Software Architecture Is Overrated, Clear and Simple Design Is Underrated (pragmaticengineer.com)

608 points by signa11 46 days ago | hide | past | web | favorite | 211 comments

jaequery 46 days ago [-]

First 1-3 years of coding, I just coded to get sht done. I got a lot of sht done.

Next 4-8 years, I started getting cute with it and applied all kinds of design patterns, Factory, Abstractions, DI, Facade, Singleton you name it. It looked cute and felt good when it all worked but it was a juggling act. There was usually like 2-3 files to touch just to do one thing. UserFactory, UserService, UserModel, User, you get the idea. It got to a point coding now felt like a burden and I started getting allergic reaction to any projects that had more than 50 files.

Next 4-5 years, I made it a mission to only code in a pragmatic, minimalistic way. Best decision I ever made, this have been the most productive time of my career. I don't look back and never going back again. A simple require and requireAll with basic OOP is all I need on most cases. Most of my project now have less than 10 "core" files minus the standard views/routes/etc. I enjoy just working on code now it makes me happy and also any devs who joins loves it too as they get it right away. I code almost exclusively in Sinatra now btw. Express is great too but I think the ecosystem isn't there yet for developer happiness.

Keeping code simple is not easy. It takes a lot of trials and errors to know what works and what doesn't. I realize I code a lot slower now than in the past and that I write much fewer lines of code. It's both good and bad because sometimes I'd even spend hours just trying to properly name a variable. But I believe this pays off at the end.

You just can't best simplicity.

JMTQp8lwXL 46 days ago [-]

Over my career, I've worked with engineers that like to over-engineer and under-engineer.

The over-engineered code looked like russian dolls: had many layers to it, and some of the abstractions offered no value. That can make onboarding to such code unnecessarily complex.

On the other hand, under-engineered code made very little of use of even simple data structures or algorithms. I like to call it "chicken scratch" code. I find it tends to be brittle, and it fails the longevity test: you end up frequently having to touch everything, and be aware of the entire state of the system at once, due to a lack of functional style. There are few enforced boundaries between subsystems in this type of code.

Like most things, moderation is key. I only introduce abstractions when there is a meaningful value-add to doing so. This is somewhat subjective, but there is a right level of application of design patterns. Not too much, nor too little.

stinos 46 days ago [-]

In my own experience, it's really a must to go through both over- and under-engineering phase yourself: only then I found I got a really clear view on what's good and bad in approach, allowing me to default to somewhere in between (with outliers when useful) and in any case with focus on simplicity (but again, if needed for e.g. performance that can be violated). All of this is the basic 'learn from your mistakes' principle in the end - and it works.

fnord123 46 days ago [-]

I don't even think it's a phase.

From what I see in my own experience, overengineering comes from not having a deadline to deliver an actual working piece of software. Unclear requirements. Fear that the code must be perfect against change without knowing what the changes might be.

Underengineering comes from confidence that the issues are too small to worry about. Or the time constraints mean I can't try to find good primitives to make something robust. And if I have to return to it because it's so important, I can re-engineer it then.

tluyben2 46 days ago [-]

I have been writing software professionally for almost 30 years now and yes, this is exactly what I see too. I find the underengineering far more productive though; because of time constraints I hack something together which works but is brittle (decades experience do help fight the brittleness even in a hurry though) and I do revisit to fix and refactor if the software survives.

fnord123 45 days ago [-]

Underengineering is far more productive but not always possible when you're defining the overarching architecture of a multi team project.

Then you have to rely on experience to do things like definelibg interfaces that can be extended later on instead of overspecifying types. It's hard to balance it just right.

stinos 46 days ago [-]

overengineering comes from not having a deadline

This really resonates: at the height of the phase I spent about 2 years once, half-time developing something new for which there was essentally no deadline. I went totally berserk doing insane stuff like implementing design patterns using templated 'reusable' classes in C++ so whenever something even remotely looked like a pattern I had a class for, I would use that class. Which then needed small fixes of course. And extra logging. Pluggabble. Etc, you get the point :) In the end everything was dropped because I found a much better, much simpler, existing way to do everything though.

But didn't happen to me again afterwards, or at least not to that extent: it was really like having had a severe electric shock; once, but never again, before even coming close to any hot wire my brain is already subconsciously screaming to get away from the thing a.s.a.p.

delinka 46 days ago [-]

Overengineering: I just left an enterprise-y company where the lead software engineer "over-engineered" for the purpose of job security. His philosophy was that if it's layered and convoluted, it's harder to replace him. Oh, we had deadlines. It had to work. Part of "working" is "passing tests" so he also wrote lots of tests that executed the written code but actually tested the mocking framework.

I hated dealing with any of his code.

psychoslave 46 days ago [-]

My experience is that you rarely *have* the time to make things right, you *take* the time. And you do so, because you estimate that despite doing so will put you off the deadline, you expect it to be a good bet on the long run.

Not doing so, time required for maintenance will increase, and competing with new feature development time. Of course these new features have deadline already to short even before you add this burden. Moreover, maintaining everything working is the top priority "right now", but making the new development reaching production is prevent the solution to fall into market irrelevance.

schwurb 46 days ago [-]

I never went through the over-engineer phase personally. I am a very very lazy person, which leads to me to correct balance most of times: Not enough abstraction, I suffer because of too much change to the code base, too much abstraction, and I feel pissed about my risk management. Since I suffer greatly when I suffer, I avoid suffering in my own code as much as possible. What also helps is that I plan most of my algorithms and program designs on paper, actually touching a computer is the last step in that process.

stinos 46 days ago [-]

too much abstraction, and I feel pissed about my risk management

Interesting, so it's not a phase for you, but for the rest it sounds exactly like my point, i.e. learning that over-engineering (too much abstraction) is a mistake and as such trying to avoid it.

schwurb 46 days ago [-]

Yes, we are on the same page there. I think due to depression during my formative years, I have a mix of slight apathy towards expectations and a slightly off motivation system, which leads me to be very lazy - really only doing what is necessary to get a mantainable product and not so much caring about what other people want. Working in two software companies during studies was okay, but I really hope to sell my own software one day!

Glad to see you making progress!

tarsinge 46 days ago [-]

I think you can under engineer and still use a pragmatic functional style.

Also yes when it’s under engineered you usually have to touch multiple parts to make a change, but in the end it usually take less time (testing for bugs included since the code is simpler).

Edit: I don’t say under engineering is perfect, but compared to an over engineered code base I prefer the first for maintainability.

JMTQp8lwXL 46 days ago [-]

In theory, I agree with you, but in practice, I rarely see under-engineered code that is functional in style. My conjecture is that people under-engineering aren't even aware of the concepts of imperative vs. functional style.

The more we can think of code as existing in a sandbox that knows nothing about the surrounding system it's in, the better. Because when you need to change that code, you don't need to juggle comprehension of the rest of the system as you edit that code. It makes reasoning about the code much simpler. This is why I agree -- code can be pragmatic and functional.

But if you tend to practice having code existing in little sandboxes, you are essentially practicing design. This might be semantics, but now you have sub-systems with well-defined borders that interact with other sub-systems in proscribed, expected ways. I would consider that, to some degree, the application of design patterns and architectural concepts -- even if it's only the inevitable consequence of writing pragmatic, functional code.

bayindirh 46 days ago [-]

> I think you can under engineer and still use a pragmatic functional style.

I actually believe this, and tried in some occasions for my personal projects.

When you develop the code further, you need to touch everywhere. It starts very neat and being extremely tidy at first, but when you need to add some significant features, the design breaks down pretty badly.

Then, to save the architecture, some significant refactoring runs are required, and boy, these refactorizations are expensive both time and design-wise.

If your project will not add new features over time and relatively compact, under-engineering can deliver pretty nice results. Otherwise, designing with some headroom and flexibility goes a long way.

jamesb93 45 days ago [-]

You should clarify that your experience us pretty much exclusive to web development which is bot indicative of a lot of work that goes on in software creation and computer science research. Just a weird nit pick of mine that most people here seem to be web devs who assume everyone else is - it creates a weird environment for discussion.

xcubic 45 days ago [-]

From someone doing webdev but without studies, is there a small list of simple data structures or algorithms that you would recomend?

EGreg 46 days ago [-]

In general I have found — over 20 years of experience architecting software - the following:

1. Platforms and reusable frameworks should be architected as well as possible. Apps can be whatever.

2. A developer who writes clean code and documents it is far better than a “10x” developer, unless you have budget for only one developer.

3. Functions should have extensibility, put the required parameters as parameters and always include an “options” at the end. Each function can have defaults that you can extend, which means you need a deep-extend method:

https://qbix.com/platform/guide/javascript#functions

4. When in doubt whether to do convention A or B, take a bit more time to do C which can handle both, and add the convention in a config. You never know when someone may need something else!

5. Similar to 4, if you can have an extra indirection, add it. So you can let others hook into “before” and “after” hooks at any point. Use events instead of functions.

6. In fact try to have event driven architecture rather than futzing around with mutating data. The easiest way to sync is to maintain a linear total order for events.

7. Think about how lookups will proceed and partition everything by those keys. Sometimes you need to have duplicate tables and keep a sync from a “primary” table to a “secodary” one in the app layer. Doing this allows you to do sharding or even go serverless peer-to-peer later!

8. Security: more checks are better than less. Pile on private keys, bearer tokens (api or cookie), and so on. Use a device keychain:

https://qbix.com/blog/2018/01/17/modern-security-practices-f...

igor47 46 days ago [-]

This list is full of hilariously terrible advice. Was that intentional? On the internet, nobody know if you're a dog or being sarcastic, and I dread stumbling across a code base where someone took some of this stuff seriously.

Examples: > if you can have an extra indirection, add it.

> Use events instead of functions.

> You never know when someone may need something else!

EGreg 46 days ago [-]

Please be specific about the issues and let’s discuss. I am serious - this list is optimized for maintainability of code. Developer time is more valuable than processor cycles, in most cases. Unless you are the kind of person who would argue that C++ introducing object orientation and virtual methods made everything slower and that extra indirection by default is hilariously bad architecture?

igor47 46 days ago [-]

> 3. Functions should have extensibility, put the required parameters as parameters and always include an “options” at the end. Each function can have defaults that you can extend, which means you need a deep-extend method:

counter-point: why not write functions that take the arguments they need? if they need more arguments later, why not add them later?

> 4. When in doubt whether to do convention A or B, take a bit more time to do C which can handle both, and add the convention in a config. You never know when someone may need something else!

counter-point: why try to anticipate future needs? maybe it won't happen. maybe it'll be something you totally don't expect. you should write code that's easy to understand and change, so it can react to future requirements. you should *not* write code that's bloated with unnecessary features, because it slows down your future ability to iterate.

> 5. Similar to 4, if you can have an extra indirection, add it. So you can let others hook into “before” and “after” hooks at any point. Use events instead of functions.

have you ever tried to figure out how a piece of functionality is implemented, only to go chasing it across 10 files because of indirection? have you ever had a piece of code that runs, but you don't know why it's running because you can't tell which things are firing what events? have you ever had to ask the question, "what happens when i execute this function", but been unable to answer because of hooks/observers that are not directly connected to that function?

my counter-point: use the barest minimum of functionality you actually need. stick with functions and maintain linear control flow with minimal side effects.

> 6. In fact try to have event driven architecture rather than futzing around with mutating data. The easiest way to sync is to maintain a linear total order for events.

if you have to mutate data, you should mutate data? not sure how events save you.

> 7. Think about how lookups will proceed and partition everything by those keys. Sometimes you need to have duplicate tables and keep a sync from a “primary” table to a “secodary” one in the app layer. Doing this allows you to do sharding or even go serverless peer-to-peer later!

wow! serverless peer-to-peer! i can't wait to re-write my app, i'm sure my customers will love the new architecture! /s -- on a serious note, while distributed sync is sometimes required (for instance, state kept in the JS front-end app that is a copy of the DB-preserved state on the backend), keeping them in sync can be a nightmare, and you should avoid this at all costs if possible.

> 8. Security: more checks are better than less. Pile on private keys, bearer tokens (api

or cookie), and so on. Use a device keychain

in my experience, 80% of all bugs are security bugs (that number may be as low as 20% if you use the OPs advice to increase your bug count). while security is important, you should avoid "[piling] it on", and instead be thoughtful about it. separate authentication (accomplished with keys/tokens/etc...) from authorization (accomplished through checks of the identity at points of functionality).

EGreg 46 days ago [-]

3. Because it explicitly signifies a place to put those later arguments, in a way that is MAINTAINABLE.

First of all, the last parameter should have a default value of {} — that is, no options, but can still be deferenced in code. So not passing options is ok.

Secondly, if you don't do this, future developers will keep adding parameters in an ad-hoc manner until you get stuff like:

context.copyImage(src, dest, sx, sy, sw, sh, dx, dy, dw, dh, filter, rotation, matrix, ...)

and your calls will keep looking like:

context.copyImage(a, b, 0,0,30,30, 20,20,40,40, null, 5, null, "foo")

Not only will it be harder to read for anyone looking at the calls, but also the future function signatures will have parameters in the chronological order they were added — which is often totally unrelated to the order they should be in, but you can only add them at the end.

Since the function should be backward compatible, all new parameters are by definition OPTIONAL and therefore can be added to a hash or object called "options".

And *YES I stand behind this*. Years ago I actually recommended this to the PHP language mailing list:

https://grokbase.com/t/php/php-internals/1042mr8yrn/named-pa...

In other words, I wanted the function call syntax in PHP look like the array syntax:

func($a, $b, $c => 3, $d => "foo");

Simple, and elegantly enforces the above convention while looking "quintessentially" PHP!

4. When you are building a re-usable framework, it's silly NOT to anticipate future needs. The whole point of a framework is future needs.

Now, you say you should weite code rhat is easy to undeerstand *and change*. If you hardcofe vales, that's easy to change - just put a variable there. But if you hardcofe a concention in 100 places, fhat's not ao easy to change. You can't just grep for a hardcofed constant.

But ot gets worse rhan that. Other code will come to rely on this "invariant", which may change later. Again, the wholw goal of my recommendations is to future-proof your code *so that future developers will write code rhat grows up around it and can go in any direction, and can play nice with each other*.

5. Suppose you are lacking middleware between A and C. So now you want to mock an input from A. Too bad, you can't. Ok, what if you want to modify something that goes on between A and C? You have to rewrite A or C.

Let me give a real example. Suppose I said that people may have one more articles they write. So I implement a User table and an Article table, which has authorUserId field on it. Simple, right?

Except it's too simple - one article can at most one author. If instead I had thought ahead and made three tables: Author, Article, and Authorship as a join table with articleId, authorId, then I could have 1 to N mappings in both directions and far more flexibility.

Now you may say — why think ahead? Maybe in the future articles can have more authors and THEN we will refactor the code! Except at that point you'll have tons of plugins and apps, some beyond your control to change, relying on the details of your implementation.

Of course, you should also use another principle I didn't mention (because it's very obvious and popular) namely to write abstract interfaces that don't leak implementation details, and keep these interfaces as small as possible, so that you can reason about large systems through these "bottlenecks". But I have found that, on the back-end, it's just a bit of extra work to add an extra indirection, whether you use it now or not. Instead of saying "we will NEVER do it the other way because it makes no sense", if it costs you so little, why not allow for it, in case later someone will want it? The interfaces are often a leaky enough abstraction for this to matter.

For example you'd have article.getAuthor() if you didn't make that extra table join indirection. And now what will article.getAuthor() return when articles can have many authors? It would return a random author, for backward compatibility. With my approach, you would prevent the "older" apps from using dumber interfaces. It's just a bit extra work for everyone, for huge wins later. And that's the point.

7. Event streams can be abstracted into pipelines and middleware. You can do undo/redo, store histories, have Merkle Trees and more. Compare SVN and Git :)

8. Sync becomes much easier when everything has a history of states. Look at git. You can just use it. Look at scuttlebutt, blockchains, or other types of merkle trees. Everything becomes super simple to reason about.

This forum could be refactored to be distributed. Everyone owns their own node of the tree and the relationship to its children (replies), and everyone else just replicates them scuttlebutt-style. Expanding a tree is fairly simple, and each branch has a merkle hash. There is no central server.

And the best part - you could start with a centralized app and gradually move to a decentralized, end-to-end encrypted model, if you only had the foresight to make sure that your tables has primary keys corresponding to how people look up some data (node, etc.)

The only thing you'd need to have consensus about is the ordering of replies to a node. And that consensus can be among the repliers or simply dictated by the parent node's owner.

8. Having a non-extractable private/public key be used to sign requests is better than JUST having a bearer token (cookie). If someone commandeers the cookie via, say, session fixation, they still won't have your private key. But they need the session id (bearer token) to look things up on the server. This is "piling on".

Then, on top of this, you can have a blockchain of keys for devices, stored across sites, so you can revoke a device when it's compromised, or authorize some new device with N previous devices.

You can have the same exact mechanism manage users in a chatroom or other merkle tree structure. This is what keybase does.

You can encrypt data on the server, with people's public key, and they have to decrypt it.

You have to make sure that the initial signup requires some sort of token to prevent sybil attacks.

In short — once again the approach is to "layer on more security mechanisms", they should all work independently.

You don't just rely on HTTPS for example, because a server or CA cert can be compromised. You hash passwords on the client with salts before sending, regardless. Once again this is called "defense in depth".

> dragonwriter 45 days ago [-]
>
> > Secondly, if you don't do this, future developers will keep adding parameters in an ad-hoc manner
>
> You seem to presume a situation where you have absolute control over the initial signature of functions added to the codebase but no ability to constrain future changes.
>
> 1. All parameters that for which a default makes sense should be optional and have a sensible default.
>
> 2. (In languages where this is an option) All parameters with a default must be keyword-only.

3. All new parameters to an existing function (from a stable release) must have a sensible default.

> When you are building a re-usable framework, it’s silly NOT to anticipate future needs. The whole point of a framework is future needs.

The point of a framework is to avoid solving the same problem multiple times in each new project. If you haven't needed to solve it twice in two separate projects, be skeptical that you need it in a framework. If you haven't needed it once, don't even consider it. Code for problems you don't have is pure waste.

Much of the advice you offer is going to produce waste because YAGNI; each time they come up the cost may be small, but in aggregate it's going to be a lot of extra zero-value code being written and maintained, bloating development and maintenance costs and timelines. Occasionally, you'll be benefit a little down the line from hitting a problem you correctly anticipated, but often you’ll suffer from having not having anticipated the real constraints of the problem when dealing with it when it wasn't a real need, meaning you’ll have to throw away your just-in-case code anyway, and all the time you'll be dealing with extra complexity dealing with problems you haven't had any real need to address but only imagined might come up in the future.

darkarmani 46 days ago [-]

> 3. Functions should have extensibility, put the required parameters as parameters and always include an “options” at the end. Each function can have defaults that you can extend, which means you need a deep-extend method:

No! Absolutely not! There is a time and a place for this, but it's nearly impossible to reason about the interface if any data can be passed in.

EGreg 45 days ago [-]

I think you misunderstand. It’s not that “any” data can be passed in. The options object is documented in every version. It’s just a place that lets future versions add named parameterd

Madeindjs 44 days ago [-]

IMO If a method take many parameters there is a chance that this methods have too many responsibility and can be cut into two methods.

olalonde 46 days ago [-]

Not sure you are aware but it seems you are describing "event sourcing" in point 6. [0]

[0] https://martinfowler.com/eaaDev/EventSourcing.html

galaxyLogic 46 days ago [-]

>started getting cute with it and applied all kinds of design patterns

Even though there are books about design patterns, taking such a book and trying to "apply" its patterns is a bit backwards I think. The idea of patterns is they describe commonly useful solutions, not designs you "should" use.

Once you started to code in "pragmatic, minimalistic way" I assume you found you could apply the same solutions you had found earlier in new contexts. Those are your own design patterns. That is how design patterns work, some patterns of design "emerge", because they are the optimal solutions.

A Design Pattern should be minimalistic, it should only do what is needed, not anything more. It should only solve its problem in optimal, minimal way. But if the problem it is solving is not your problem, you should not use it.

Scarbutt 46 days ago [-]

Even though there are books about design patterns, taking such a book and trying to "apply" its patterns is a bit backwards I think. The idea of patterns is they describe commonly useful solutions, not designs you "should" use.

This nails it, and author *does* talk about it:

Similarly, knowing about common architecture patterns is a good thing: it helps shorten discussions with people, who understand them the same way as you do. But architecture patterns are not the goal, and they won't substitute for simpler system designs. When designing a system, you might find yourself having accidentally applied a well-known pattern: and this is a good thing. Later, you can reference your approach

easier. But the last thing you want to do is taking one or more architecture pattern, using it as a hammer, looking for nails to use it on.

saiya-jin 46 days ago [-]

I agree with you, but that's not how people usually progress. Its more in line with initial discovery of 'the best and battle-tested way to design code'. Immediately they try to apply patterns anytime they see an opportunity for it. They must be taken more seriously from now on, right?

I get it, I went through exactly same hoops. My guess is, we all desperately want to be those aged and wise devs that nail the implementation in first go, without hesitation, knowing exactly what to expect and avoid, covering all corner cases with some elegant snippet. And learning about patterns feels like the surest way to get there.

Of course as we know now that's far from truth, path to seniority can't be fast-tracked by reading a book or two and that's it. But inferiority complex is rife with junior devs, I mean you think you know a bit and then you encounter a codebase where you are completely lost. You see tiny piece of code that takes forever to decipher and fully understand. Who would feel great at that moment

KISS was a thing 20 years ago, any probably even 40 years ago, there is no need to think now its different

galaxyLogic 45 days ago [-]

It is understandable that everybody wants to learn especially junior developers, and it is a good thing to learn, and what better way to learn than try out different things.

Unfortunately then rather than getting something useful done we often just get some learning done, perhaps learning how NOT to do it :-)

The same issue I think affects the tools landscape. People want to use the latest hyped things because ... they want to learn how to use the new tools. The new tools might not be better, but you don't know until you try them.

Where you make a great point I think is that the most important thing to learn is: The simplest solution that works is typically the best. They used to say "YAGNI", You Ain't Gonna Need It.

feketegy 46 days ago [-]

Most devs prepare for the abstraction nirvana. I see a lot of fellow devs creating complicated code, because "in case we need to switch out the database down the road" or "what if we want to run the web app in CLI"

In 20 years of programming I maybe seen one or two times a large application switched database engines and I've never seen a client want to run his/her web application in CLI...

The art in programming is to decide whether you need that abstraction or not.

UK-Al05 46 days ago [-]

I find the benefits of abstracting away database, isn't so you can swap.

It's that' the high-level business logic does not have any low-level unrelated persistence code in it.

Double_a_92 46 days ago [-]

> in case we need to switch out the database down the road

I would still kinda consider that. But the solution wouldn't be to create some massive abstraction. Instead I'd try to separate logic from database access code. I.e. avoid reading data from the database while you are in the middle of some calculation, or in the GUI somewhere. If you need the data do it beforehand and pass it in a generic data format.

nabnob 46 days ago [-]

I'm not sure that was the best example to use of an abstraction that leads to more complicated code, with database code I feel like it's way simpler to have it separate from your business logic. That way, there's fewer places where you need to make changes anytime there's a schema change.

Abstracting away the database also makes it easier to write unit tests for your business logic.

Vinnl 46 days ago [-]

It's not just programming, though. A large part of it is defining the product roadmap: code can get a lot better if a product team has the guts to unequivocally say that some features are not going to be part of the product, ever, such as running the web app in CLI. That's not just a programming decision - though I guess influencing the product roadmap could be seen as part of the art in programming as well.

downtide 46 days ago [-]

Yeah I had a manager that was quite good at throwing out functionality. Didn't like it at the time, but they were mostly right - of course they were just trying to save money.

james_s_tayler 46 days ago [-]

I like this. Because time after time after time I have seen the opposite. Managers saying "but we might need it again later" and...

We. Never. Have.

And even if we did... its still under source control if you really want to dig it out.

codr7 46 days ago [-]

Second that, after 34 years in the game I'm always on the lookout for dead or dying code. Every line I can put out of its misery is one less to deal with in the future.

And I find that removing dead code often unlocks further improvements that become obvious without the noise.

Even if the functionality is needed down the line, it tends to require massive rewrites to catch up.

james_s_tayler 46 days ago [-]

I agree, but with a caveat. I think more often than not, you're right, worry about things like that just tends to be a waste of time, because while it might happen it mostly doesn't. Vendor lock-in is a good example. Probably not worth losing any sleep over.

However, I definitely see value in making code easy to throw away. That's something slightly different than making something easy to swap out. Entire front-ends being re-written is something I've seen a couple of times. But when they were super-glued to the back end that was very, very expensive to do. They were hard to throw away. I think clean separation between boundaries means when you want to throw something away you can. Doesn't mean it will be easy, but a heck of a lot easier. You don't necessarily always want to replace certain things either, sometimes you just want them gone.

schwurb 46 days ago [-]

Of course it all depends on the application at hand, but databases and other system endpoints are something I treat as volatile. However, I don't commit to the madness of replicating a whole database interface in our codebase. Instead, I find it sufficient to bundle all database access in a single point. (i.e. by using a singleton class). That way, later change is much easier and it's a very low investment in maintenance.

GuiA 46 days ago [-]

Well yes. It all sounds all so easy when you put it like that.

The problem is that, in my experience at least, you can't just teach junior engineers how to go straight to phase 3. You have to go through phase 1 and 2 to really develop a sense for what makes a solid, streamlined design.

Some never get there - either because they become set in their ways early, or because they work in organizations where the wrong kind of thing is encouraged. Some get there faster - because they've worked with mentors or in codebases that accelerated their learning.

But like with any craft, you have to put in the hours and the mistakes.

(Yes, there are John Carmacks in the world who go through all those steps within 18 months when they are 12, but they are 0.0001% of the programming population)

userbinator 46 days ago [-]

I think the state of CS education is actually responsible for a lot of overabstraction; students are taught early to worship abstraction (because of all the buzzwordy benefits it allegedly brings) and apply it liberally, and not taught when *not* to do it. A good way to counter that might be to get them to read early opensource code (e.g. first few versions of UNIX, some BSD stuff, etc.), which I think is mostly an example of "abstraction done right".

GuiA 46 days ago [-]

Yes, curricula tend to focus on writing code over reading code, which is a shame. Looking at existing code, extending it, refactoring it, etc would ideally be something that students do as much as writing things from scratch.

taneq 46 days ago [-]

Carmack is actually a good example of professional development. A lot of what he talks about is how his style is evolving and the pros and cons of different approaches.

GuiA 46 days ago [-]

Yes, that's a good addendum. Like any other craft, even the masters have always more to learn.

As they say, it is only when you get your black belt that you are really ready to start learning karate.

taneq 46 days ago [-]

> I realize I code a lot slower now than in the past and that I write much fewer lines of code. It's both good and bad because sometimes I'd even spend hours just trying to properly name a variable.

I think the important thing happening here is more than just naming. You're taking the time to fully consider what you're doing with the new variable in order to name it. That's time very rarely wasted.

psychoslave 46 days ago [-]

There are other factors here:

- how much time will the code be reread?

- does the carefully selected name does make sense for other readers, future self included? That is, does it help to make things easier to understand for them?

- a good name can be reused extensively over your whole career, so a few hours might well be well invested time from this perspective if you plan to keep a foot in the technical side of the story all along. Actually, a good name is surely something you might have the opportunity to reuse outside a coding context.

croh 46 days ago [-]

Well said. Similar to you, I code now almost exclusively in Flask. I don't want to spend days and night learning (and remembering for interviews) unnecessary abstractions and apis. Instead I prefer to spend more time on CS fundamentals, if I have to. Sad part of this story is broken hiring. Your resume doesn't get short-listed unless it comes with new hyped-shiny-toy. But this can encourage you to put more efforts on finding good employer.

Apart from juggling act you mentioned, there is another caveat. Many devs don't understand exact use-cases of these design patterns and use them in wrong context.

On foot note, If you don't have good team of engineers similar to OP, best way to craft your art is -

- pick up a good library in your subject

- start copying it line by line

- when copying, try to understand everything

- this will teach you lot about designing softwares

Even though this sounds like stupid and time consuming, it is not. Believe me. You don't have to even reach 100 %, just try to reach 33 %. You will learn lot by this in short period.

paxys 46 days ago [-]

This describes my career perfectly. And at every stage I inevitably get annoyed at other engineers not in the same stage as me.

tarsinge 46 days ago [-]

This mirror my experience too.

I think at its core the issue is that code duplication is irrationally seen as a bad thing. But from my recent experience of the last few years with ultra minimalists approach making a change to a non abstracted code is so much faster. Yes it's boring and feel unsophisticated, but when you only have flat functions vs an architecture tightly coupled to a business process, it's a matter of hours vs days/weeks.

In short I would add to the title "because reusability is overrated". Especially when the trade off is complexity.

JMTQp8lwXL 46 days ago [-]

I've seen code duplication fail me. In my current role, any time we want to spin up a new Node.js process, we "fork" an existing scaffold project. (Some of the repositories are actual forks; many more just copied the scaffold code and init'ed a new git repo). Whenever anything changes to the scaffold, there's no easy way to percolate those changes down the forks (then they all have to be versioned and deployed). It leads to a lot of copy-pasting and busy work that could instead be spent solving more pertinent company problems. Nobody ever considered how, once you end up with a growing X number of copies of what is nearly identical

code, how to manage it. It's an increasingly unscalable problem for us. It would be 100% more efficient if all the development happened in a monolith that kept clear lines between the projects that exist as "forks".

gmueckl 46 days ago [-]

There are also cases where code duplication is better than the alternative. Sometimes, you end up writing the same pattern (often a hand full of lines) over and over again in different modules and these copies can't easily be moved to a common place. In this case, it is likely be better to accept the repetition than to mess with high level architecture to worship the gods of DRY. Slavishly following DRY is often a cause for overengineering a system into a tar pit.

TeMPOraL 46 days ago [-]

DRY is unfortunately commonly misunderstood as a principle of avoiding repeating the same *code* patterns, whereas it's beneficial effects are entirely about avoiding creating repeated copies of a single *concept*. There are many ways to avoid or mitigate consequences of copying concepts in your system, not all of them involving abstracting similarly-looking code into a method.

Sometimes, a bunch of code repetition shouldn't be DRYed, because they're different concepts, just looking the same. Your language most likely doesn't have means for necessary structural abstraction - Lisp-style macros. If it has, you can sometimes address these cases cleanly and efficiently with a macro by correctly noticing that the concepts are in fact related, they're instance of a same structure or behavior (much like a book on physics and a book on literature are unrelated if you look at the topic, but related if you look at the structure).

And sometimes the best thing to do is to mitigate the consequences - through a quick unit test that fails when you change one thing without changing the other, or even a note in comments. "These two things are related, but we're not enforcing it." Or, "these two things are unrelated, even though they look the same".

collyw 46 days ago [-]

I have come to a similar conclusion lately, though I would say "code reuse" as a goal is the problem. Someone said you shouldn't aim for code reuse, but avoid duplication, which seemed to sum it up better for me.

james_s_tayler 46 days ago [-]

Out of curiosity what kind of projects are that small?

I guess I have hobby projects that are that small, but all my professional work is large, enterprise systems that wouldn't fit in 10 files if they tried.

Makes sense when things are so small to only use what you need. Sounds like you made a reasonable decision for the kinds of things you work on. But when you get past a certain size actual architecture becomes very beneficial.

Of course it's also possible to have a massive enterprise system without any architecture. Believe me it's not very fun.

flukus 46 days ago [-]

I find 90% of the complication in large enterprise projects come from the developers trying to write too much architecture and not enough simple, boring, imperative code. At the moment I'm staring at a validation framework and 6 layers of validator inheritance who's job it is to verify that one number is bigger than another one. That's just for one tiny part of the overall system, nearly everything has to be in some inheritance hierarchy or pattern to minimize duplication or even worse, abstracted into an internal library that makes the entire organisation tightly coupled. God forbid you declare more than simple data class per file. We have tens of thousands of lines of code just to extract a few values from a csv file, tens of thousands more to write xml documents all because someone had to prove how clever they were.

Once you take away all these unnecessary complications you quickly find a lot of "large enterprise software" could be a few scripts dropped into a cgi-bin directory, some could even be a shell script + cron. I think we'd be better off with this as the starting point for all enterprise projects and should not become a "real project" until they past a point of complexity where they really deserve to be.

I'll take small stand alone spaghetti scripts over bloated architectures any day. At least they can be refactored without taking the whole system, all it's libraries and half the enterprise into account.

james_s_tayler 46 days ago [-]

I currently have the opposite problem. Staring a system with zero frameworks where every time the developer wanted to do something they didn't reuse an existing solution, they simply wrote a new one. So there's 10 different ways to create a customer and now there is a requirement to change

some validation. It's a major high-risk overhaul of the system to change the validation on a single field of a single entity.

I've seen big enterprise systems that had solid architecture and I quite enjoyed working with them and I've seen the trash-fire variety too.

Abstraction can be wonderful. Abstraction can be ghastly. It's how you wield it. Well designed systems are just that. Well designed. They have solutions that fit their current problems well and make just enough room for the problems of tomorrow, but not for the unanticipatable problems beyond that.

Poorly designed systems don't have solutions that fit their current problems well (or at all), and/or don't have solutions that are amenable to tomorrows problems or have solutions well suited to tomorrows problems, but not todays, or have solutions that fit imaginary problems they will never have.

I think that's what the article fundamentally hints at. Taking the time to look at the problem and taking the time to derive and refine a well fitting solution. If you can do that, you've won.

frgotmylogin 46 days ago [-]

The enterprise project I have been on for quite a while now is overengineered, but the overengineering was done by some really smart people with years of overengineering experience. It works and isn't quite an unmaintainable, dangerous monster, but as someone who is very much in the short, simple, readable code camp, it drives me nuts.

pytester 46 days ago [-]

>Out of curiosity what kind of projects are that small?

A lot of open source projects with a tight focus which solve a specific problem very well.

>I guess I have hobby projects that are that small, but all my professional work is large, enterprise systems that wouldn't fit in 10 files if they tried.

Most of my enterprise work could be broken up into loosely coupled subprojects of that size similar to open source. IME that doesn't tend to happen, but it could.

StreamBright 46 days ago [-]

Not OP but couldn't you split up you project to smaller projects?

>> But when you get past a certain size actual architecture becomes very beneficial.

This article and only rejects the convoluted architecture approach with design patterns and suggests that you can come up with your own design without using these. It is not arguing that there is no need for architecture at all.

james_s_tayler 46 days ago [-]

>Not OP but couldn't you split up you project to smaller projects?

My own personal stuff I can do whatever I like with, work stuff I don't have much say over. Could they be split up? Oh, yes. Does it need to be 1 million lines of code? No... I think it could probably be about 1/5th to maybe even 1/10th of that.

But still, even if it's split into microservices or just smaller modules, the aggregate lines of code required to solve the entire business problem is still a great deal larger than will fit into 10 well crafted files. Was just curious the nature of the work that naturally fits into that size.

>This article and only rejects the convoluted architecture approach with design patterns and suggests that you can come up with your own design without using these. It is not arguing that there is no need for architecture at all.

I think this article is great. I took this article to be advocating for taking a good hard look at the problem at hand and really nutting out a solution that fits it well. It doesn't reject design patters, per se, it rejects not properly thinking things through.

marcosdumay 46 days ago [-]

Not microservices.

Way too many (but not all) enterprise codebases can be divided into small non-interacting pieces that share only a small bit of code. If so, diving them transforms the problem from a monster program that nobody can ever understand into a lot of tractable ones that a single person can read.

GordonS 46 days ago [-]

I've been on a similar journey, and I've seen this pattern repeat itself again and again!

1. Hack any old shit together, but it works

2. When you actually have to maintain what you previously wrote, you realise (1) doesn't work so well. Then design patterns seem like an epiphany, and you Cargo-cult the shit out of them, using them *everywhere*. You dogmatically eliminate *all* code duplicatation, use mocks with wild abandon, and are not happy unless you have 100% test coverage. For bonus points, you also overuse abstraction. A lot.

3. When you actually have to maintain anything you previously wrote, you realise what a tangled mess of abstraction you have made - you can't simply open a file and ascertain *what it's doing*! You also realise that the tests you wrote to achieve 100% coverage are *crap*, and don't really prove anything works. You finally reach a zen-like state, realising that *simplicity* is key. You shun all forms of dogma, and use patterns and abstraction, but only *just enough*

downtide 46 days ago [-]

When I couldn't program I almost achieved more! I saved time by picking stuff up and glueing it together. Then later spent ages learning specific softwares, plugins and their wiring only for them to fall out of favour. Later frameworks etc.

A web outfit I worked at should have concentrated on a few small plugins/components that would have handled most of their sites. Instead other behemoths emerged, that added pain and complexity to what should have been very simple sites. Only the author understood the ins and outs of a half finished product, that ended up bastardised for each project, resulting in multiple version hell. But hey this was before good 'Git'ing. Oh for hindsight.

iask 46 days ago [-]

Thank you for sharing. Same experience here. I felt I was the only one going down this path. Many projects I look at has too many unnecessary layers files etc.

I joined a company recently that has a simple app for end users to take orders over the phone, perform lookup and refunds. Something that can be built in a few days, seriously. When I looked at the code - WTF!!! The previous dev over architected this thing. Unnecessary layers, interfaces etc. one simple change can take hours.

I think developers need a little bit of management experience to understand the impact of these complexity. At the very end, companies just want something usable to stay in the game...a Honda and not a Rolls.

amelius 46 days ago [-]

It looks like you were applying design patterns "just because". Obviously this is not a good thing.

A better approach is to take some time and think about all the requirements of your project, and to take into account what requirements might be added later. With that in mind, you can choose the abstractions that you need, and from there start coding. That way, your design patterns start working for you instead of against you.

> naikrovek 45 days ago [-]
>
> yes, and the problem is that carefully engineering your solutions is not what's taught; applying design patterns is what's taught, both in college and at most/all large employers (certainly all of the ones I've worked for.)

StreamBright 46 days ago [-]

Exactly. I never got into OOP design patterns and my co-workers could not convince me this is a good idea. I thought for a while that I am crazy but then I got to know Erlang and Clojure. Joe and Rich set me straight on software design.

>> Keeping code simple is not easy. It takes a lot of trials and errors to know what works and what doesn't.

Refactoring helps. I usually achieve 20-40% reduction with the first refactoring.

> jacquesm 46 days ago [-]
>
> Even OOP has its place. I once wrote a simulation of a physical system with a lot of moving parts and OOP made the problem tractable and the code readable and maintainable. I don't think any other style of coding would have had such an enlightening effect on that particular problem.
>
> > mac01021 46 days ago [-]
> >
> > Are you able to provide any more detail about what the simulation had to do, and how OOP helped?
> >
> > > jacquesm 46 days ago [-]
> > >
> > > Lots of mechanical linkages interacting with each other. Move one and everything else moves as

well. Gears, shafts, levers, chains (modeled as independent links), that sort of thing.

StreamBright 46 days ago [-]

Totally. For me just AbstractUserGeneratorFactory does not make sense.

You can have a look here to get examples:

https://docs.spring.io/spring/docs/3.0.x/javadoc-api/allclas...

james_s_tayler 46 days ago [-]

It's actually a pretty incredible framework.

While it might just look like word soup from that particular vantage point I'd venture to say that Spring Framework is one of the most successful examples of OOP in history. It has massive adoption because of its sheer flexibility which all of those abstract generator factories give it.

Granted, I think Ruby on Rails is also an amazing project. For slightly different reasons, but then again, they're different use cases serving different paradigms and different groups of people.

It isn't the paradigm that makes something good or bad or better or worse. It's the quality of the implementation.

StreamBright 46 days ago [-]

Sorry I am not trying to turn this into a framework discussion. I just used that list the exhibit the sort of naming that I do not really like. I like if I can tell what my code does. I usually use verbs in naming methods create, update, delete, add, remove, run, wait, etc. and try to name the classes (or more likely just packages, modules) after what they do.

Regarding your comment about the quality of implementation, I agree. This is why I like paradigms that are simple enough for everybody to understand.

wonderwonder 46 days ago [-]

I've always been happy with just OOP and dependency injection. Anything more and things start to get difficult to follow. Currently working on a a legacy system that uses micro services and it takes hours just to figure out where the code is that needs to be changed and to trace how those changes will be propagated through the system.

dgellow 46 days ago [-]

What about tests? In my experience simple code without abstractions often becomes a pain to write tests for. For example that's one of the main reason I see to use some form of Dependency Injection and other indirections, even if in practice you have only two implementation of each dependencies (once in your tests, once the real one).

TeMPOraL 46 days ago [-]

Basic dependency injection is just functional style - code getting its dependencies as arguments. I feel it's often actually *simpler* than having code manage its own dependencies. I didn't think that until recently, though, because my primary exposure was always bloated Enterprise Java DI frameworks written in pre-Java 8 style. I'm not saying the frameworks were bad per se, just that the amount of incidental bloat prevented me from understanding the core insights of "dependency injection".

dgellow 46 days ago [-]

I gave DI as an example, but other types of abstractions can also make an implementation more complex (i.e: less simple) but easier (or just make it possible) to test, which can be quite important (dependending on the context of course).

Now for DI being a functional programming principle, I don't know, I guess you could argue for this. I personally learned it as a way to satisfy the "D" from the SOLID principles, so in a context related to object oriented programming. But there is always some level of overlap between paradigms.

TeMPOraL 46 days ago [-]

Depending how you look at it, you could argue that classical approach to SOLID's dependency inversion is an amalgamation of two separate concerns - dependency injection for ensuring that neither "higher" nor "lower" level depend on the other directly, plus a type system restricting what operations are available to both. There are so many ways to look at and reason about the problem of structuring programs that pretty much every year I discover a new perspective on an old thing that blows my mind.

Now my enlightening moment about dependency injection was this: it's literally as simple as

passing an argument to a function. In a functional approach, you may be passing lots of values and closures expressing dependencies with surgical precision, in an object-oriented imperative approach you might pass an instance conforming to an interface just once. But it's the same concept.

enriquto 46 days ago [-]

> Next 4-8 years, I started getting cute with it and applied all kinds of design patterns, Factory, Abstractions, DI, Facade, Singleton you name it.

What's cute about this madness? There's nothing uglier than that! Simple code is cute. The simpler, the cuter.

hoseja 46 days ago [-]

It's an English idiom. See [1], point 3.

[1] https://en.wiktionary.org/wiki/cute#Adjective

enriquto 46 days ago [-]

Thanks! I ignored this meaning (not a native speaker). Feeling a bit ridiculous now...

abtinf 46 days ago [-]

In the spirit of being helpful: “ignored” implies you knew the meaning and intentionally did not address it. I think you meant “I was unaware of this meaning” or “I didn’t know that.”

jbn 46 days ago [-]

not to nitpick any more than necessary, but I don't think that's entirely correct: "I have ignored" implies that GP knew the meaning and intentionally did not address it, while "I ignored" is correct and also means “I was unaware of this meaning” or “I didn’t know that.” ... at least that's how I see it.

woolcap 45 days ago [-]

"I ignored..." = I was aware, but consciously chose not to address it, but perhaps softer than, "I consciously chose not to address it. "I ignored the fact that the sun was out, and returned from the beach sunburned." (I knew the sun would burn me, but went out anyways, with perhaps a shade of not realizing how much I would get burned.)

"I was ignorant of..." = I was unaware. "I was ignorant of the fact that the sun causes sunburn, and returned from the beach sunburned." (I didn't realize the sun would burn me.

enriquto 46 days ago [-]

according to wiktionary [1], this is indeed a possible meaning of the verb "to ignore", but it is marked as "obsolete".

This is a case of false friend when translating from french, where this meaning is the first one [2].

[1] https://en.wiktionary.org/wiki/ignore#Verb

[2] https://fr.wiktionary.org/wiki/ignorer#Verbe

jbn 46 days ago [-]

Right, and it's likely that this verb was imported into English from French :)

TIL!

winrid 46 days ago [-]

You can break things up without overdesign right?

I think you should do what is easiest most of the time. However, that is hard to measure. Easiest now, or when you need to finish this and move onto the next thing without spending two more sprints fixing bugs?

I prefer small/reasonably sized components because I can easily cover them in unit tests and sleep easier at night. I built a survey builder at one company (think mix of survey monkey/qualtrics) and that is probably over 100 files. But the codebase is straightforward and simple (no complicated inheritance, one tree data structure for pathing, lots of code reuse)...

JMTQp8lwXL 46 days ago [-]

Breaking things up is, by definition, design. Having meaningfully enforced boundaries between subsystems, that collectively add up to forming one larger project, is applying the concept of design.

lmm 46 days ago [-]

You can define it that way, but in that case my position would be: devoting time and effort to design leads to worse design than not doing so.

JMTQp8lwXL 46 days ago [-]

It depends on how carefully you've considered requirements before you begin coding. Sometimes when I start coding, it triggers the realization that I need to change the design, because I failed to deeply consider something well enough.

When you're designing (paper, pen, boxes, arrows, no code), it can be easy to hand-wave or fail to consider real problems you might encounter.

lmm 46 days ago [-]

Indeed. I'd argue that this is so common that it is better to begin coding first, and allow any "design" to emerge, rather than trying to design before coding. Perhaps with extreme care one could achieve a decent design up front, but it would surely take much more effort than simply being willing to refactor as one goes.

TeMPOraL 46 days ago [-]

I've learned to prefer doing both. I start with spending enough time with pen and paper/whiteboard to get a coherent picture of what I want to build and how, and then I start coding it. I flip between "design" and "code" whenever I start to feel that current activity is getting hard. Getting increasingly confused as to what functions to write, how, and where to put them? I switch to design phase and sort it out at a higher level. Starting to feel I'm losing touch with the ground, or that I'm not able to figure out which design option is better? I go back to code and write what I can until the code itself reveals answers.

In essence, I see creating software as a dance between higher-level, structural concerns and low-level, immediate needs. Since one mode solves the problems found in the other, I try to be working in whichever mode I can make progress most effectively, and flip as soon as that changes.

yodsanklai 46 days ago [-]

You can't beat simplicity, but software aren't planned entities. They evolve from the collaboration of multiple persons with a variety of skills and personalities, working together to meet deadlines.

afpx 46 days ago [-]

The only problem is that it takes at least 10 years to get to that point. No one has found a shortcut, yet.

nine_k 46 days ago [-]

"Simplicity does not precede complexity, but follows it."

This is why it's often worth it to come up with a complicated design, work on it without implementing it as much as possible, think it through, understand where the core functionality lies, and make a new, much simpler design.

d--b 46 days ago [-]

Let’s see how the OP’s system looks in 20 years. Then we’ll see how clear and simple it has remained.

The OP is railing against a culture that never existed. Banks software architects are not in their offices smoking cigars and making UML diagrams that they send to coders, only to realize later that they made the wrong trade off.

What happens is:

You design a system for what it’s supposed to do. You do it the way the OP says: nice ad hoc charts, talk to a lot of people, everybody agrees, all is swell.

Then time goes by, new integrations are made, newer tech comes out, costumers need change, business orientation changes. And what used to be neat starts to become ugly. People cut corners by putting things where they don’t belong to save some time. And then it’s all downhill from there.

There is a toilet analogy that works well. If you go to a bar and the toilet seat is very clean, then you'll make sure to clean it before leaving. But if the toilet is already dirty, you're not going to be the one cleaning other people's pee, so you just add your own and leave.

The same is true in software architecture, once it doesn't look neat, everybody just pile up their crap in the way that demands the least effort. "The system is shit anyways".

I find it a little easy to say: "ha look at those guys, they spend hours trying to sort out system architecture, while all you really need is common sense".

gerbilly 46 days ago [-]

> But if the toilet is already dirty, you're not going to be the one cleaning other people's pee, so you just add your own and leave.

Good analogy, but in code it's more than just disgusting to clean up after others. Changing code that was poorly written by someone else may cause bugs, bugs that now become your problem.

The goal of every programmer faced with such a codebase—as in the dirty bathroom analogy—is to get in, do his business as quickly as possible, and get out. Iterate this over time and the problem just keeps getting worse.

It's like the tragedy of the commons, where each programmer pollutes a common resource because the incentives are set up to reward that kind of behaviour.

This leads the codebase to become a 'Big Ball of Mud', the most popular architectural pattern in the world: http://laputan.org/mud/

edelans 46 days ago [-]

It reminds me of the broken window theory https://en.wikipedia.org/wiki/Broken_windows_theory : same as the toilet analogy, but more classy

snapetom 45 days ago [-]

> Banks software architects are not in their offices smoking cigars and making UML diagrams that they send to coders,

You'd be surprised at how common this is, especially in large companies that play, "let's pretend to do technology." I'm leaving a large hospital where I've spent half my time butting heads with our "architect" who's skills have been frozen since 2005. Leadership is all eager to chase modern buzzwords like "machine learning" and "AI" but this guy is advocating for outdated crap.

peteradio 46 days ago [-]

And then there's the only available bathroom that is filled 3 ft deep with tp and shit and you must cut paths through to make brown. That is where the magic thinking happens.

bigbluedots 46 days ago [-]

It's pretty rare these days that systems are maintained for that long. More than likely there'll be a rewrite every few years anyway to keep up to date with $EXCITING_NEW_TECH.

james_s_tayler 46 days ago [-]

Hmm maybe in the Bay Area?

Everywhere else I've seen software that is on average a decade old.

gizzlon 46 days ago [-]

That does not mean that typical service will live for a decade though. You don't see all the ones that did not make it.

alkonaut 46 days ago [-]

Architecture for something that is only actively developed for say 5 years is pretty uninteresting. You can make better or worse architectural/design choices, but you never really test whether the choices were sound if you don't maintain it for a longer time.

collyw 46 days ago [-]

Yet there is COBOL code still running.

cryptica 46 days ago [-]

This article has some contradictions:

>> Third, we had practically no references to the common architecture patterns and other jargon referenced in common software architecture literature, such as Martin Fowler's architecture guide. No mentions of microservices, serverless architecture

Then a few paragraphs later:

>> Is your architecture split into different microservices? Mention why you decided against going with a monolith, that might have some other benefits

Another contradiction (which mostly contradicts the general premise of the article):

>> We created plenty of diagrams, but none of them followed any strict rules. Just plain old boxes and arrows, similar [this one describing information flow] or [this one outlining class structure and relationships between components]

In the last link ([this one outlining class structure and relationships between components]), the article says:

>> If you have previously worked with the [VIPER] architecture, then the class breakdown of a RIB will look familiar to you. RIBs are usually composed of the following elements, with every element implemented in its own class:

... and then it shows some kind of class diagram which looks vaguely like UML in which the classes have highly architected names like 'Interactor', 'Presenter', 'View', 'Builder'... Nothing to do with the underlying business domain. Doesn't look like simple design to me. The recommended approach looks more like complex VIPER architecture.

kdmccormick 46 days ago [-]

Good observations. I think a more accurate portrayal of the author's experience would be "Clear and simple Architecture is underrated".

moksly 46 days ago [-]

The value of Enterprise Architecture doesn’t come in to play until you’re an actual Enterprise.

We operate more than 300 IT systems, from a myriad of different and switching (courtesy of procurement) suppliers. These systems are operated by 5000-7000 employees, and range from making sure employees get paid and patients get the right medicine to simple time booking apps. Most of these systems need to work together, and almost all of them need access to things like employee data.

Before we had a national strategy for enterprise architecture, which defines a standard model for organisation data, all of those 300 IT systems did it their own way and most of them actually thought we liked that so they came with no APIs. Imagine having to manually handle 5000-7000 users in 300 different IT systems...

That’s the world without Enterprise Architecture, and we’re still paying billions in taxpayer money to try and amend it. Because you don’t move 300 IT systems, some of them running on COBOL on mainframes, over night. And that’s just our municipality, there are 97 others with the exact same problems.

Don’t get me wrong, I get the sentiment of the article and I actually agree with most of it. The thing is though, developers have very different opinions about what “simple design” is, I know, because I’ve build a lot of the gaffa-tape that integrates our 300 IT systems and not a single system has had remotely similar APIs.

blub 46 days ago [-]

I've seen you mention your organization and the challenges you're facing few times and I'm curious what kind of architecture books or principles you'd vouch for based on your experience.

moksly 45 days ago [-]

It’s build around our own version of TOGAF, but I’m not sure I’d really recommend that to anyone. It’s also more political than technical and suffers from a lot of “not invented here” even in competition between different government agencies and changing bosses.

A good example is the OIO standard we use to model most our abstract data design. It’s basically a local standard, which means it’s different from the EU standards that do the same. Which again means, that we had to work with Microsoft to get OIOSAML working with ADFS and are still working with them for Azure AD, and it may all be in vain when we eventually swap to EU standards as the rest of Europe catches up.

The thing is though, we started the journey before there were EU standards, and a lot of the decisions that seem bad today were right at the time. Over all, it’s still a pretty huge benefit to what was before.

To get back to your question. The thing I’ve done that has been the most useful in EA hasn’t been TOGAF or any of the other EA focused frameworks. It’s been the year of political science I took at the university, I think it equals to part of the American MBA but more focused on Enterprise Admin and HR. Because Enterprise Architecture is mainly about understanding the business on its terms and finding the compromises to make your tech sector understand it. I think being able to communicate and understand your business is a lot more important than whether you map things in X framework. I mean, your developers are probably going to understand your PowerPoint drawing just as well as your UML/ArchiMate anyway, and the less tech details you define, the better because the article is actually right about developers knowing better how to build

things. If you tell them how the data is mean to be understood by any system that receives a User object, then you won't have to tell them how to handle it beyond that.

corodra 46 days ago [-]

>The value of Enterprise Architecture doesn't come in to play until you're an actual Enterprise.

Probably the smartest thing ever said when it comes to design patterns.

To put it in non-tech terms, a lot of design patterns equates to learning how to build a suspension bridge when building a back patio to a house. There's value, sure, maybe. But don't kid yourself. 80% of projects don't survive for more than 3 years at best. Most of which never really get "updated" after a year or two. Nor see teams more than half a dozen people.

ajuc 46 days ago [-]

That's not architecture, just standarization.

bamboozled 46 days ago [-]

In my career thus far, I can honestly say I've never, ever, ever seen an "Architect" who actually provided valuable inputs.

Not trying to say they don't exist, but I've just never witnessed someone with that title actually have a positive impact on any project I've worked on.

The only semi-positive value I've seen an architect have is when they counter another architect to allow the engineers get on with their work without interfering.

Maybe the issue with the job comes from the connotation that an architect is someone with supreme insights? Where as most usually, they just over simplify things and expect engineers to deal with the implementation details (the hard part).

Timberwolf 46 days ago [-]

In my experience, architects who are valuable to their teams tend to be the ones who rarely do any "architecture" themselves; instead they work their arse off trying to smash apart every last blocker to the engineers in a team being able to own architectural responsibilities themselves. (This may include asking smart questions to help a team who don't really do systems thinking start engaging with it). This inevitably ends up off in the EA realm grappling with Conway-type questions: not so much "how should we structure our software to make it good?" as "how should we structure our organisation so it naturally produces good software?"

Sadly these people are also rare as it requires a combination of sufficient technical skill and the ability to effectively navigate the people side of the equation.

The "white paper" style of architect is very frustrating in comparison, not least because they are too removed from the context and impact of their decisions. This results in a situation where a team views their architect as merely a source of additional work, much of which is frustrating and pointless if not outright damaging to the system being built.

tootie 46 days ago [-]

I was an enterprise architect for about a year and it was dullest most soul-sucking job I ever had. In a sense, it was incredibly cushy. I had zero responsibility. I could easily just drop technical decisions on teams and not have to deal with the repercussions. But it really just drove me nuts. And I hated the other architects because they had set this system up and seemed perfectly content.

My role before and after as a director was always to give my tech leads a really long leash. I try to never force decisions on them, but rather let them work their own way and my job is just to make sure they've considered the project goals correctly and their solution is going to fit.

petjuh 46 days ago [-]

We have a great architect right now, but he's really just an engineer designated as the "architect". He also codes sometimes.

rumanator 46 days ago [-]

> We have a great architect right now, but he's really just an engineer designated as the "architect".

Why is anyone assuming that an software architect is not supposed to be a software engineer?

The software world is not divided as the civil engineering world, where architects are responsible for meatspace UX and engineers are tasked with bringing the UX to life.

In the software world, software architects are expected to be experienced software engineers who are tasked with coming up with design decisions upfront that render the software development project feasible and economical by picking which technologies to reuse, which modules/services to develop independently,

and how to run in production.

Which of hear tasks is not the job of a software engineer?

TeMPOraL 46 days ago [-]

I feel being an "Architect" is trying to do half of the job that's atomic, unseparable, because the "architecture" half informs and is informed by the other half, "writing and running code", both of them working best in a tight feedback loop. An architect not writing code has to rely on engineers in their team to communicate to them all the insights gained by implementing the architecture - which is a really bad idea, because it's already hard to precisely articulate your feelings about the code to yourself, and now you have to explain that to another person and hope they understand.

bradenb 46 days ago [-]

I feel like an "Architect" should not be a standalone role. The architect for a project should be an engineer working on the project that can make decisions about the underlying architecture when a decision is needed.

noobiemcfoob 46 days ago [-]

Much of an architect's role won't be visible to developers beneath them and -- like a manager -- involves coordinating with other projects or other business units. That a specific project exists at all to work on or is otherwise a discussion topic is often the result of an architect's work.

pvorb 45 days ago [-]

In my company, every software engineer also has the software architect role. This way everybody is aware that they are welcome to think about the architecture of software. There are no dedicated architects. This works quite well in my experience.

drawkbox 46 days ago [-]

A programmer, engineer, creative and product developers job is to create simplicity from complexity.

The job is to tame complexity via simplicity, not make a complex beast that needs more taming.

Sometimes engineers take something simple and make it more complex which is against simplifying either due to bad abstractions or proprietary reasons or obfuscation for job security or to ego flex. Anyone can make something more complex, making something simple and standard takes lots of work and iterations.

Nature is built with simple iterative parts that look complex in whole, systems that work well mimic that. Math is built with simple parts that leads to amazingly complex and beautiful things. Art is built with simple marks to create something complex and beautiful. Humans as well, and they desire simplicity so they can do more and build on that.

mikekchar 46 days ago [-]

I'd make a slight caveat with this. The our job is to make something that is as close as possible to the complexity of the problem. You don't want to make it more complex for obvious reasons. However, you *also* don't want to make it less complex, because then you are removing fidelity. Let me aim a slightly playful jab at the GNOME people for "simplifying" by removing features that I actually need. Only slightly playful as it's the reason I had to give up GNOME. ;-)

ryanjshaw 46 days ago [-]

Uber is barely 10 years old. They can get away with this. Wait until it's 2 or 3 times that age, and its (present or future) regulators sign new laws into place that require massive changes or reporting feeds across multiple systems engineered and documented in this unprincipled fashion. Probably after a couple more privacy breaches or self-murdering car incidents. Nobody will be able to figure out how it all fits together, and the compliance team and auditors are going to throw a fit.

That's when all those architecture processes, repository tools and languages suddenly make a lot more sense. Uber deals with extremely sensitive personal information, and the move towards self-driving cars means they deal with highly safely sensitive systems. The dismissive attitude towards these tools in what should be a highly disciplined engineering organisation disturbs me, but I come from a highly regulated environment so perhaps I was just raised that way.

pmf 46 days ago [-]

> the compliance team and auditors are going to throw a fit.

I've never seen an auditor give a shit either way. They're just box ticking robots.

hdfbdtbcdg 46 days ago [-]

And if they can't tick the box? You fail the audit...

niceworkbuddy 46 days ago [-]

I think there is distinction between documentation in development progress and documentation afterwards. IMHO article is about the first. After something is done, you can write thorough documentation of product with UML diagrams and whatnot.

obstacle1 46 days ago [-]

A big problem IME is people tend to define "simple" as "written in a style I prefer". For example you can extract a series of 10 obviously-related methods from some 2000-line God class into their own class, but have others who are used to a more procedural coding style complain that the indirection is "hard to read" because they need to open a new file. This despite the facts that others find the God class "harder to read" because it contains 2000 lines of code doing everything under the sun, and that class is objectively harder to maintain/change for everyone because nobody knows what things are necessary to change to achieve some goal, because there are no logical boundaries between code functions so you can't tell what needs changing without reading everything.

Cue endless bikeshedding in the name of "simplicity", which nobody is using an objective metric to define.

stinos 46 days ago [-]

2000-line God class ... "hard to read" because they need to open a new file

Might be me, but I've always found this a rather strange argument: either they aren't using 'go to definition' which means that to be able to read the other code they have to scroll through the file manually, leaving where they are, and then go back. That's not really convenient? Or they are using 'go to / peek definition' and then it doesn't really matter it's in another file?

goto_self 46 days ago [-]

I have counterpoints on both ends here.

It's not always possible for a "goto definition" function to work in dynamic languages.

If you're in a large file, it's quite easy to jump around with line marks (a la vim), searching, etc.

stinos 46 days ago [-]

Good point about dynamic languages. But line marks also work across files, right?

goto_self 46 days ago [-]

Yes. Line marks are amazing.

rumanator 46 days ago [-]

I have a hard time understanding the author's point of not using UML but somehow boasting that they used "plain old boxes and arrows" to create "plenty of diagrams".

UML is nothing more than a bunch of "plain old boxes and diagrams", but which have concrete, objective meaning that has been specified and thus help share ideas as objectively as possible. UML is a language, and languages are tools to communicate and share information.

Using ad hoc box and arrow diagrams invented as they go along means they decided to invent their own language, which may or may not be half-baked, that is not shared by anyone other than the people directly involved in the ad hoc diagram creation process.

If the goal is to use diagrams to share information and help think things through, that sounds like a colossal shot in the foot. I mean, with UML there are documents describing the meaning of each box and each arrow, which help any third party to clear up any misconception. What about their pet ad hoc language?

In the end the whole thing looks like a massive naive approach to software architecture, where all wheels have been thoroughly reinvented.

verall 46 days ago [-]

Because UML is generally about defining processes, and it is easy to accidently try to poorly "code" parts of the system in UML, processes that might be easier represented in code. If there is distinct process that is complex/important enough to be architected, by all means use UML.

Normally, at a high level, where people are architecting, what is more important is flow of information and containment of responsibilities. UML is not really designed for describing these situations, and trying to wedge this type of information into a UML diagram can get confusing and can encourage architects to focus on the wrong things.

When people say "box and arrow diagrams", I think that to mean boxes=infomation+responsibilities arrows=information flow.

rumanator 46 days ago [-]

> Because UML is generally about defining processes

It really isn't. In general UML specifies diagrams for relevant system views, but it's centered around diagrams that represents the structure of software projects, not processes. Perhaps UML's most popular diagram is the class diagram, which is complemented with the component diagram and deployment diagram. UML modeling software focuses mostly on structural diagrams, whether to generate source code or dump DDLs. Most of the diagrams used to directly or indirectly represent processes, such as sequence diagrams and communication diagrams, are hardly known and far from popular. Flow charts/activity diagrams are hardly seen as UML, and UML doesn't even provide anything similar to the age old data flow diagrams.

sebcat 46 days ago [-]

Data Flow Diagrams as described in "Structured Analysis and System Specification" (Tom DeMarco) are lightweight and provides a common way to describe a system with a focus on the flow of data.

The book also goes into detail on how to apply it. e.g., the value of having simple diagrams and have separate diagrams to break down the more complex processes in detail.

It's not rocket science, but I have found it helpful in the past to communicate ideas or existing designs.

hdfbdtbcdg 46 days ago [-]

This reminds me of the framework va libraries argument or ORM vs raw SQL. Yes frameworks and ORMs can be constraining and limit clever solutions. But when you need to add complex features to a complex project you are always glad that every other programmer that came before you was constrained and that things use a familiar pattern.

jillesvangurp 46 days ago [-]

I've encountered spring/hibernate projects in the with lots of performance and integrity issues that were easy to straighten out by just cutting out the orm layer and replacing it with non magical, simple SQL. Magic is nice when it works but when it stops working and you lack the skills on the team to make sense of it, things get ugly quickly. This happens exactly when you are adding complex features that stretch the abilities of the framework you are using as well as the teams understanding of that framework.

I'm not a big fan of micro services. But one nice feature is that they are small and usually quite easy to grasp. That makes putting new people on them to do some changes a lot more feasible. A big monolith is much more complex.

I'm torn between building nice monoliths and doing micro services. IMHO for a small team micro services are rarely worth the complexity and overhead and I like the simplicity of having a nice monolith. However, with multiple teams in bigger organizations, it's a better fit. The risk to watch there is Conway's law where your org chart and architecture start resembling each other. The key risk here is the constant staff changes that necessitate having a speedy onboarding and ensuring there is a path forward when key people leave. Complex monoliths are a risk in such situations.

Simplicity and predictability are key here. This does not have to translate into ORM but it often does. IMHO with modern languages and framweorks, there's a trend for more code that does what it says vs. annotation code where all the magic happens in annotation processors that are opaque. This is playing out in the Spring community right now where the latest trend is using Kotlin DSLs to replace annotations and annotation processors. A side effect is that this allows using the graal vm to get rid of JVM startup overhead.

Part of that is losing ORM. E.g. the experimental Ko-Fu framework for Spring is taking that to the extreme: https://github.com/spring-projects/spring-fu

james_s_tayler 46 days ago [-]

I actually love magic, but boy oh boy are you right when it comes to hitting a wall and getting absolutely stuck when magic happens.

I've spent a bunch of time thinking it through and I've come to the conclusion that it isn't actually the magic that is the bad part. It's the lack of the discoverability that is bad. I guess that's why it's called magic right? Because you don't know what the trick is, yet it appears to work...somehow.

I call it dark magic. Magic that is not discoverable and makes no attempt to help you discover it. On the other hand, if you put magic into a solution because you are a wizard and you want to leverage the stunning super powers that magic gives you, if you can also tell everyone how the tricks are being done... it's actually possible to get the best of all worlds.

rumanator 46 days ago [-]

I agree. However, I don't believe anyone has ever been constrained or limited by UML. They might be

constrained by not knowing even the basics of UML, but that's not UML's failing.

Double_a_92 46 days ago [-]

The difference is that you can't really use proper UML to quickly explain something on a whiteboard, unless you were fluent in it. I personally get mental inhibitions when I have to quickly decide if the arrowhead needs to be hollow or filled, or if the arrow itself needs to be a line or dotted, or if the box needs to have rounded corners or not... Especially if it doesn't matter for the idea that I'm trying to explain (maybe even just to myself).

rumanator 46 days ago [-]

> The difference is that you can't really use proper UML to quickly explain something on a whiteboard, unless you were fluent in it.

That's not true. Class diagrams are boxes with arrows, and so are component diagrams and communication diagrams and deployment diagrams.

If you can draw a box and lines on a whiteboard, you can use UML on a whiteboard.

AFAIK, the only thing that's not explicitly supported in UML is data flow diagrams (i.e., convey the data perspectice instead of describing software components and their interactions).

uber99953 46 days ago [-]

Services at Uber are pretty much all stateless Go or Java executables, running on a central shared Mesos cluster per zone, exposing and consuming Thrift interfaces. There is one service mesh, one IDL registry, one way to do routing. There is one managed Kafka infrastructure with opinionated client libraries. There are a handful of managed storage solutions. There is one big Hive where all the Kafka topics and datastores are archived, one big Airflow (fork) operating the many thousands of pipelines computing derived tables. Almost all Java services now live in a monorepo with a unified build system. Go services are on their way into one. Stdout and stderr go to a single log aggregation system.

At the business/application level, it's definitely a bazaar rather than a cathedral, and the full graph of RPC and messaging interactions is certainly too big and chaotic for any one person to understand. But services are not that different from each other, and run on pretty homogeneous infrastructure. It takes pretty strong justification to take a nonstandard dependency, like operating your RDBMS instance or directly using an AWS service, although it does happen when the standard in-house stuff is insufficient. Even within most services you will find a pretty consistent set of layers: handlers, controllers, gateways, repositories.

barrkel 46 days ago [-]

What you describe is an architecture, of course, and it didn't happen by accident.

uber99953 46 days ago [-]

It's an architecture that's almost completely mute on how business-level functionality should be organized.

barrkel 45 days ago [-]

Generally software architecture solves for non-functional requirements, rather than functionality. Product managers organize business-level functionality.

cat199 46 days ago [-]

Umm:

```
+ all stateless Go or Java executables
+ running on a central shared Mesos cluster per zone
+ one service mesh
+ one IDL registry,
+ one way to do routing
+ one managed Kafka infrastructure
- handful of managed storage solutions
+ one big Hive where all the Kafka topics and datastores are archived,
+ one big Airflow (fork) operating the many thousands of pipelines computing derived tables.
+ Almost all Java services now live in a monorepo with a
+ unified build system.
+ Go services are on their way into one.
+ Stdout and stderr go to a single log aggregation system.

= +11 singular/unified things, forming a single, larger system.
```

" It takes pretty strong justification to take a nonstandard dependency ... Even within most services you will find a pretty consistent set of layers ... "

Maybe I'm misunderstanding, but how in the world do you get 'bazaar' out of this?

phillipha ydon 46 days ago [-]

I went to a course by Udi Dahan once, about CQRS. One of the people asked him a question about CRUD. Something along the lines of how would you use CQRS for simple CRUD operations. And Udi was like "just go to the database".

The guy kept asking the same question different ways like Udi didn't understand the question. The response was always the same. Then Udi said, if you need to go to the database, go to the database, don't over complicate things.

It was like a lightbulb for me, having spend ages always trying to fix everything into an abstraction of some sort instead of just getting stuff done.

q-base 46 days ago [-]

I am very much a proponent of simple and pragmatic design. It should be the default, especially for small - midsize companies. I do however have my doubts once you get to very large financial institutions for instance, where you have a vast portfolio of products spread across numerous departments and potentially countries. On top of which comes heavy regulation.

In order to keep this system-landscape somewhat coherent, then I can actually see a need for enterprise architecture. Or put another way, I can't really see how it should succeed without it. The default should still be simplicity, but to keep every department from building their own version of components and keeping security at the forefront you still need guidelines and direction in my opinion.

Not that this necessarily is in opposition to the article though.

merlincorey 46 days ago [-]

This resonates very well with my experiences throughout my career.

The best design experiences from an at-the-time and with hindsight typically result in a collaborative document with a clear narrative structure explaining most importantly what and why as well as how (and as the article mentions, the trade offs involved) such that even junior engineers and potentially even non-technical contributors can understand what we're doing and why.

cryptica 46 days ago [-]

Design is part of architecture so it doesn't make to compare them.

The best architectures are usually the simplest ones which get the job done.

To design the simplest architecture possible, you need to know exactly what "get the job done" means. Many software architects have no vision when it comes to seeing all possible use cases for the product so they don't know what are the minimal structural requirements. So either they underengineer or they overengineer based on what is more profitable for them as employees of the company.

Underengineering is also bad but it's rare nowadays because it usually doesn't align with employee salary incentives so we forgot how painful it is to maintain a code base which copy pastes the code everywhere.

> SamuelAdams 46 days ago [-]
>
> Right, there's the concept of JBGE:
>
> http://agilemodeling.com/essays/barelyGoodEnough.html
>
> I think too many people want to apply a "silver bullet" to all projects: IoC, Docker containers, auto-scaling, etc. But sometimes I'm just tossing data from an API into a database somewhere. I don't need all that complexity.
>
> Other times, I'm building an enterprise product with three fully-staffed agile teams, spending a million dollars annually for five years. Architecture that enables those teams to work in a cohesive way becomes very important, so an IoC pattern might save us a lot of time down the road.
>
> Great architects know when to underengineer and when to overengineer.

liha 46 days ago [-]

It’s interesting how OP talks about the process as something unique and groundbreaking. Honestly that’s how 90% of the tech companies handle architecture and design - a bunch of people whiteboarding solutions and drawing box diagrams and writing down notes. In 12 years I haven’t worked at any company that uses the tools he mentions. Experience usually drives the output and the result of the mentioned “process”. An experienced engineer in the room is more likely to bring up non-functional characteristics and related concerns such as performance, security, high availability etc. You can choose not to have a formalized architecture process or review and you can also choose to just draw boxes which link to each other without completely making sense - like a class with an arrow pointing to a machine and another arrow pointing to a process (which is fine until a couple of years later, someone looks at a dangling process in the diagram and wonders which machine/container it’s running on). Obviously Ymmv because it’s not some “predictable” process and purely relies on drawing out the collective experience and intelligence of the room

andreyk 46 days ago [-]

Boils down to this: "So what is the role of architecture patterns? I see them similarly in usefulness as coding design patterns. They can give you ideas on how to improve your code or architecture."

The whole idea of patterns is to identify often useful, and possibly non-obvious, ideas to be aware of when designing the solution. It's great to start simple, but tricky to make things both simple and robust/powerful - and that's what patterns are supposed to help with. This ends with:

"Software architecture best practices, enterprise architecture patterns, and formalized ways to describe systems are all tools that are useful to know of and might come in handy one day. But when designing systems, start simple and stay as simple as you can. Try to avoid the complexity that more complex architecture and formal tools inherently introduce."

What this misses that if you start simple and stay as simple as you can, you may undershoot and be stuck refactoring code down the line; a fine balance is needed, and patterns are definitely part of a toolset that a good engineer should be aware of when trying to nail that balance.

> james_s_tayler 46 days ago [-]
>
> I really agree about undershooting. I like to try and overshoot by about 15%.
>
> It's definitely a big mistake to overshoot by say 50 or 100 or 200%. But overshooting by just a little often leaves me feeling like "thank God I did that" more often than it does "hmm I guess I really didn't need that".
>
> Balance is absolutely key.

perlgeek 46 days ago [-]

Despite the provocative title, the author argues for software architecture, just doing it in a manner that suits the organizational culture.

He somewhat decries traditional software architecture material, which I find off-putting. IMHO the best approach is to be aware of the techniques/patterns/references architectures, and use just the parts that make sense.

> rumanator 46 days ago [-]
>
> > Despite the provocative title, the author argues for software architecture, just doing it in a manner that suits the organizational culture.
>
> The problems demonstrated in the blog post go deeper than (and are not explained by) organizational culture. They convey the idea that the task of designing the architecture of a software system was assigned to inexperience and ignorant developers who, in their turn, decided that they didn't needed to learn the basics of the task, and winging it as they went along would somehow result in a better outcome than anything that the whole software industry ever produced.
>
> There is a saying in science/academia that is more or less "a month in the lab saves you hours in the library", which is a snarky remark on how people waste valuable time trying to develop half-baked ideas that match concepts that were already known, thought through, and are readily available if they only put in the time to do a quick bibliographical review. This blog post reads and awful lot like that.
>
> > lensopra 38 days ago [-]
> >
> > The saying in software is "Weeks of coding can save you hours of planning."

growlist 46 days ago [-]

In my experience the further away from fierce commercial factors, the greater the tendency towards cargo-cultism. Hiring for roles in government related work in the UK is awash with acronyms and buzzwords, as if it's the case that with enough methodology and certifications we can regulate failure away. Problem is: things still seem go wrong in all the same old ways despite all the latest greatest fancy new techniques. But hey, all our developers are TOGAF certified these days, so that's something!

> james_s_tayler 46 days ago [-]
>
> For some reason I read "methodology" as "mythodology" and I thought "That's genius! That's the perfect portmanteau to describe the phenomenon of people trying to learn and adhere to 'methodology' but then really just adhering to the lore and the myth! I'm stealing that!"
>
> Then I read it again and it didn't say that. But I think that should become a new word. Mythodology.
>
> > growlist 46 days ago [-]
> >
> > Nice. Even better, use it as the name of a company hawking snake-oil methodology consulting.

a_imho 46 days ago [-]

Engineers at higher levels, like staff engineers, are expected to still regularly code.

As they should. The non coding software developer is one of my pet peeves, call it architect if you want. I can appreciate the idea of collecting a paycheck producing wiki entries, drawing diagrams and making slideshows, but they are no substitute for leading by example. In fact, my empirical finding is that the tendency to describe software in prose is usually inversely correlated with the technical ability to create the executable.

jameslk 46 days ago [-]

Sounds great for a tech company with highly skilled engineers. They can afford the type of talent who will be thinking thoughtfully and have the time to do so. Startups seem to attract similar talent, and when not, don't always have the same problems anyway.

But what about the companies that can't afford the best engineers and don't have the bottom up culture? What about the companies that hire overseas IT agencies who do exactly what they're told and no more (it's safer that way when communication and timezones are working against you)?

I've worked in companies both the former and the latter. I've seen the top down "architect" role work better in the latter.

The author even seems to admit this, although briefly:

> To be fair, these same companies often want to optimize for developers to be more as exchangeable resources, allowing them to re-allocate people to work on a different project, on short notice. It should be no surprise that different tools work better in different environments.

This best summarizes it. Different approaches work better in different scenarios. That's really what the title and article should be about.

rumanator 46 days ago [-]

> Sounds great for a tech company with highly skilled engineers. They can afford the type of talent who will be thinking thoughtfully and have the time to do so.

The blog post says nothing of the sort. It focuses on two aspects of software architecture which are entirely orthogonal to the design process: using a common language and tools to communicate and describe their ideas (UML, documentation) and leveraging knowledge and experience to arrive at a system's architecture that meet the project's requirements.

Deciding to invent their own personal language (their "boxes and arrows") and take a naive tabula rasa approach to software architecture does not change the nature of the task.

A rose by any other name...

james_s_tayler 46 days ago [-]

I couldn't agree more. A lot of times people espouse a particular worldview without computing through the 2nd and 3rd order effects in different contexts.

If anything one of the fundamental things to get right is to pick an approach suited to the context.

cousin_it 46 days ago [-]

A couple decades ago we had a world that mostly standardized on the LAMP stack. It was an architecture that solved everyone's webapp needs, switching projects was easy, life was good. Then SOA happened on the server side, JS monoliths happened on the client side, and here we are, worse off than when we started.

Hokusai 46 days ago [-]

> However, no one person owned the architecture or design. The experienced developers did drive this.

The lack of formality does not mean a lack of the role. If "experienced developers" are the ones doing the design. They are de-facto architects.

> No mentions of microservices, serverless architecture, application boundaries, event-driven architecture, and the lot. Some of these did come up during brainstormings. However, there was no need to reference them in the design documents themselves.

So, the teams were thinking in patterns to discuss and express themselves. But, then decided to hide that and do not show the reasoning in the documentation, for reasons. That makes the job of understanding the conclusions of the discussion harder for people from outside their group.

I am all for transparency. If your company has architects but calls them "experienced engineers". If you use patterns and then remove them from your documentation. Your company is going to lack the transparency to allow everybody to participate.

Everybody has seen this with on-line rants. People raises and falls by politics. When they are one of the "expert engineers" they talk about cool company and meritocracy. When politics make them fall, then there comes a rant and how

now the company "is not at it used to be".

I like to spend my time doing software engineering instead of politics and gaining upper management favor or fighting peers. Clear responsibilities help with that when a company is big enough. Like any system, a quantitative change - number of employees - may lead to a qualitative change that needs a change of approach. To try to run a 1000 people company like a 50 people startup is like trying to design in the same way a small server with a few queries per minute and a critical server with thousands of queries per second.

To each problem its own solution.

closeparen 46 days ago [-]

Central, top-down architecture is *extremely* political. You have to fight with bigwigs who don't know your problem domain and don't live in your codebase to make it reasonable, or even possible, to solve the business problems on your plate when they inevitably don't fit the 10,000 foot 5-year plan.

Pushing down architecture responsibilities into the hands of senior engineers with specific problems to solve / features to build eliminates that form of politics. They are not disguised architects, because designing the architecture is only phase of the project. They also have to live with the architecture. This is a *great* thing.

jbn 46 days ago [-]

architecture is political because an architecture (or a system) always ends up mirroring the organization that produced/operates it. A well-known fact in OB, and one reason why one wants to engineer (i.e. architect) the organization simultaneously with the system, otherwise you create silos or disjoint systems that can't talk to each other.

blub 46 days ago [-]

Interesting that this mentions Uber.

The way I understand it, in automotive companies have to prove that they carefully designed their systems based on the appropriate standards and they have to have the documentation to prove it in case there's a complaint from customers. A simple document won't cut it.

Then there's Uber, which is mentioned by the author and which killed a person through their gross negligence and then got away with it scot-free. I wonder how good their payment systems really are.

Can anyone from Uber comment whether sometimes they get a little bit less money or duplicate payments? Do transfers still work in conditions of poor visibility?

Maro 46 days ago [-]

I agree with the post, and this works at companies like Facebook and Uber, which have very high hiring bars, impact-oriented internal cultures and can afford to pay a bonus if you help the company be successful.

The interesting question is, what should all the other companies do? They cannot hire the best people, that aren't cash-rich, so cannot incentivize people. If you rank companies, what about the ones hiring the bottom 25%? They also want to write software to help run their business.

I'm not saying software architecture is the answer, but the [hidden] assumptions in the post break down and don't apply.

sid1138 40 days ago [-]

I have been coding for over 40 years and have designed many systems from small embedded systems the size of my little finger to huge, globe-spanning telecommunications systems. For all of those systems, the various teams' designs were basically as Gergely Orosz talked about. A few times I did some UML drawings, but the time to get them "right" was not worth it.

So, what is my role as an architect (I have been doing that role for about 15 years)? For various start-ups, I made sure that designs were compatible with each other (primarily through APIs and protocols). This was especially important at the startups since each team was running full speed to get a project done and often forgot the company's big picture.

For larger companies, I would be the interface between the customer and engineering to ensure both sides understood the requirements, expectations, and deliverables.

In all cases, I would also continue to code as well as document. An architect who doesn't code quickly loses sight of what is real and what is important in the ultimate expression of the design - the code itself.

vnorilo 46 days ago [-]

I think the underlying art is in striking the right balance between the specific and the general. In studying PLs for almost a decade now, I always ask PL designers I meet to try and distill their wisdom in one sentence. Andrew Sorensen said something along the lines of "Avoid all abstraction".

That shocked me, because my pet theory at the time was that abstraction was basically a force multiplier: good

abstraction makes everything exponentially better, while bad abstraction makes everything exponentially worse.

Nevertheless that quote stuck, and later I've started to appreciate it in terms of the YAGNI arguments.

Regarding abstraction, I've come to believe that a common anti-pattern is *deductive abstraction*: working down from a general pattern towards a specific implementation. This resembles what a sibling comment called "applying all sorts of design patterns from books to my code" (and not getting great results).

The opposite, *inductive abstraction*, is starting from a specific task and introducing gradual abstraction. Abstraction in lambda calculus is a beautiful example of the concept. Now make that thing a lambda term!

There's a Bret Victor classic that touches these themes [1].

[1]: http://worrydream.com/LadderOfAbstraction/

Michielvv 46 days ago [-]

I personally feel that software architecture isn't so much overrated as too much focused on abstract patterns instead of how to best solve common problems.

e.g. last year I needed to model an invoicing system, although this has been implemented hundredths thousands of times, there is very little generalized information on how to best do that so that it doesn't fall apart next year.

The areas that currently are better at this are mostly related to security and operations.

agentultra 46 days ago [-]

This is great... but one thing I think we _do_ need to change a bit is around *specification*. Knowing when to use blueprints as opposed to a sketch on the back of a napkin (or on a whiteboard). UML diagrams are not helpful to a lot of projects. But if you're dealing with concurrency or hard problems involving liveness or safety -- having a model one can verify and check goes a long way.

My litmus test for knowing when to bust out a TLA+ or Alloy model is: *what's the worst that could happen if we get this wrong?* and *are there behaviors we cannot allow the system to have*?

I find many developers, especially senior ones, develop a misplaced confidence in their knowledge and skill. And usually this is justified. We can generally get the majority of the system correct enough to be useful within a couple of iterations. Once that system is in production however it can become too difficult for even the most genius among us to find that one behavior of billions of possible ones that keeps triggering our SLOs.

That's because once we break down our behaviors into discrete steps and variables we find that many systems have behaviors that are 50 or more steps long involving several variables. The bad behavior that we didn't account for is there and we didn't know it existed until it went into production and started frustrating our customers.

I don't suggest we specify entire systems using formal methods like this, but for the important parts, I find it's worth avoiding the trouble than it is tolerating the risk and frustrating users.

Nice article though -- consensus building among peers is one of the least-taught skills and one of the most important.

vemv 46 days ago [-]

> No mentions of microservices, serverless architecture, application boundaries, event-driven architecture, and the lot.

I see why one would want to escape the mindless name-dropping that can be prevalent elsewhere, but at the same time, things have a name/fame for a reason.

There is pragmatic value in studying patterns (especially from good sources), and citing them in your internal documents so everyone is on the same page.

jacquesm 46 days ago [-]

Clear and simple design *is* optimal software architecture. Oversimplification and architecture madness are sub-optimal.

vemv 46 days ago [-]

> Oversimplification and architecture madness are sub-optimal.

Let me point out, you are essentially saying "bad things are bad" here.

jacquesm 46 days ago [-]

That doesn't seem to stop people from building systems in exactly those two ways. Yes, bad things are bad. Now stop doing bad things, it would make my professional life a lot easier.

Cobbled together Filemaker pro software running major factories; a million lines of code and 40 people to solve a problem that would take 3 people and 10% of the LOC if properly architected, and so on.

vemv 46 days ago [-]

Probably I agree with you.

I was mostly pointing out the weak writing: over<x> and <y> madness are *always* sub-optimal, for all values of x and y.

Richard_East 46 days ago [-]

This is why ex-startup-founder product managers are so in-demand if they ever go back as employees. Releasing your own software to the market, even with the help of a small team, demands a level of design finesse which is difficult to realise in a big corporation. The results in terms of customer feedback, market fit, product functionality are also easy to evaluate for hirers compared to NDAd or vague prior work experience.

Markets are not fair or uninfluenced by luck, and not every product or startup will succeed financially or provide the lifestyle the founder desires. But I've found that ex-founders make excellent picks as employees since they have the ability to work seamlessly across an organisation, particularly with software devs, and understand fundamentally which product features and developmental changes are worthwhile pursuing.

PeterStuer 45 days ago [-]

"I know of banks and automotive companies where developers are actively discouraged from making any architecture decisions without going up the chain" ... "So these architects create more formal documents, in hopes of making the system more clear, using much more of the tools the common literature describes."

Having consulted in such environments you have to remember that most of the large projects there are implicitly expected to fail, so CYA has become ingrained very deeply into the culture.

I stopped taking these jobs once it fully dawned on me that getting results in terms of on budget, on time, delivery while delighting users was never the prime objective, but using each project as a battleground in a perpetual internal jousting match for power grabs was.

cestith 45 days ago [-]

As a developer I find I'm paid very little for lines of code. I'm paid primarily for making decisions and for recording those decisions clearly.

Application source code is one way those decisions are recorded. Call graphs, data flow graphs, help text, application documentation, API documents, troubleshooting guides, monitoring code around production applications, tests, configuration and build management files, and commit messages are others.

Given that point of view, here's my advice.: Make the source and the other artifacts as simple as possible, but no simpler. Cross-reference where it helps most, but not excessively. Compose things from modular pieces rather than building monoliths when you can. Document the parts and the whole.

mpweiher 46 days ago [-]

Software Architecture just *is*, regardless of how you rate it.

Your software is going to have an architecture, whether you make conscious decisions about this architecture or not.

In the absence of a coded/codified architecture, your architecture is going to be implicit in the code. Currently there is very little choice in this, as we don't really have a useful way of putting the architecture in the code.

So your choice is either (a) implicit in the code or (b) implicit in the code + explicit in non-code documents. Neither of these are good choices, opinions differ about which is less bad, and those opinions also vary with the project and with experience.

Of course, the choice would go away if we architecture were a code artefact.

psychoslave 46 days ago [-]

The safest general characterization of the non-continental software architecture tradition is that it consists of a series of footnotes to the quote "Simplicity is the Ultimate Sophistication[1]"[2].

[1] https://quoteinvestigator.com/2015/04/02/simple/ [2] https://www.age-of-the-sage.org/philosophy/footnotes_plato.h... if you didn't get it already. ;)

barrkel 46 days ago [-]

A clear and simple and sufficient design is good architecture; an unclear, complex or insufficient design is bad architecture.

The article seems like a bit of a tautology to me.

cryptozeus 46 days ago [-]

Good article but some parts are outdated like who uses UML these days ? Saying you did not create diagrams using any

architecture tools is but obvious, no ?

SamuelAdams 46 days ago [-]

I just started grad school this fall (September 2019). My "systems analysis and design" course spends three weeks on UML, Data flow diagrams, and CASE tools.

This would have been a great course, 20 years ago. No sane business uses these tools today. The military might, but that's about it.

cryptozeus 46 days ago [-]

Yeh schools are way behind real world!

ccanassa 44 days ago [-]

As a Python developer, I always fell that I am talking to aliens when I have a conversation with Java developers. I've already built several large systems in Python/Django and nobody in our field talks about these patterns. We tend to follow more broad and philosophical principles like DRY, "we are all consenting adults" or the "Python Zen".

MertsA 46 days ago [-]

Overcomplicated designs is like the rocket equation for software engineering. Adding unnecessary abstractions, caching layers, abstract business logic, etc isn't an additive effect, it's a multiplicative one. This is how you take something that could have been a million dollar project and turn it into a billion dollar project that is completely broken on launch like healthcare.gov was.

zarkov99 46 days ago [-]

It changes. When you don't understand the problem, yes, simplicity is your best bet. Keep the code clean, clear, because you are going to have to change it a lot. As the understanding of the problem increases, and this does not always happen since sometimes the problem changes too quickly to be understood, then a good architecture can make an enormous difference.

l0b0 46 days ago [-]

Reading (OK, skimming) about this is a bit like a time machine. It's the kind of realization I expect people had in the early 2000s after fads like UML imploded massively. Is anyone taking architects like these seriously these days? Makes the article seem a bit like patting themselves on the back for not being bonkers.

eternalban 46 days ago [-]

Is the OP claiming that he/uber built systems that have no discernible architecture?

Confusing methodology with architecture is not helpful. Even the simplest system has 'architecture'. Now this can be arrived at via: happenstance methodology, or expression of internalized well known types, or via a more formal process.

tradichel 35 days ago [-]

Make sure you are preventing data breaches when you skimp on architecture via proper threat modeling and best practices. Some recent breaches result from poor architectural choices. @2ndsightlab

repler 46 days ago [-]

I feel like people get hung up on UML conventions and try to follow every process exactly and create every piece of documentation perfectly.

That was never the point. The point was to be able to communicate complex concepts in a way that people could understand and work with.

Bravo for having done that!

cuillevel3 46 days ago [-]

I remember reading somewhere, that it's wise to avoid naming your classes after design patterns. It only leads to discussions about technicalities.

Makes sense to apply this to bigger designs, discussing the concepts is more important than following the book by the letter.

jmull 46 days ago [-]

Hm... good software architecture is always clear and usually simple. So I think the title here is self-contradictory.

I do agree with usually using custom diagrams rather than, e.g., UML ones. Diagrams need to communicate. The problem with the formally defined ones is that they communicate what they are specified to communicate, not what you need to

communicate. As a result you end up with perhaps multiple diagrams to cover the concept, or you need to add text or an ill-fitting custom overlay or custom exceptions, etc. Also, people who don't create these diagrams regularly (a large part of your intended audience) have trouble remembering all the intricacies of the diagram language, so important concepts are made obscure rather than clear.

Anyway, you want to be focused on communicating certain concepts, not building a correct diagram.

ahaferburg 44 days ago [-]

I read one paragraph, and don't even know what it's about, and I get pestered to sign up for something with my email. Not cool.

oftenwrong 46 days ago [-]

Nobody sets out to build something that is over-engineered. Over-engineering happens in pursuit of a "clear and simple" solution.

dqvsra 41 days ago [-]

Just don't mix view with any business logic, keep them separated, and things will go right! ;)

foobar_ 46 days ago [-]

OOP is overrated. OOP programmers are the equivalent of bible-thumping evangelicals.

mrpickels 46 days ago [-]

I read the comments and see there are two types of engineers - conservatives and liberals, those who work for big corporations, draw UML diagrams with factories, bridges and facades and throwing arguments that because of some regulations or privacy policies your architecture can change and you need to be prepared for it. Those guys are right.

More liberal engineers are saying that keeping code simple is the key and to keep it simple you need to be smart and creative. Those guys are right as well.

Now back to reality, the conservative developers will always work on the code that was written by liberal developers because the latter deliver sh*t in time that works and carry the business on their shoulders, where the first makes it work in another scale.

Conclusion - there are different types of engineers and our responsibility is to accept that humans are not machines and somebody likes to CREATE vision and value, others like to manage huge system that are complex.

> Gibbon1 46 days ago [-]
>
> 30 years ago? I talked to an project manager that designed and built factories. He said there were three kinds of engineers and techs he hired. Design, Construction/Shakedown and Maintenance. Design requires being creative and methodical. Construction and Shakedown requires the ability to gleefully beat chaos into submission. And Maintenance is the methodical following of rules and procedures. He hired three different groups for these tasks because they would go insane or be overwhelmed if they were doing the wrong job for their temperament and skills.
>
> > rgoulter 46 days ago [-]
> >
> > I like the interpretation of 'conservative'/'liberal' as applied to engineering practices which Steve Yegge wrote (in a lost Google+ post): "acceptability of breaking code in production".
> >
> > 'Conservative' developers *really* wants to 'conserve' what's there. I feel that description suits both the 'draw UML diagrams with enterprise patterns' as well as 'loves dependent types' kinds of people.
> >
> > In this sense a liberal 'keep code simple' is more about things like "You Ain't Gonna Need It", and focussing on writing what code is needed now. (Since it doesn't matter if it needs to be broken later as requirements change).

known 46 days ago [-]

Unlike Simple Design, Redundancy is built into Software Architecture

pmf 46 days ago [-]

If I had more time, I would create a cleaner and simpler design.

> kmote00 45 days ago [-]
>
> "I apologize for the length of my letter. I'm afraid I did not have the time to write a shorter one." -Blaise Pascal (et. al.[1])
>
> [1] https://quoteinvestigator.com/2012/04/28/shorter-letter/

kissgyorgy 46 days ago [-]

One of the best design decision we come up with for a cluster configuration synchronization feature between nodes of our appliance born during an ad-hoc conversation between 3 senior engineers. We made the decision in like 2 hours. No long and unnecessary meetings with architects or week-long planning. It was made for a 1 million dollar deal and we finished in a couple of months. You could probably guess how the "architecture" looked like; we downloaded the config files from a web server running inside the cluster. That's it.