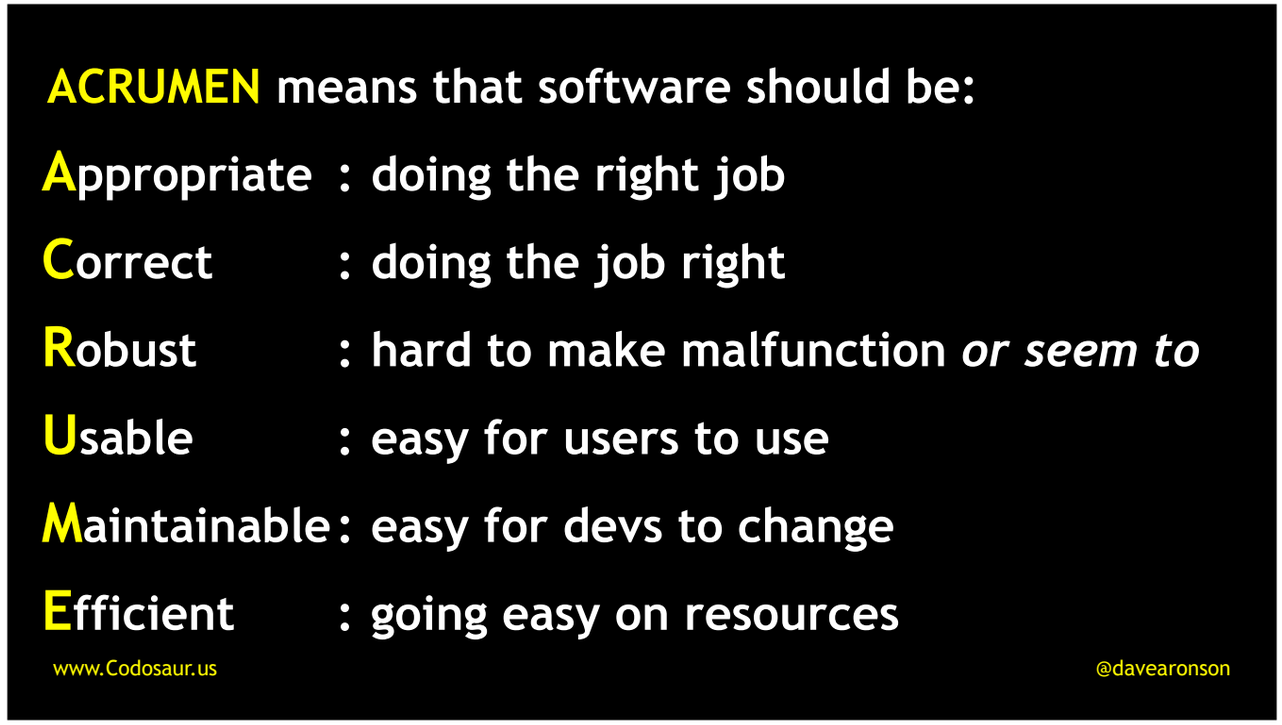
www.Codosaur.us

@davearonson

. . . easy for the users to use and . . .



. . . for the developers to change. (The other way round, not so much.) Last, *dead last* despite how we developers tend to worship this, it should be . . .



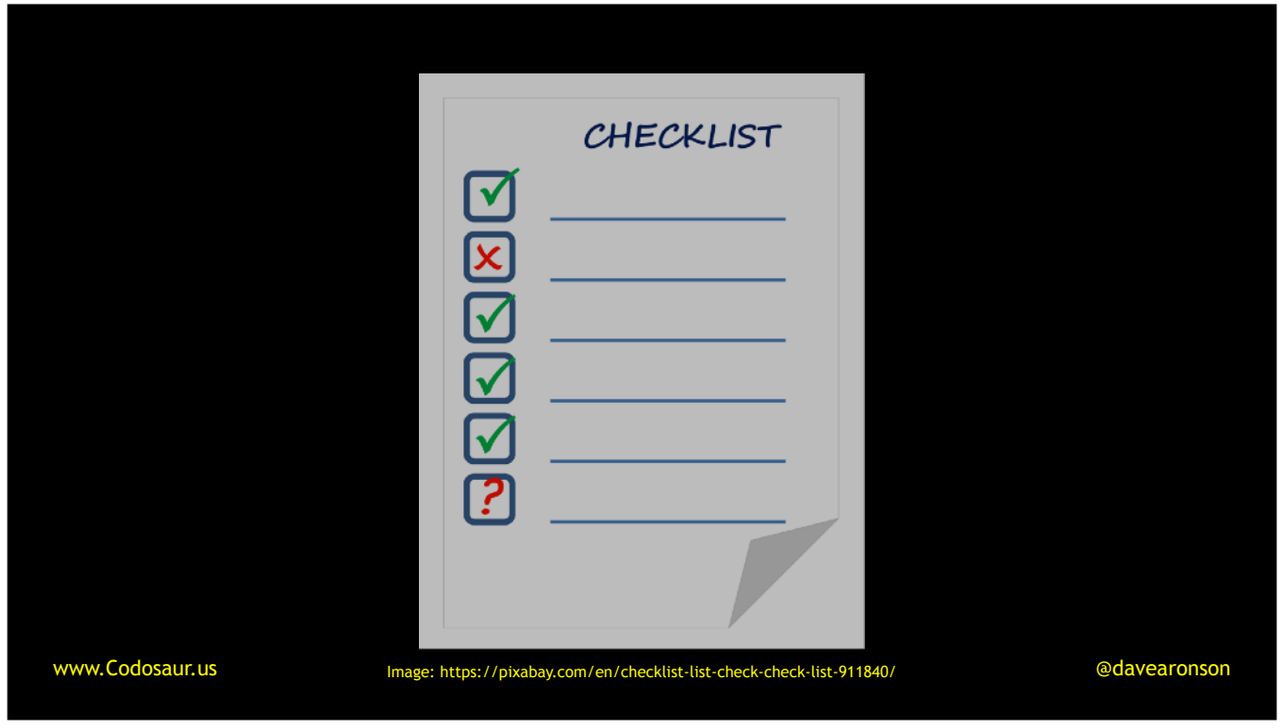
. . . easy on resources, not only the technical ones that we usually think of, but *other* kinds as well.

Now, I've said a few times that it consists of six aspects, and I've told you about six, 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.

While the basic definition is fresh in our minds, I'll address a few . . .

. . . frequently asked questions. The first is, aside from going into detail on the tips, how do we actually *use* ACRUMEN itself, the . . .



. . . list?

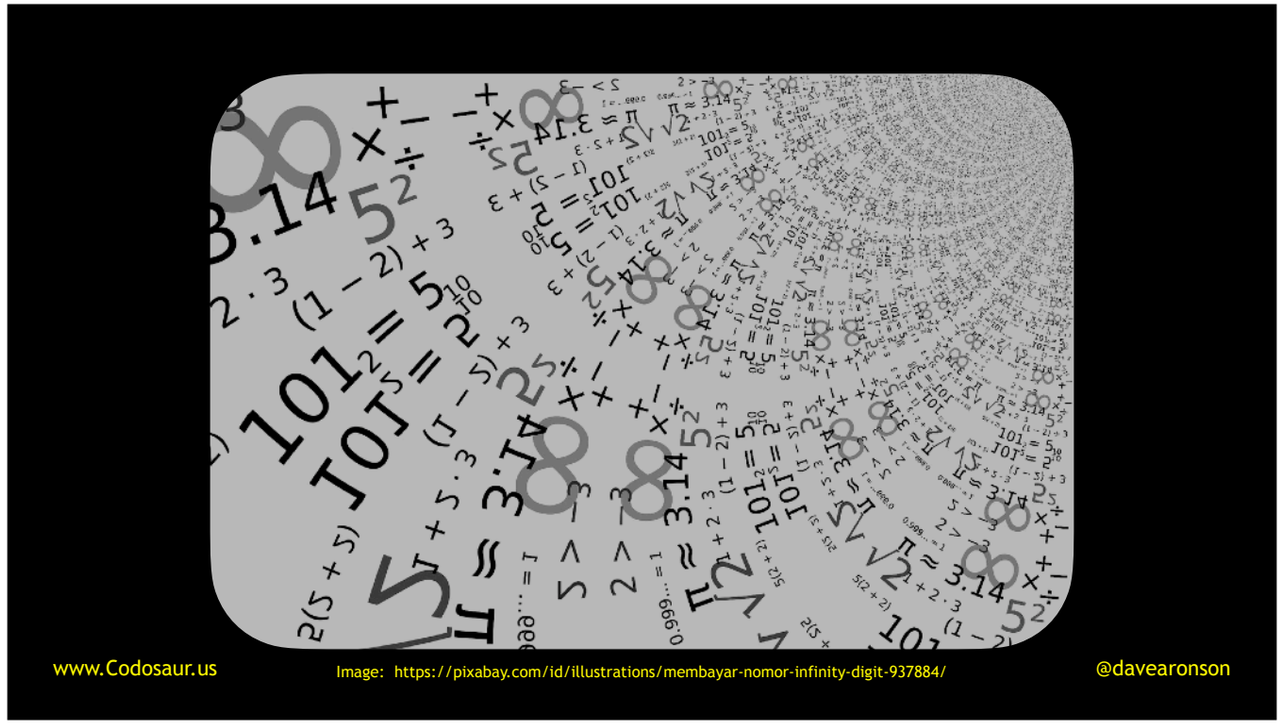
Mainly, we can keep it in mind as a *checklist*, when writing or evaluating software. We can ask, *is* it Appropriate, *is* it Correct, and so on, or *how* good is it in each aspect, on a scale of 1 to 10, or by simple triage, or is it *good enough* for *our needs?* And if the answer is ever that it's 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. 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* the system to be in each aspect.

That leads us straight to the third most frequently asked question (after which we'll backtrack to #2): . . .



. . . how can we quantify this, and boil it down to one number that shows the quality of a piece of software? Mainly I advise that you *don't* do that! Instead, at the very least, . . .

Aspect	Score (out of 10)
Appropriateness	8
Correctness	10
Robustness	7
Usability	3
Maintainability	7
Efficiency	4

www.Codosaur.us

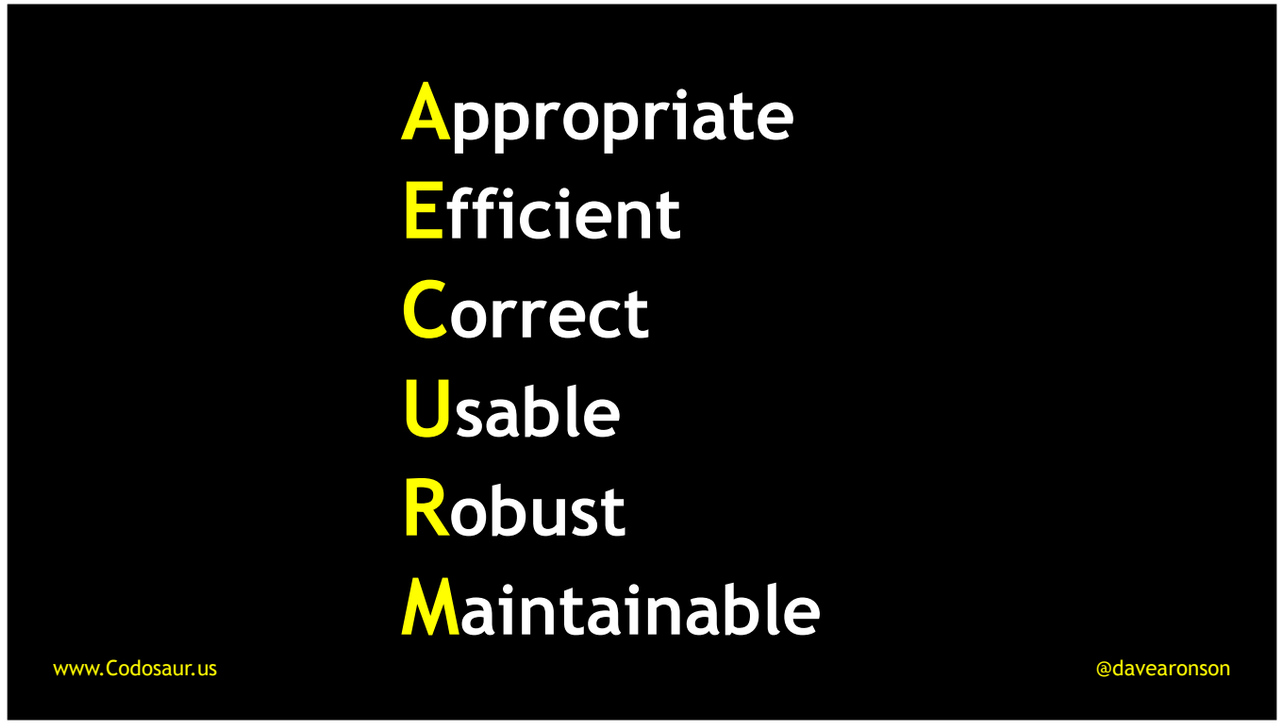
@davearonson

. . . keep *six* numbers, one for each aspect. Otherwise, you lose too much valuable information. A single number *might* tell you that the software is good or bad, but a set of *six* numbers will tell you *how*. For instance, with a chart like this, we're probably talking about a program that does most of what's needed, does it absolutely correctly, but not very efficiently, I would bet slowly, impacting the usability. It's also fairly robust and maintainable, but could still use some improvement there too. These numbers can help prioritize further work on it.

Now let's backtrack to the *second* most frequently asked question: . . .

. . . 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 may well vary. Consider a company-internal command-line physics simulation tool, using a standard algorithm that will never change. It needs to do the right job, may need to be very efficient, but maybe we can make do with a rough approximation, rather than a precisely correct number that would take much longer to calculate. It might not need to be so usable because it's for ourselves, not customers, nor so robust because of the limited interfaces and fewer things to go wrong, nor so maintainable, because the logic is never going to change. So, its list may well look more like . . .



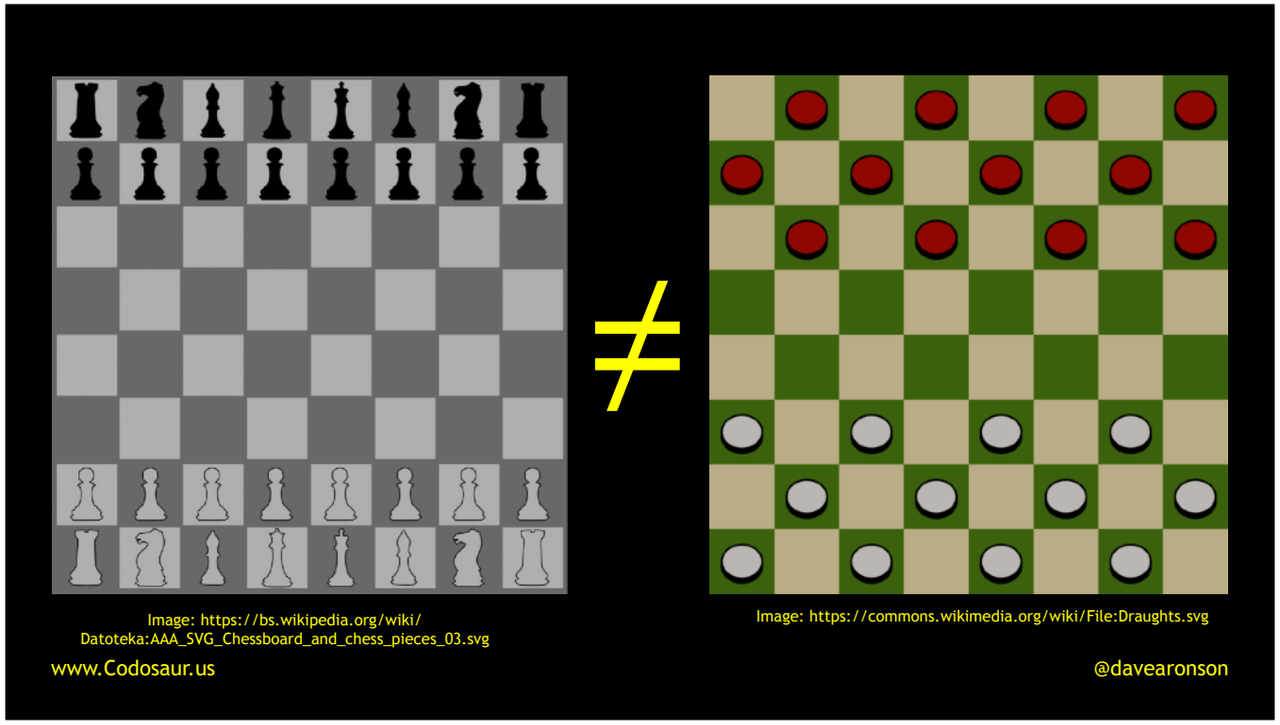
. . . this, AECURM, rather than ACRUME. The only real constant is that Appropriate will always be at the top. We'll see shortly why, because now we're going to look at each aspect in more detail, and up first is of course . . .

. . . 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 point, let's try a little thought experiment. Suppose you want a program to play . . .



. . . *checkers*, and I write for you the world's greatest *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 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! Ideally face to face, or as close to that as possible. Unfortunately, we don't usually get that opportunity. Second best is to bring in . . .

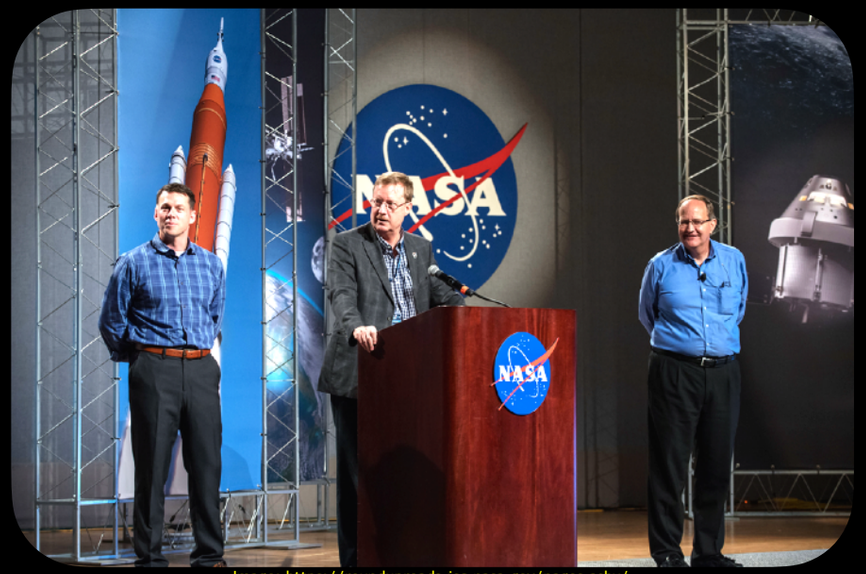
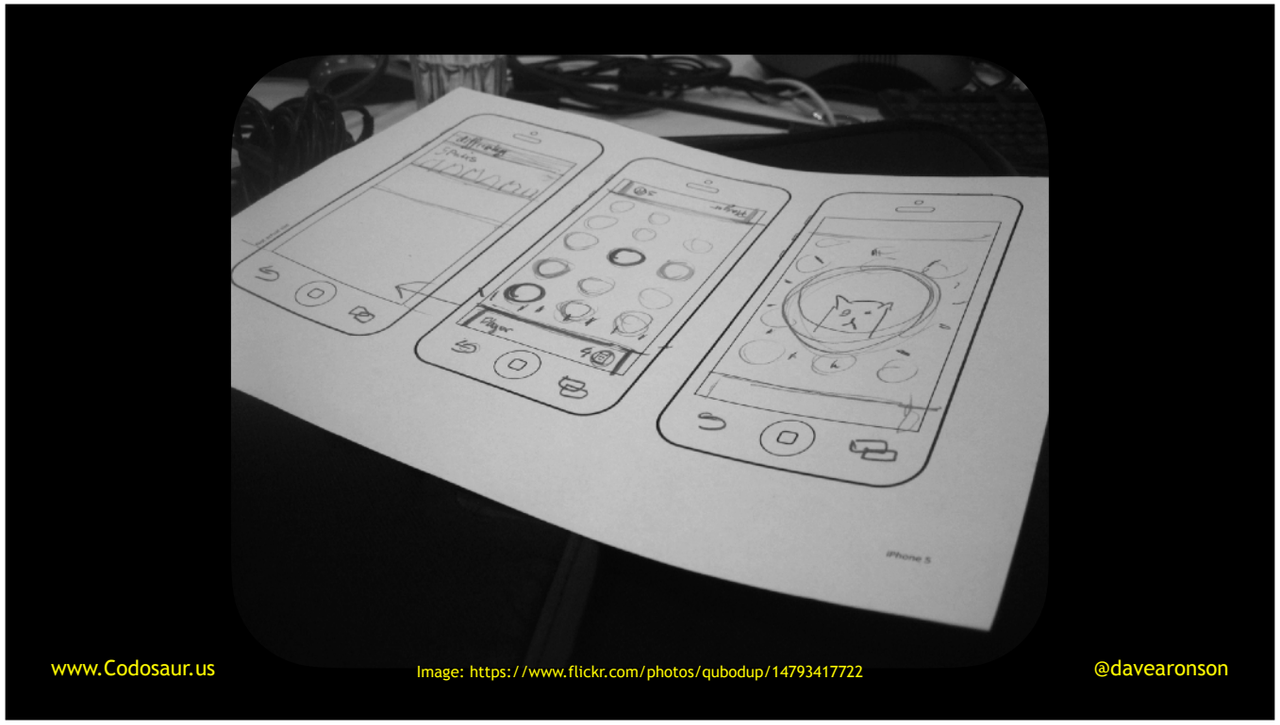


Image: https://roundupreads.jsc.nasa.gov/pages.ashx/699/10%20things%20to%20know%20and%20share%20about%20the%20Eclipse%20Across%20America

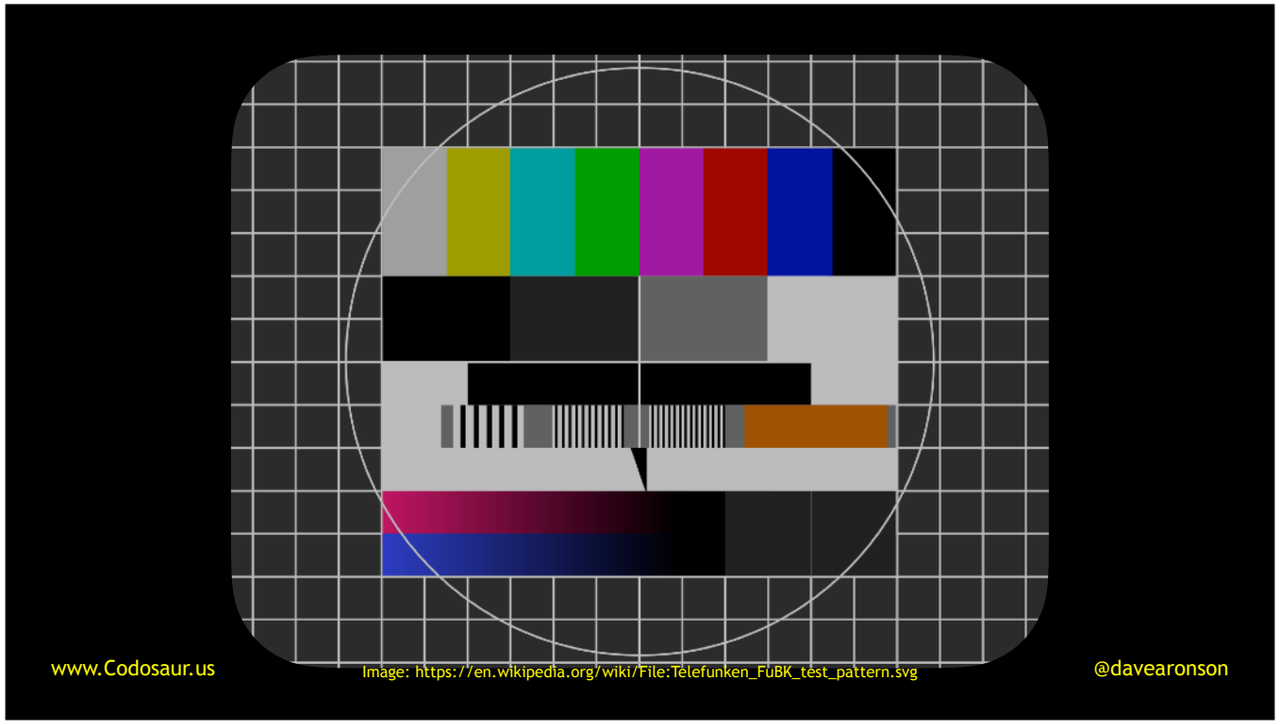
. . . the experts, which in this case would be Requirements Analysts. But on this planet, we usually don't get those either, at least outside of huge companies. So we usually have to settle for occasional remote or indirect contact with at least a representative of some stakeholders, like a Product Owner 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!) I think we all have. Ideally, show them these *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 from users, or a timer, or a sensor, or another system;
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.

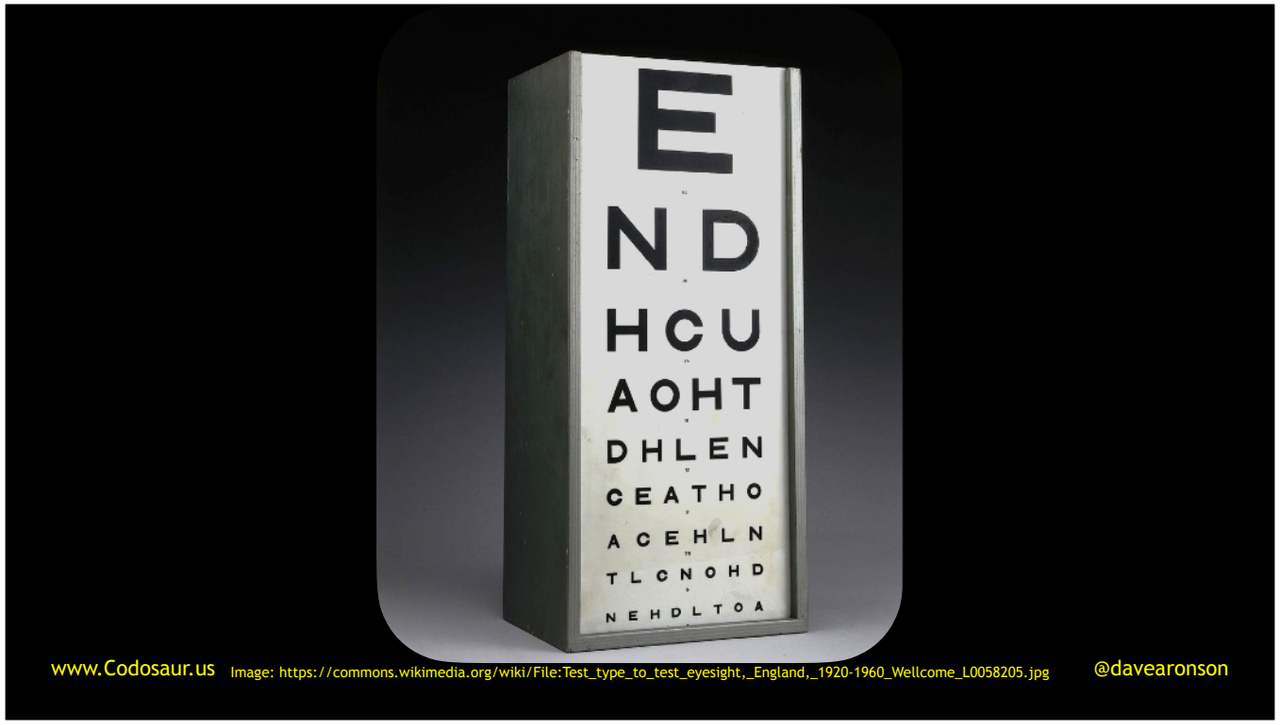
Our next aspect is . . .

. . . correctness. If our software doesn't have this, then, it has bugs, and nobody likes that!

Nothing can actually *stop* us from *writing* code that isn't correct, at least with the tools we have today. So, the big question is: . . .

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

As you probably know . . .



. . . *tests* prove whether or not our code is correct . . . *assuming* of course 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 . . .

www.Codosaur.us

Image: http://www.publicdomainpictures.net/view-image.php?image=25030

@davearonson

. . . 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. We should still add our own . . .

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

Even then, though, normal tests like these can only prove the correctness of cases *we thought* to test. There are some advanced techniques, though, that can help find unusual cases we didn't think of. For instance . . .

. . . property-based testing tests whether some desired property holds true for all valid inputs, usually some relationship to the 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. If it finds an input that makes our property fail, that means that there is an edge case that we didn't consider.

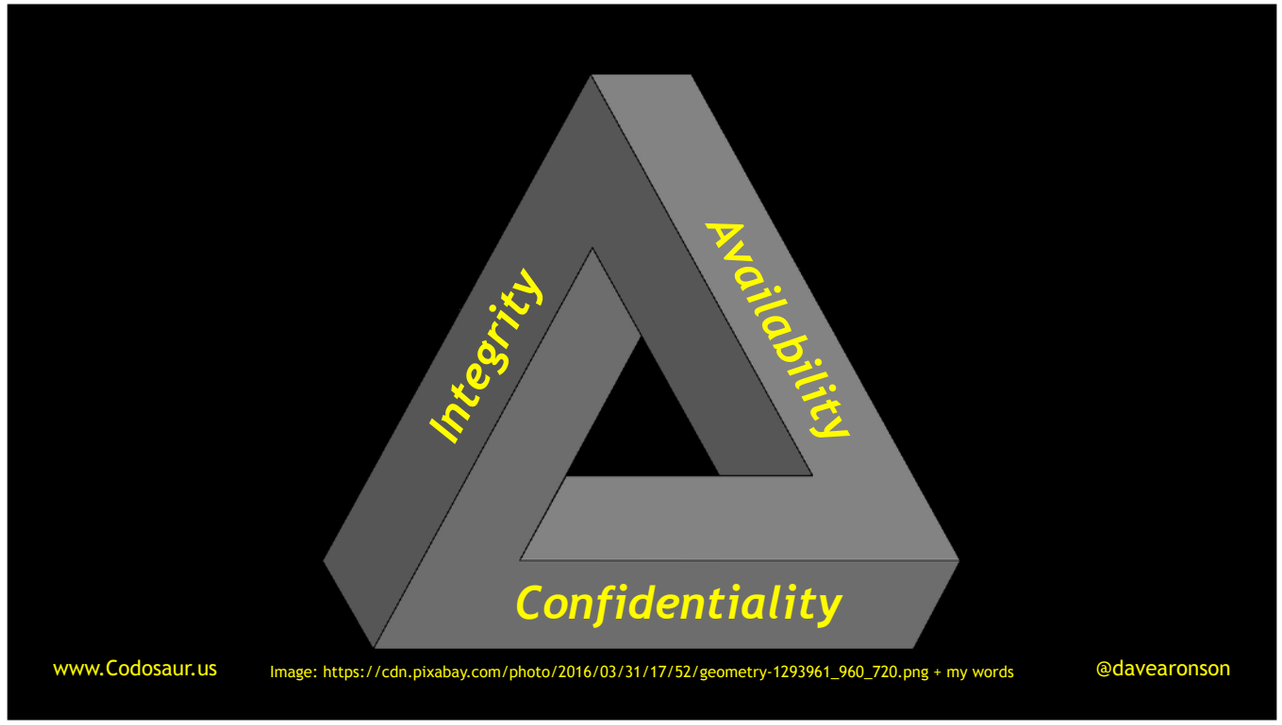
Mutation testing runs our tests against slightly altered versions of our code. Each altered version should make at least one test fail. If not, that means that our code isn't *meaningful* enough for the mutation to make a *difference* in its *behavior*, or our tests aren't *strict* enough to *catch* the difference the mutation made, or both. BTW, I also speak on mutation testing at conferences, and you can find some of my videos on Youtube.

Anyway, we should have enough test coverage, of assorted kinds and levels, 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 simply 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, 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, of Confidentiality, Integrity, and Availability. So, robust software does *NOT:*

www.Codosaur.us

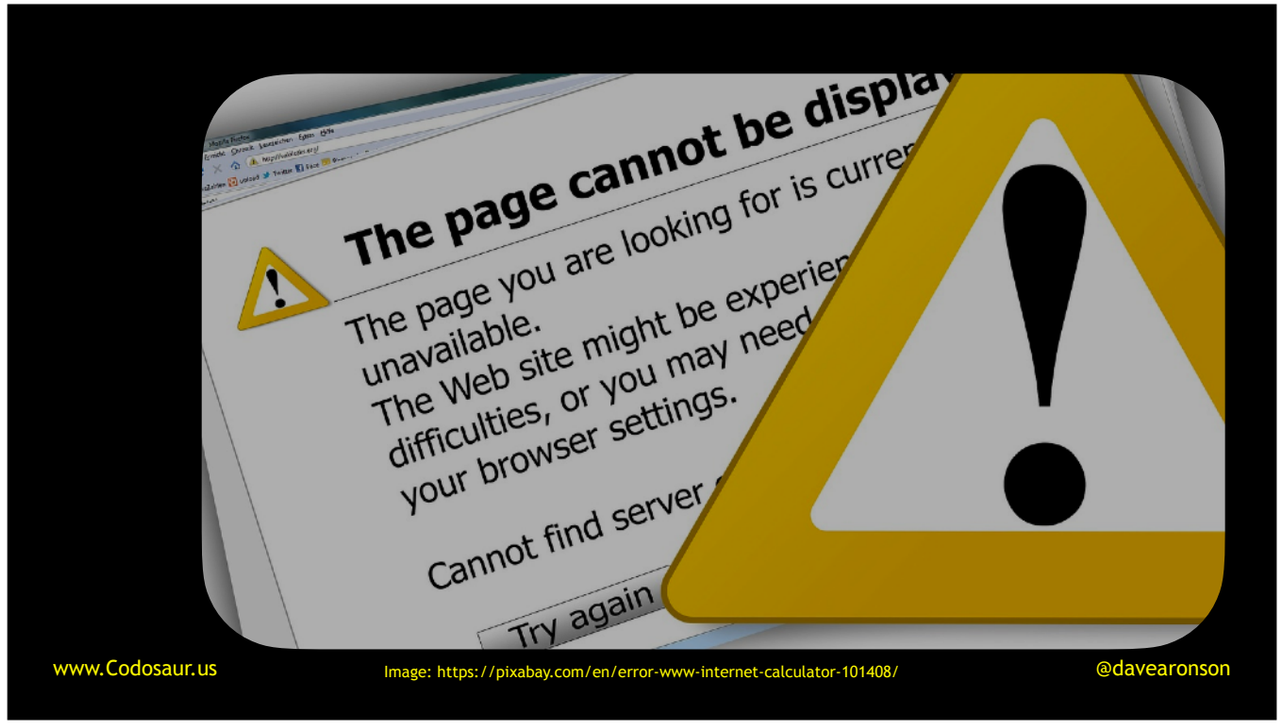
Image: https://commons.wikimedia.org/wiki/File:Somebody_blabbed.jpg

@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, . . .



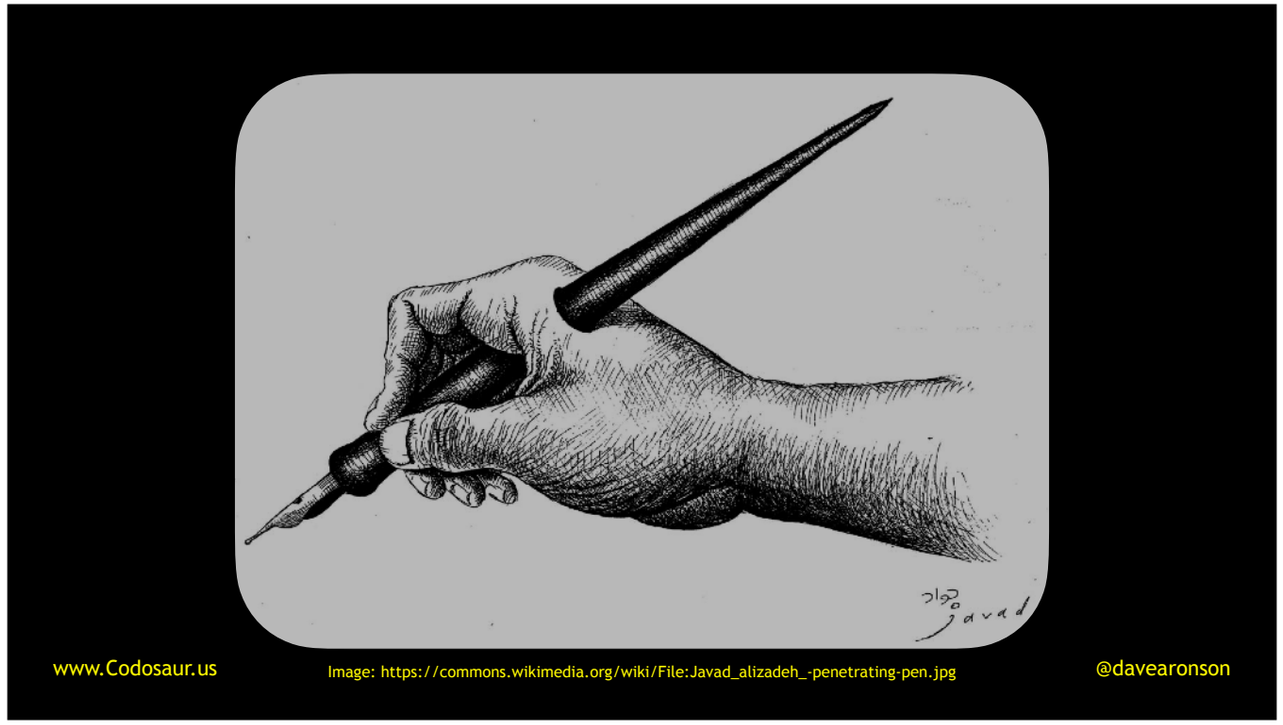
. . . or become unavailable when it’s not supposed to.

A robust system should uphold these principles even when under attack, by someone trying to . . .

. . . *force* it to violate them.

So how do we achieve all *that?*

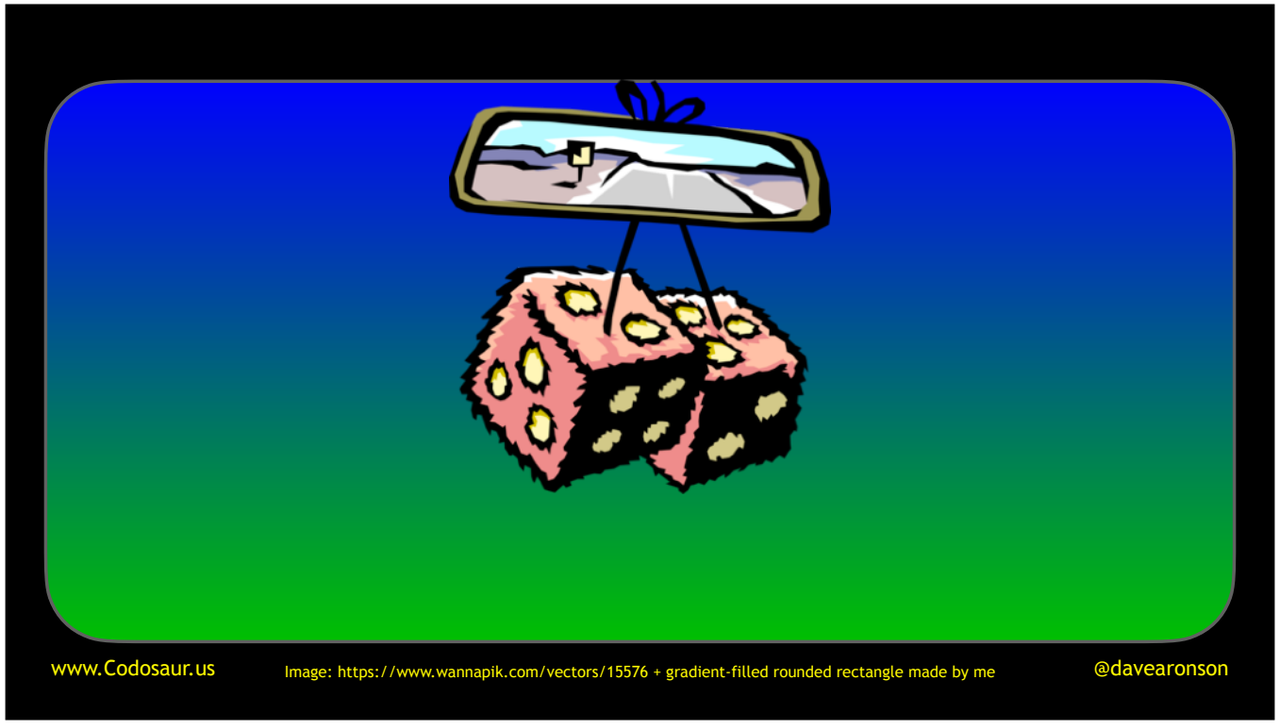
Once again, we could bring in the experts, and in this case, that would be . . .



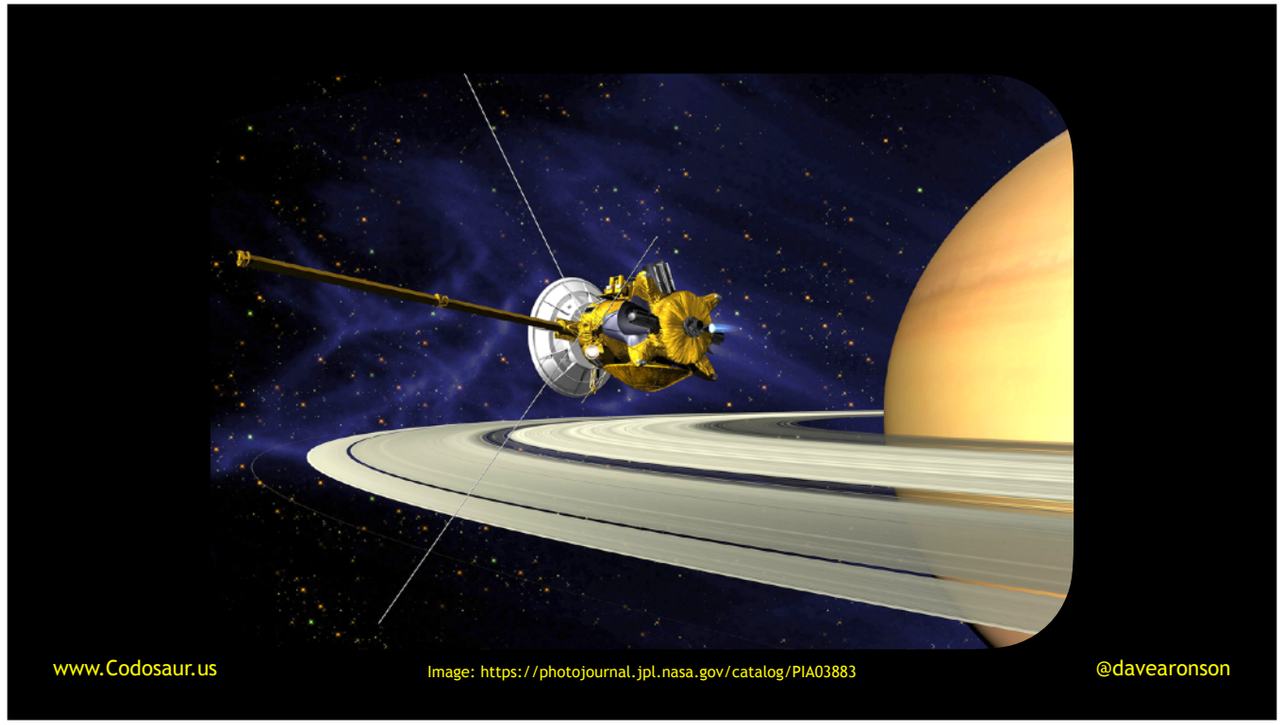
. . . penetration testers, or for short, pen testers. (You can see why I couldn't resist using that image!) The good news is, you don’t have to work for a huge company to use them. Many work for independent computer security companies, that you can hire on contract. However, they are usually expensive,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

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 . . .



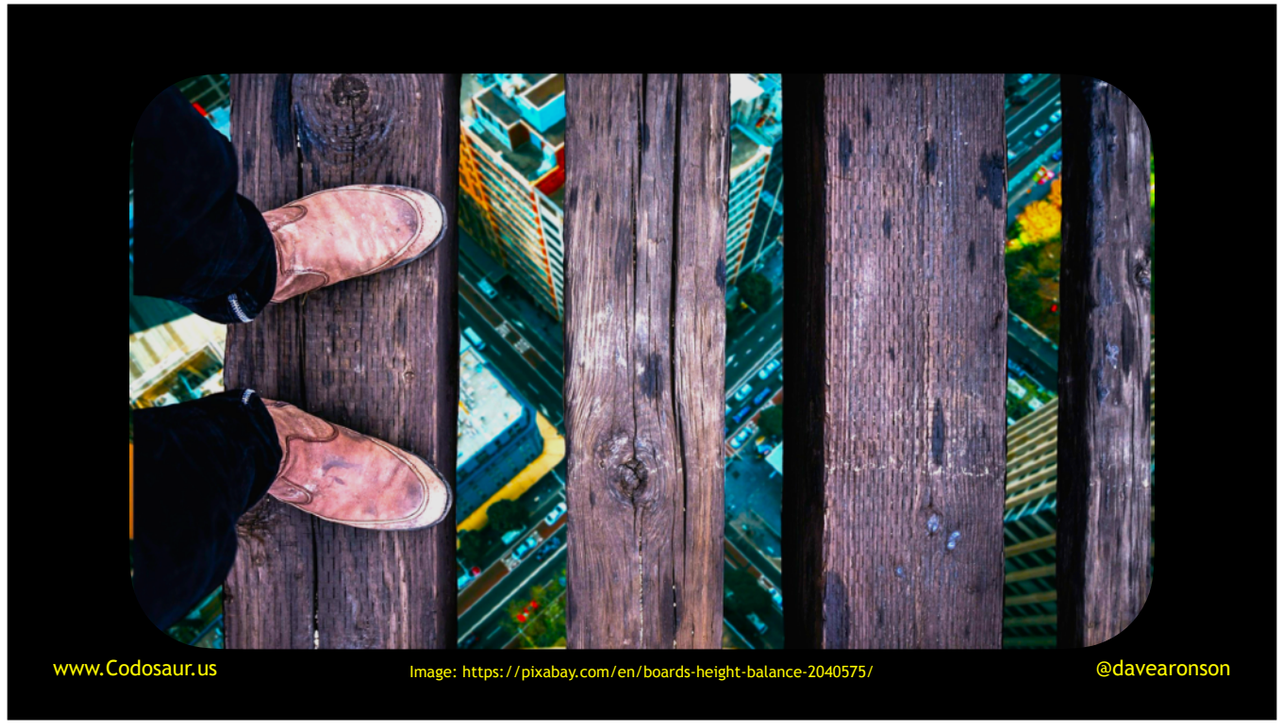
. . . fuzzers, which test our program’s reactions to various kinds of invalid inputs, in the "fuzzing" technique I mentioned earlier . . .



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

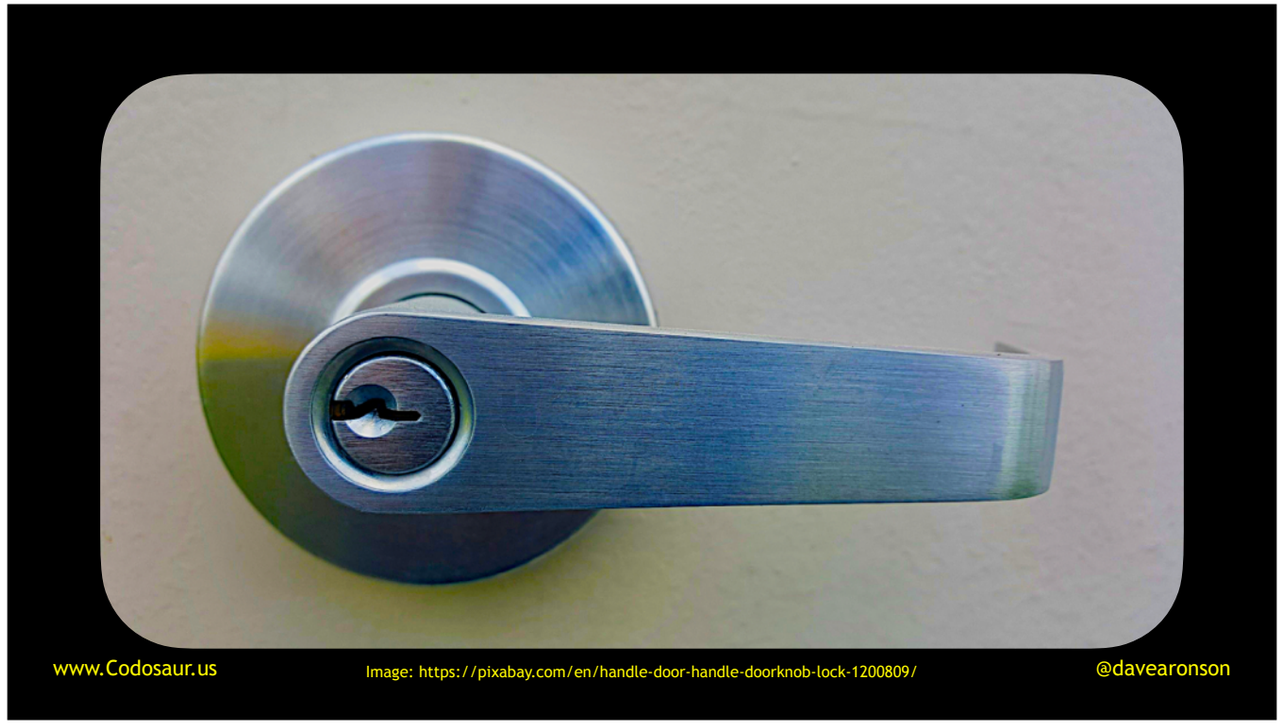
Many of these are available as open source.

But even without their software, we can still get a long way by using their *mindset*. The main 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 brainstormed and run out of answers to such questions, then for everything we've come up with, we must somehow . . .



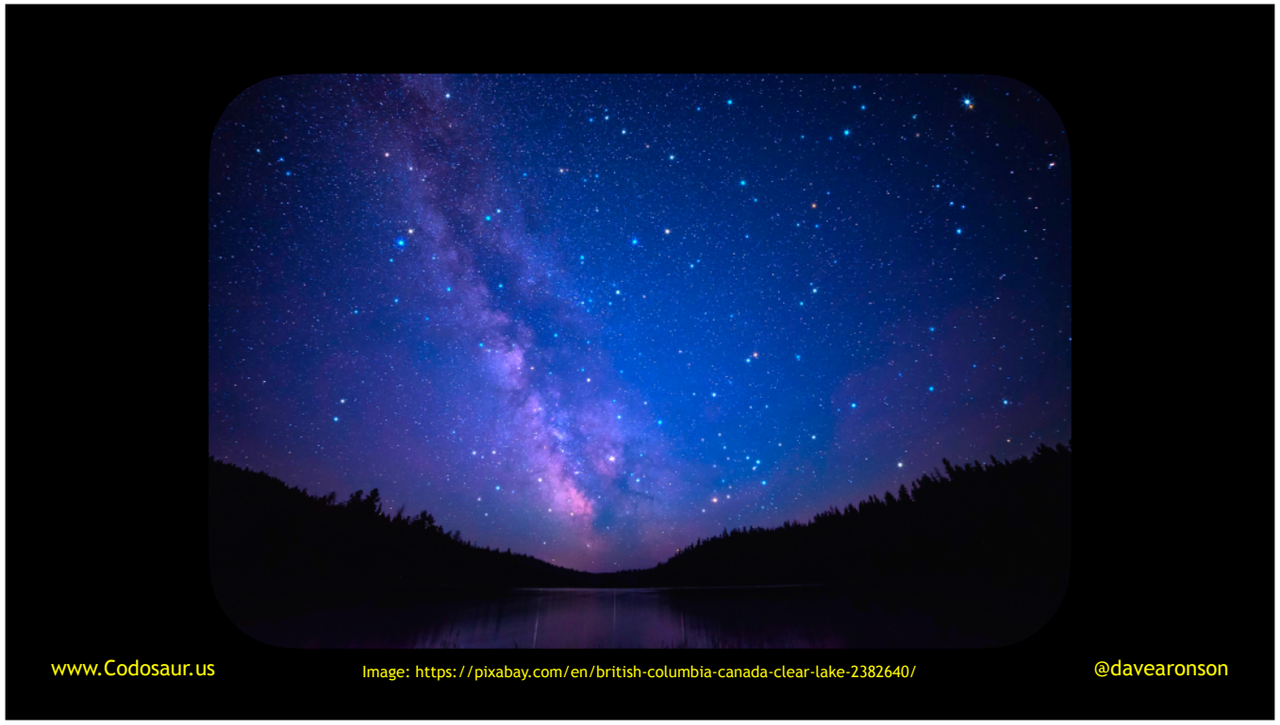
. . . handle it. Yes, that's extremely vague, but how to handle something is going to vary immensely, depending on exactly what it is. Our software should handle *all* reasonably foreseeable problems, from simple user error, to system problems like a full disk, and even external problems like losing a network connection, as gracefully as possible, while giving as little information as possible to potential *attackers*. 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: . . .

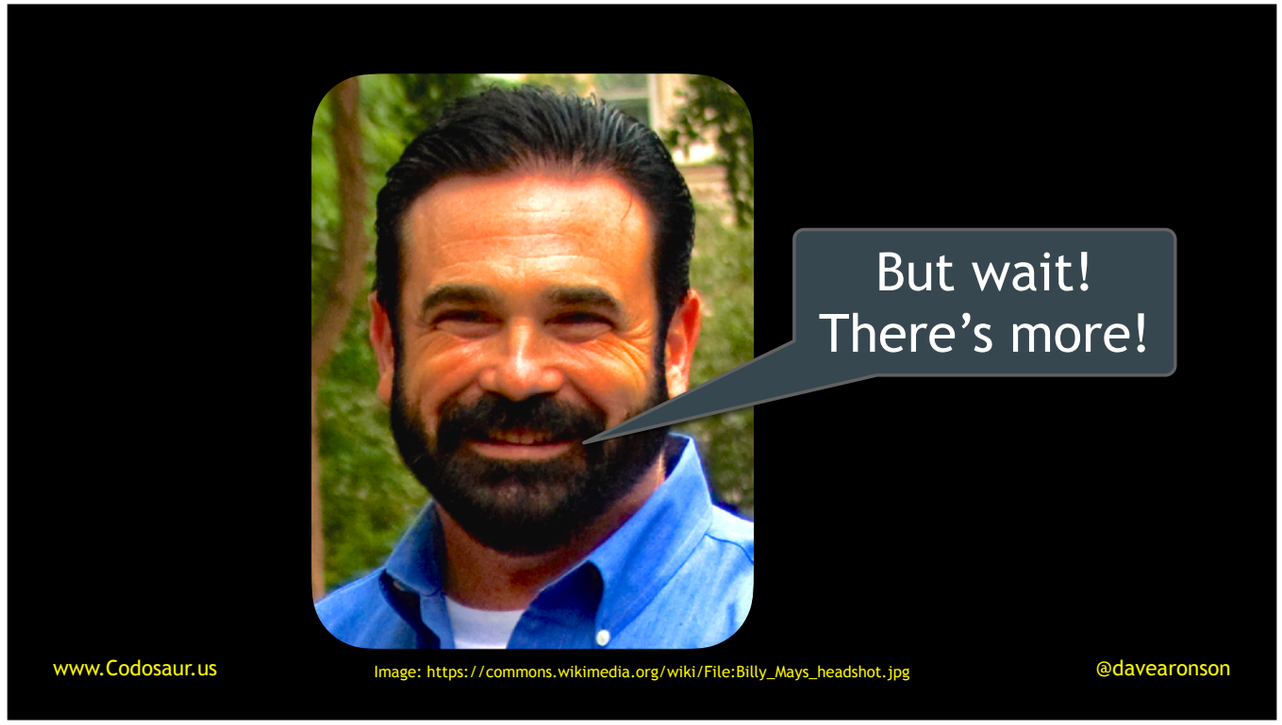
. . . usability. If our software doesn't have this, our users will become frustrated, and may stop using or recommending our software. That could be *disastrous* for a software vendor, or a software-as-a-service company! Also, hard-to-use software can lead the user to do the wrong thing. At least eleven of the 82 people collecting endorsements to run for President of Iceland in 2024, actually just meant to endorse some other candidate. Some of them only found out about the error when someone else endorsed *them*.

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 . . .

. . . *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, or color vision (my own main challenge), hearing, or fine motor control. But there are other whole *types* of challenges we should be aware of, like lack of literacy, at least in our character set. The users may lack certain knowledge, such as pop culture references. The users may even be of low *intelligence*. Yes, we may 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. . .

www.Codosaur.us

Image: from https://commons.wikimedia.org/wiki/File:FarmBot_Genesis_Web_App_on_Different_Devices.png

@davearonson

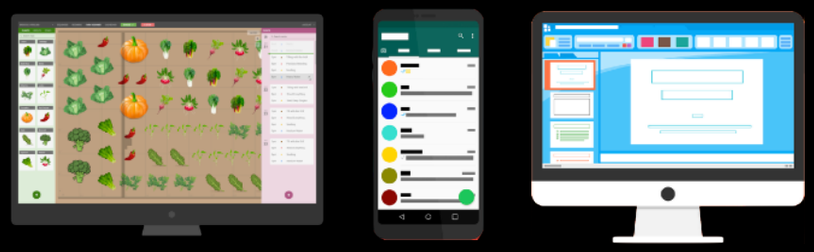
. . . a web app . . .

www.Codosaur.us

Images: as on prior slide, plus https://pixabay.com/en/whatsapp-smartphone-mobile-phone-3126713

@davearonson

. . . a mobile app . . .



www.Codosaur.us

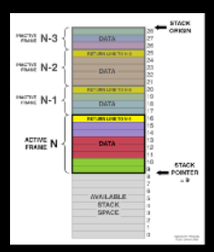
Images: as on prior slide, plus https://pixabay.com/en/presentation-slide-animation-title-1794128

@davearonson

. . . a desktop GUI app, or . . .

www.Codosaur.us Images: as on prior slide, plus https://en.wikipedia.org/wiki/File:Version_7_Unix_SIMH_PDP11_Emulation_DMR.png @davearonson

. . . a command-line app . . . or . . . an API, be it through . . .

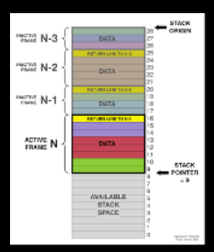


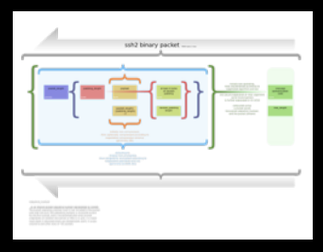
www.Codosaur.us

Images: as on prior slide, plus https://commons.wikimedia.org/wiki/File:ProgramCallStack2_en.png

@davearonson

. . . function calls like with a library or framework, or a wire protocol, whether . . .





www.Codosaur.us

Images: as on prior slide, plus https://en.wikipedia.org/wiki/File:Ssh_binary_packet.png

@davearonson

. . . binary or . . .

. . . textual, or whatever.

So how do we achieve all this?

Once again, ideally we can bring in . . .

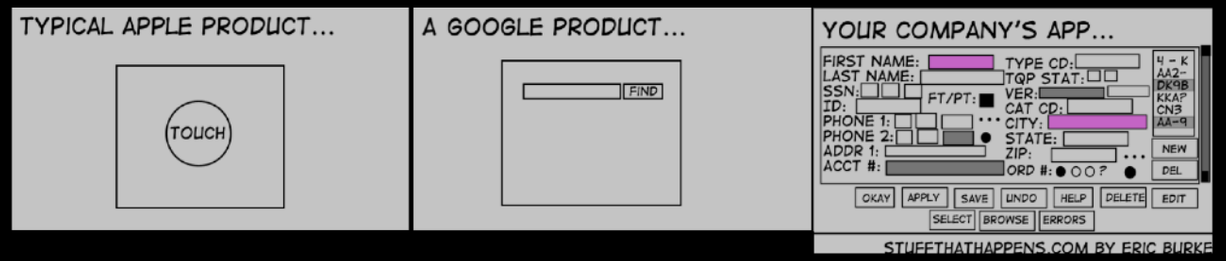
www.Codosaur.us

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

@davearonson

. . . the experts. The bad news is, the people called Usability Experts are mostly really about accessibility. The good news is, we have a wide range of other professions we can get help from! We mainly want a User Experience expert, or at least a User Interface expert. But even a web *designer*, or even an old-fashioned *print* graphic designer, has training in principles of practical visual design that can help us, at least in that aspect of usability. For the non-visual parts, great progress has been made in API design. This is helped by TDD, or test-driven development, letting you specify the API while you're using it, to write tests, rather than finding out that it's hard to use, later when it's hard to change.

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



www.Codosaur.us

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

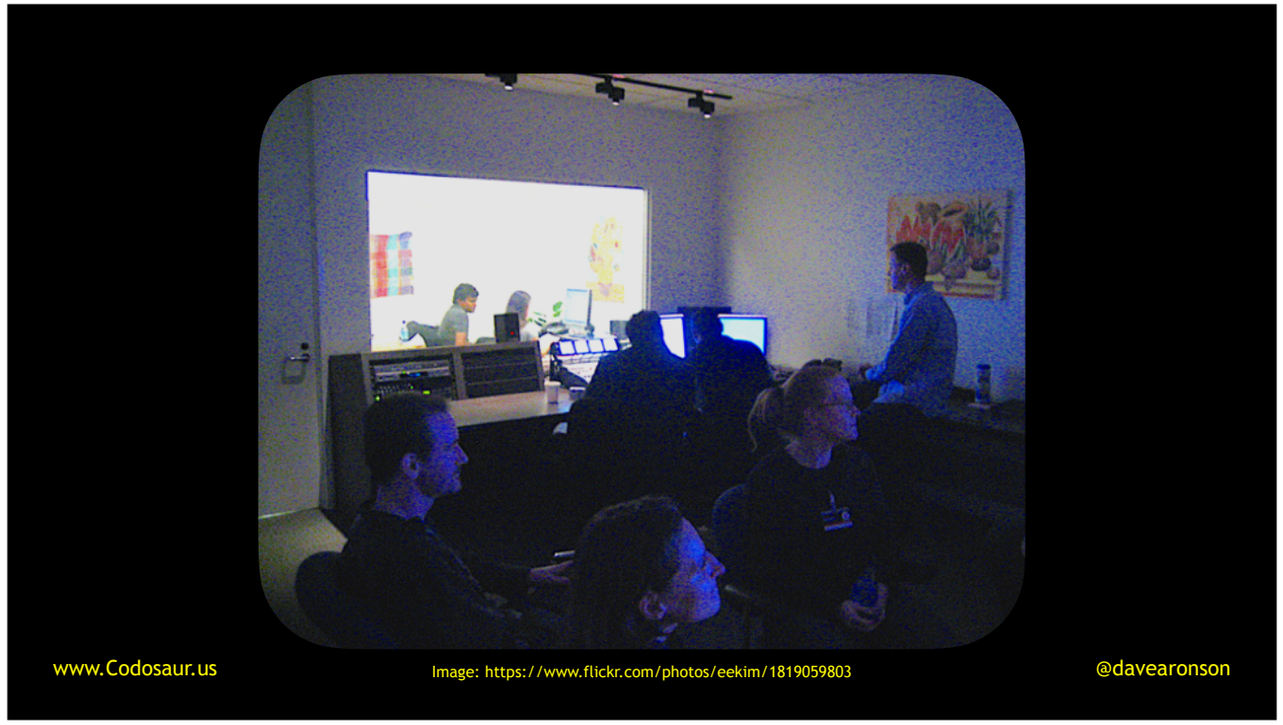
@davearonson

. . . 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, compared to the cluttered unusable mess from “your company”. I think many of us will recognize some of our own work in that.

Another thing we can do, if the software is something we can use ourselves, is to do so, or . . .

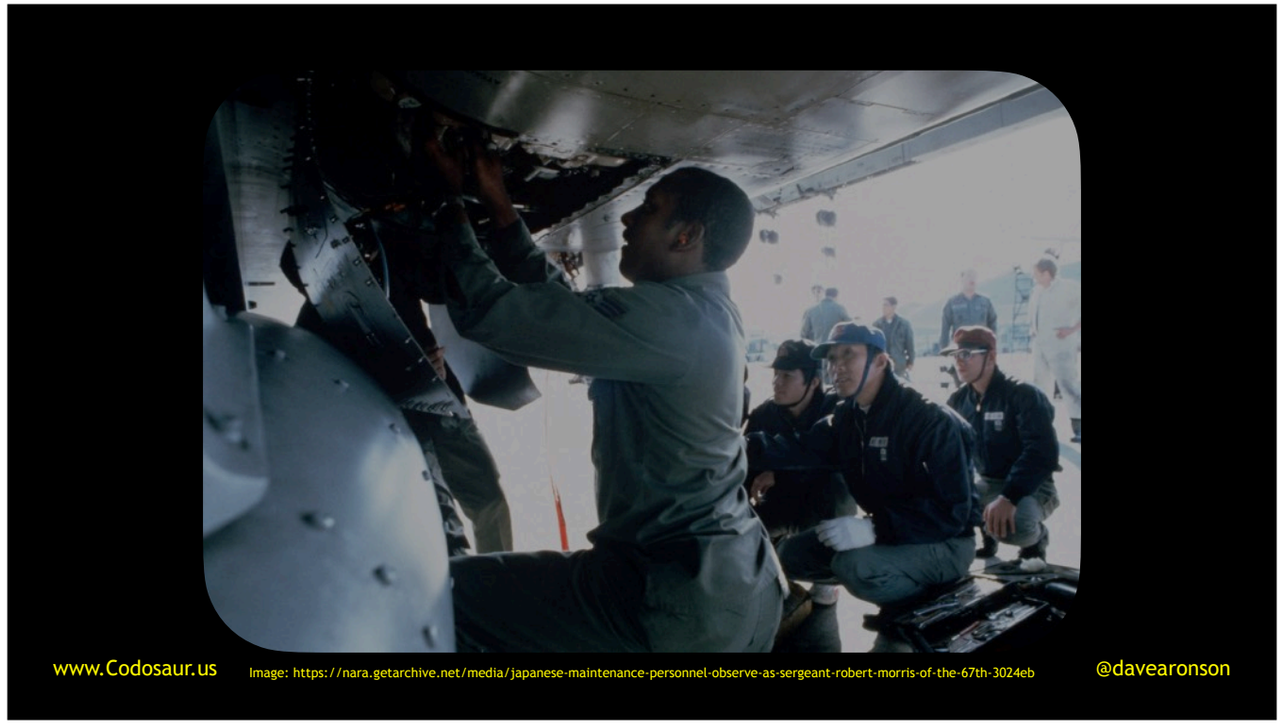
. . .“eat our own dogfood”. But remember, if we find our own software easy to use, that does not mean that our users will! We have inside knowledge, that makes it much easier. But if there’s anything *we* find difficult or unclear, it will be much worse for our users. So, dogfood it mainly to find the pain points.

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



. . . tested! We can bring in some of our typical users, even 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? Then fix their pain points, do more of the good parts, and lather, rinse, repeat.

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!) But I'm going to add 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 . . .

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

. . . new to our project.

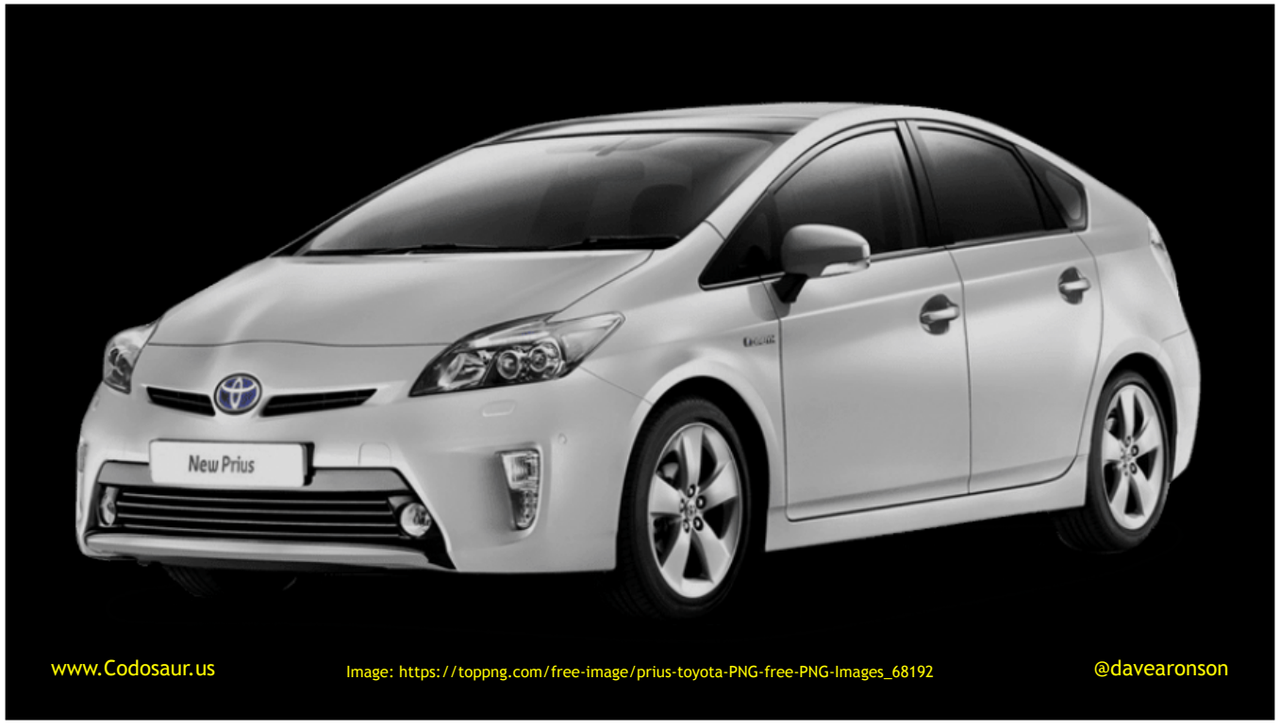
Now how do we achieve all this? For better or worse, the vast majority of software engineering advice is aimed squarely at this. So, rather than expound on lots of generic principles like good naming, or Single Responsibility, or low coupling and high cohesion, I'm going to stick to my theme and tell you how . . .

. . . testing can help with maintainability. Some of you may already be in the habit of using tests as a sort of *documentation* of how the code should be *used*. This is *especially* useful in languages that support doctests, which are tests built right into the documentation. But also, the tests we wrote to verify any *prior* changes we made, 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 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 users complain. And *that* speedup is why I mentioned fear at all.

There are also lots of . . .

. . . tools we can use, like linters, complexity analyzers, and just cranking up the warnings on our compilers or interpreters. These will give us lots of hints how to improve our code, mainly in its maintainability, and occasionally uncovering subtle bugs.

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, bandwidth, and screen space, but there are other kinds, such as the user's *patience* and *brainpower*, and the company's *money!*

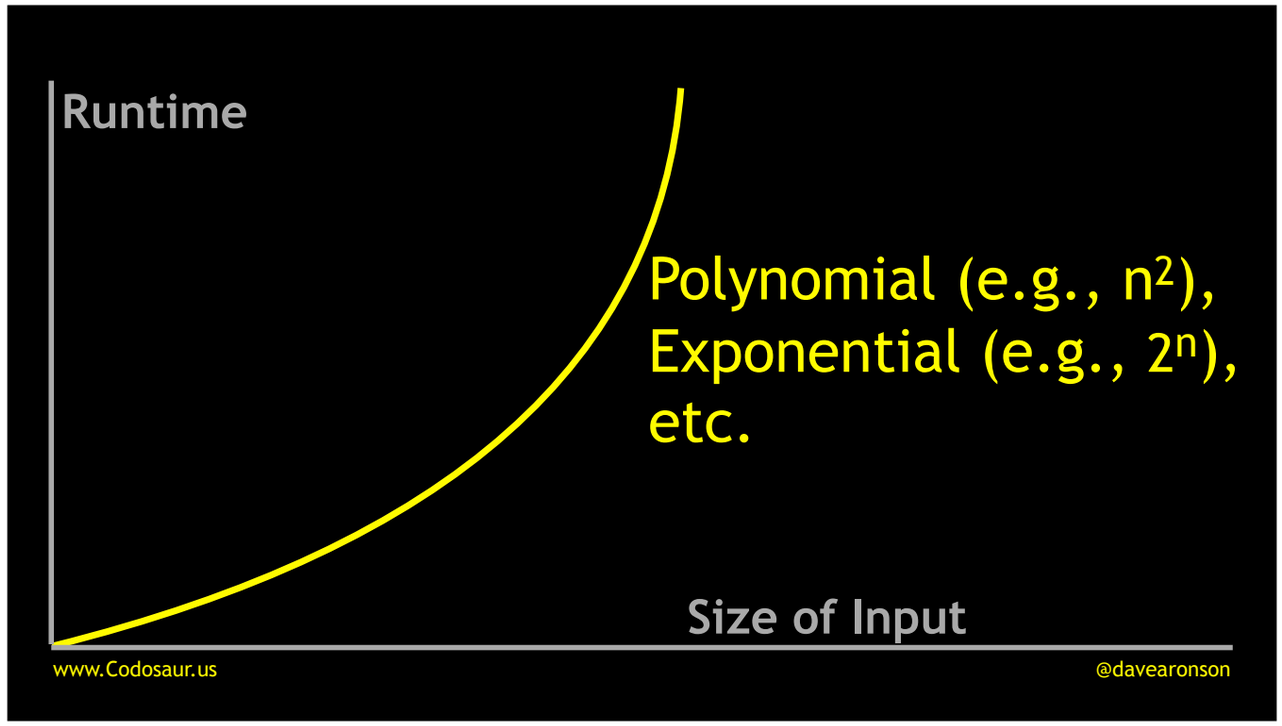
So how do we achieve efficiency? There are many kinds of resources, and many ways each can be abused, so there are many many different kinds of inefficiency, but for the sake of this discussion I'm going to focus on fixing the most obvious and common kind: slowness.

I'm sure we've all had 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!

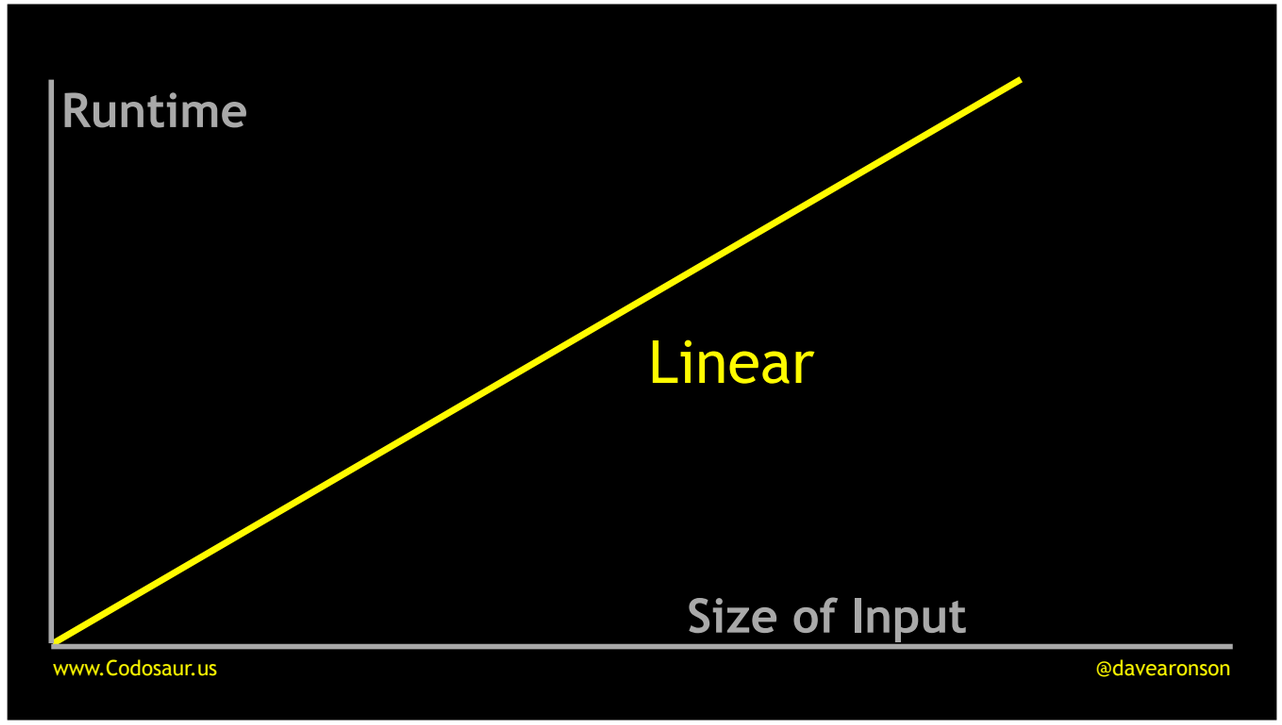
Measure it instead! Humans aren't really very good at spotting the inefficiencies, but there are *profilers* and *packet capture programs* and such, that will tell us *exactly* where, or at least when, we're using too much CPU, RAM, bandwidth, etc.

Once we've found *where* or *when* it's slow, though, there's still the question . . .

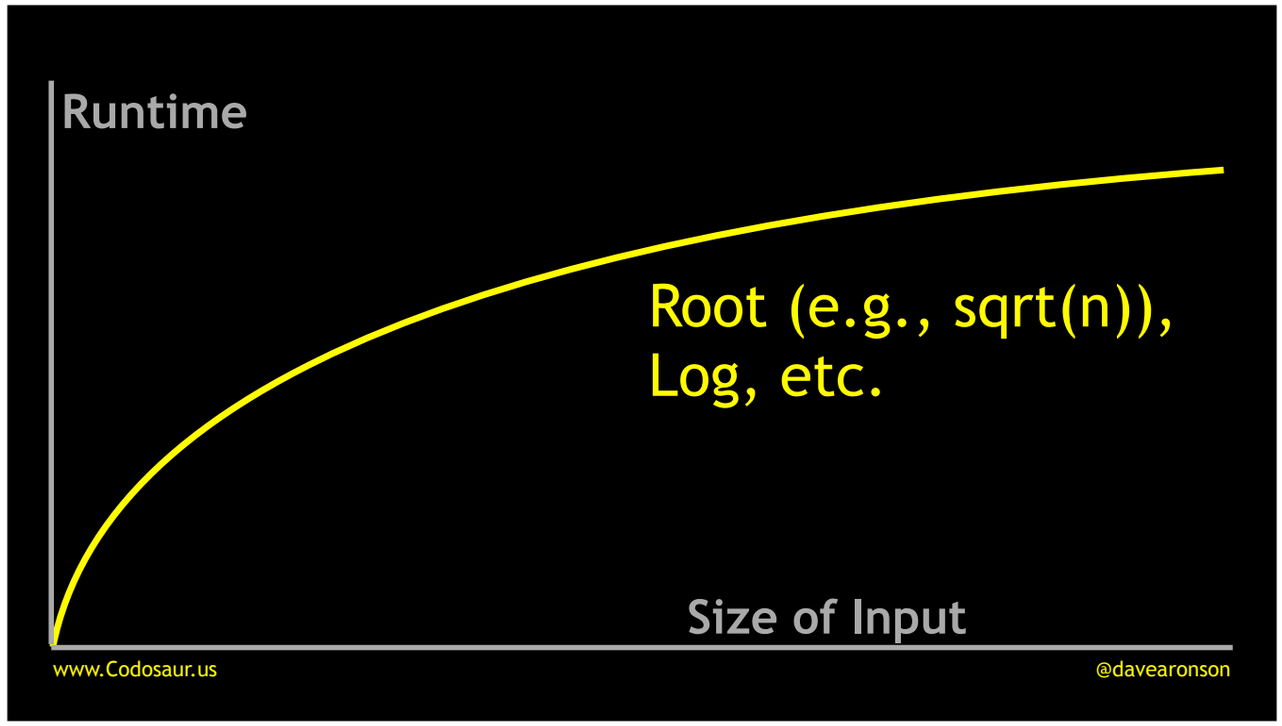
. . . why is it slow? Certain kinds of programs tend to have certain problems. For instance, a distributed system may be doing too much communication, or using a slow network. A database-driven system may have an inefficient query or data model. But in the general case, usually the problem is either 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 . . .



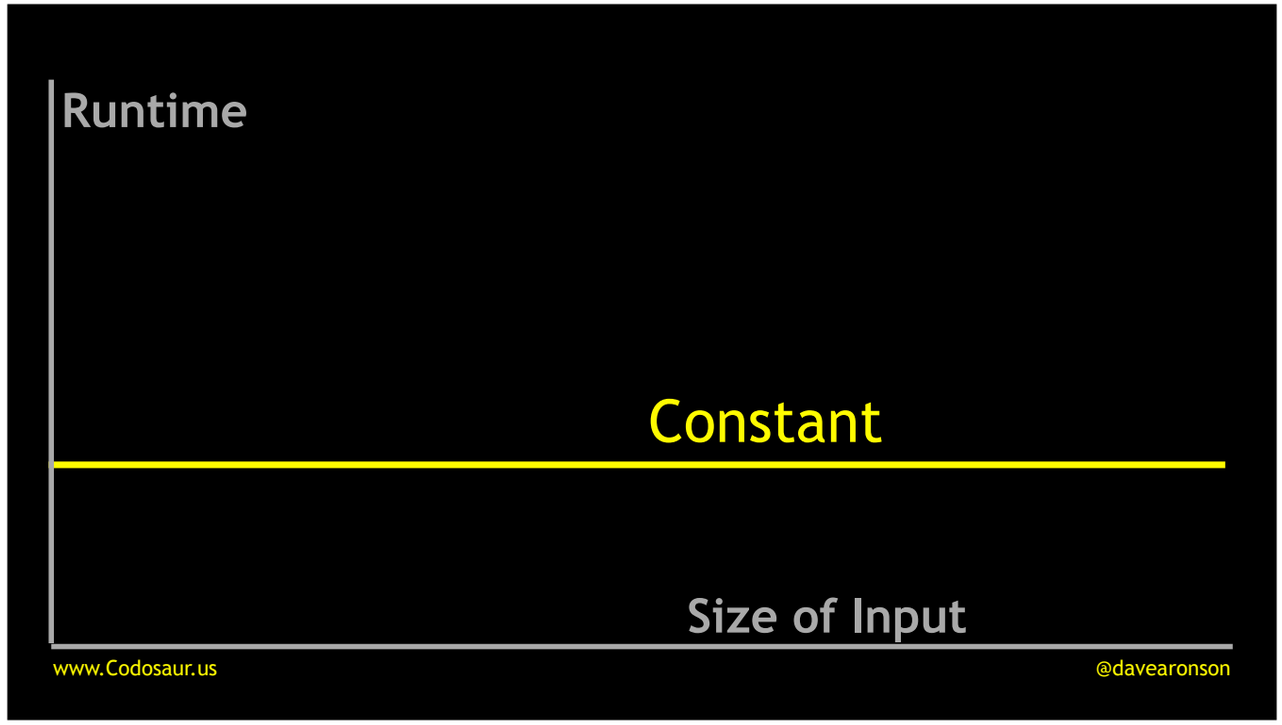
. . . polynomial or exponential runtime, when thinking about the problem a little differently could let us use a better algorithm, such as one with . . .



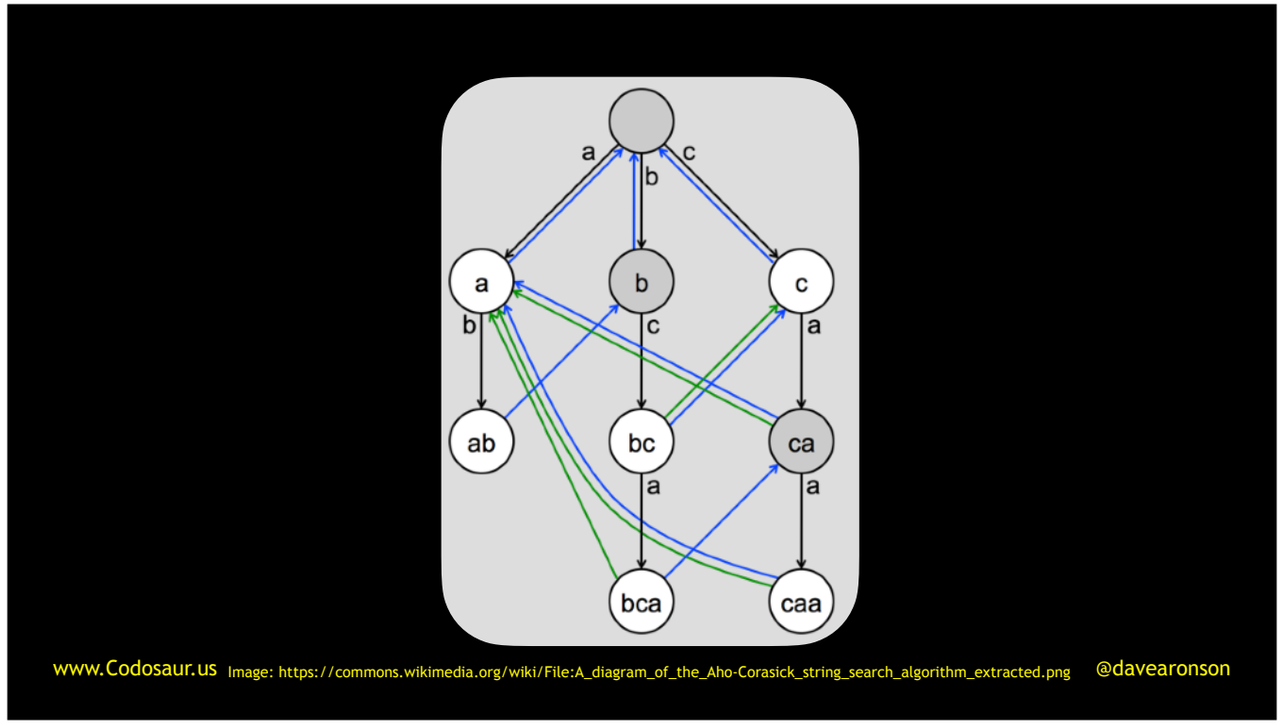
. . . linear runtime, or better yet . . .



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



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

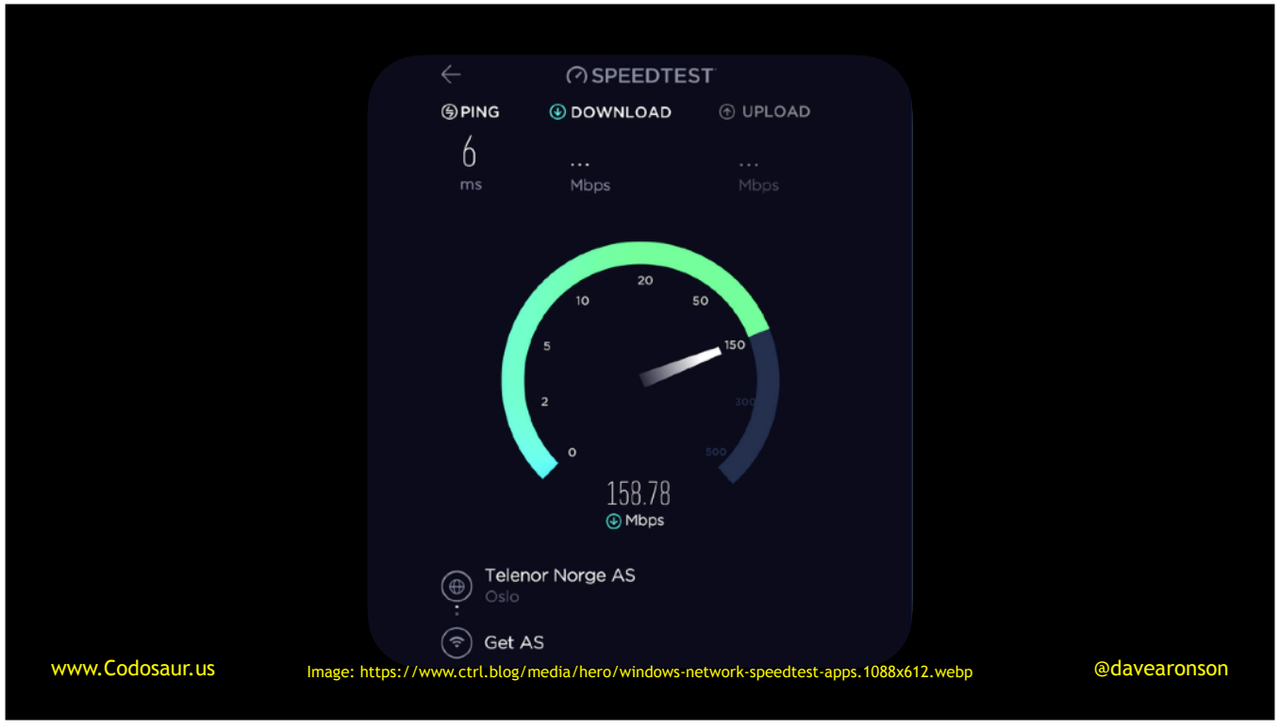


www.Codosaur.us Image: https://commons.wikimedia.org/wiki/File:A_diagram_of_the_Aho-Corasick_string_search_algorithm_extracted.png @davearonson

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

The upshot is that we should be familiar with the basic common data structures and algorithms, and how to 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 . . .



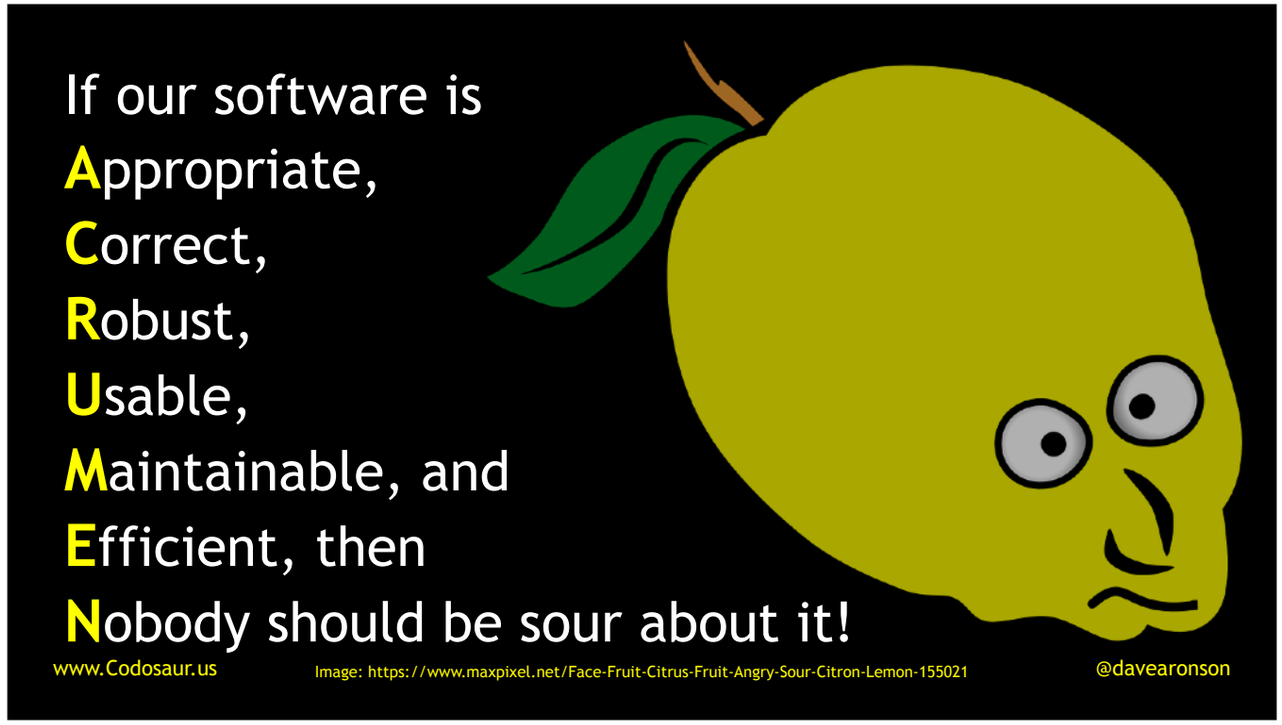
www.Codosaur.us

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

@davearonson

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

In conclusion . . .



. . . if we 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.

And now, . . .

T.Rex-2024@Codosaur.us
twitter.com/DaveAronson
linkedin.com/in/DaveAronson

Codosaur.us/acrumen
Codosaur.us/reds/acrumen-toptal-24-slides

. . . it’s your turn! If you have any questions, I’ll take them now, if you think of something later, there’s my assorted contact info, plus the URLs for my page with more info on ACRUMEN, and for the slides, complete with a full script, which I’ve *mostly* stuck to. Any questions?