Intro to Empirical Software Engineering, What We Know We Don't Know : A Review

Muh. Riansyah
Faculty of Computer Science - University of Indonesia
Kampus UI, Depok 16424
muh.riansyah@ui.ac.id

***Abstract*—Almost in Software Engineering is belief.**
***Index terms*—** Belief, Empirical

1. Introduction

I actually want to start us off with an exercise. Imagine you're looking at a function and it's 40 lines long a pretty long function and you can break it down into say four ten line functions taking a big function and making some smaller functions so exercise one raise your hand if you think that the small functions will be easier to work with in the big functions coop I see most of you raise your hands cool down yeah on average okay so question number two it's flu season raise your hand if you think that a vaccine prevents diseases thank God I think that's everybody don't have to kick anyone out so last question raise your hand if you think it is more likely if you believe more strongly that small functions are easier then you believe that vaccines prevent diseases sorry oh is that raise your hand if you are more confident in your belief that small functions are good then that vaccines prevent diseases and I see a couple of very brave souls have raised their hands but almost everybody else who raised their hands for both times put it down why is that why can we believe two things competent two things but believe one more well for vaccines we have we've got medical studies we have historical data we have the elimination of smallpox we have clinical trials we just have so much but for small versus big functions we have a feeling in experience our opinions and we might have some logic like it's obvious that small functions are easier, but then again it also sounds obvious that injecting a virus into your veins isn't going to make you healthier right our logic can often be flawed now. This talk is not about small versus big functions not telling you to write big functions this talk is not called write big functions but I do want to underscore some really important point almost everything in software is a belief it is something we have experience about it is something we have opinions on but it's not something we have hard data in most cases we just don't know but we can find out we find out through empirical engineering or ESC for short that is the science of taking claims about software and dissecting them testing them observing them to find out what's really true and what just feels good my name is Hillel Wayne and I am here to talk to you about what this is how we do it and it's why it's so important and some of the things we've learned on the way but first some bookkeeping I'm going to be name-dropping about 35 different studies in this talk you don't have to write them down if you go to this link I have every single one online annotated and linked you can just go there and find them and look at them that way also every question being asked through the app. I will be answering if I don't get to it in this time I will be also uploading all the answers to that linked to so you'll be able to see every question people ask and the researched answers there and with that we're ready to begin so I think the first question you have to ask is why

1.1. Reason to do empirical research

why do we care about this why is it so important to be empirical and I see three reasons

1) Inefficient affect on GDP. The first one is the easy one I'm a developer I want to get better I want to know what works and what I just feels good by studying empirically. I can find that out but that's probably not that convincing to any of you right. I mean it's one thing to sort of have a more realistic argument of oh we should be better but quite different to actually do something the other reason extra density that's important is financial the tech industry makes up about 10% of the United States GDP that's 1.5 trillion dollars a year if we are 1% and that's really low estimate 1% inefficient that's the GDP of Iceland we are setting on fire every single year but that's it the large-scale that's not local to us and here's what I think is.
2) To protect ourselves. The most important the most subtle but most valuable reason to study the empiricism of software we do it to protect ourselves you see the most common most popular paradigm in software is charisma-driven-development. There are experts who are good at speaking we're good at writing and they tell us what we should be

doing maybe we do scrum because it works for our company maybe we do it because that's when everyone else is doing. Maybe we need a big data cluster to crunch our gigabytes of data. Maybe we're just following Google's lead. Maybe we do object oriented. Because people stand on commerce stages and say you must use S.O.L.I.D, but yes see empirical engineering just cuts all of that away it's what helps us distinguish what's fact from what just a Salesman is telling us empirical engineering is what tells us that well in 2014 MacBook Pro can crunch 50 gigabytes of data 100 times faster than a hundred server spar cluster it tells us that experts separated by hundreds of miles writing the same kinds of problem will make the same mistakes in the same places with I'm sorry is that like part is that is that something we can like sort of hide ok ok yeah so fundamentally ok yeah I'll just ignore for now no so fundamentally the reason we study ESC is to protect ourselves from the thought leaders and the Predators.

2. Example of Empirical Research

Below are list of example empirical research in Software Engineering. But doing so is very very hard it's complicated to study people are complicated we're studying people and people are more complex than atoms. I mean, take that original question I asked, our small functions easier than big functions. Well how do we even define better do we define it with a code metric, like cyclomatic complexity some people do and some people do studies that find that's the case but that just pushing the question back how do you know that cyclomatic complexity is better we don't instead we can find maybe a more goal-oriented result.

2.1. Effects of Clean Code on Understandability

We say better means well the code is easier to read or easier to modify or easier to debug that makes sense to everyone who and turns out that we have not that many studies on whether clean code is actually easier to read and I have looked i've looked pretty dang hard about this and i was able to find a couple of studies that actually studied this and they both said the same thing small functions are easier to read easier to modify and much much harder to debug so the evidence is mixed then again those are small sample sizes with lots of caveats and well look people are complicated so complicated we're doing such complicated things that some engineers think this is impossible we cannot study ourselves we cannot get hard data on what we do and in doing so we've made a mistake that a lot of engineers make a lot of smart people too. Because we don't know how to do something it can't be done but just as people are very smart they're also very clever and there are people who just as we put our lives into building complicated systems have put their lives into studying us.

3. Type of Research

So I want to give an example of what this looks like there are many different kinds of research we do. I'm going to break them down to three kinds quantitative, qualitative and code mining. In terms of what recently looked like this is one of my favorite papers. ever I know it looked small but it's.

3.1. Fixing Faults in C and Java Source Code: Abbreviated vs. Full-Word Identifier Names

well it's four-page forums pages to a sheet double-sided so it's actually about 40 pages in total this amazing paper is called fixing faults in C and Java source code abbreviated vers forward identifier names yeah not the most exciting title and it's a pretty small topic - what makes it interesting it's interesting because it shows perfectly how we do research and why it matters. let's take a simple example I have a I have a codebase and one of the variable names is employer_number okay descriptive name descriptive title great is it easier to debug this verse debug this emp_num easier to read the code easier to find faults what not so this is what they did they want to study which of these would be easier to work with and what they were doing is called a qualitative a quantitative study it's what we most commonly think of as science we have two groups a control group and a modified group control group we do not touch my group we make some tweak - then we have them both to a task and see which one does it better in this case the control group was debugging code and the other group was debugging code where all the full names were replaced with abbreviations then we can see which one does it better and then we can know which one's better now I see some of you looking skeptical I need me right there's lots of what we call confounding factors things that can explain our results without our fundamental premise being measured for example experience maybe one group did a better job because they all have 10 years more experience there might also be alternate explanations maybe this only matters when you're working with assembly and if you're working in Python the difference changes there are a lot of different confounding variables that can ruin an experiment if we aren't careful and if we know them we can design our experiment to control for them make sure they don't matter make sure they don't affect the results but you have to think of them in advance so they did they tried to figure out what could possibly give alternate explanations for whatever they were seeing and I'd actually can make this a quick demo quick exercise take 30 seconds think down write down now No maybe ask the person next to you really want to try to come up with possible confounding factors things that might explain the results besides just that give you all 30 seconds everybody back up eyes back on me great everybody hub has everybody to have some things.

How many people got like one thing. How many got like two to four? How many got five? six ? Let me read all the things that they found I even compared it to your list

experience level of the developer's education level of the developers programming language used size of the codebase studied density of bugs in the code base formatting of the code classification of bug time of day fatigue level prom domain sample size experience with the problem social media exposure how many of those did you miss yeah for the record they didn't find any difference so it turns out that while we know that descriptive names are really important there's no evidence that a full word is required when an abbreviation can fit in and this was only over a hundred people so it's not a hundred percent validated but it does show we can study something and get real results about what matters who here find that result surprising well so today it doesn't really make intuitive sense that a code base that emp_num is going to be no harder than employee number right see quantitative studies are science but they're not enough we also need qualitative studies this is the studies of people's experiences of their opinions of their ideas how they flow it is the exploration part of science it's how we get the ideas we want to test in the first place we need to explore so they explored they did what's called an ethnography they sat down and watched developers debug code in the real world with no sort of controls no suggestions just watch them and they saw that the two groups had different ways of debugging when you have the full word identifiers people tended to skim they used the name as an anchoring point lexicography or in themselves around the code and quickly jump between where they thought the bugs might be this worked as a debugging technique the people who had abbreviations though they more methodically went top-down understanding the context and the full flow of the code this also worked so changing the names did subtly change how people debug code but in both cases there was advantages and disadvantages they both worked qualitative studies let us actually know what is we're seeing what is interesting what we want to explore now both of these studies these clinical trials these of sonography x' are about people and people are tricky to study something that's easier to study that was code right it just sits there it's not going to change on us it's not going to be tired or sleep-deprived and code also has one bigger one big advantage it scales if I asked you average code base what's the unit testing coverage how would you find that out anybody want to make raise your hand anyone yep yeah but where do you get the UM code so the question becomes how do we I'll just go ahead I think you have the right down the track here so I mean we can try code that analyzes like our code base to find the unit testing coverage but how do you get enough samples how do we know the average well there's 100 million repositories on github we've just found them all crunch the data done problem solved and this does work.

3.2. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation

Here's another study that was done similar to this on code smells these were people who looked at 30 open-source projects each of which had been around for over a decade and studied what the anti-patterns were what they did wrong and where the defects in the code were based on what was changed their results were twofold one was that yes if there are code smells the code is more likely to be buggy in that area that's probably obvious to a lot of us the second thing that they found and this is a little bit more surprising is that fixing the anti-patterns did nothing to the bugs it turns out the two were correlated certain kinds of code led to more poorly designed low quality code and more bugs but they were independent fixing one didn't affect the other so this means that we can use code smells to isolate where we should be looking for bugs but we can't fix the bugs we just fixing the code smells we have to actually figure out what the bug itself is this is an example of how we can use code mining to very quickly and efficiently get insights into how things work but code mining does this very effective has traps of its own you see it's not a controlled environment we're looking at the field we're looking at the world and that makes for very complicated noisy data and we have to be very careful about that.

3.3. A Large Scale Study of Programming Languages and Code Quality in Github

Who's heard of this paper came out in 2017 a large-scale study of programming languages in code quality and github see a few of you because it was a pretty moment to study it was the first it only has about a hundred million lines of github and it was the first site that showed clear sophistical significance between different kinds of programming languages using commits and bug fixes and commits they found that functionally programming languages were safer than imperative languages static typed languages had fewer bugs and dynamic type languages manual memory languages were buggier then controlled garbage collected languages this came out a couple years ago and while the aspectus was small it was hailed as one of the best evidences for the port importance of programming language now there's one more part of this entire process that I have neglected to tell you we don't trust papers a paper is interesting it's insightful but it's not trustworthy people researchers make mistakes too in order to see a paper and actually benefit from it we have to do it's called the replication we have to get another group to do the same experiment and see if they get the same results that makes us more confident the study on naming was done replicated several times successfully as far as we can tell that's a pretty consistent this though just this year we tried to replicate this a group tried to analyze the same repositories and get the same results and in doing so they found a small problem you see this imagine you

have a commit that looks like this add in fix operator the other group was flagging it as a bug because it had the word fix in it and all about one third of the commits they studied were false positives once this was a con for every school difference went away they could not find any evidence that one language was better than any other language doesn't mean it's not true it just means you don't have evidence yet so code mining can be very effective and get us really deep insights but we have to be careful we have to make sure that what we're doing actually makes sense now something interesting about all of this that I've just been sharing you might be noticing a pattern I've named several things that we think are matter programming language how we name our things how we look at code smells all that stuff and nothing seemed to have a really strong effect write abbreviations totally fine code smells don't actually identify bugs it turns out that this talk is what we know we don't know and we know we don't know pretty much anything see software engineering is a very very young field some of the founders are still alive today it's a feel about systems and any system is going to be complicated any system is going to have some things that are obvious and false and some things that are insane and totally correct we don't know that doesn't stop us from programming we can still build pretty incredible stuff just as humans are complicated and clever we're really good at doing things in on certain situations so we don't know the answers and that's okay but just as we don't know the answers nobody else does either and that's what it comes back to so many people tell us here is how you must code they're the people who tell us you must use agile but they don't know they're just saying that they just do believe that though the people say agile is a waste of time they don't know that they just believe it they're just saying that and that's the key here we don't actually know anything nobody does and that means anybody who's certain about what isn't is it true about software is probably wrong and probably trying to sell you something we have to be methodical we have to understand the limits of our fields and learn how to push them we have to be careful and methodical and explore and short we need to understand now I've shared some things that don't work most things don't work but there are some things that we've studied that we are pretty sure make a difference we have done many experiments in many contexts and they've all found significant persistent positive or negative effects I'd like to share some of you with this to show you that there is some hope here.

3.4. Beyond Lines of Code: Do We Need More Complexity Metrics?

I'm going to also focus on the field that matters most to me I do what's called formal verification the study of making programs probably correct unfortunately that's not been cited by anyone so complete waste of time but one thing that has been pretty heavily studied is defect finding software defects how do we know where the bugs are in our code and a second how do you how do we prevent bugs in the first place that's what I'd like to talk about these two categories and what we've learned about them so first question how do we find bugs I've already shared one thing that works identifying code smells and seeing where they are helps us trace down where the bugs are. but that's probably not enough for most people we want an automated tool that helps us more carefully more accurately identify code. that led to an explosion of code measuring techniques. who here has heard of cyclomatic complexity. Who's here has heard of function points clean code most people these are techniques people try to use to measure the quality of software and maybe they work but in terms of finding where bugs are most likely to be in code there was one technique that works much better than all of them lines of code more lines more bugs now you might feel cheated by this because again we want an automated tool that we can point our code and find where the bugs are lines of code doesn't help us just saying there's a thousand lines probably bug somewhere in there just doesn't really do anything for us and as far as we can tell.

There just really isn't a way to just look at a code base and find where we can find the bugs so we don't look at the code base instead we mind the org chart.

3.5. The Influence of Organizational Structure On Software Quality

you might have heard of Conway's law code reflects the organization that produced it and it turns out that is empirically true in both positive ways and negative ways if you have code if you have a system in the organization a functioning organization that is cross-cutting and complicated the code for that system is going to probably be buggy this has been empirically verified if you have a lot of different people that touch a code base it is more likely to be buggy if you have a lot of different groups that touch a code base it is more likely to be buggy not in the rate of change but in the rate of types of change and this is a pretty consistent persistent effect so it's not necessarily a technical thing that we look at but the social thing our hierarchy is our VCS our git blame that help us identify where the bugs are going to be that's though in the general case in the specific case we know that in certain contexts it's easier to find bugs for example in a distributed system about nine out of every ten critical bugs that crash the entire distribute system are either uncaught exceptions you know the kind you find with the unit test or configuration errors so if you look at those two things you'll cut out maybe 90% of your crashes we also know from some surveys that about half of the worst bugs that take the longest to fix our requirement or design issues so if you just sit down and dry our decision table before you start coding you will probably save your company a few hundred thousand dollars but that's all in the finding of bugs ideally we don't want bugs in the first place right that's harder there's a lot of things we've studied on this and most of them seem like they work and seem to work in practice for us but when we put them to the test they just fall apart take oh no test-driven development now I'm gonna be very clear here testing is great everybody thinks testing is

great in fact it's so great that it's almost impossible to find Studies on it it's what's called a parachute study something so obvious nobody bothers to study it this term comes from medicine well there's no double-blind studies showing that parachute save lives so how do we know for the record I did spend about three days hunting down really old studies and they all agree that yes testing has an overwhelming benefit keep writing your tests the question though is does test and development work better who here knows the test room development is aa great most of you for the for the people don't it's a very tight cycle where you first write a failing test then write the code that passes the test then refactor it's really widely lauded a lot of people really love it I personally love it I do it all the time

3.6. Realizing quality improvement through test driven development

I recommend friends do it but does it actually make a difference well we have one said he's saying yes this came out in 2006 it was the first long-term study on test row development it found that it did reduce defects but also added about 20% more testing time to your system which made the effects kind of uncertain maybe it was the Chi GD maybe was just we spent more time testing we've done a lot of follow-up studies since then and as far as we can tell no there's really not a difference doesn't make a difference that much to quality either as far as we can tell test-driven development is no better or worse than any other disciplined controlled testing technique this is personally a huge bummer to me because I as I said loved doing it and it's kind of frustrating to know that this thing that I know helps me probably doesn't work but that's being what empirical means it means accepting the results accepting the data you if we don't see that even if we don't like the data and it turns out pretty much every other technique we've studied pair programming type systems etc don't really have that much effect either they feel like they help they probably don't except for one technical practice there is one technical practice that we've studied again and again and know for certain not just finds and removes bugs but is dramatically effective at doing so code review now there are some caveats here you can't review that much at a time you can review that many lines of code at a time but in those constraints the effect is absolutely enormous most of the rigorous studies they've seen on this say it finds about 60 to 80 percent of all the bugs in the code and even better than that that's the secondary effect it turns out that only one out of every four comments that basically block the code significant software defects is about functionality the other three are about code quality so very roughly for every bug it finds which is again about sixty to eighty percent of all of the bugs it spines about three situations where we can just make the code better more maintainable spread knowledge share knowledge code review is simply fantastic and no other technical practice comes close not pairing not TDD not testing in general not types not even like formal proofs to be honest these are still great things and I still recommend doing them but far and away TDD is the one technical practice we are absolutely 100% certain is effective nothing else comes close to code review at the end I said though technical practice for a reason we haven't studied software engineers as much as we really should have as I've made clear but we have to be knowledge workers in general we've studied them for a hundred years and we know without a doubt that there are three things that have a profound impact on the output of any possible knowledge worker any possible manual laborer anyone doing anything

3.7. Impact of a Night of Sleep Deprivation on Novice Developers Performance

Sleep deprivation stress levels and hours worked and these effects are absolutely enormous an unstressed well-rested oh not overworked team that is happiness job will produce orders of magnitude better code better output better systems than otherwise as just one example of one of the few cases we've studied software engineers in this context this was a study about what happens if you skip a night of sleep one night so you know your hackathon what you're doing at the end of the hackathon instead of at the beginning of the hackathon and if you skip one hour of sleep if you sleep one night of sleep for the first hour of coding after that just the first hour on simple tasks you're about half as productive also other studies show that if you miss about a week of two hours of sleep at night you are basically about as bad off as a person who skipped an entire night of sleep so chronic sleep deprivation can be just as bad also also it turns out that when you are sleep-deprived you can't tell your work is worse so if your team is sleep-deprived there is literally nothing you can do to make up for that no practice will make your code any better then they would make if they were well rested and well it's not just sleep I mentioned also time worked and hours worked right and stress one of my favorite studies that's come out recently is the games Sutra study on game developers they interviewed 700 game developers on 700 separate teams and we know that down to 270 different code bases total seven different games and among other things they found that when a team entered crunch mode that is over work to get a game done in time they produced games that were worse on every single metric reviewer scores profitability user satisfaction sales everything then the teams that simply cut scope or push their deadlines those groups were burning more time and money not to mention the health and safety of the developers on a worse outcome so the question you've probably heard that correctness you should be doing testing or review or pairing why haven't you heard about sleep why haven't you heard about stress levels for correctness a lot of reasons to be honest because these are a long-term subtle effects as opposed to short-term ones because they are diffuse and insidious because it's very hard to trace them back to their source and because it's not in our control things like stress and sleep are a product of things like deadlines scope creep bad managers bad company culture things that our organization level social not technical you see there are

some things that US engineers can do that will improve our code like code review but ultimately at its core software engineering is knowledge work it's about us putting our minds to the best use we can and I find that beautiful it really exalts what exists that humans can do but at the same time it means that anything that impairs our ability to think is going to cause much worse effects than anything else can so yes if we want higher quality we need to do our code review we need to be careful but if you really want high quality and high productivity well that can't be demanded of engineers it has to be enforced at the organization the change must come from the top now that is just software engineering just empirical engineering in software defects there's other fields we've studied - education human-computer interfaces performance all that stuff. I shared software defect because that's what matters most to me I don't know what matters most to you maybe it's something else all I can do is encourage you to look for yourself I encourage you that this is even worth looking at in the first place hopefully I've done that a little at least a little bit of that hopefully I've convinced you of the value here if so I'd like to end by talking about where we can get started what's the best introduction to both doing the research and finding it so there's two books I definitely strongly recommend the first is making software this is how I got into it the first half is about the practice of research how we do the research the pitfalls everything the second half is the things we've learned this book is absolutely fantastic I reread it once a year if any of you have a safari subscription it's free online there too they also have a site never work in theory org which has high quality open source research I'd also recommend reading that other book is a counterpoint the leprechauns of software engineering this is by person laurent bosavi who is skeptical the idea of empirical engineer obviously I disagree with him on that but what he does in this book is show how it is that people misinterpret research how claims turn into urban legends so it is a very good book for learning the methodology of evaluating research that's mostly about how we do research in terms of finding it that's a trickier problem who here has heard of the academia industrial complex basically goes like this almost all research is done by universities universities have their stuff published in scientific journals to read a scientific journal you have to pay either$30 per article or belong to an organization that pays$10,000 a year for access I'm guessing you most of you aren't in that so you can find the paper you can read the abstract you can't actually read the paper there are a few ways around this though if you have an ACM digital membership that's about a hundred dollars a year you can read all the memberships in their system if you go to this place called the archive a lot of scientists in protest and rebellion of the system upload their preprints there if you go to the scientists actual website they probably have their stuff hosted there too if you email the scientists they'll happily share it but by far the most efficient most effective and easiest way is to use SCI hub SC i - h ub if you put in a paper into the site it will just immediately give you the entire article no problems no questions asked the problem is I can't actually recommend this because it's incredibly fast and convenient and workable and has great UI but it's also a little bit illegal because you know copyright rulings so you really you're really supposed to you know if you want to be really moral about this you kind of have to pay the $30 per paper so I'm definitely don't go to that website definitely but it all you can and don't go to there and don't like follow them on Twitter just don't so in

4. Conclusion

Software Engineering very powerful very difficult. But it helps us distinguish what is correct from what we believe and what is useful from. What is either negative or uncertain it's great for I guess humility and actually improving and protecting ourselves now a couple of things to wrap up really quickly. First as mentioned you can go to the site and you can see all the reference. You can also read the references for yourself and it would recommend that everything I've shared has been colored by my opinions, how I see the world my own biases maybe when you read it you'll come up with something different maybe you'll think it's correct maybe you'll think it's garbage. I do recommend though checking for yourself because you should see for yourself what the research says and not just trust a person on the stage selling their consult shilling their consulting business speaking of shilling I'd like to end by clearly talking about what I do I work in a field called formal methods that is sort of the art and science of producing large-scale bug-free designs essentially software blueprints. I teach workshops and consult for companies clients have included Netflix Cigna protocol labs Skala tea medium math so far they seem pretty satisfied with me so probably a good sign if you're interested in this just either go to the site or come talk to me after I'll happily answer any questions on what I do and how it works and of course you're always welcome for the rest of the conference to ask me any questions you have about empirical engineering and with that my name is Helene and thank you for listening to my talk