JARED: Today's episode is brought to you by Naming Things, the revolutionary code refactoring technique that promises to solve one of the two hardest problems in computer science. Do your variables have names like data, temp, and thing2? If so, congratulations, you've successfully documented your confusion for future generations, including yourself next Monday.

Welcome to Dead Code. I'm Jared, and today we are talking, among other things, about naming things, building abstractions, writing code that operates at a higher level of abstraction, and writing code for humans and for machines. So, without further ado, let's get into it.

Adam, welcome to the podcast.

ADAM: Thank you very much. I'm happy to be here.

JARED: Could you introduce yourself to our audience?

ADAM: Sure, I'd be happy to. So, I'm Adam Tornhill, been a programmer for 30-plus years, still enjoying it, still writing a lot of code. I'm also the founder of CodeScene, a tool for managing and prioritizing technical debt. And then my background is a degree in engineering and a second degree in psychology, which is one of my big interests. So, that's pretty much me.

JARED: Cool. And I recently saw a talk from you titled "Writing Code" that I thought was really interesting, and there's a ton of interesting ideas in there. But I wanted to sort of start off, you mentioned having degrees in both engineering and psychology, like, what led you to exploring those two domains? They don't tend to be...I don't know anyone with a psychology degree that works as a software developer.

ADAM: It's probably a rare combination, but I kind of ended up there, more or less by mistake and curiosity. So, I started to write code in the 1980s on a Commodore 64, always loved that, so it was kind of natural to become an engineer. And I started to work as a software developer. And after a couple of years working for large product companies, I kind of noticed a pattern, and that pattern was that it was ridiculously hard to succeed with software.

And what I mean by that is that most of the projects I saw, they failed, and they failed miserably, like, you know, shipping late, the quality wasn't up to expectations, quite often we had to cut the scope. And I kind of started to think about why is it so hard to succeed with software? And I didn't thought that I would find the answers within the tech industry because then we would already be doing things differently.

So, instead, I decided to start to look at psychology because psychology is very much about how we think, how we reason, and how we solve problems, and also a little bit on how we collaborate with each other. So, in fact, all of that is, like, relevant skills. So, that's when I started to look into psychology.

And, originally, I'd only planned to do, like, an introductory course, but turns out that psychology is really fun. So, I ended up taking, you know, prolonging one year at a time. And after, suddenly, I found myself having spent six years studying psychology and standing there with the degree. So, now I thought I have to do something out of it.

JARED: So, in your talk, you mentioned that the human brain has so many cognitive bottlenecks that programming should be quite hard for us. Can you explain what those are and how they sort of manifested in day-to-day programming?

ADAM: Yeah, sure. So, there are tons of cognitive bottlenecks. And that’s kind of what surprised me when I started to study psychology. But to mention just a few, we all know that human memory is imperfect, right? We tend to forget things. And we might even remember things that never happened, right? So, human memory is imperfect.

But I think what really affects us as programmers is another type of memory called the working memory. So, working memory is like the mental workbench of the mind. We use working memory when we perceive, integrate, and manipulate information in our head. And the big problem with working memory is that it's a strictly limited cognitive resource. So, we can hold maybe three, maximum four things in our head at once, and reason effectively about them.

And when I talk about this stuff, I always ask the audience, like, think back to the last class you wrote. Did you have more than four instance variables? Basically means you're doomed, right? You're operating at the edge of your cognitive capabilities. And that's quite often the reason why things go wrong.

JARED: Yeah. So, you know, you mentioned these three to four things as the rough limit for working memory. But, you know, the software systems that we work on are extraordinarily complex sometimes. Like, what are the strategies you've discovered to help software developers deal with this sort of fundamental limitation?

ADAM: Yeah, if these limitations are the bad news, then the good news are that the human brain is very good at finding workarounds. But we need to help the brain a little bit. So, one way of doing that is by raising the abstraction level, and we do that using a technique called chunking.

So, chunking is something that comes from psychology. It basically means take lower-level information, and group multiple lower-level items into a higher-level concept. You can still only hold, like, three, four things in your head, but now each thing carries so much more information.

So, what that means in software is abstractions. So, at the lowest level, we have the humble function. A function is a great chunking mechanism, right? It introduces a name that can serve as an abstraction, as a reminder, as a handle to a piece of information. And then, of course, when we talk about software architecture, we have subsystems, services, layers, all these kind of abstraction mechanisms that help us fit more information into our head.

JARED: So, you found that developers can work significantly faster working on healthy code than complicated code. How have you measured this, and what specific code characteristics do you think have the biggest impact on this sort of productivity?

ADAM: Yeah, that has been a long journey. It's a metric called code health. It's something I've been working on for the past 10 years together with my team. And basically, what we wanted to have we wanted to have a code-level metric that could predict how hard a piece of code is to understand for a human. And it turns out that it's really hard to get a bunch of developers to agree upon what good code is or high quality code is.

JARED: [chuckles]

ADAM: I mean, we all have our own point of reference there, but it's quite easy to identify what makes code bad, so that we can probably agree on, right, like, ridiculously long functions, lots of duplicated logic spread all over the place, low cohesion, that type of stuff.

So, we built that metric by looking at not what makes code good, but what makes code bad. Basically, we looked for ugliness in code. So, we measure, like, 25 factors and then using algorithms, we can aggregate whatever we find on these 25 factors and categorize every single piece of code as either good or bad, depending on how hard it is to understand.

And the numbers I mentioned they come from a piece of research that we did a couple of years ago. It's called the Code Red: The Business Impact of Code Quality. And what we did was, simply, we looked at how does this code health metric correlate with stuff that means something to the business, like, faster and better?

So, we collected the data from real-world projects, looked at what are the task completion times, and how many defects do they push into production? And then we correlated that with code health, and we found a very strong correlation, meaning that if you have healthy code, then you can implement a feature 10 times faster than someone that has unhealthy code. And on a similar scale, with the healthy code, you will cut the risk of a defect with a factor of 15. So, it's a massive defect reduction. So, that's where it comes from.

So, when I talk about healthy code and beautiful code, it's not only a vanity metric. It's not only a vanity thing, right? It's something that's substantially impacting the business.

JARED: That's interesting. So, you talk about the sort of absence of ugliness rather than sort of the presence of specific, more positive qualities because those are hard to define. I'd love to hear more about how you arrived at that definition and why that's as useful as it obviously is when you talk about these, just how much more productive developers are working on this healthy code.

Yeah, I think, originally, I wrote about it in “Your Code as a Crime Scene” in 2014, or something like that. And I took a lot of inspiration from psychology there as well. In particular, there is a subfield of psychology called the psychology of attractiveness.

And I stumbled upon this absolutely fascinating study where a bunch of researchers they have taken photos of individuals, hundreds of individuals. And then they're using computer software to start to morph these photos together digitally. And what they found was that the more photos of individuals they morph together, the more attractive the end result, which is a bit surprising because when you start to morph multiple photos, the end result goes towards an average.

So, the takeaway was simply that beauty in humans is average, but it's average in the sense of mathematics, which is a very rare thing. So, that's when I started to think about code, that maybe the same holds true for code. Maybe beautiful code is average code with very, very few surprises, something that's super straightforward. So, that's where I kind of looked at beauty being this negative concept. Beauty is the absence of ugliness. So, it was kind of a long and bumpy road, but yeah, that's how I got at this.

JARED: Yeah, that really fits with sort of my experience. I've always said that if code looks unusual or surprising, that aspect should reflect something unusual or surprising about the problem being solved. If the code looks mundane, it should be doing something mundane. And if the code looks surprising, then I should be hopefully surprised about what it does because my attention is finite. And I would hope the code would guide me to the unusual aspects of it.

One thing you mentioned is how God classes can act as sort of developer magnets, where everyone ends up making...you end up with a lot of people making modifications to a single, or sometimes there'll be a couple of them, classes or objects in the system. How do teams identify these sort of architectural bottlenecks and address them before they become serious code health issues?

ADAM: Yeah. So, God classes, they are probably some of the worst code smells you can ever have. The thing with God classes is that they, I mean, detecting them is fairly straightforward. So, you basically need to look for a couple of patterns that you combine.

One of them is that you want to look at how cohesive that particular class is. Because if it turns out that that class is doing multiple different things, then they are at risk for becoming a God class. And what I mean by that is that you tend to stuff multiple business responsibility into the same class.

Then the second thing you want to look for is, are there any God functions inside them? That also tends to be a very typical pattern where you have these long, long functions. They're always long. And the reason they're long because, again, they are low in cohesion. They do multiple things. They tend to lack abstraction. So, there tends to be a lot of complex conditional logic controlling the flow for them. And they also tend to be, you know, since they are so low in cohesion, they typically need a lot of context to do their job. And you typically see that in having a very long argument list to the functions, right?

And as a consequence, the resulting class, the God class, they always become really, really long in terms of lines of code. And one of the problems I've seen is that not only is that piece of code really, really hard to understand for us as individuals, it's disastrous for a team. Because one of these workarounds that the brain uses, that we talked about earlier, is that we don't recall all the details of code normally, right? Very few people can do that. Instead, we have a simplified mental model in our head that let us reason around the code.

And with a God class, the consequence of having all this logic stuffed into the same place is that a God class tends to attract so many different developers, potentially working on different teams. And now we have tons of people changing the same code, but for different reasons, since they work on different features. And that makes it virtually impossible to maintain a stable mental model of what the code does.

And the obvious risk is that when I go into that code and want to fix it or change something, then I'm either doing that based on an outdated mental model of how I thought the code worked a week ago, but has since changed. Or I need to kind of relearn all the details and rebuild my mental model. So, I'd be in a constant onboarding mode, which is extremely expensive. So, quite often, there's a strong correlation between God classes, God functions, and defects.

So, these are really some of the worst code smells we can have. And we can, of course, talk a bit about how to fix them as well.

JARED: Yeah, what are the steps to addressing these once you've realized you have them in your system? Because I work on a lot of e-commerce systems, and, naturally, the order ends up being sort of this magnet for functionality because it is probably the most important object in the system. But it's also sort of the central point where everything else…it relates to all the most complex logic in the system, and everything sort of focuses in on it. And consistent efforts have to be made to keep it from becoming a God object any more than it actually has to be.

ADAM: Yeah. So, I think the first thing we have to realize is that we're not going to fix our God class or our God object in a few days of work. This is going to be a long, long-running process.

So, typically, the first thing I try to do is to identify what are the different responsibilities inside this class or module? And, usually, you can get some ideas already by looking at the function names, right? And what I try to do next is that before I even make a larger refactoring, I try to refactor for proximity. And what I mean by that is that I take functions and methods that are related to the same business responsibility, and I try to put them next to each other.

So, I kind of create small islands of responsibilities inside a potentially really large file. And once I have that, then I start to extract the responsibilities into separate classes, one by one. And I try to change as little as I can in the actual logic. I really just want to rip out these methods. And what I try to do is I put them into classes and try to select a name that really reflects the responsibility, because with a good name, I have now introduced a chunk, making it easier to reason about the code.

And this is something I do highly iteratively because, due to the nature of God classes, they will be coordination magnets. So, you have basically a whole team working in the same code. So, you want the iterations to be really, really short. Otherwise, you will never, ever be able to merge your beautiful refactoring branch back, right? Because our colleagues have been working on the main line in the meantime. So, that's typically how I go about it. Then, of course, the specifics, they can vary a lot. And in really, really severe cases, I might use different patterns. But basically, it's about divide and conquer.

And the nice thing is when you do that, if you use techniques like the hotspot X-ray analysis that I've talked about in other contexts, we basically look at, you know, inside this class, there could be potentially thousands of lines of code. Are there any specific functions that are being worked on more often than others? If yes, then those are, like, the first candidates to be extracted and modularized. And we use behavioral data like that. You will pretty soon find out that a lot of that code hasn't been touched in ages, so maybe you can leave it as is. And then you find it and find the more volatile parts, and those are the ones you really encapsulate.

So, I hope I managed to explain that process.

JARED: Yeah, that makes a lot of sense. And you mentioned sort of behavioral code analysis in there and how that looks at how teams interact with code over time. What's the most surprising pattern you've discovered about sort of team dynamics and codebases by analyzing that kind of version control data?

Yeah, I think one of the main benefits of behavioral code analysis is that it makes it possible for us to put numbers on things instead of making decisions based on gut feelings. And one pretty strong pattern I've seen is that the worse the code or the lower the cohesion in a class, the more developers that piece of code tends to attract. And that is not only due to the low cohesiveness of the class that has so many responsibilities.

It's also because with poor code quality, we tend to get more defects. So, there are probably a lot of quick fixes, different people, different heroes jumping in and fixing that stuff. And, again, it influences some mental models. But it also makes it hard to refactor the problem because there are so many developers working in parallel. So, when refactoring, we also have to consider that socio-technical context.

So, that is one of the patterns that you can confirm using version control data, using behavioral code analysis. And then there are, of course, other patterns that are super important to know about.

One of the most important is probably the hotspot pattern. And this is the one that surprised me the most, not that it exists, but in how strong it is. And that is that no matter what level you look at, if you look at the system level and look across all your files, or you look inside a God class and look at the functions, what you will see is that the majority of your development work is spent in a very, very small portion of the code, usually as little as 1 to 2%. And those 1 to 2% of the code tend to be responsible for 20, sometimes even more than 50% of your work. So, what that means is that using behavioral code analysis, we get a window through which we can view quality and choose to prioritize remediations where they have the largest impact.

JARED: Yeah, that makes sense. We spend so much time looking at our codebases, just in their current state, whatever that happens to be, that it can be hard without dedicated tooling to have really any idea how things are changing over time in any meaningful way. You mentioned in the talk the Sapir-Whorf hypothesis and how that influences developers' thinking. And you bring up some fairly unusual languages. Should developers learn radically different programming languages to sort of expand their problem-solving capabilities?

ADAM: Yes, I think that's one of the most important things we can do. So, Sapir-Whorf, the hypothesis, for those of you who haven't heard about it, it's the idea that the language we speak influences the thoughts we can think. And the same holds very, very true in programming. And, in particular, learning multiple languages is not only about learning multiple different uses of syntax, right? It's more about learning different thought processes. So, to be impactful, we should try to learn languages from different paradigms.

So, if you're already programming object-oriented languages like Java, or C#, or whatever, then it definitely pays off to learn something functional. And I know that, you know, functional programming we can do those paradigms inside Java and C# to a certain extent. But using a pure functional programming language really forces you to think differently.

And there are also some really fascinating languages that are far from mainstream today, like the whole family of array languages, where you find languages like APL and J. And learning these, they really, really turn your brain inside out. And it makes you think differently about problems. So, when you come back to your language of choice, (be it Java or C#) you will find that you look at problems from a different angle. And it gives you a completely new mental toolset for addressing problems in a more effective way. So, that's definitely a recommendation from me.

JARED: Yeah, recently talked about a few different languages on the podcast, like OCaml is one that I personally find really fun to work in and very different. I'm a Ruby programmer, so it's quite different. But we've also talked about some more unusual ones, like Forth recently, which is another language that really forces you to stretch how you think about solving problems with software.

One specific topic that came up in your talk was the use of null and its sort of relationship with special cases. Tell me, what are your thoughts on null? What do we do with null?

ADAM: Oh, null. Null is probably the thing that annoys me the most about programming. I mean, it has been labeled a billion-dollar mistake, and it really, really is. Because the problem with null is that the moment we set a reference or a pointer to null, we are basically introducing a lie into our codebase, right? Because now we can no longer use our APIs. So, an example of the use is to have a string, and the string has an API. There are certain operations I can do in a string. But if I have a reference to a string and I put that reference to null, then all bets are off. I can no longer use the API.

And what's worse is that my decision to introduce null tends to ripple through the codebase, and now all the clients, all the surrounding code needs to be aware of that decision and introduce these null checks, which is quite ugly because it adds an extra burden, an extra cognitive burden on us, right? So, that's one thing that will unnecessarily fill up our working memory.

And the thing is that when you start to learn different languages, different paradigms, you see that the null problem has been solved in different technologies, but it kind of never really made it into mainstream. So, one of my favorite examples there, and a language I took a lot of inspiration from, is APL. So, APL has this fascinating concept that the built-in basic type is an array.

So, all the functions in APL, you do them on arrays, never on individual numbers or individual strings. It's always a collection of things. And once you start to think about null, basically, the problem we try to solve with null is that, yeah, we may either have something. We might have an object, or we don't have anyone. Once you start to think about it and start to express that as being either an empty sequence to represent the absence of something, or a sequence filled with one value to represent a presence, then we'll find that you can design away the need for null in your application. Just treat the absence of something as an empty sequence. And you can still apply all your API functions. You can still use streams and Fluent APIs. So, that's one of the simplification techniques with the biggest impact that I've started to explore over the past decade.

JARED: Now, how can developers working in, say, object-oriented or other languages sort of apply that? Because I can put all of my values in arrays in Ruby and run map over them instead of calling the APIs, but surely, that's not going to produce better software than I'm writing currently.

ADAM: Yeah. So, what I think is really important is that if you make it a habit to always return sequences, and, of course, I try to avoid returning raw sequences. So, I never return something like a built-in list, or a built-in array, or whatever data type you have. I want to turn these into domain objects, too. And, of course, the API for your domain object should encapsulate the operation you want to do. And then it's up to you internally to encapsulate the knowledge that, yeah, this is a sequence, so let's just iterate over it and invoke whatever elements we have there. So, it's a lot about information hiding, too. We always want to program at the level of our domain rather than using all these built-in types. I think that makes a big difference, too, in terms of abstractions.

JARED: Yeah. And I think you use the word domain in there, domain-driven design. One of the sort of core concepts in there is objects that represent collections. And at least in the software systems I've worked on, that particular pattern is underused. There's a lot of passing, in the case of Ruby, arrays around when perhaps we could build up abstractions around those that better reflect our domain, though our arrays are...they have so many useful methods on them. It's hard to get away from.

ADAM: They do. They do. I mean, I absolutely agree. I also find that it's a very underused pattern. But the extra level of encapsulation and information hiding that we get is so much worth it because it becomes really, really hard to make a large sweeping change if we pass around the raw data types, right?

JARED: Absolutely. So, domain-specific languages, something that we love in the Ruby world, absolutely can make programming accessible to non-programmers as you showed with an example project in your talk. What are the key principles around designing a DSL that actually reduces the cognitive load rather than sort of adding another layer of complexity that the programmer has to learn?

ADAM: Yeah, we are indeed touching upon it. So, the key here is to get our solution domain, which in our case means the code we write, to get that as close as possible to the problem domain, right? Which is, like, our requirements, the very concepts we are writing our software for. And the closer we can align these two, the easier it is to iterate between these two worlds. And this is really important because human problem-solving is inherently iterative. So, typically, we start out with an imperfect idea on what we want to build. We build it, and we observe the outcome. We refine our mental models, and we express that refinement in our next iteration of the code. And, of course, that iteration is easier if these two models are closely aligned.

So, with domain-specific languages, we basically take a general programming language like, say, Ruby, Python, Clojure, and we build it up to the level of the problem we're working in. And that's a technique that's so much more natural in certain languages compared to others.

JARED: I guess what I'm curious about is, are there specific patterns in DSLs that you've seen work more effectively than others? I think, you know, in Ruby, we always point to...people often point to RSpec as this example of, it reads very well. The resulting tests can be understood by people that have very little experience with it, at least at a high level. But ultimately, sort of the under-the-hood details come to matter. And people find that the DSL is just yet another thing that they have to learn when they'd much rather just be writing raw asserts and testing things at a lower level, and it's a constant argument. Is there something that makes some DSLs better than others?

ADAM: I mean, of course, I might be biased from my experience. I won't claim that I have a general answer. But one thing I've observed is that if you have an easy way of building a DSL, and that's particularly true if your language already supports embedding a DSL, extending its syntax, like Clojure does, for example, or Common Lisp, then what I found is that you can write quite some narrow DSLs.

So, it doesn't have to be this big investment where you spend weeks developing a new language. It could be something that you do in 30 minutes. And that is really, really powerful. So, some of the best DSLs I have found have been for quite narrow domains, where it was just, like, the natural solution. And maintaining that kind of code is pretty straightforward because as soon as I understand the domain, I will immediately see in the code what I have to change.

So, definitely, I think it's about affordability, that your programming language has to support it. We shouldn't have to write a parser in order to build a DSL. It should be something that we can just extend the existing syntax of a language together.

JARED: So, CodeScene can help identify when you've got these bottlenecks, these modules, or whatever they are, that you've got a lot of developers making significant changes to. And we talked about why those are issues. How do you recommend organizations go about restructuring both their code and maybe even their teams or the organization itself to deal with those kind of coordination bottlenecks?

ADAM: So, the first thing I always recommend is that before you even start to think about refactoring or paying down tech debt or whatever, then put a quality bar on what's already there. Start by ensuring that your code health doesn't deteriorate further. So, that's step number one. If you do that, then you're already better than 80% of all organizations out there. So, it's a really good first step, and it's harder than it sounds.

The next thing I do is that I try to...I always recommend that prioritize based on how you work with the code. So, look for these development hotspots, and sometimes you identify those hotspots, and it turns out that that's healthy code. When that happens, you're in a very good spot. It's exceedingly rare. Quite often, these hotspots tend to be in worse shape than the rest of the code. But, again, the good news is that they are a very small fraction of the code, so maybe just a few percentages. So, those would be my initial refactoring targets.

And what I try to do next is that I always want to understand, okay, how are the organizational dynamics here? Are there multiple teams contributing to these hotspots? In that case, then I might have to be much more careful with refactoring and doing it in much smaller steps. If there is a better alignment between the code and team, then I can be a little bit more aggressive because just the people on that team need to know about it.

And I always try to identify the root cause, and then my tip is to start looking for that root cause in the technical issues. Because quite often, a lot of the organizational problems we see they are symptoms of a technical root cause. Quite often, you have low cohesion. You have these

God functions. You have tight dependencies that shouldn't be there. So, addressing these technical issues they kind of liberate your organization. And suddenly, you can find that now people on different teams they can start to work in a more isolated way because you have decoupled the code or modularized it at a level where the code aligns with the tasks you work on.

JARED: That makes sense. That makes sense. So, looking forward, as codebases grow ever larger, more complex, what do you want developers to remember most about writing code for humans rather than machines?

ADAM: So, let me flip that because there's some recent evidence that suggests that code that's easy for humans to understand also tends to be code that machines prefer. So, it turns out that an AI can actually perform better on healthy code than unhealthy code. So, I think there's a double win here.

So, I think my number one recommendation, if I could just give you one thing, is to really, really put an extra emphasis on naming things. It's hard; we all know that, but the better you are at naming concepts, the easier it is to reason about the code. Good names make wonders, not only for your code, but also for your LLM.

JARED: That's a great tip. Where can people go to follow the work you're doing online?

ADAM: So, people can follow me on LinkedIn. I'm relatively active on LinkedIn. I also blog occasionally at codescene.com, and more rarely at adamtornhill.com.

JARED: Cool, thanks for coming on the podcast.

ADAM: Yeah, thank you very much for having me, and thanks for the conversation.

JARED: It was really interesting to see Adam tie together just a whole bunch of different, really, I think, important ideas about what makes good code, the nature of God objects and how they attract functionality, and how they snowball, and the idea of hotspots within a codebase. It sounds like the work he's doing and the research that he's done has really tied together a whole bunch of sort of different ideas that I've encountered in my career to build a really nice way of looking at sort of code health, which I find super interesting.

This episode has been produced and edited by Mandy Moore.

Now go delete some…