LIVE: Uncle Bob on Software Fundamentals in the Age of AI

Matt Pocock | 56:39 | transcript saved 22 Aug 2026

https://www.youtube.com/watch?v=zcLPGC-tvgk

Introduction

Today I have prepared something special for you. I have a guest — someone I've wanted to talk to for a long time. Someone who has been as passionate about software as I have, and has been a developer even longer, and who is currently actively making a name for himself in the field of agent technologies. We have Uncle Bob Martin, live, in his robe and ready to fight.

Uncle Bob, hello. Congratulations.

Hello, and thank you. Glad to be here.

This is great. Very glad to see you.

Very glad to see you.

The Bathrobe

For those who don't know the history of the robe, it's probably worth explaining — what's going on there, and why has the bathrobe become an important part of your look?

This happened, it seems, two years ago. I was on the porch in my bathrobe. It was six in the morning, and I started thinking about how terrible SQL was, because of all these SQL injections, and how pointless it was to use a text-based language to access a database, for security reasons. I was in a bathrobe, took out my phone, and expressed everything I was thinking — and that's how the morning "bathrobe" speech was born. It turned out to be popular, so I repeated it a few more times.

So is this the mood you came with today? Is this the same Uncle Bob, early in the morning?

I haven't had my coffee yet, so just don't touch me. Six in the morning, thinking about SQL — that was enough for me. I'm done with that. It's actually ten in the morning now, the robe has been removed, I'm fine, I'm drinking my first Diet Coke, so everything is good. Very good.

Who Is Uncle Bob

Okay, well, now there's a polo shirt. What is Uncle Bob's story? Most of my audience are developers, but there are also people here who aren't involved in development. How do you introduce Uncle Bob, this phenomenon, especially to those who are not developers?

Phenomenon — I don't know about that. I am a programmer. I've been programming for a very long time, over half a century at this point. My first program was in 1964. I was 12 years old. It was a small model computer that my mother gave me for my 12th birthday, and I programmed it by putting little wires on the pins. It was essentially a three-bit finite state machine, but at 12 years old it fascinated me immensely.

So you were 12 years old — how did we get to the point where, 50-some years later, Uncle Bob is doing his thing?

Let me think about it. I just started learning everything I could about programming. My father bought me

a book about Fortran, a book about COBOL, a book about PL/I. I read them all. I didn't have machines to run anything on, so I wrote programs on paper and ran them in my head. I got a job writing a little code at 16 — that was temporary — and then I got a real job at 18, and I've been a programmer ever since.

Clean Code

So you've been in the trenches for a long time, and I think you've written a pretty important book.

A few. One of them, I think, was important. I wrote a few more, but yeah, one of them seems to have been successful. It was a good book.

Clean Code, if I remember correctly?

Yes, yes. Here's the second edition.

This is probably the one I hear quoted most often when people talk about good books for software development. It's incredibly popular, incredibly influential.

Discovering AI Agents

That's why I'm interested in talking to you today — you've been very influential in the pre-AI era, and you worked in that era for almost as long as it's actually existed. How have things changed for you, Uncle Bob, now that AI exists and has become a reality?

It kind of caught me off guard around December of last year, around Christmas time. I was just experimenting — I'd already been playing with ChatGPT, a little bit with Grok, a little bit with this, a little bit with that — and I wasn't particularly impressed. Then I had a period where I thought, "Maybe these things are a little more interesting than I thought," and I got an agent.

The first agent I got seems to have been an early version of Grok. I just asked it to write me some code, and it did it pretty badly, but it wrote the code. I was in the middle of a project at the time, so I thought maybe this thing could help me, and I started involving it in the work. But I kept having to redo everything after it, because it kept making a mess — always left "dog tracks" behind. I thought: it's interesting because it's fast, but at the same time it's annoying because it slows me down.

And then I started thinking: wait — because it's fast, it can do things I can't.

CRAP and Mutation Testing

Back at the very beginning of the 2000s, a couple of innovations appeared that interested me. I thought, "Oh, these are good ideas," but they were completely impractical.

One of them was called CRAP — that's an abbreviation — and it was a way to assess code quality using tests. You run a code coverage analysis, measure the cyclomatic complexity of each function, mix those two metrics together in a formula, and out comes a score showing how "crappy" your function is. Back in the early 2000s I thought this was a great idea. I applied it to a large project I was working on, and yes, there were a lot of "crappy" functions — but it took forever to go through each one, fix it, and rewrite the

tests, even though everything worked anyway. So I decided I couldn't afford to waste my time on it. It was interesting, but I put it aside.

Another innovation was mutation testing. That also caught my attention — a little program goes through your source code and changes minus signs to plus signs, "less than" to "greater than," "equals" to "not equal," and so on. For each change, it runs the entire test suite, expecting a failure because something significant was changed. If the test suite doesn't fail, that's a "surviving mutant," and it needs to be destroyed. I ran it on that same project around 2000. I had to run it all night, because the test suite took four minutes to run and I ran it several hundred times. It found a bunch of surviving mutants that I was able to fix. But again, that was impractical — I couldn't include it in the standard project build script.

Putting Agents to Work

So there I was, sitting last December — or maybe it was already January — thinking: "Wait a minute. These agents are fast, and they don't care how boring the job is. They'll do whatever I tell them. So why not run CRAP on everything they just wrote?" It would flag the crappy code, and then clean it up. I watched it do that, and I thought, well, that's pretty cool.

And why not also run mutation testing? It would perform mutation testing — maybe it would take 30 minutes instead of an overnight run — and then it would plug all the holes and make sure everything was covered with tests. I thought this could be a way to clean up all the mess these things leave behind, and maybe a good way to actually make it practical.

So I kept going down this path, adding more tools and working with these agents, until I got them to the point where they were doing pretty well. Today, my principle is this: I put agents to work. I let them run these tools, and I try really hard to make sure I don't have to look at the code at all — I can trust what they do. And then I do other things to make sure the code is still decent. I look at the CRAP scores and make sure they're low, I do a selective code review from time to time, and I run a bunch of other tests. But overall, that's my goal: if these things are fast — and they are fast — and if I can get them to do the job well, then I won't impose my own slowness on them. My contribution is a code review or inspection at some level of detail. They're fast with code. I'm slow with code.

Environment vs. Code

So I'll let them work on the code, and I'll deal with the environment myself to make sure everything is okay. So far, so good. Your goal is to start moving away from the code itself, from manual testing, to build a framework around it so that you don't have to interact with it yourself, and so that the agent is as tightly constrained as possible so that it can't make a mistake.

Why Does Bad Code Matter?

I think there's an assumption here I want to address first, which is that the phrase "dog poop" doesn't sound very good with a British accent. So what, and why is dog poop bad? Why is bad code bad? Why should we care about this? Since these things are so fast and can move so fast, why are we worried?

LIVE: Uncle Bob on Software Fundamentals in the Age of AI

Matt Pocock | 56:39 | transcript saved 22 Aug 2026
https://www.youtube.com/watch?v=zcLPGC-tvgk

Why do we bother with all this scaffolding? Can't we just keep going until the bugs are gone?

When Agents Drown in Their Own Mess

One of the things I noticed very early on, sometime in December, was that I was working with this little agent, Grok. I would make him do something and he'd leave a bunch of mess behind, and instead of cleaning it up, I'd make him do the next thing and he'd leave even more mess, and then I'd make him do the next thing. I noticed he was slowing down, and I noticed he was having difficulty. He'd get into a mode where he'd change one thing but inadvertently break another, then he'd have to fix that but inadvertently break something else. It started going in circles, and I thought: okay, these agents may be fast and relatively intelligent, but they're just as susceptible to dirty code as humans.

Maybe not to the same degree — maybe there's a difference in threshold — but the threshold still exists. The code can become so messy that agents can no longer handle it, and then they simply start spinning in place, making the mess even worse, and become unable to work. They even gave up on me. One day the agent basically said, "I just can't work with this anymore." I'm paraphrasing — he didn't use those exact words, but it was clear that's what was happening.

From Rules to Deterministic Checks

So your answer to this is: how do I remove this junk? I think when most people encounter this, they start loading the agent with instructions. They say, "okay, I'm going to feed information into the agent that does the implementation," and they pile up a CLAUDE.md or an AGENTS.md, and every time they see something bad, they add a rule. Whereas what you're doing is a deterministic mechanism — you're using automated checks. A lot of other people are, in my terminology, trying to control it, trying to direct it. So do you direct them at all, or how does that work in your approach?

That's what I did at first. My early requests were things like: here's how to do test-driven development, here's how to write clean code, here's what your code should look like, you have to follow all these rules. You end up with a five- to ten-page document describing all the best practices for the code — I could probably upload a whole book in there. But I noticed that agents, models, whatever you want to call them, treat these rules in a Pirates of the Caribbean style — they're more like guidelines than actual rules.

Can I give you credit? That's the exact metaphor that came to mind for me. Good, we're on the same page. They definitely relax the requirements, and there are technical reasons for that. I did a little research into why models behave this way, and it turns out there's a phenomenon known as "lost in the middle." When the context window inside a model fills up, the information at the very beginning and the very end matters more than what's in the middle, for technical reasons — the agent will effectively ignore what's in the middle. And whatever you say at the very beginning ends up in the middle once the context grows long enough. Maybe the first three sentences will still get prioritized, but sentence 50, sentence 80 — they disappear. They're just somewhere in the middle, and the poor agent is trying to deal with this huge, growing context, trying to extract the important bits, and the stuff in the middle just vanishes.

Deterministic tools don't disappear like that. I think the main thing when working with agents — and it's

very difficult to do — is to reduce the initial query to the absolute minimum, so that as much information as possible stays in the priority zone, and then apply the deterministic tools on top.

The "Smart Zone" and the "Stupid Zone"

I totally agree. I call it the "smart zone" and the "stupid zone" of the context window. That's not my term, that's Dex Hardy's term — I stole it, and it's very accurate. Especially in the early part of the context window, say the first 150,000 tokens, the model is pretty sharp. But as you go further, the attention ties in the transformer get really strained, they blur. It's like everyone's screaming, every token yelling in a crowded room, and the room keeps getting smaller. You can't hear the signal over the noise.

That totally resonates with me. So you've given up on "control" and gone back to some of the old automated testing methods. It sounds like, because these checks don't fall into the context window the way instructions do, you can just layer them on, over and over. If you're using a language with strong types and tests, what does that look like to you? Is there ever too much automated testing?

Deterministic Tools in a Loop

That's one of the things I'm working really hard to figure out. Obviously there has to be a point where it's too much — eventually you'll slow the agents down to the point where they're slower than a human, and at that point you've lost the game. Why do that? But as long as you can maintain a higher performance level than a human, you're still ahead. From what I've seen so far, I can get a margin of, say, a factor of two, three, or four — they're still moving pretty fast, although I am slowing them down a lot right now.

When you use these deterministic tools, you put them in a loop. A lot of people like to talk about loops these days. You put the agent in a loop and say: you have to change the code until this tool says it's good. And now the agent goes in circles, over and over — okay, I have to do this, I have to do that, I have to add more tests here, I have to reduce the cyclomatic complexity, I have to separate these functions, I have to do all this work. And it takes a lot of time to get the code in line. So you sacrifice speed for higher quality, and at some point that has to give way — but I haven't found the end point of that yet.

Right now I'm trying to get multiple agents interacting, passing work off to each other: one does the work, the next one checks it, someone else tests it, someone else improves it, and so on. There's an insane amount of communication overhead, and yet it's still much faster than a human could do it.

Multi-Agent Systems

Let's talk about that — multi-agent systems, because I find them incredibly exciting. I've always had a certain suspicion — and I think we may disagree here — of people who say, "oh, I got rid of all my staff, now I have a hundred agents, each one has a role, they all communicate with each other, they have their own email inboxes," and all that. That's never worked for me. But okay, I'll make an exception for the implementation and validation phases.

I think the advantage here isn't having two highly specialized agents so much as having one agent do the implementation — like a red-green-refactor approach: the developer just has to write a test and make it

pass, he doesn't have to make the code perfect. Then the reviewer comes in. He doesn't have to do any research, because he's already got the diffs from the developer — he knows exactly what he's checking. And then you—

Multi-Agent Development: Advantages and Disadvantages

Can add a lot more instructions, because the task becomes a lot less constrained. So I'm really interested in your thoughts on what the developer does, what the reviewer does, and what these other refinement agents do.

I haven't experimented with that yet — tell me more about that.

There are two advantages to using multiple agents in this way. The first is that you can run them in parallel — you can have, say, three programmers working at the same time, and my little laptop can support a lot more than three. The other advantage is that when you focus the agents on one specific task, you keep the context window under control. The problem of "lost in the middle" becomes a lot less significant. So you can add a little more — not a lot, but a little more — rules at the beginning, and they'll follow them better. You can also set up a system where agents are born, they do their jobs, and they die, so that the next one starts with a clean context.

Those are the advantages. The disadvantage is that the startup time is long — the agent takes 10 or 15 seconds just to start up, and then it has to figure out all its context again. So there's this specific startup cost.

My Agent Pipeline

I try to focus the task as much as possible, so I run a set of specialists.

The job of the first specialist is to take a human-written document and turn it into a Gherkin spec and a QA procedure. Gherkin is the "given, when, then" construct — a high-level acceptance test. The QA procedure is essentially a system test: you run the system through the interface according to the procedure. I have them write it from a human perspective — "you are a human, you work with this system through the interface, you have to prove that the system works" — and that produces the two documents: the Gherkin document and the QA document.

Then they go to the programmer. The programmer's job is to write unit tests and code that implements the story and also makes the Gherkin scenarios pass.

Once that works, it goes to the cleaner, whose job is to do a quality review and a general code review — clean up all the chaos the implementer created, because at that point there's usually a lot of mess.

Then I send it to the hardener. The hardener does mutation testing, and he's absolutely ruthless — he'll mutate the code, demand 100% coverage, check every equal sign and every "less than." It takes quite a while.

At the end it goes to the QA agent, who takes the written QA document and turns it into an executable script that manipulates the system and produces a deterministic result.

If you can get through all of that, you'll have a pretty solid working program. I've had big success with this approach. A task that one agent could do in five minutes with questionable results takes about an hour to go through this whole pipeline — which is still profitable, because a human would spend half a day on it. So I get maybe four or five times the productivity, and much higher quality than any human could provide. And it's not just that you're trading productivity now for more later — it's an investment in your own codebase.

Context Window Trajectory

What I found really interesting — something I've been thinking about lately — is that you're not just manipulating the context window, there's also the idea of context window *trajectory*. Within a session, if you direct an agent a certain way, everything that follows in that same context window will continue to follow that trajectory. If you tell it, "okay, maybe we should test the interface here," it's going to test the interface again every time, no matter how many changes you ask it to make afterward. The only way to clear the trajectory is to clear the context window. So an implementer agent that's just trying to make everything work has a much looser trajectory than one that's trying to hit 100% test coverage. Does that align with your mental model?

Yeah, definitely. There's a pretty well-known effect in these models — it happens to people who aren't even programmers. You're talking to the model about, say, what the best coffee is and how to brew it properly, having a nice conversation with the agent about it. Then someone walks by and mentions the latest soap opera they saw on TV, and that gets into the context window. You didn't put it there — the person walking by did. But from that point on, all the coffee references start turning into soap opera references. The model can't tell the difference. So your idea of a trajectory is spot-on: as long as you can keep the model's direction consistent — everything in the context window pointing the same way — it won't produce the crazy hallucinations people often run into.

Or even just plain inconsistency. Hallucinations are always a danger — whether it's hallucinations or context you've provided, it's a kind of perceived risk, and your approach of running everything through these different filtering agents is a great way to eliminate that.

Codebase Structure and Architecture

You talked a lot about the implementation phase, and we can come back to that. I'm also curious about the planning you do up front — specifically, how you think about the internal structure of your codebases. Having a good set of tests is good — mutation testing and all that is good — but if you have poorly designed APIs or poorly formed modules, how does that interact with all these automated checks, and how much do you even think about it?

Up until about last month I was doing it manually. I'd ask agents to build me something, and then I'd interrogate the agents: what's the structure here, how does this module interact with that module, what are these modules anyway, how do they communicate with each other? I'd ask these questions and then be scared to death, because the answers were often terrible. So I'd design the structure of the modules myself and tell the agent, "this is how the modules should actually be separated, and this is how you

should interact with them" — I'd give them an implementation plan, and then they'd implement it.

That's a complex question, actually. I also had my agents build me an architecture viewer, so I can bring up a nice little UML-style diagram showing the modular structure of the system and where the dependencies go. I can click on a module, see the submodules inside it, click on those, and it'll actually bring up the code. So I can drill down as much as I want and see the architecture of the system at any level.

That's been very useful. I also built another deterministic tool where I can specify which module should depend on which, and which shouldn't, and how the dependencies should be routed. It outputs a compact specification file that agents can't violate — there's a checker that runs at the end, and if they violate the rules, they have to fix something. Usually that means dependency inversion, an interface implementation, or splitting a module in two to satisfy the rules. I'm working on automating this further, but I'm not very successful yet.

I'm in the same situation as you — you get a huge advantage from well-designed modules. Could you explain what that advantage actually is? Why is it important to have good structure for these modules?

Structuring Code So Agents Can Understand It

Well, it's the same argument as in the dirty-versus-clean-code debate. Anything that's well-structured, with clear interfaces, is easy for humans to understand because we're good at separating things in our minds. Models do the same thing; agents do the same thing — maybe with a slightly different threshold of perception, I'm not sure yet — but they work much better if they can focus on a module with a defined trajectory, so the model doesn't get confused about the topics inside it.

Okay, I'm using the words "model" and "module" — I want to make that clear. We get it, we get it. But it's important, and it's the same argument — it's the argument about coffee and soap operas. If you load everything into the module, the poor agent is going to think, "What the hell am I doing here? How can I do anything here?" If you get it all neatly organized, it's going to work just as well as it would with a human.

Deep Modules and Interfaces

So you probably get more value out of your test suite too, because that's what I always think about. I guess I have another question for you related to that. I have to say I'm a huge fan of your work, but I'm also a huge fan of John Ousterhout's work — who isn't? You probably have his book somewhere. His concept of deep modules seems really exciting to me: you can have bad modules, shallow modules, with a big interface and almost nothing hidden inside, or you can have deep modules with a small interface and a lot of hidden information inside. It occurred to me that this is really good for models, because they can read the interface without having to go into the implementation. Does that resonate with your vision? Isn't that how you approach things?

Yes, absolutely. Do models pay attention to interface names? They pay attention to structure. That lets them avoid reading the code underneath, which is both a danger and an advantage. If the code is

consistent, that's fine. They also pay attention to tests — they read the tests to understand what the system is doing. So yes, anything you do to improve the structure of the code will help the models understand it better.

By the way, there's a long discussion between me and John Ousterhout in the appendix of this book, and it was really interesting. He and I had a great time — well, I don't know how much fun he had, but I had a lot of fun.

I watched the whole thing from start to finish — I saw your discussion on YouTube where you talked about this, and I really enjoyed it, which is why I wanted to invite you, because I enjoyed it so much.

What Would You Change in the Book Now?

There are a couple of ways I could go about this. Is there anything in your book that you would change or update now? I'm thinking about some of the advice, like building small functions and keeping them compact. Is there anything — because we've talked a lot about the fact that a lot of things haven't changed, right? These approaches have always been useful; we just never had the resources to do them. These ideas have been good for a long time, we're just tweaking them a little. But is there anything we should throw away as obsolete? I hate to ask this question because it puts pressure on you, but do you have an answer for that?

Thresholds and the CRAP Score

So, thresholds — that's one of the points. It seems agents can handle different levels of complexity better than humans. They have much better short-term memory — huge, and absolutely accurate. So one of the things I do is increase the allowable size of a function, by adjusting the CRAP score. For a human, I'd keep the CRAP score below four. But for agents, I set it at six, and I think maybe I'll raise it to eight. I'm trying to find where that threshold is, and it's not easy to determine.

But what does that mean in terms of going from four to eight? Is it four to twelve? What's the difference — is it like a 20-line function or a 100-line function?

It really comes down to cyclomatic complexity, which is the number of paths through the function. If you have 100% coverage, a CRAP score of six means there are six paths through the function, and they're all covered by tests. That's the real goal of CRAP — cover everything with tests, then limit the cyclomatic complexity. I've had a few discussions with agents about this — and by the way, you can't trust any discussion with an agent, but I'm having them anyway — and they seem to think six is probably pretty good. You don't really trust them, but that's fine.

Disciplines Like TDD Don't Transfer to Agents

But I think there's a difference in thresholds, and there's another factor. In my books I talk about disciplines. One of them is test-driven development — I'm a big believer in it. But it's a human discipline that exists because people are wired a certain way. I can't and won't impose that on agents. I don't think it makes sense to have an agent write one line of test, then one line of production code, then the next

line of test. It makes sense to a human — at least for me, it's a huge benefit to my job — but for agents, I don't think so.

So I let agents behave more like John Ousterhout: write a function, then write a test for it, then write the next feature and a test for it. I let them do that, even when I told them to strictly adhere to test-driven development — they always come back to that eventually. So I think that's probably okay.

The main takeaway is that it's probably wrong to impose human discipline on an agent. It's not wrong to impose human values on an agent, but maybe we should change some of the thresholds. The disciplines themselves — these patterns of behavior — I don't think should be imposed.

That's great, I really like that. So it's about short-term memory, right? TDD is great when you have very limited short-term memory, like humans — you have enough to write a test, then enough to make the test pass, and then you can go get a coffee or something. Okay, that was great, thank you.

Planning Before Handoff: Waterfall vs. Agile with Agents

We talked a lot about building the agent's product itself — creating an implementation, modular architecture, and so on. What do you do before you hand off to the specifier? How much planning do you do before you enter that development cycle? Doing the wrong work in the cycle is a waste of resources, right? How do you think about that?

Well, the temptation is to have a human write everything down in detail and then hand it off to an agent. It's a very old temptation — we went through this in the '70s. It led to the waterfall development model and all that, and the Agile revolution was a response to that, or a reaction to it. I don't know how successful we've been with the Agile revolution, but it was a way of saying, "Wait, all this heavy upfront planning is just messing things up, because the result is never really like the plan."

Well, the temptation with agents is to do the same thing: okay, we'll plan it, then hand it off to the agent. I've tried that — in fact I tried it this week — and it's always a disaster. The result is always the same: you make all these plans, and then when the agents start working, you realize as a human that they can't stick to the plan, because you didn't foresee everything, and they're not as smart as you. So they go off and do some crazy stuff, and you have to stop them, go back, rewrite the plan, and start them again.

So I gave up on that. I said, "Okay, wait a minute, let's try an agile approach." I don't know how that's going to work — it might not work very well — but let's try it. Let them do a story or two, then we'll look at the architecture, and maybe I'll have to intervene manually and fix a few things, then a few more stories, and so on. That might be a better approach.

We might never — let's avoid that manual organization step at the end. Although I'm trying to figure out a way to do it, I don't know if it's possible.

I think about it like this: you have a huge amount of work to do. It used to mean building a product, and building a product could take days, weeks, maybe months. Now that time has decreased. But the planning at the beginning, and the testing at the end, has remained the same.

LIVE: Uncle Bob on Software Fundamentals in the Age of AI

Matt Pocock | 56:39 | transcript saved 22 Aug 2026
https://www.youtube.com/watch?v=zcLPGC-tvgk

Iteration Cycles Are Faster, But the Real Work Isn't

We as developers are expected to have faster iteration cycles, but what was actually quite complex has remained just as complex and takes the same amount of time. Doing the wrong thing is still very much something you can do.

I totally agree that a lot of people are doing this kind of "max planning" now, where they take their spec, think about it, run it through seven different agents or something like that, and come up with a better plan, and then try to implement it. Doesn't that sound good? Agents love to write plans — oh my God, they just love it. They'll decorate the plans, make them all beautiful and detailed. And then it all falls apart at the end. I think you're seeing this in the industry right now.

There's this movement toward specification-driven development, and my impression is that it probably won't work. My own experiments haven't worked very well. I'm now thinking maybe we should go back to the Agile ideas — do a little bit, get feedback, do a little bit more, get feedback, reorganize, do a little bit more, get feedback, reorganize.

The House-Building Analogy

I used to tell this story when I lectured on Agile. Suppose it cost you a dollar to make changes to a house — including the foundation, the roof, everything. Every change you suggested to the contractor would cost you a dollar. How would you build a house? Would you hire an architect and pay them thousands of dollars to come up with a perfect plan, then pay a contractor a dollar to build it all in one go? Or would you go to the contractor and say, "I want the foundation here, make it this way." "Oh no, that's bad, okay, let's change the foundation." "Put the kitchen here, put the living room there." "That's two dollars." "Oh no way, let's switch them around." "Let the kids walk through — oh, the traffic pattern is terrible, move the stairs."

Obviously the last option is probably better, and that's basically where we are right now. It costs a dollar — well, maybe two, maybe five — but the cost of changes has come down to almost zero, as far as I think is possible. That's a prediction I'll probably end up losing, but still: the cost of changes has come down so much, why bother with all that pre-planning? It's expensive. Why not just tinker, tinker, tinker, tinker until it looks right?

I agree one hundred percent. I can't help but agree.

What Even Is "Spec-Driven Development"?

I have a huge problem with the label "spec-driven development," because what is spec-driven development? You could have about ten different interpretations of it. Every time you give information to an agent — is that prompt engineering, or is that spec-driven development? It's like when you used to have a colleague sitting across from you, your buddy, and you'd say, "Just fix that header loading problem." Is that a spec? Is that the spec I gave him? It seems like every time I say, "Okay, let's just get this straight up front," people call that specification-driven development. But the difference, for me, is: do

you keep those specifications? Do you go back to them? So what's your attitude toward that — do you keep a list of all your specifications in a repository, or how do you do it?

Specs Are Ephemeral — the Real Spec Is the Tooling

No, I don't. Specifications are ephemeral — they disappear, they change all the time. I play with them and then they're gone. There's nothing like source code anymore. We humans used to write the source code, so that was the final specification. Well, that doesn't exist in the same way now. The source code is still there, but we humans don't write it. A lot of people miss that feeling — the sense that there has to be something human that defines everything in advance. But at the end of the day, even the things agents create were created by humans.

Here's what I've been doing lately: instead of creating a spec that defines what I want, or even what I have, I look at the end result and say, well, this is the spec. I have a bunch of tools — I have a CRAP tool, which works for Clojure, for Java, for Go. I've written a few of these, or I have my agents write them. I have a mutation tester, I have my agent toolkit — I have all these things. I tell people: don't download them, I wrote them for myself. You should point your agents at them, let the agents learn from them, and then build your own.

I think that's a much better way to get the gist of something and then adapt it to your own specific needs.

Agents Read Everything; Humans Read Nothing

What always amazes me about agents is that if you send them something, they actually read it. That's very different from humans — if you send someone a huge spec, you get maybe a 20% success rate, and honestly 20% is probably too generous, maybe more like 5%. But if you give the spec to an agent, it's much more likely to actually read it. And the flip side is that what agents write, people don't read.

Right. So they expect us to read everything they write, and we just don't. It's funny, it's so one-sided. In the long run, I wonder how human relationships change because we're so used to talking to agents — I dictate to my agents, it's like I'm just talking to a friend. I'm really interested in how life imitates art over time. I don't know, it's fascinating to me.

Tactical vs. Strategic Programming

I'll ask you maybe one last question, and it's a pretty big one. I'll mention John Ousterhout again, because he has a great definition of the different types of programming — tactical versus strategic. Let me grab the book — here it is. Tactical is the sergeant on the battlefield, the person who fights the battle. Strategic is the general, the person who directs the war.

I think we both agree that's a good way to put it, and agents are very good at tactical but very bad at strategic. So for people who are just starting out, and AI has already eaten everything tactical, how do they learn strategic programming? I think a lot of people want that from you, Bob — they want, "please give me your brain so I can use it, understand why mutation testing is so useful, tell me how to think

about these modules." Your job now is to give people your brain. How do they learn this?

How to Learn Strategic Programming

I get asked this question a lot, and I don't have perfect answers because I really don't know, but here's how I think about it. First, in terms of how a programmer should learn — whether at university or somewhere else — they should write code. You should write code for, say, a year. I don't know exactly how much, but enough that you understand what the agents are dealing with.

The next thing that should happen is: when you get hired by a company that's actively using agents, you, as a young graduate, should be treated like an agent. The person running the process — maybe a lead engineer, someone with a lot of agents running — is the one acting strategically. They should treat you like an agent, give you the same tasks as the agents, and force you to use the same deterministic tools the agents use. You should spend a few months in that state, being terribly inefficient but learning a lot. And once you get through that ordeal, maybe you can be trusted to run your own agent. I don't know — it'll be something like that.

You can't completely give up on code. I told people about ten years ago: if you've never written assembly language, it's worth spending a weekend on it just to see what's really going on behind the scenes, because if you're writing Java all day, you're living in a fictional world. There's still magic you don't understand.

Learning Through Layers of Abstraction

Spend a weekend on assembly language, and you'll finally understand what's actually happening. I think that's relevant to this learning journey — you have to go from the basics, binary code, through assembly, some basic code like C, then higher level like Python, and then you deal with agent operations and deterministic tools, and then you end up strategically managing the agent under supervision. I don't know how to phrase it better. It's difficult, isn't it? There are two things going on here: the agent is a layer of abstraction over the code. So you have the code, and it's interesting that you mention it as a layer of abstraction — a viewer, almost. That's a great way to learn: get those abstractions over the code so that as you go deeper into it, you understand it much better.

But if you have someone who's just sitting over the books doing tactical work, a company will look at that and ask, "Why are we hiring this person when we have Uncle Bob's squad of five optimizers who can do this for a fraction of the cost?"

How People Learn Strategic Programming

What resources, books, or approaches help here? I'm trying to understand how people get this information, because the feedback loop in strategic programming is so long — or traditionally has been. People who get fired after six months or so may never master strategic programming, because their mistakes are only revealed maybe nine months later. They just never see their mistakes.

But with agents, because things have accelerated so much, you can actually get feedback on your

mistakes earlier. I'm interested in how you realized your agents were making mistakes back in December — when you looked at them and thought, "this is a complete failure." How did you determine that?

At first I just looked at the code and saw it was nonsense. But that wasn't the most important thing. The most important step was the next one, where I watched them struggle. I saw the agent struggle, and I recognized that struggle because I had been through it myself. That's one of the problems: a newbie will come in and not recognize that struggle.

So how do you learn that? I learned it the hard way — the school of life.

Recognizing the Struggle

How do you teach that to someone who's just starting out? There's a huge amount of information on this — the old books, the ones nobody reads because they're old. Turn to Tom DeMarco or Ed Yourdon, or... I can't remember the title right now, but *The Pragmatic Programmer* — there are a lot of these old books, and they're really good. If you study them when you're young, you get a sense of what this is like: a high-level strategy game. You have to filter out some of the archaic stuff, because a lot of these books were written in the '70s or '80s, but that's when those lessons were learned.

So I'd go to those books first and study it that way. And then, of course, you have to learn it yourself. That's why I think people should be agents for a few months, to really understand what it's like. Become an agent — have the agent delegate tasks to you. You become a subagent to that agent.

Why Fundamentals Still Matter

I have another question, a short one: it seems like the fundamentals of programming still matter. Why is that, and what do you say to people who claim they're not important?

The fundamentals of programming are important for the same reasons they always have been. I remember who said it — I think it was Dijkstra, though I could be wrong: software is the most complex thing humans have ever tried to do. More complex than any other task we've ever attempted.

Software is the most complex thing. Fundamentals are a way of organizing that complexity into a form that not only humans can understand, but our models can understand too — after all, our models are made in the image and likeness of humans. So fundamentals are still relevant because they organize complexity so it can be understood.

Now, there are people who think fundamentals don't matter. They'll learn, and they'll learn the hard way, and it won't take long — though it might take longer than I think, because agents are pretty good at it. But I've seen them hit a wall, so I know that wall exists, and I don't want to hit it again.

The Same Story at Every Level of Abstraction

We're seeing a very interesting series of parallels. You mentioned the level of abstraction — we're now at a level above the compiler. Before, our level of abstraction was the compiler. Before that, it was assembly language. Before that, it was binary code. Now we're here at the model level.

LIVE: Uncle Bob on Software Fundamentals in the Age of AI

Matt Pocock | 56:39 | transcript saved 22 Aug 2026
https://www.youtube.com/watch?v=zcLPGC-tvgk

At each of these stages of increasing abstraction, people at the lower level were complaining, "Oh, this is going to break everything, we won't even have a job. It's become so easy that five-year-olds can write code." Back then they were still talking about binary code. It's the same thing at every step. And here we are at the next stage, and people are saying the same thing again: "Oh, this is going to break everything." No, it won't. The same rules apply. All the same fundamentals are there for all the same reasons. The rules you throw away are the ones you pick up off the floor a year from now, shake them out, and remember why you needed them.

Closing

Thank you very much, Bob. There's a great quote — I think it's from Plato — where he says writing will make people stupider. People have been arguing about abstractions since the Greeks. It's funny.

I think everyone watching — how many viewers do we have on stream? 1,500. Incredible. I think we can all agree this was a fantastic conversation. Bob, thank you very much. Hold up your book again so people can see it — let's take a look. *Clean Code.* And it looks like it's 11 a.m. your time — go back to your robe!

Great. Thank you very much, everyone. I'm ending the stream. Bob and I are going to stay on a little longer.