Hi, I'm Uncle Bob and this is Clean Code. Teksting av Nicolai Winther Ah, welcome, welcome to episode 6 of Clean Code, Test Driven Development. Remember, episode 5, we talked about form? We learned that comments should be rare because, to the extent possible, we should write our code so that it expresses itself without needing comments. is important. We learned that we should take care with our blank lines and with indentation. We learned that file sizes should be kept short and that we should never make our readers scroll to the right. We learned the difference between classes and data structures and the proper usage of each. We learned that classes protect us against new types while data structures protect We learned that boundaries crisscross our system separating abstractions from concretions and that source code dependencies should cross those boundaries pointing towards the abstract side and away from the concrete side. We learned that databases do not contain business objects. Instead, databases contain concrete data structures. Therefore, the boundary between the application and the database should be crossed by inserting a layer. That layer will depend on the database and translate the data structures into business objects for the application. The layer should depend on the database and it should depend on the application. The application, being abstract, should not depend on the layer. So now, in this episode, we're going to learn about test-driven development. Fascinating. I was unaware that this discredited technique was still being promoted. Um, yeah. We will learn why code rots, and how a good suite of tests can eliminate that rot. We will learn the three laws of test-driven development. Laws that are so strange, so absurd, so utterly counterintuitive, that they might actually make some sense. We will learn the step-by-step approach to test-driven development, the red-green-refactor approach, to writing tests and keeping code clean. After that, we will address the most common objections to test-driven development. Indeed, Uncle Bob, there are so many logical objections to test-driven development that I discounted the technique long ago. you persist in touting this fallacy. Well look, if you'll give me a few minutes to make my case, then I'll address all your objections in segment six. Can you listen for that long? Yes, Uncle Bob, indeed I shall, but if I am not convinced… Finally, after addressing those objections, professionalism, craftsmanship, and discipline. So grab a hold of something, hold on tight, TDD is coming out tonight. Our moon is unique in the solar system. It's large compared to the earth. by comparison. All the other planets have teeny-weeny little moons when you compare them against their own size. Not Luna though. Luna is a big one. It's 2159 miles in diameter, a little over a quarter the diameter of the Earth. It weighs 80 quintillion tons, which is about 1.2% the mass of the Earth, and at its surface the is there. Having that big satellite swinging around us like that stabilizes the Earth's rotation, keeps the North Pole pointing north and the South Pole pointing south. Without that stabilization, the Earth would wobble like a top and that would make the climate a lot more variable than it already is. It's questionable whether complex life could have evolved without the stabilizing Of course, we're grateful for more than just stable climate. The moon may have played a significant role in the start of life itself. The tides that the moon raises in the oceans create a stirring effect, especially at the coastlines. This may have helped to mix the primordial soup that gave rise to us. But why do we have such a moon? Where did Luna come from? force because the Earth was spinning too fast after it first formed? If so, then why does the moon orbit in the plane of the ecliptic, the plane of all the other planets? If the moon came out of the Earth, wouldn't the moon orbit in the plane of the Earth's equator? So perhaps the moon was an explain why the moon orbits in the ecliptic, but it can't explain why the moon's orbit is circular or so close to the earth and it can't explain why the moon's rotation is tidally locked to the earth. A captured body should have a distant elongated orbit and its rotation should not be tidally locked. The moon and the earth have the same isotope that they very likely formed out of the same part of the solar nebula. The atoms are the same isotope ratios. So maybe the moon and the earth are brother and sister. Maybe they formed at the same time. The problem with that is that the moon is much less dense than the earth. The earth has a massive iron core. The moon apparently does not. In fact, the moon seems to be made out of the Earth's crust is made out of, low density silicates. If they formed out of the same part of the solar nebula, why doesn't the moon have any iron? The current favorite answer to all these questions is startling, and it took a long time for astronomers to work out the details. Four and a half billion years ago, just 30 still cooling down from the violence of its formation, a planet the size of Mars slammed into it. The energy released by that collision was enormous. It melted the two bodies before the collision was even complete. The two dense iron cores in the two planets slammed through the outer crusts and merged, coming to rest in the final molten body. But the lighter silicates involved and formed a circular ring around the new earth. That circular ring gradually coalesced into our moon. This explains why the isotopic ratio of the two worlds are almost identical. They formed out of the molten mixture. It also explains why the moon has no iron core. The denser iron stayed and is circular. It's in the ecliptic because the collision itself occurred in the ecliptic. The splash of materials and the formation of the ring occurred in the ecliptic. And then frictional forces within the ring circularized all the ring particle orbits, leaving the moon in a circular ecliptic orbit. In fact, the moon was much closer in those days than it is today. Since then, of course, the moon has been inching away from the earth by gradually stealing angular momentum from the earth's rotation. Over the last four billion years, the length of the day has grown from five hours to its current 24. It's taken that long for the moon to reach its current distance. So is this what really happened? Well, it's the explanation that fits the evidence best at the moment. of how the moon was formed, but we can imagine it. I mean, think of what a hoot it would have been to stand outside and watch that collision. Code rots. We've all seen it. Systems that start out with a good design and relatively clean code eventually degrade over time, The code gets tangled, warped and perverted. It starts to stink like a piece of bad meat. As the design rots, it becomes rigid, fragile and immobile. When you need to make a change, you can't just change it in one place. Instead, you have to touch the code in many places. And each time you touch the code, you risk breaking it in still more places. Debugging becomes complicated, estimates grow, and fear dominates. Why do we allow this rot to persist? Why don't we fix it by cleaning the code? Because, Uncle Bob, if we attempt to clean it, we are liable to break it, and then the agonizers will be engaged. Agonizers, right. We don't clean it because we're afraid. some code up on your screen that was ugly and the first thought through your head was wow this is ugly I should clean it and the next thought in your head was I'm not touching it because you know if you touch it you'll break it and if you break it it will become yours and so you back away from the code your fear in the system to gradually rot into chaos. And that rot slows us down. It slows everything down. Changes that should take days take weeks instead. As our uncertainty in the codebase grows, so do our estimates, and yet we still miss them. And the project descends into the quagmire of rotten, tangled, messy code. Sometimes the mess gets so bad that we contemplate a dedicated effort to clean it. We might set aside as much as a day or a week or even more to make a concerted effort to clean the code. At first, our employers might even support this idea of a dedicated cleaning effort. productivity, and we've promised them that if we can clean the code, well then we'll go faster and our estimates will shrink. But cleaning code is hard and fraught with risk. When we clean a module, we've got no good way of knowing that that module still works properly. Perhaps we've introduced subtle bugs or race conditions or data corruptions. The ad hoc manual tests we've been running to make sure that the system behaves properly begin to fail for reasons we can't easily explain. We start to debug, and the debugging time grows, and as it grows, it eats into the time we thought we'd have for cleaning. After weeks of cleaning and debugging, we've finally run out of time. bugging than we'd hoped, we think the system is at least a little bit cleaner. So we hand the new cleaned system to QA. Once QA gets their hands on it, they run their manual and very expensive suite of system and regression tests. And unfortunately they return with a huge pile of defects that our simple manual ad hoc testing never could have found. are daunting. They contain inexplicable behaviors, data corruptions, even crashes. The list of new defects is so long and so puzzling we don't even know how to estimate how long it would take to repair it. After much wailing and gnashing of teeth we finally conclude that the only sane action is to set the clean system aside and to revert to the pre-cleaned I have frequently been required to order whole teams into the agony chambers for such failures. Now look, this doesn't happen every time. But I've seen enough of these sad scenarios to know that they're not at all uncommon. And once you've been through one of these disaster scenarios, it's enough to convince you you never want to try a big cleanup effort again. to clean. Our estimates grow with every passing year, our productivity continues its asymptotic plunge toward zero, and yet we are helpless to do anything about it, paralyzed by our fear. And so the moral is, we can't clean code until we eliminate the fear of change. What if we had a suite of tests that was so comprehensive that virtually no bug could escape it? What if that suite of tests could execute in a matter of minutes? What if any programmer could run that suite of tests at the mere click of a button or the typing of a simple command? And what if that suite of tests could never get out of date with the system? What effect would that suite of tests have on the fear of changing the code? Uncle Bob, you are wasting my time. Such a test suite would be impractical to create. Impractical? On the contrary. It's been done. I've done it. Indeed. Then continue. tests running. The application is Fitness, something that I've been working on since 2002. Fitness is an open source project. It has thousands upon thousands of users. It's 70,000 lines of code. It's a web-based system with a big thick middleware and a persistence back-end. In short, it's a significant application, the kind of application that tends to rot if it's I've measured the code coverage at about 90%. Actually, I think it's a bit higher than that because much of the code executes out of process where the code coverage tool can't see it. So, I estimate the coverage to be closer to 95%. What you're looking at as you watch these test pass is the QA process. If these tests pass, we ship. There's nothing else we do. There's other procedure. If these tests pass, and they pass in just over a minute, we ship it. That's how much we trust this test suite. If it passes, we ship. Now it's true, this is an open source project. If there's a bug, nobody dies. Millions of dollars aren't lost, at least not that I know of. And yet we have thousands upon thousands of users and our bug list would fit on a page or two. A long bug list? That comes from irresponsibility and carelessness. I'm not talking about issues. I'm not talking about things that the software could do a little better. I'm talking about defects. When you have a long list of defects, that can only mean that the development team has been behaving unprofessionally. But I have often demonstrated my displeasure with long bug lists, and yet they continue to appear. In the fitness project, we keep our defect list short by maintaining a test suite that has very high coverage and yet very quick run times. It's really hard for a defect to get through that kind of defense. Having defects under control is just one of the benefits of this suite of tests. Another benefit is that we just aren't afraid to make changes to the code. We're not afraid to clean it. Alright, let's clean some code. And let's clean it fearlessly. I have a function here. This function is called HTML. I can run those tests for you and you will see that they pass. There's some code coverage information down the side here. You notice all the little green bars down the side of that function. That tells us that we have a hundred percent code coverage on this test, on this function. If you didn't see those green bars, well they're kind of faint, but I can see them and I can tell you I've got And you'll see they just run very nicely. Good. This code is a mess. It's got string buffers in it. It's got flags in it. Look at these flags in here. It's got funny little strings in it that don't seem to make an awful lot of sense all the time. HTML tags, data structures that represent HTML tags. You don't need to understand much of this to follow this refactoring. Notice that the author has got some functions here, add attributes, add tabs, add child HTML. I don't like the way that this partitioning works, so I'm going to get rid of it. And I'm going to get rid of it like this. I'm going to inline the add tabs function, calls with its implementation. I will do the same thing with add attributes, just getting rid of the old partitioning. There's add child HTML, I'm going to get rid of that one as well. Replacing the function call with its implementation, and of course, all the tests still pass. Now, I'm going to repartition this. This HTML function seems to have a lot of state. There's that state information probably belongs in a class. So the first thing I'm going to do is cut the entire implementation of that function and replace it with an instance of an HTML formatter. Passing in the depth. And I'll call the format function on it. formatter so I will create it as an inner class there it is lovely I will create a field for the depth parameter good I'll go back to the call I will create the format function it's going to return a string of course and I will replace the defunct implementation with the real implementation that I had just my tests will still pass they do that's very nice now I've got a class in which to put all those ugly variables and the first thing that I note is that this one is called a buffer well what it really is is the HTML it's being accumulated so I'm going to change that to HTML the next thing that I don't like is the fact use string buffers. The string manipulations are fast enough. They actually use string buffers under the covers, which means that I can turn HTML into a string. Of course, this breaks all the calls to append, but that's all right. What I'm going to do is I'm going to find every call to append, and I'm going to replace it with a call to plus equals. Actually, not a call, this replace works for some of them it doesn't work for others but there is a method to my madness so just watch what happens here all right now some of these look fine some of them don't of course there's some leftover parentheses these in particular don't work correctly let's go back up to the top any case where there were multiple appends that doesn't work right but that's alright because I and I can replace them now last thing I need to do is find all these dangling parentheses which I can find one at a time as I scroll upwards just finding any little place where there's a dangling parentheses, there's one there's one, let's see, any more, yeah I missed a few, there's one there there's one there, oh yes, and there's a couple there, right there, somehow missed them, oh yeah, there's that one, there's that one, and I believe the tests will still pass, yep, they do, that means I can get rid of this html.to string and just return html, little loop right there I think I can replace that with a function named make tabs very similar to that last one which was add tabs but it doesn't add it to a string buffer oh look we found at least one more yeah we'll go ahead and replace it that's fine yes go ahead and replace it it should not be a it should not be a plus equal an equals it should be a plus equals it should not be passing in the Let me change the signature to remove that HTML from the call. Good. Make tabs should return the tabs. And that should not be an HTML. That should be a tabs. Lovely. The first make tabs up at the top is right there. That one actually I can leave there because I can do that. And this should still pass. It does. Next, if head is not null, head plus equals head. Well, I think I can do something like this. HTML plus equals head equals null question mark that colon head. I believe that's the same thing. I can find that out easily enough by running the tests. the test passed just fine so I don't need that comment anymore this little bit of code I can remove I can call it make head how lovely and in fact I can probably just pull that right up to there yes that still works just fine it's adding the tag name I think I can just take that and replace it with the function make tag mmm make tag yeah fine and that I think I can just bring up here with another plus sign isn't that pretty and now the attributes well I think I can pull that out into a function named make attributes I will flag in there's make attributes that should be a plus equals and the make attributes function down here a little bit of cruft left over I should make that to attributes makes that look a lot nicer now that function makes a lot of sense go back up to the top and I think I can just do that that passes very So now, the next thing is this big if statement, and it is a big if statement. I mean it goes all the way from there down to there, and then look at that funny little else with that hanging out there, which looks a lot like that. And here's what's going on there. If there are no children, then it puts the ending on the tag, which doesn't require another closing tag. children then it puts the simpler ending on the tag and then it's going to have to end with G that so I think what I can do is simplify this bit of code by multiplying the if statement so I'm going to do this if has children is true there and I believe that will still pass it does and now I think I can take that else clause right there and bring it up to there that passes and that nice so now clean this up a little bit now I think I can this and reword it like so HTML plus equals has children question mark that otherwise that I believe that that will work I'll just comment this one out yes that passes the test which means that I can eliminate that code and now I can this bit here, I can pull out into a function called make tag end, which I think I can just pull right up here. Lovely. Now I think I can take this entire if statement with all this goop in it I'm going to not pass in HTML. There's the make children. That should be a plus equals. The HTML there should be initialized, and that should be named children. This should still pass. It does. And now I think I can do this. This line is getting a little bit long here. shorten it. How about that and that? That's nicer. A couple more things. We've got the tail and the inline. I think I can do them pretty simply. If tail is equal to null, then how about this? HTML plus equals tail equal null question mark nothing otherwise tail. I think that's kind of obvious. All right. That should still work. It does. I can eliminate the comment. I can take this little bit right there, call it make tail, and just do this with it. Plus make tail. that can probably come out that should still work lovely and now I can probably do the same thing with the is line thing inline thing let's see HTML plus equals is inline if it is inline then we're going to return nothing otherwise it'll And if I did that right, then this should pass. It does, which means I ought to be able to take this and turn it into a function called make line end. That ought to be able to go up in there. That passes. How nice. Probably ought to do something like that. the end tag that's probably left over down here and make children oh yes there it is right there ugly ugly ugly ugly um i should probably do this let's take that code out of there back up here we'll do this where do I want that I really want that like right there should only be done if there are children. Let's see if that passes. Yes. But I think otherwise that and that means I don't need this good and now this could be be brought into there let's see make children plus make tag make end tag and then let's bring the tail down here and let's bring the line end up to there and that should still pass and now i don't think we need that html variable let's get rid of it Make the tabs plus make the head plus make the tag and the attributes and the tag end and the children and the end tag and the tail and the line end. Let's see if that passes. Isn't that pleasant? I mean, that is a much nicer function than it was before. No funny variables, no flags, no nothing. You read it, it makes perfect sense. You don't really need to look deeper into it to see what's going on. If you want to, of course, you can. got this horrible make children function yet to refactor but we could continue that. I won't hear but if you want to you can. That was a cleanup. Notice how fearless it was. Notice that I wasn't worried about it. My tests were covering me the entire time. Tests let you clean your code. and all because of that lovely suite of tests. Okay, so maybe you're thinking this. Maybe you're thinking that this is all well and good for an open source project where all the developers work for free. But that in the real world of software development, a test suite like this would just be too expensive and time consuming to create. That's just a load of dingo's kidneys. These tests save us a massive amount of time, and they have from day one. From the very first line of code until now, fitness has been a test-driven project, and we can develop it faster and safer than any previous project I've been involved with. There are fewer defects, we debug less, we code faster, we code better. In short, we can dance rings around projects that don't have tests. need that suite of tests. It mitigates defects, it shrinks debug time, it speeds development, and it eliminates the fear of cleaning the code. Uncle Bob, your passion is compelling, but you have said nothing so far that answers my objections. Indeed, you have not explained how such a test suite is even practical to create. How, Uncle Bob, do you create such of tests. Test driven development is a discipline and as a discipline it has a set of rules. Abandoning those rules and just writing a whole bunch of tests whenever you feel like it is not a discipline and does not make you a test driven developer. of a doctor washing his hands before doing an examination, or the discipline of handling a handgun at a shooting range, or the discipline of behavior at a martial arts studio. These disciplines may seem extreme and austere, but they are imposed to ensure safety and effectiveness. The first three disciplines of test-driven development are known as the three laws. To the uninitiated, these laws seem extreme and absurd, just as hand washing seemed absurd to the doctors of the middle 1800s. The first law is, you are not allowed to write any production code until you have first written a failing unit test. Illogical, Uncle Bob. Tests must be written after the code. You see, I told you, it just feels wrong, doesn't it? It's like the first time somebody shows you how to grip a golf club, or the first time your drum teacher shows you how to hold the sticks, or the first time you put your fingers on the home row of a QWERTY keyboard, it just feels wrong at first. But don't get too excited, because the second law is gonna feel a lot worse. The second law is you're not allowed to write more of a unit test than is sufficient to fail, and not compiling is failing. Uncle Bob, first you suggest that we write tests before code, then you tell us we must stop writing those tests as soon as they don't compile. That would happen could we complete a thought? Your logic is seriously flawed. Yeah, it feels crazy, doesn't it? It's like that strange thumbs forward grip you use with a semi automatic pistol, or the bizarre follow through you use with a good bowling throw. It's just not the kind of thing you would do unless your instructors were continuously badgering you to do it. But things are about even crazier because the third law is weirder still. The third law is you're not allowed to write more production code than is sufficient to pass the currently failing test. Uncle Bob, I see no logic in having programmers constantly hopping back and forth between production code and test code. anything without constantly interrupting ourselves. Yes, that's exactly what I'm telling you. If you follow these three laws, you'll be stuck in a cycle that is perhaps 20 seconds long. You'll write enough of a test to fail, and then you'll have to write just enough production code to make it pass, and then you'll keep bouncing back and forth between the two. Little test, little code. Little test, little code. in these ideas. Coding this way would clearly be slow and tedious. Programmers would constantly be interrupting themselves. They'd never be able to think through a single thought. Programmers know how to write code, Uncle Bob. They know how to write if statements and while loops. They don't need an absurdly conceived set of laws interfering with what they already know how to do. why you think that. I thought that way once myself. So before you draw your final conclusion, consider the following. Imagine a group of developers following these three laws. Pick one of them. It doesn't matter who and it doesn't matter when. Sometime in the last minute or so, on executed and passed all its tests. And it doesn't matter who you pick, and it doesn't matter when you pick him. Sometime in the last minute or two, or three, or even five, everything they were working on executed and passed all its tests. Indeed. What would your life be like if everything always worked a minute or so ago? How much debugging do you think you would do? Clearly, with so little time invested in creating the defect, removing that defect would not be protracted. Right. I mean, it's hard to spend an awful lot of time debugging something that worked a minute ago. to undo the last thing you did so you can retype it. Are you good at the debugger? Do you know all the hot keys? Do you know how to set a break point and then set a watch point and get to the break point three times and then get to the second break point twice and then set the watch point so that the variable gets to 37 so you can start debugging? Are you that kind of champion debugger? This is not a skill to be desired. You don't get to be that good at debugging without spending an awful lot of time debugging. I don't want you spending your time debugging. I want you spending your time writing code that works. Logical. An increase in productivity must lead to a decrease in debugging skill. Therefore high debugging skill must be correlated with low productivity. Fascinating. I've lost virtually all of my champion debugger skill. I very seldom use a debugger. Oh, I do use it. Not very often, but on rare occasions, I'll have to chase down some bug. And typically what I'll do is I'll set a breakpoint in a test that's failing. Then I'll step into the production code one or two times. And I'll see the bug usually right away, because it's in code that I just got done writing within the last few minutes. mean you won't ever have horrible bugs. You know the kind that take a day or even a week to find. This is still software, it's still hard. But if you follow these three laws that'll happen far less frequently. So if I told you that you could shrink your debug time by a factor of two just by following these three stupid laws, do you think it'd be worth it? Actually I think you could probably cut your debug time by a factor of ten, but Let's not say that. Let's just call it an even factor of two. Would cutting your debug time in half make those three laws more attractive? It would be a factor, Uncle Bob. Well, don't answer yet, because if you follow these three laws, you also get... Somehow or another they give you a nice zip file. And when you unpack it, you'll get all the software. And maybe you'll also get a PDF. That PDF will be a pleasant manual written by a tech writer. At the end of that manual, there's an ugly section of appendices where all the code examples are. Where's the first place you go? You go to the code examples because you want the truth. wrote you want to go to the code and see what's really going on if you're lucky you can copy and paste that code into your application and then fiddle with it to get it to work those unit tests we're writing are the code examples for the whole system you want to know how to create an object there's a unit test that creates that object every way it can be created you want to know how to call called. The tests are a low-level design document. They're written in a language that you understand. They are utterly unambiguous. They're so formal that they execute, and they can't get out of sync with the application code. They're the perfect kind of low-level design document. Indeed, Uncle Bob. The cost of documentation has always been high, and its reliability This seems to logically resolve that issue. So now how much would you pay? Well, don't answer yet, because... When you write your tests first, you have to design the production code to be accessible from the tests. Since you haven't written that production code yet, the tests have a tremendous influence on the design of the production code. And that influence is to make the production code testable. Writing tests first makes production code testable. And another word for testable is decoupled. The only way to test lines of code is to access them from the tests. But the only way to access them from the tests is to decouple the functions that contain them. have a system that is far less coupled than otherwise. In short, you get a better design simply by writing your tests first. Incredible! The long-sought goal of highly decoupled systems suddenly rendered trivial through the production of tests? Uncle Bob, I am indeed intrigued. in reliable low-level documentation and even improved design. So now how much would you pay? Well, don't answer that yet. Because you also get something that's worth more than all those other things put together. Remember in episode one we talked about how code rots? like a piece of bad meat? In the last segment we identified why that happens. It happens because developers are too afraid to clean it. But what if you could do this? What if you could run a suite of tests that proved to a high degree of confidence that the system worked? What if you knew that those tests what if you implicitly trusted those tests? The tests would eliminate the fear. That's exactly right. You wouldn't be afraid to change the code. You could see a little bit of ugly code on your screen and you could fix one small part of it, clean a little bit up. Then you could run your tests and see that you hadn't broken anything. and see that you hadn't broken anything. And you could repeat that over and over and over, cleaning, testing, cleaning, testing. The tests allow you to clean the code. The tests stop the code from rotting. And that's the real power of those three laws of test-driven development. The test suite that results from following those three laws eliminates the fear of change. I mean, if you see something a little bit messy, you just clean it. You don't even think about it, because the fear is gone. Those tests will protect you from inadvertently breaking anything. Oh, the tests aren't perfect. You can never have perfect assurance. But those tests can be very, very good. So good that you are not afraid to change the code. I'd like you to think about something. important? Why are there so many books written on these topics? Why is so much effort applied to getting design and architecture right? It's because we want the structure of the system to be flexible, maintainable, and scalable. When we add a new feature we want to make sure that the structure of the system is flexible enough to allow those changes without breaking everything. And good make systems more flexible and maintainable. But nothing makes a system more flexible than a suite of tests by a huge order of magnitude, because that suite of tests eliminates fear. If I give you a perfectly designed system but no tests, you'll be afraid to clean it. You'll be afraid to improve it, and so over time On the other hand, if I give you a terribly designed system, but a comprehensive suite of tests, you will not be afraid to improve it. And so over time, it will gradually get better and better. The bottom line, if you want a flexible system, get a suite of tests that you trust. Logical, Uncle Bob. Entirely logical. from a questionable premise that you trust your tests. You have not yet established how that trust is achieved. How much trust do you have to have in those tests? I want you to think about those tests as though they were a parachute that you're gonna jump out of an airplane with. That's how much you have to trust them. After all, changing a is very risky. It's a lot like jumping out of an airplane. And you better have a good parachute. So how do you get a suite of tests that you trust with your life? Follow the three laws. Follow them implicitly. Follow them always. If every line of production code was written to make a failing unit test pass, then you will trust your test suite. tests after the fact, then you'll never trust that test suite. You'll always worry that it's got holes in it. This is because testing after the fact is boring. You already know the code works because you've tested it manually. Writing tests after the fact is some kind of procedural milestone. It's not a necessary part of getting your code to work. You already know it works. You tested it manually. make work. It feels like a waste. And that means you're going to take shortcuts with it. You're going to take shortcuts with your parachute. There are some portions of the code that are just difficult to write tests for. You've tested them manually so you know they work and therefore you don't feel it's worth it to write a unit test. And so your test suite has holes. And you aren't it has holes. Those three laws of test-driven development may have sounded pretty stupid at first, but nothing has had a more profound effect on the way software developers work since the invention of the screen editor. Those three laws change the way you work on a minute-by-minute basis, and that change designs, and the courage to change and clean code. Nothing I have found has more reliably prevented and reversed code rot than the three laws of test-driven development. Yes, Uncle Bob, I see it all now. The logic is impeccable. But that which is logical is not necessarily practical. that test-driven development is also practical. Then the time has come for clarifications. Permit me to show you a demonstration. picks up right where part one left off, right with the demonstration. It's a full half-hour presentation of one of the first and most popular test-driven development demonstrations, the bowling game. You'll see it go from a quick design session to a flurry of red-green refactor cycles through several major refactorings right up to the surprise and shocking ending. objections and complaints of our mirror universe visitor. I'll tell you why TDD is not slow and I'll answer what to do about your manager if he complains about all the tests you're writing. I'll tell you who tests the tests and why it doesn't matter that tests can't prove correctness. I'll answer the charge that test-driven development is too dogmatic and I'll tell you why writing strategy. We'll talk about legacy code and how to test GUIs and databases and I'll let you know just what I think of programmers who don't want to write tests and think that test-driven development is too hard. Finally, we'll talk about professionalism and why test-driven development is But say goodbye to Bob and all his kin And he would like to thank you folks for kindly droppin' in You're all invited back next time to this locality To have a heapin' helpin' of some more of TDD We'll be right back. We'll be right back. We'll be right back. We'll see you next time.