JARED: So, you've written and spoken rather forcefully about the perils of using integrated tests, going so far as to call them a scam. Why are integrated tests a scam?

J. B.: Well, they think they're going to solve a testing problem for you. In fact, they even make the testing problem worse. You might as well just put all your money in the middle of the room and set it on fire. I mean, those are the reasons why we block almost every number who tries to call us, right?

JARED: Hello and welcome to Dead Code. My name is Jared Norman, and today, we're back talking about testing. I'm joined by J. B. Rainsberger, and we're going to talk about what kind of tests to write, why we write them. We're going to talk about whether integrated tests are, in fact, a scam. And, I think, most interestingly, we're going to chat about how we frame TDD. What is doing TDD right? Does that mean anything? And how should we frame our experience as we learn to test first?

Hey, J. B., welcome to the podcast. Can you tell our audience a little bit about who you are?

J. B.: Absolutely. So, my name is J. B. Rainsberger. A lot of people know me as JBrains. No, I'm not affiliated with JetBrains. I had the name first.

JARED: [laughs]

J. B.: I've spent the last 25 years or so trying to help people, mostly programmers, write software in a way that makes them more comfortable and less stressed. And that started with teaching people sort of at the dawn of extreme programming in the agile movement in the late ‘90s and early 2000s. And, these days, I have a relatively small group of people that I work with regularly, and I also work with companies. And I'm generally trying to just help them take a lot of the pain, and suffering, and stress, and anxiety out of software delivery.

So, I still tend to work with programmers most of the time because that's sort of what I'm known for. But I do also help, you know, emerging technical leaders who want to spread their wings and become coaches, or mentors, or trainers. I help people who aren't programmers who are maybe anxious project managers or struggling managers. I don't live in that world, but I can certainly help them.

And, you know, I'm interested in the people side of the work as well. So, even though we have a tendency to start by trying to solve a programming problem together, a lot of the work really does come down to, like, why can't you two people talk to each other, or why is there this weird policy in the way? Or why do you think it didn't work when you tried to do this two years ago? You know, it's Jerry Weinberg's old chestnut remains pretty true that behind every technical problem is a people problem. It's always a people problem.

JARED: Right. Yeah, no, I agree with you there. My company is in the software consulting space doing e-commerce software. And it's very interesting to see how many of the problems that we face are not technical ones despite, you know, us billing ourselves as technical experts.

J. B.: That's sort of full circle then. E-commerce was where I started my career. I worked at what eventually became WebSphere Commerce at IBM starting in the late ‘90s and early 2000s.

JARED: Cool. Very cool. So, the bulk of the teams that I work with tend to be doing some kind of mix of test-after-test-driven development. They sometimes have some broad idea of the proportions of tests they're aiming for, either something like a testing pyramid or we often see test suites that are heavily skewed towards end-to-end tests of some kind. And it's rare that more than one or two people on any given team can coherently answer the question: When do you write each kind of test and why? So, let's start. I'll put the question to you. What kinds of tests should I write, and when should I write them, and why?

J. B.: This is one of those wonderful kinds of answers because I could take a deep breath and launch into probably 17 minutes of analysis about how to decide when to write smaller scope tests and larger scope tests and what is the audience of your tests, and all that kind of stuff, the difference between the customer tests and programmer tests, the difference between unit tests and integrated tests. And what is a unit test anyway? And what's a unit? And what about microtests? And when do we use test doubles, and when do we not? Blah, blah, blah, blah, blah, blah.

Ultimately, though, the answer that's perhaps less satisfying but nevertheless is more accurate and eventually helpful is, what are you afraid is broken? Kent Beck famously wrote all methodology comes from fear, which means that we don't have to have rules to govern our behavior until someone's afraid that someone else is going to do the wrong thing. And, in programming, it's very similar.

So, Philip Plumlee was the guy I learned it from over 20 years ago: Test until fear is transformed into boredom. So, then my question is, why isn't it boring yet? What are you afraid of? I think one of the things that programmers routinely get wrong is they feel like there's this proper proportion of all these kinds of tests that they're supposed to write, and if they can just get the proportion right, everything will be fine, and I don't think that's true.

I think what matters more is that when you decide to write a test, and I'm specifically talking about automated tests, when you sit down to write an automated test, what are you afraid is broken? And if you're worried about one or two lines of code over there, then I'm probably going to push you in the direction of writing a smaller scope test, something like a micro-test. Let's extract a function that contains the two lines of code you're worried about. Let's write microtests for that because you can write them thoroughly. You can run them quickly. You can understand them within the narrow context of I'm not sure if these two lines of code are doing what I want, and feel very confident about that.

But if you are instead thinking, well, I'm not sure what I'm testing, but I have the vague feeling that something's wrong over here, then that's the perfect time to think of a bigger scope test whose purpose is not necessarily to check something specific but is trying to survey an area. Like, I'm not sure what's broken here. I'm just worried that it was hastily written. I'm worried that it was written a long time ago. I'm worried that Jared wrote it, and I don't trust him yet.

JARED: [laughs]

J. B.: There's a whole bunch of reasons why I might just be worried that there's a problem in this room. And if there's a problem in this room, then I'm probably going to want to start with a smaller number of bigger scope, more integrated, more system-level kinds of tests. But then I'm not concerned about, do we get all 12 edge cases of this decision table? I'm concerned with, hey, what happens when we put these pieces together? Do they do anything we didn't expect? Or how well do I understand how that other thing over there behaves? Or there's something wrong here I can't put my finger on.

And casting a wide net is more important to me right now and uncovering unexpected problems than checking something that I'm pretty sure is wrong in a specific way. And I think that's, like, even just that difference; I think if you stop the average programmer on the street and ask them, “What is good about a big test, and what's good about a small test?” they'll struggle to articulate that. I'm not saying they're idiots. I'm just saying that that's not how they've been trained to think, and that's what I want to do.

I want to help...one of the ways that I can help is to train people to think that way, to focus to what is it that you're trying to achieve with this test. And often, when I ask them that question, their face goes blank. And the only thing they can say is, “Well, we're trying to get 80% code coverage,” or “My manager told me I had to write tests,” or “I read in a book that unit testing is a good idea.” Okay, cool. That's a starting point.

There's something about what's difficult about your life right now that made that resonate with you. Let's figure out what that is, figure out what your goal is. And then, you probably know what kind of test you need to write. Do you know that these kinds of tests are good for that problem, these kinds of tests are good for that problem? And so on. But it really does have to start with, like, what are you afraid is broken here? Tell me more about why you think you need to write that test at all.

JARED: Right. And it's interesting how you frame that in terms of talking about what you are afraid is broken. Does this apply when I'm thinking of writing new code? Like, are we talking about, oh, I'm afraid code I haven't even written yet is broken? Like, how does this relate to when I'm testing first?

J. B.: It can already be broken in your head. So, when I think about trying to write something new, so I'm sitting down to write some code. What are some things that might go wrong? There might be some complicated logic, and I'm not sure I'm going to get it right. So, I want to focus on

that. Or I'm integrating with some unfamiliar technology, and I'm not sure what I need to do there. So, let me focus there. Or I'm writing the kind of code that I routinely get wrong. I flip Booleans in my head. Alarming. The number of times that I will confidently write a line of code and then it's wrong, and then I put not, and it works.

JARED: [laughs]

J. B.: I've been doing that for 25 years. I recognize that as a blind spot of mine. Cool.

JARED: I do it in English, too. I have a habit of missing negatives in English and writing the opposite of what I mean [laughs].

J. B.: Yeah, exactly. “Did you mean to say, ‘Not there?’ “Oh yeah, yeah, yeah, yeah.”

JARED: Yeah, exactly.

J. B.: So, there's already three different classes of problems with the code in my head. One could be that there's just too many parts, and it's hard to keep them all straight. And sometimes the answer to that is writing tests, and sometimes the answer to that is just writing it down on a piece of paper beside you and getting the thoughts clear in your mind. Sometimes the problem is that I'm integrating with technology that is either unfamiliar, or difficult, or painful, or whatever. And I want to focus my energy on getting that integration right. If I get that integration right, the rest is easy.

Or this other thing where, you know, there is a classic mistake that I just keep making, and I need some way to compensate for making that. I need to feel free to make that mistake and have it cost less. And that can be rubber ducking, pairing, writing tests first, having testers who check my blind spots, all the kinds of things that you learn about from reading any book on how to build habits.

The purpose of these systems is not to stop you from making mistakes. The purpose of these systems is to allow you to make the mistakes and have it not hurt so much. And so, any of those could be the reasons that the code in your head is broken. And so, it doesn't have to be code that you've already written. The code that you're about to write is going to exhibit some of those problems. And I've got different kinds of tests I want to write in different...or different focus areas, let's say, for those different problems.

JARED: Right. Well, that makes sense. So, you've written and spoken rather forcefully about the perils of using integrated tests, in particular, going as far as to call them a scam. What do you mean by integrated tests? And why are they a scam?

J. B.: Yeah, before I even go there, I would like to...I've said this before, and I'll say it again. So, the whole it's a scam thing is a cute inside joke with my wife. Anything she doesn't like, she calls a scam. So, I adopted that immediately, and I thought that was fantastic. I didn't expect people

to take it quite as seriously as they did, but that's part of the law of raspberry jam, you know, the wider you spread it, the thinner it gets.

And I have softened my stance over the last 20 years. And so, I haven't gone back and rewritten it everywhere, but you'll notice that the articles are now called “Beware the Integrated Tests Scam.” And, in fact, I was on stage in Budapest at CraftConf 10 last year and presented the idea again under the name, the kinder, gentler name, “Beware the Integrated Tests Scam.”

I got into a lot of arguments with people about whether I was saying that integrated tests are always and forever a terrible idea. That's not what the word scam is meant to convey. Scam is a little bit more like anti-pattern. Anti-pattern doesn't mean bad idea. Anti-pattern means bad idea that seemed like a good idea at the time. Good idea in theory. Bad idea in practice.

And so, the scam part really comes down to the thing that you think is the solution actually makes the problem worse. That's why I use the word scam and stand behind it. That if you rely too much on integrated tests to solve a specific kind of problem, and it's a natural choice to make, it's like taking medication that makes the headache worse. And if somebody tried to sell you that medication, they would be convicted of fraud. And that's the sense in which I actually do stand behind the word scam.

Okay, so, integrated test, what is that? Why does he say integrated test and not integration test? An integrated test is any test that runs multiple interesting parts of the system at once, even though you really only want to check one of them and maybe even forget the last part. Just a test that runs multiple interesting parts of the system at once. You get to decide what interesting means. You're not going to hurt my feelings. It doesn't matter. If you want to think of it as more than one class, cool. Doesn't bother me. More than one module, across an interface boundary, across a process boundary, doesn't matter.

The important thing is, when an integrated test fails, I can point to one part of the code with confidence and say, “There's the problem. The problem is here.” The smaller the test is, so the less integrated the test is, the more isolated the test is, the more likely I can point to the part that's broken, as opposed to at the other end of the spectrum; there's a problem somewhere in this room. So, an integrated test then, the scam part of the integrated test is, when I try to write unit tests, I'm going to have gaps because I'm not perfect. I'm going to miss stuff. No matter how many unit tests I write, I'm going to miss stuff because the system keeps getting complicated, so I can't catch up.

And, at some point, someone's going to say, “Hey, all our unit tests pass, and there's still a bug. What's the deal?” And how you react in that moment really determines a lot about where the project's going to go. Because some people react by saying, “Well, we need an integrated test to see if the pieces hang together correctly.” It's the phrase they always tend to use. I don't know why. It's not my problem. And when they do that, they can sort of vaguely find the part where they made a mistake, where there was one unit test is missing, or the other unit test is missing,

or the two unit tests are mutually contradictory. And then, when they put the pieces together, they don't quite work. And the integrated test helps them discover that. It's true.

The problem, then is what happens if you keep going in that direction too far? I'm intentionally not creating a straw man argument. I'm not saying that, well, if you rely on integrated tests for everything, that's a bad idea. Clearly, everyone knows that. But it's more like the “How many grains of sand does it take before it becomes a beach?” question. Which is not so easy to sidestep. Like, how far do you rely on integrated tests before you're overpaying for a guarantee?

So, the metaphor that I like to use is, imagine that you're trying to paint a wall, and the way you've chosen to paint the wall is you've drawn a line 6 feet back from the wall, and you've got a bunch of cans of paint. And one by one, you pick up a can of paint and hurl it at the wall. Now, the cool thing about that is that you'd be surprised how much of the wall you can cover pretty well in, like, four cans of paint. You can get a lot of the wall covered with four cans of paint.

But once you start to notice that there are particular parts of the wall that you're trying to cover, the edges, the corners, the part that's farthest away from you, you start to notice that there's some cracks in this strategy. That this strategy let you move quickly, but at some point, what you should probably do is pick up a fucking brush and paint the corners.

JARED: [laughs]

J. B.: And the unit tests are more like using a paintbrush. And the problem isn't that throwing paint to the wall never works. The problem is that when you know that throwing paint at the wall is no longer working and the people watching you know that it's no longer working, and the people helping you know that it's no longer working, but nobody picks up the brush. That's what I don't get.

And so, when I talk about Beware the Integrated Tests Scam is when someone convinces you that the way to solve this problem is you need more people throwing more cans of paint at the wall. And what you end up with is a wall that is never completely painted with five coats of paint in the center and one thin coat of paint at the corners, and some corners that never get painted. And that can be okay, but it's an expensive way to work when you could just try both.

And the thing that drives people back into the arms of integrated tests is that they don't have a good way to identify, or to find, or to make precise problems at the integration points. And this is often why they then call them integration tests. To me, an integration test is meant to test the integration between two parts. It should live at the boundary of the two parts, not run everything else that those things touch, but to focus on the integration between those parts. Mock objects can be used to create integration tests, and, in fact, I routinely do that. I call them collaboration tests. Contract tests are integration tests when they don't try to run big parts of the system.

And so, the scam is that if you rely too much on integrated tests, you make it harder to do what you want, not easier. And then, you know, you have more and more big tests. Those big tests

don't care how badly your system is designed, which means since you're a programmer and you're productively lazy, you're probably going to cut corners because you can get away with it.

You're going to have bad days, and the tests are not going to stop you. They're not going to put positive pressure on your design. They're going to allow you to build tangled code that happens to work. And as soon as you go down that road, the tangles are just going to get worse and worse and worse. And they're going to guarantee that you're not going to take the time to write unit tests because the code's too tangled. I'm not going to write unit tests for this. I'd have to refactor it too much. And that's the scam. That's the merry-go-round.

Now, the more integrated tests I write, the more tangled my code gets, the more integrated tests I need. And, at some point, the CTO says, “All right, folks, I gathered you all here today because development has ground to a halt company-wide. So, on Monday, we're going to throw it all away. We're going to build it again in Python, and we're not going to make the same mistakes this time.” And that rarely works.

JARED: No doubt. So, I have a couple of questions coming out of that. So, the kind of integrated tests we see test multiple, you know, pieces in conjunction. There's kind of two that I want to talk about. The first, I think, probably...I'm getting the impression that you don't necessarily take issues with it, which is where when we're interacting with some third-party code or something, we choose to put some layer that we control between, you know, the rest of our application code and that third party.

And often, it's useful to, or we find it useful, to write tests that test that that layer actually does do what it's supposed to do with that third party. And so, we need to be running the third-party code and our wrapper for it in order to make sure that that collaboration works correctly. And we usually don't want to mock a third-party tool because that's the thing that we're expecting to change. I work in Ruby. So, you know, I certainly can't even rely on having, you know, interface types or something there to save me from some kinds of change.

J. B.: Everything is an interface all the time in Ruby. That's the beauty of Ruby, PHP, Smalltalk, pure JavaScript, right? Everything's an interface all the time.

JARED: Yeah. And so, you'd call that an appropriate place to use this kind of, you know, to test multiple pieces in conjunction?

J. B.: Yes-ish. So, tell me about the second type, and then I'll come back to that.

JARED: Okay, so the second type is, in the Rails ecosystem, you know, we're building web applications. We have controllers that serve requests via actions. And we don't have any kind of dependency injection or anything that we can do at that layer. The framework instantiates those objects and calls them. So, whenever they call, you know, we have some hilarious mocking tools and stubbing tools in the language because it's so dynamic that we can do some very funny things.

But, typically, what we end up doing for those top-level controller tests or request specs, there's a couple of different kinds of tests that we can do at that layer, is we do make these integrated tests that test the whole thing. I see people writing tons of these tests, and I'm not a fan of that. But we typically do end up writing, you know, at least one or two success and a fail at that layer and then leave all the sort of detailed testing of the objects underneath to something at least resembling a unit test. How do you feel about that?

J. B.: So, lovely. I'll come back to the technology integration point part first. I will indulge in a bit of total blaming here.

JARED: [laughs]

J. B.: I don't love that, especially after I just was apologizing for how blamey and horrible integrated tests are a scam was, and now I'm going to do the thing that I was just apologizing for, but I can't resist.

I remember when I first looked into Rails in the early 2000s, I don't remember exactly the timing, but since, like, 2005-ish, when I started seeing Rails as a possible off-ramp from doing web application development in Java. And, naturally, I was interested in test-driven development, so I was curious about how that would work in Rails. And I saw the intentional coupling at the center of Rails, that that was heralded as a feature, not a bug, that this was a deliberate design choice that allowed us to have the whole convention over configuration thing. Loved it. I thought it was wonderful.

This was even in the days before, like, some of you might not even be old enough to remember the days before ActiveRecord was its own thing, when it was just all this jumble of stuff. And because it was all a jumble of stuff, the only tests we could write were end to end. And it was a huge deal when somebody went in and built the first RSpec Rails test running environment that would allow you to do something like, oh my God, I can render a view without going through the controller. Like, I'm living in the future.

I bring all that up because I remember what it was like in those early days. And I had already started to try to teach some of this stuff. Back then, I called it test-driven J2EE because that was what was popular at the time. And this was when the ideas around integrated tests are a scam were starting to take shape. I had a couple of friends who were also interested in Rails, and we were all, you know, TDD types. And I looked at them, and I said, “This integrated tests are a scam thing is going to be a real problem in Rails over the next five years.”

And I think it was two or three years later when I started to hear Corey Haines popping up with the fast Rails tests talk. And he was starting to talk about, okay, we need to break this stuff apart because as soon as you try to write more than about six months' worth of code in Rails, it's big enough that this end-to-end testing focus is going to choke every company that tries. So, I say that as a way of, you know, my ego wants to say, “Haha, I told you so.

JARED: [laughs]

J. B.: Like, I knew this 20 years ago. Welcome to the party." But the kinder, gentler version of me wants to say, "This problem was always going to be there, and no matter what else we try to do, the solutions are going to remain the same." So, when I talk about how I approach these things, I want more. We knew this in the '50s, and it's a big problem, and that's why it keeps coming up and less old man yelling at clouds. Okay.

So, let me come back to the technology integration point thing. So, here's a pattern that happens over and over again. I'll use the database as an example because I am old and the local relational database was the first thing that we had to deal with that surfaced this problem. So, in the old days, you used to not...actually, it's funny; it's come full circle.

In the old days, they used to not let you install the database software on your machine. You had to run it from some central thing because the licenses were really expensive. So, back then, the first bit of advice was you need to be able to run the database on your development machine. So, that advice has come back in vogue now that everybody thinks that the database is just a thing living in the cloud somewhere.

So, first things first, I want to be able to write tests that allow me to run a local database instance because that's just less expensive. It's easier to run. It's easier to fix, blah, blah, blah. Now, here's what tends to happen. People have a tendency to either try to mock the database. So, for example, they mock their SQL interface. It'd be like, I'm going to mock Arel, or I'm going to mock ActiveRecord. Seems like a good idea at the time.

It's going to make my tests faster. And the end result is those protocols are so low-level and complicated, intentionally so, because they're toolkits. They're supposed to allow you to do everything. So, they have to be the lowest common denominator of what you might need to do. They are the opposite of convenient, by design.

And so, as you said, one of the first things we do is we come up with some kind of a pattern of usage API. Here's a library whose job is to represent the 14 ways that we talk to the database, which is the way we do things around here. These are the usage patterns. That's why I call it a pattern of usage API. They are the patterns that have emerged from trying to write code that interacts directly with the database.

And here's what tends to happen. People write code that talks to the database. They write tests that talk to the database. That's a reasonable thing to do. I applaud them. I think it's a wise strategy. We've gone down the mock the database, or mock Arel, or mock ActiveRecord, or mock JDBC road, and that did not work. More trouble than it was worth because we can't control their interfaces. They're too complicated. The mocks are too complicated. Pain and suffering. Just write integrated tests.

And here's the mistake they make. They then copy and paste the same boilerplate from data access layer module to data access module. They write essentially the same code for when they want to talk to the customers' table, and the orders' table, and the line items' table, and the products' table. And, really, how many times do you have to write a test that checks for select star from table find by primary key? At some point, find by primary key is a thing you learn how to do that has nothing to do with customers, orders, line items, whatever. It's domain-neutral.

And, eventually, the tests that you want are tests for, hey, can I do select all from table by primary key? Can I do delete by primary key? Can I do select this column? Can I do some fairly common SQL commands and use them with this version of the Postgres library, or use them with this version of the MySQL library? What you're really building is a convenient way to talk to that technology. And the mistake they make is they fail to draw a clean distinction, an actual dependency break between their repository implementations, the things that turn rows into customers and customers into rows, where you do all the mapping, and the database table names, and the column names, and all that stuff, and actually executing that thing on the database.

If they would draw a sharp line there, then the test that they think they want for their customer repository, orders repository, whatever, aren't for that. They're actually tests for select by primary key, delete by primary key, select star from table, blah, blah, blah, blah. And those are actually database driver tests. They're not application tests. They're database driver tests. And if there's another project down the street that wants to use the same version of Postgres, they probably want the same library that we use because it's a more convenient thing than the low-level database library.

And so, once you do that, once you make that clean break, the interface goes in front of your little library. And now the rest of your system, we can write tests for the rest of your system that mock this thing, this little pattern of usage API. Because you, when you decide, when you try to write your tests and use mock objects and find out the mock objects are too complicated, and this is why mock objects are terrible, and I want to kill myself, you can change the API. Those mock objects are telling you to refactor the API. So, do it. But if the mock objects are telling you to refactor JDBC, you're out of luck. That's a three-year-long RFC process. Good luck.

And so, people can go read a wonderful article by Matteo Vaccari called "How I Learned to Love Mock Objects" that illustrates this point by talking about an SMTP server. Don't mock the mail server. Mock the post office. When you do that over and over again, you end up with integrated tests that only check the point of integration between the last layer of your code and the first layer of theirs. And then, you can extract an interface from your code and mock that everywhere else in your system.

And the rest, 95%, 97%, 99% percent of your tests are faster. And the only ones that actually need to be integrated with the database are the ones that actually talk to the database. And they're not, this is the key, they're not tests for your application. They're tests for your database

driver. They're tests that demonstrate I know how to talk to Postgres 14.3.2 on Ruby. And that's it.

And once you know how to do that, you can make the contract of that library anything you want so that you can mock it simply and use it in the rest of your system. What a lot of programmers do well is extract code to sort of get 80% of the separation between the rest of the system and their database pattern of usage API. And the last 20% is where the rest of the value is.

Like, if they really do say, “I'm going to treat this like a third-party library. I'm going to treat it like a third-party dependency. It is a convenience library on top of, you know, Ruby Postgres,” now that's where the real value is, and the rest of the system doesn't care. It doesn't even know you have an SQL database at all, and it never will. And those tests can run lightning-fast.

And so, ironically, that's where we really want integration tests: tests that focus on the point of integration and don't go any farther towards the rest of your system. It's literally the integration point between the last layer of your code and the rest of their stuff. And the job of that one layer of code is to be the anti-corruption layer that protects the rest of your system from the dirty SQLs, or the dirty HTTP server, or the dirty SMTP server, or the dirty messaging system, whatever it is.

JARED: Cool. So, the bulk of the developers that I meet, you know, that's in the Ruby community primarily, who do test-driven development, learned it by basically trying to do it on the job, seeing what did and did not work. You know, my company, almost everyone's read “Growing Object-Oriented Software, Guided by Tests” and/or Kent’s book on TDD. I find that's a minority of people more broadly, even among people who are doing some form of test-first development. That seems to lead to a like, significant disagreement as to what TDD actually is, whether any particular test-first approach is the right one, so to speak. So, how do I know if I'm doing TDD right?

J. B.: The short answer is you probably are. And the fact that what's right for you is not what's right for the person next to you is not a problem to solve. We used to see articles that came out around the early 2000s, mid-2000s that were very heavily focused on TDD isn't about tests; it's about design. And you could sort of hear that there was an implicit you idiot at the end of the title, and I probably wrote a couple of those articles. So, I'm sorry. I was young and foolish.

What's really changed for me is sort of how I think about what that really means. Test-driven development solves different problems for different people at different times. Not long ago, less than ten years ago, I think, I started talking about four stages of TDD. I'm not going to go into the full details here, but I want to highlight the idea of four stages of TDD is four different ways of practicing and focusing on TDD, depending on what your biggest problem is.

So, early on, a lot of people come to TDD because too many bugs, and they're drowning in bugs. They don't know what to do about it. They heard that tests is a good idea. And they came to test-driven development because someone told them, well, “Hey, if you write the test first,

then it's like you're pre-bugging your code. You're not just fixing defects, but you're actually avoiding them in the first place," and that sounds marvelous.

So, people become interested in it for that reason. That's what got me in. And I actually now call that test-first programming, which means the only rule is no production code without a failing test because all you're trying to do is reduce the cost of bugs. You're trying to find them either as early as possible after you made the mistake or even to avoid the mistake in the first place. And if you write the test first, you are going to avoid a lot of mistakes that you would have made, and the ones that slip through, you'll find sooner. Fine.

For those people, in stage one it is about testing. Let it be about testing. It's fine. But, eventually, that gets boring. Eventually, writing code that behaves the way you want to gets boring. And now you're ready for stage two. And in stage two, now you're doing test-driven development. And the only thing you're adding is, I'm going to refactor now. My original idea about how to design this didn't quite work. Instead of throwing it away and starting again and writing the new solution test first, I'm going to allow myself to change code after I've written it.

And that refactoring allows you to change your mind. And because your bottleneck is no longer writing code that works, your bottleneck is being able to change it inexpensively. The expense of changing it becomes the problem. So, then you start practicing not just writing the tests first but also refactoring as you go. And you play around with how much to refactor, and blah, blah, blah. But what happens is that's the stage where all the articles come that say, "Test-driven development is about design, not testing, you idiot." And, in fact, I just slipped. We even experimented with calling it test-driven design for a few years.

JARED: Yeah, I've mentioned that on the podcast, too.

J. B.: Didn't have to change the initialism, right? I used to think of that as an evolution of the technique, but it's not. It's merely an evolution of someone's learning. It's another path of learning, another step on the learning path. Okay. After a while, refactoring gets boring, like the good kind of boring. I can do this. But you know what's annoying? Sometimes I have to refactor for, like, 10 minutes, and sometimes I have to refactor for, like, 6 hours. And I can't predict how long I'm going to need to refactor.

When I go to add a feature, I'm going to want to do some refactoring at the beginning, clean up before, you know, make the change easy, then make the easy change. Kent's been telling us that for 25 years. But sometimes it's a lot of work, and sometimes it's easy. The volatility of how much refactoring I need to do becomes my bottleneck. Like, I can't predict how much I'm going to need to do. And that's when we adopt the idea of smaller steps.

In stage 3, we do everything we did before, but we take smaller steps. Write the smallest test that could possibly fail. Write the least code that could possibly pass. If you like Kent Beck's idea of test and commit, otherwise revert, do that. But what matters is small steps. And when you practice small steps, the cost of refactoring gets amortized throughout all of your work.

You refactor all the time. You have fewer nasty surprises. You have a small number of very nasty surprises and otherwise, it's pretty boring. It becomes a power law, and that's stage 3. And then, eventually, even that gets boring. Like, okay, we always make little messes but never make big ones. Cleaning up is never a big deal. We know to clean up a little bit at the beginning. Boy Scout rule. We know to clean up a little bit before moving on. This is all pretty boring.

And then, the bottleneck becomes, okay, how do I do this differently in different environments? Like, there's a difference between what kind of tests I write in Java and in Ruby because one has a type checker and the other doesn't, or how I work in a solo project compared to working with a bunch of very experienced programmers, compared to working with a bunch of very inexperienced programmers, or working on back-end processing compared to web. Or, like, what happens when I want to do something with CSS? I still have no idea how to do test-driven CSS. Fine.

At that point, your adaptability becomes a bottleneck. Like, how do I choose all the things that I've learned? How do I use them to make a coherent strategy based on my environment? How do I know the cause and effect between writing more integrated tests or fewer, writing more microtests or fewer, focusing on the syntax of contracts or the semantics of contracts? That kind of thing. So, all this is to say that what they have in common is writing the test first. But when somebody says to you, "Oh, no, no, no, you have to make a test list first," or "You have to work in small steps," or "You have to work outside in or inside out," these are all just preferences. They're all just optimizations.

The only rule is, when it comes time to sit down and write production code, start with a failing test, and everything else is just technique. How well do you know which problems you're trying to solve, what you care about? And which things will help you? And that's where the people that you described, Jared, the people who picked up TDD in the streets and just figured it out by going by what works, I think they have a real advantage because they did exactly what I teach. They had an actual problem in mind. They saw a connection between writing the test first and solving their problem. They tried it. They saw what worked and what didn't. They, you know, did a kind of genetic algorithm way of figuring out how they should practice test-driven development.

And as long as you're willing to change your mind when it's not working, as long as you can go somewhere and ask for advice and help, and as long as you're willing to try again, you're doing it right. Everything else is just details.

JARED: That's really interesting framing. I think that people are always looking for hard and fast rules about how they should do something, and you've sort of turned that on its head and said, okay, you know, the golden rule of test-driven development still applies, but it is really the only thing that applies so long as we're doing test-driven development in this moment and everything else is...it was a very long way to say, "It depends," but [laughs] --

J. B.: As soon as they say, “It depends,” you have to ask them, “On what?” Otherwise, it's a useless answer.

JARED: Yeah. Yeah. And then, you know, what you're describing sounds like a sort of a growth path where you look at all of the different ways that people out there will tell you are the correct way to do TDD. And you evaluate them with the question, you know, when is this a useful and valuable way to do TDD for me?

J. B.: Almost all of them are right for somebody at some point. And it's even important to keep in mind that with four stages of TDD, you are going to change. I first started because I was drowning in bugs, and then that got boring. And then, I realized, hey, I keep designing myself into a corner. What do I do? Oh, here's this refactoring book. Let me read that.

And, you know, it goes back to kind of where we started a bit that being aware of what kind of problem you're trying to solve and seeing the connection between the technique, in this case, test-driven development, and the problem you're trying to solve is really the important thing. And people who only read this stuff in books, who only read blog posts, who only argue with each other at meetup groups or whatever, I mean, that's important. But the people who only do that get caught up in trying to build in their mind this rich decision tree of when to do the right thing. But it's not until you actually practice and get feedback, get hit in the face a few times from some bad experiences, that you build what becomes your style.

You know, we like to say that good judgment comes from experience, and experience comes from bad judgment. You need to be willing to let yourself have some bad judgment. And so, when people ask me, “But what's the right way?” I literally can't answer them anymore. There are some things that I do that are probably helpful. But if you're asking me what's the right way to practice test-driven development, I'd say the only rule is when you sit to write production code, you don't write any until you have a failing test. And, at the risk of blowing people's minds out there, I think you can even break that rule, and it's still test-driven development, as long as you're at least thinking in tests.

JARED: Well, it has been a pleasure having you on the podcast, J. B. Where can people go to learn more about these ideas and follow you online?

J. B.: If you search the web for Jbrains, J-B-R-A-I-N-S, you'll probably find me. jbrains.ca is where you can start reading about me. Unfortunately, dot com is some Japanese consulting firm that got there before I did.

But I have two blogs, yes, I'm that old, where I still write occasionally. One of them at thecodewhisperer.com is more of the nuts and bolts programmery stuff with code samples and all that, and blog.jbrains.ca is where I talk more about the wider software development stuff. But if you put Rainsberger into your favorite search engine and you filter out the marathon runner and the football coach, you'll probably find me.

JARED: Awesome. Well, thanks for coming on.

J. B.: Thanks very much for having me. I appreciate it.

JARED: I really found J. B.'s framing of test first through stages where we learn about what we care about and what we should be thinking about as we approach testing, and we make that approach more valuable to us in the initial stages as we get better at the skill. And then, later on, we learn to make it more flexible so that it’s valuable in more kinds of situations. I'd never encountered anyone framing it like that. And I think that's definitely going to impact how I look at the skill and how I teach the skill moving forward.

As always, this episode has been produced and edited by Mandy Moore.

Now go delete some…