Hi, I'm Uncle Bob and this is Clean Code. I am the architect. What? What? Wrong movie. Functions. They're what we do. Virtually all the code we write goes into a function and all the code that actually does something goes into a function. Functions are the first tier of organization. They're the first container that we put lines of code into. So it's probably a good idea if we know how to do them well. How big should a function be? 20 lines? 30 lines? Do you know? and if you follow it, your functions will always be the right size. Have you ever wondered whether or not you found all the classes in your design? Do you worry that classes might be hiding somewhere in your code, undiscovered? It turns out that classes tend to hide in large functions. That's one of the places where they can be buried. In this episode I'm going to show you how to find those classes by getting your functions to be the right size. Oh there you are, devil! Always running away. You've heard the rule that a function should do one thing? Sometimes it's said like this. But what does one thing mean? In this episode, I'm going to teach you what one thing means in no uncertain terms. Because this episode is all about function science! I'll bet you thought I forgot about the astronomy lesson. No, I didn't. Where does the sun get its power from? In the 1890s, scientists believed that the big questions in science had all been answered, that there were no new discoveries to be made. In 1897, A. A. Michelson actually said that the future role of science was simply to add decimal places to results already obtained. There were a few things that bothered them like the age of the earth or the sun's power supply, but most scientists felt that those were minor issues, just little quibbles that would eventually get resolved with time. By the early 1800s physicists had worked out that if the sun were powered by normal chemical means like fire, then it would have enough fuel for perhaps 10,000 years. were convinced that the earth was at least 100 million years old. And since they had impressions of plants in those old fossils, they figured that the sun must have been shining for at least 100 million years. In 1887, Lord Kelvin solved this problem by showing that the sun could derive its power from its own gravitational collapse. in order to maintain the observed energy output. For an object the size of the sun, that meant it could power itself for several hundred million years. But by that time, the biologists and the geologists of the day were demanding even more time. Darwin was not willing to accept that a hundred million years was enough time for life to evolve, and the geologists were finding ever deeper and deeper layers of rock. So while Lord Kelvin had solved the problem of a hundred million years, now the biologists and the geologists were demanding billions of years. This standoff was eventually broken by a set of happy accidents that began in 1895. In that year Wilhelm Roentgen accidentally discovered x-rays. He had a faint shimmer came from some phosphorescent material that he had coincidentally set nearby. Roentgen quickly showed that these x-rays were very penetrating. He could fog photographic plates that were wrapped tightly in construction paper. He was even able to take a picture of the bones of his own hand. A year later Antoine-Henri Becquerel decided to experiment with phosphorescence. He felt phosphorescence was a source of x-rays. So he took material that would glow after being exposed to sunlight and he put it next to photographic plates that were wrapped tightly with construction paper. If those

plates were fogged then the phosphorescing material must be a source of x-rays. One of the phosphorescent materials he used was crystals of potassium uranyl sulfate, And it worked like a champ too. Every time he put these crystals out in the sun with a tightly wrapped photographic plate, the photographic plate got fogged. He was even able to create silhouettes of metallic objects that were placed between the crystals and the photographic plates. So Becquerel was right. The phosphorescing materials were producing x-rays. The sun went behind clouds and it stayed there for several days. He had an experiment all ready to go. He had the crystals and the plate. But without the sun to make the crystals glow, it was useless. So he took the experiment, the crystals and the plate, and he put them in a dark drawer waiting for the sun to shine again. Days later, Becquerel removed the experiment from the drawer. But then he did something unexpected. he developed the photographic plate what made him do that the crystals had never been exposed to the sun they hadn't glowed there was no way they could produce x-rays why did he develop that photographic plate we will likely never know but when he did the results were astonishing the plate was fogged somehow x-rays were being produced by those crystals even though they hadn't the sun. And those x-rays were penetrating the construction paper that wrapped the photographic plate and fogging the photographic plate. Becquerel tried it again and again. He repeated the experiment over and over and every time the photographic plate was fogged by these crystals. The x-rays coming out of these crystals would penetrate cardboard or construction paper. He could make But that power wasn't coming from the Sun as he had first thought. No, that power was simply coming out of the crystals. And that implied something very disturbing. How long had these crystals been emitting X-rays at this level? The only reasonable implication was that they'd been emitting X-rays since they formed, but that would have been millions of years. Where was that power coming from? The amount of energy was staggering. Becquerel had discovered radioactivity quite by accident, and with it he discovered the immense energies contained in nuclear reactions. He didn't know it at the time, but he had also uncovered the power source of the Sun. be small. The second rule of functions? They should be smaller than that. How big should a function be? Back in the 1980s we had a simple rule. A function should be about a screenful. Of course, in those days we were talking about a VT100 which had 24 lines, four of which were used by the editor. So the real rule was about 20 lines. Nowadays, Of course, these young eagle-eyed programmers all have multiple screens the size of a wall. And the pixels on those screens are the size of uranium atoms. So they can cram hundreds of lines into a screenful. So the screenful metric, it doesn't apply anymore. How big should a function be? is okay, maybe five, six, okay, ten is way too big. I can see the expression on your faces. I know what you're thinking. You're thinking I'm nuts. You're thinking why would anybody make their functions that small? But consider this, in a four line function how much indenting do you think you're going to see? How many nested if while or god help you try catch blocks will you see? In a four line function, there's just not a lot of room for indenting. So what would the bodies of an if statement or a while loop or a try catch block look like in a four line function? They'd have to be

function calls themselves, wouldn't they? I mean there's no way to bury a bunch of logic in a four line function. be extracted as functions themselves. And what do all functions have in common? Remember our last episode. They all have names, well chosen, highly descriptive names that tell you what the function does. And remember that scope rule. Remember that functions in a small scope should have long descriptive names. of lots of small little functions, then all those small little functions are going to have nice, long, descriptive names. What's more, you will extract the predicates of your if statements and while loops into even more nice, small, well-named Boolean functions. And all those lovely names will use the correct parts of speech, well written prose. I need to see an example. Show me an example. Back in 2007, I was browsing through Fitness when I noticed this wonderful example of a long static function. Its name was testable HTML, which by the way is a noun and not a verb, so it violates our rule for function names. is an acceptance testing framework based on a wiki. Testable HTML is a function that surrounds a test page with appropriately inherited setup and teardown pages. This function has 46 lines in it, and they're loaded with duplicate code. It's also got two arguments and two local variables that are used throughout all 46 lines. divided into several functional areas and when you have variables that are used by all those areas then what you really have is a class. After all a class is just a group of functions that use a common set of variables. So the first step is to invoke the extract method object refactoring in order to convert That class will then be constructed with the original arguments of the function, and then the invoke method will be called. Next, we promote all those local variables into fields of the class, and we initialize those fields in the constructor. This will allow us to extract many smaller functions without having to pass arguments between them. As you can see, the invoke function has a repeating structure. lines repeat four times. The three lines aren't absolutely identical. There are some minor differences. So the first thing we're going to do is eliminate those differences and extract a new variable so that the three lines are parametric. Now, we can extract out those four repeating groups into a single function named includePage, and then we can clean that function up a little bit. Now we can see that there's yet another repeating structure that has four instances. page name and then we can extract all four of those repeating versions out into a single function named include if inherited operations into nicely named functions. We can extract the code that includes the setups into a nice function named includeSetups. We can also extract the code that includes the teardowns into a nice function named includeTeardowns. Next, we can get rid of all that string buffer code strings and appending them all together in the invoke function. We can consolidate it even further by putting all the content altering code inside the if statement. Now at last we can extract out the predicate of that if statement and replace it with something that describes what it does very nicely. sentence that describes precisely what it does. It reads like well-written prose. So now the invoke function tells you exactly what it does. And so do all the other functions in this class because they're all small, simple, well-named functions. It surrounds a test page with setups and teardowns. So we can rename this class to SetupTeardownSurrounder, and we can rename the

invoke function to Surround. Are you out of your mind? I was pretty much expecting that reaction. Let me see if I can tell you what you're thinking. You're thinking that you're going to be lost in a sea of tiny little functions that you won't be able to find your way around inside the code. You're thinking that all those functions are going to take a long time to call. The function call overhead will kill you. that writing all those little tiny functions is going to take you a long time. Well, that's just nonsense. You're not going to get lost in a sea of functions. You're not going to get lost in a forest full of little tiny methods, because those methods, those functions, have names that you gave them. If you've done your job, if you've put names on your functions, well-named classes, and if those classes live in well-named namespaces, then nobody can get lost. Those names are the signposts that will guide you through the code. You can't get lost if you've used good names. Why do we feel so uncomfortable with the idea of small functions that have names? What is it about that that frightens us? I'll tell you what it is. It's the fact that long functions are familiar to us in a very deep way. Take a long function and turn it on its side and it looks like a landscape. We humans evolved to recognize landscapes. We know where we are by recognizing landmarks. That makes us feel comfortable in a very primal kind of way. we humans, at navigating complex terrains by recognizing landmarks. How do we know our way around inside of a large function? We recognize the landscape. We know, for example, that the axes are scaled in the third major indent after the second comment block. We know that the timing factor is adjusted in that little divot of code that follows the big nested try-catch block. We've memorized the landscape and we recognize the landmarks, and that's a very comfortable feeling to our reptilian hindbrain. But if you drop someone new into the project, they're going to look around this landscape of code and recognize nothing. They'll be completely bewildered. They won't know where to go or even in the code in hopes that they can find some safe place to camp before they're eaten by a saber tooth. On the other hand, if you drop someone into code that has lots of signposts, well-named functions inside of well-named classes inside of well-named namespaces, then you'll find that they'll be able to navigate their way around without getting eaten by the lurking terror birds. easily and then become productive in short order. I want you to think back to when you were twelve years old. You probably had a filing system for your room like I did. Everything on the floor. Your socks were at the foot of your bed. You wore them yesterday, you might as well wear them today. Your underwear, similarly, was at the head of your bed. This filing system suited you well. You knew where everything was and you felt very comfortable with it. But then your mother would storm into the room and say, oh my god this room is a pigsty! And she'd start to ruin your filing system by actually putting things away. She'd add insult to injury by forcing you to help in the demise of your careful and comfortable filing system. Eventually, your mother would finish destroying your filing system, and then she would give you a stern warning to keep your room clean from then on. After she left, and once she was well out of sight, your reptilian hindbrain would reassert itself. You would go get all the things that had been put away and put them back on the floor, re-establishing

It's not until we get to be about 25 or 30 and we're living with someone else that we realize that this is not a good strategy for working and living in a team. It's hard to be a team when you're surrounded by a mess all the time. When other people are involved, the on-the-floor strategy breaks down pretty quickly. And so then we remember what our mother taught us as she fruitlessly cleaned our room, put things away into drawers and cabinets and closets. We put them away so that they don't get trampled on by the other people we live with. We put them away so that we can find them later and so that others we live with can find them later. We put them away sometimes into containers with names on them so that when we come back to those containers later, we will know what's in them. year old files things away. You feel comfortable with everything scattered through a whole pile of deep indents. Later on as you mature and start working in a team you realize that the best place to put code is in nicely named a time long ago when we worried a lot about function call overhead. We knew back in those days that building a stack frame was very expensive and that passing arguments to functions took a long time. I remember once back in the 1970s we worked on an 8085 microprocessor and we were writing in C and in those days we could measure function call overheads in Our computers are so fast and our compilers are so good at optimizing that a function call takes a nanosecond, maybe less. Worrying about that kind of overhead is entirely misplaced and counterproductive. Instead, we should take advantage of the speed and power of these machines to partition our software for readability first. Yes, I know, some of you out there are working in very constrained embedded real-time environments. And yes, for you, function call overhead is still something of a concern. But it should only be worrisome in the innermost loops of your system. I could understand if you wanted to inline some functions in the core loops. But outside of those core loops, function call overhead is miniscule. Pascal once wrote, I'm sorry this letter is so long. I didn't have time to make it smaller. Of course, Pascal was right. It takes time to eliminate redundancy. It takes time to organize our thoughts and clear our heads. It takes time to make things small. And Pascal was right to apologize because leaving something big is irresponsible careless, and just downright rude, it places the burden of breaking down your thoughts and understanding them on your readers, all of them. You might be saving yourself a little bit of time, but you're costing everyone else a lot of time. What's more, you are likely to have to come back to that code in a month or a week or even a day, and you will have to break down your thoughts and And so leaving a function long is going to even cost you, as well as everyone else, a lot of time and effort. Don't be silly. Don't be penny wise and pound foolish. Cleaning up a large function just doesn't take that much time, and it pays back within hours or days with huge dividends. Do you know what a long function really is? A long function is where the classes go to hide. Did you ever wonder whether you had properly partitioned your programming to classes if you had found all the classes there were to be found? way to do that? In fact, I know of one. I'll show it to you. What is a large function? A large function is a scope. That scope is divided into different sections of functionality, usually seen as major indentations. Those different sections communicate with each other

using variables that set of variables in a long scope accessed by many different segments of functionality? Well, of course what you've got is a class. In fact, long functions can almost always be refactored into one or more classes. Show me an example! The example we're going to study next comes from this book, Literate Programming, written he describes a programming system in which you wrote your code as little snippets of Pascal or C or something like that, surrounded by explanatory text. The programming system would then gather the snippets together and weave them into a functioning program. The code you're looking at now is a Java translation of a Pascal program that Donald Knuth wrote this way in this book. So, for all intents and purposes, this is generated code, which explains why it's such an awfully messy gob of goop. This program prints a report of the first thousand prime numbers. It prints them on five pages, each page having four columns by fifty rows. I've written a test for this program. It simply compares the output with a previously saved report that I call gold. If the two match, then the test passes. This is an example of a characterization test as described by Michael Feathers in his wonderful book Working Effectively with Legacy Code. You write characterization tests when you have a gob of legacy code that you want to refactor. thing. Upon examination we realize that this program is one big function with a whole bunch of variables. Whenever you have a whole bunch of variables controlled by one big function, what you really have is one or more classes hiding in there. So what we're going to do is we're going is take this entire function and move it into a separate class all its own using the extract method object refactoring. We'll call the new class prime printer helper which of course is a terrible name but we'll go back and fix that name later once we understand more. For the moment let's just clean up a few odds and ends and then we'll run the tests which of course pass. Now, in order to break this function up into smaller functions, I need to take all those local variables and promote them up to be fields of the new class. I'm going to do that indiscriminately. I'm just going to promote them all. I understand that makes a bit of a mess up there in the class, but we'll fix that later once we understand this more. For the moment, I just need them out of the way. Once I do that, the tests still pass. It'd be pretty easy to take that function and cut it straight in half. The top part would calculate the prime numbers, the bottom part would print the nice little report. So let's extract that bottom part out into a function named printNumbers. the array of prime numbers into it and then we can call that new function from main that leaves us with a class named prime printer helper that has two functions in it one generates prime numbers the other prints a report from an array But generating prime numbers and printing numeric reports are completely different concepts. They don't belong together in the same class. So let's take that print number function and pull it out into its own class. We'll name the new class Number Printer, and then we will move that into its own file at the top level. Next, we'll fix a few things and do a few little cleanups. Then we'll get Main to use the new class to do the printing, and of course, everything still works. of course there's a bunch of cruft left over now in the old prime printer helper class so we can clean that up and that leaves us with a class that does nothing but generate prime numbers so we can change its name

to prime generator that move it to the top level and then get main to call it And so we've taken one very large, ugly, tangled function and we've turned it into two classes that are ready to be further refactored into a whole bunch of little functions. where classes go to hide. So the next time you look at a large function, you'll know that's where all the classes went to hide. I'm sure you've heard the old rule Do it well and do it only. This sounds like good advice, and it is. But there's a problem. What does one thing mean? The problem with the term one thing is that it's ambiguous. It could mean anything. Remember that function we were just looking at, testable HTML? and then render the whole thing in HTML. Is that one thing? Certainly to the caller it looked like one thing, but to the reader, wow, that function sure was full of a lot of stuff. When we refactored that function into many smaller functions, we did something interesting. First, we just pulled the functions into different levels of abstraction. The first part of that is pretty easy to understand. If your function is composed of many different sections, then it's clearly doing more than one thing, at least from the point of view of the reader. The second part of this is a little more subtle. more than one level of abstraction, it's doing more than one thing. So for example, string buffers, path parsers, and page crawlers are all implementation details at a low level of abstraction. Whereas testable pages and inherited pages are all business concepts at a high level of abstraction. We don't want one function manipulating both concepts. We want to separate of abstraction into different functions. So in order for a function to do one thing, it must not cross levels of abstraction. But abstraction levels are fuzzy. For example, page crawler and path parser, which one's higher? And where would you put string buffer? In between them, above them, below them? This fuzziness of abstraction level is undesirable. to tell whether or not a function is doing one thing. A deterministic way. One that you can't argue with. Does such a way exist? Indeed it does. How can you be certain that your functions do just one thing? I've got an answer for you, but you're not going to like it. In fact, you're probably going to hate it. to hate it because the answer is so obvious and because it's so frightening. How do you do one thing? You continue to extract functions until you cannot extract any more. After all, if you can extract one function from another, then the original function was doing more than one thing by definition. The implications are hair-raising. It turns out that if you want your functions to do one thing, then you need to extract, extract, extract, till you just can't extract no more. I call this extract till you drop. And of course, the end result of all of this is that our classes are going to be composed of functions that are all about four lines long. In fact, I want my functions so small that my if statements and my while loops don't need braces. I look at braces as an opportunity to extract. Oh, excellent, excellent. Show me a really famous example. This is the video store example, the famous example from Martin Fowler's wonderful book We're going to take this ugly code and we're going to turn it into a set of very nice small functions right now. Just look at these tests. They're a mess. They break all kinds of rules. The worst thing they do is test the whole program by looking at the text of the reports. That's equivalent to testing a system through the UI, which is always, always a mistake. factor these tests and we are going to

extract till we drop even in the tests the details of this process are available in another screencast for our purposes we're just going to speed through the first 25 minutes of that screencast and then we're going to look at the results I'm sorry. okay now we're going to slow down just a little bit to look at the statement class there's a few the strange positioning of a few variables. We're going to get through that pretty quickly. Okay, now let's watch as we refactor this generate function. This generate function is a big ugly lines of code and those five lines of code are pretty clear it clears the totals it creates a header and adds the rental lines it adds a footer simple next we're going to clean things up a bit move a few functions around so that they read nicely so that things are where you expect them to be now let's kind of big but for the most part it's just a great big loop so I'm going to reach into the body of that loop and extract it as a method and give it a singular name rental line I wonder what that switch statement does. Well, there's that comment up at the top that tries to tell us determine amount. So let's take that whole switch statement and extract it out, put it in a function named for the comment, determine amount, and then we can get rid of that comment. of a few silly structures. Finally, we can extract out the formatting of the text leaving rent-a-line as a nice pretty concise little function that shouldn't confuse anybody. Feature envy. It doesn't use any of the variables of the statement class and in fact it does manipulate the rental class. So I think we ought to move determine amount into the rental class. We can do the same thing with DetermineFrequentRenterPoints. It also smells of feature envy and ought to be moved to the rental. And now all that's left behind in the statement class is stuff about formatting the statement. has been moved into the rental class. But now as we look at the rental class, we can see that determine amount and determine frequent renter points still smell of feature envy. They used the movie more than they used the rental. So we're going to move them again. This time we'll move them into the movie, but we'll leave behind in the rental pass along the days rented. class. But since the switch statement switches on the type of the movie, we're all set up to replace that switch statement with polymorphism. So the first thing we're going to do is go to the tests and make the tests appear to be polymorphic by creating derivatives. Then Next, we're going to push down the determine frequent render points and the determine amount function out of the movie class down into the derivatives. Next, we're going to get a code coverage tool out and take a look at what's executed and what's not. And in the derivatives, we will find all the code that's not executed and we'll kill it. That will effectively get rid of all the switch statements and also all of the combinatorial logic in determined frequent render points. in the derivatives. from all that refactoring, leaving the movie class and the rental class as minimal as possible. In the original screencast, which you can get separately, this refactoring took an hour. We took one very large, ugly function, and we dragged it through a whole bunch of classes We dragged it out of the statement, into the rental, into the movie. We created three derivatives. We dragged that code down into the derivatives. We literally turned this code inside out. And what was left behind is a nice little trail of simple little functions, each one with a nice name, each one that explains itself perfectly. Pretty amazing, isn't it?

One big function refactored into a whole bunch of little functions smeared out over a whole bunch of little classes. This is much better. It's much simpler, it's much easier to understand, much easier to read, much easier to change and maintain. This is how you want your code to be. You think I'm serious about this? You're damn right I am. If you want your functions to do one thing, if you want your functions to be nice and small and have nice names, if you want to find all the classes buried away in your program, if you want your code to be nice and small and easy to read and easy to understand, if you want others to read your code and think it's pretty much what they expected, Okay you scouts, come on, let's gather around, come on, hurry up before the coons get you. Come on, come on. All right, before we head out, a few things to remember. First of all, remember that the first rule of functions is that they are small. And what's the second rule? That's right, that's right, they're smaller than that. Remember that lots of little well-named functions will save you and everybody lots of time because they're going to act like signposts along the way, helping everybody navigate through your code. overhead. In almost every case, worrying about how long it takes to call a function is probably misplaced. Remember that making functions small will save you and everybody time. Remember that classes hide in long functions, and that if you want to properly partition your program into small. Remember that functions do one thing and the only way to really be sure that a function does one thing is to extract till you drop, right? Extract till you drop. Let me repeat that. If you can extract one function from another, you should because that original function was doing more than one thing. If you do all these things my scouts you will be keeping your code very very clean indeed. Alright now let's head out. So this has been lots of fun and we've learned a lot but there's other things we need to talk about. For example how are we going to organize our classes handling and side effects switch statements and structured programming you're not going to want to miss the next exciting episode of clean code episode 4 function structure Oh yeah, and one more thing. If you'd like to see the full-length versions of all the refactorings we did, you can purchase them separately at cleancoaters.com. There's two full hours worth of them. I'll be talking the whole time describing what I'm doing and why. All right. Nå er det en ny kvartal. In which you write snippets of code surrounded by explanatory text. Text. Text. Text. Text. Text. Text. Text. Text. We'll be right back. You are so beautiful You are so beautiful Today There's no sun You're the only thing I want You are the deal of the day You are so beautiful To me You are so beautiful Tre, you are so beautiful, and your day takes you still. Dea me dea, o Pai, Dea me dea, o Reem, Dea me fulgurum, Dea me fulgurum, Oh, my God.