Hi, I'm Uncle Bob and this is Clean Code. We'll be right back. In the last episode, we made the point that we want our functions to be very small, maybe four lines of code or so. We want to have as few levels of indent as possible. One is the ideal. And we don't want very many internal braces within our functions. Our functions should be well named, and they should be organized into well named classes within well named namespaces. functions are where classes go to hide. That if you want to find all the classes in your application, you should keep your functions as small as possible. We also made the point that in order for your functions to do one thing, again, you should keep your functions as small as possible. And so we introduced the discipline of extract till you drop. Now, in this episode, we're going structure. And to do so, we're going to get some help from the craftsmen at Eighth Light Inc. I'm Colin Jones. Eric Smith. I'm Michael Martin. So my name is Steven DeGutis. We will make the point that function signatures should be small. The fewer the arguments, the better. We'll also talk about what types should be passed into those arguments and which types shouldn't. structure, it appeared to be a bag of random variable and function declarations. We're going to talk about a way of laying out our classes so that the organization is self-evident. Do you know why switch statements cause such harm to software structure? Did you know that if statements have the same problem? In this episode we're going to talk about how to get rid of most of them and what to them so that they don't cause trouble. Did you know that assignment statements were also considered harmful? That may seem strange to you, but we've become ever more convinced that many software problems can be avoided by significantly constraining state-changing operations and side effects. Have you ever seen a function that was so full of error checks and null checks that you couldn't tell what it was doing? functions that are chock full of nasty nested trike hatch blocks that obscure everything. We're going to look at a way of structuring our functions so that error handling is done in a clean and maintainable fashion. In 1972, Edsger Dijkstra told us that GoTo might not co-authored the software engineering classic, Structured Programming. We're going to talk about this fascinating discipline and discuss why it's so important even today. So get yourself ready, because we're about to dive into the topic of function structure. The Sun. It's a ball of gas a million miles in diameter. It weighs 2 times 10 to the 30th kilograms. That's 300,000 times more than the Earth weighs. Imagine the gravity of that object. And think about the inner core of the Sun, the innermost 10%. What must the pressures in there with half a million miles of hydrogen above it being hauled down by that incredible gravitational field. And what must the temperatures in there be like having been compressed and compressed by all that weight? Indeed, the temperatures are as high as 14 million degrees Kelvin. That's more than enough to rip the electrons off the hydrogen Those protons are all positively charged. They repel each other like these magnets do. And the force of that repulsion is inversely proportional to the square of the distance. Every time I cut the distance in half, the force quadruples. So any protons that are on a collision course are likely to veer away from each other, keeping their distance. another force. It's called the strong nuclear force and it's powerfully attractive. It's what binds protons together into

nuclei. It can be a hundred times more powerful than the force that's trying to repel them. Unfortunately, this force has a distance limitation. If the two protons don't get to within 10 to the minus 13th centimeter of each other, they However, if two protons can be coaxed to get within 10 to the minus 13th centimeter of each other, then they will race together, ferociously attracting each other, fusing into helium with such enthusiasm that a tremendous amount of energy is released. This process is known as fusion. fuse, we've got to overcome this huge repulsive force. Remember that the repulsion is inversely proportional to the distance between them squared. So in order to bring them to within 10 to the minus 13th centimeter of each other, we've got to overcome a repulsion that's 10 to the 26th somethings. And it doesn't much matter what those somethings are when there millionth of a Newton or roughly the weight of a gnat. But in the core of the Sun the temperatures are so high that all those protons are moving at 600 kilometers per second and the pressure is so high that the density is a hundred and fifty times that of water. Those protons are always within a billionth of a centimeter of each other and the net result of those extremes is that four tons of hydrogen fuse into helium every second. Now that may seem like a lot, and it is. But there's enough hydrogen in the core of the sun to last for 12 billion years, and we've only used 6 billion so far, so we've still got time, no need to worry yet. The energy produced by this process keeps the temperatures and pressures in the core very high, The Sun is in a constant state of dynamic equilibrium. It's like Atlas holding up the world, constantly pushing outwards, furiously burning fuel just to stay the same size. One day, six billion years from now, a huge nugget of helium will have accumulated in the core of the Sun. But that's a story for another day. How many arguments should a function have? In general, a few words better. Function arguments are hard. It's hard to figure out what each parameter means. They're hard to read and they're hard to understand. Each one can confuse and confound. One can break your flow as you're reading down the code. Each one can cause you to do a double take. If we want to conform to Cunningham's rule and keep our code pretty much what we expect, then we need to limit the amount of confusion and double takes generated by function arguments. So rather than treat function arguments as conveniences that we use with reckless abandon, amount of restraint and discipline. We should treat every function argument as a liability and not as an asset. The best functions have as few arguments as possible. Zero is best. One is okay. Two But it's bordering on sloppy. Personally, I don't like my functions to take more than three arguments. Even three is usually too many. I try to keep two as sort of a guideline. The problem with three is that it can be hard to remember the order. Once you get more than three, it's challenging to remember what they do. So remember, we don't want our readers to be doing double takes. Besides, if three or more variables are so cohesive that they can be passed together into a function, why aren't they an object? I even feel this way about constructor arguments. Constructors follow the same rules. Pretty much the same guidelines as before. I much prefer to use a set of nicely named setter functions over a constructor with a whole bunch of arguments. setters to build up the class and add the data. Some sort of structure you pass to the constructor. You can

always pass a map of options or something like that. Yeah, passing structures and maps into a constructor is okay in a pinch, but I still prefer to use the setter functions. And I know what that means. It means that there's a short period of time when the object is incomplete and therefore dangerous. make sure that I don't use such incomplete objects inappropriately. So, to be kind to my readers and to keep my own sanity, I try to limit my functions to no more than about three arguments. Generally, I don't like passing Boolean values. are doing is you are loudly declaring to the world that you've just written a function that does two things. It does one thing for the true case and another thing for the false case. What you should have done is to write two functions, one for the true case and one for the false case. A Boolean hanging out in an argument list can be a significant source it's true? What does it mean if it's false? If the name of the method and the argument don't make this perfectly clear, then this can be a significant source of puzzlement and error. I don't know about you, but when I see a Boolean in an argument list, my eyes kind of blur over. I mean, I don't know if it's the right state or not. I just kind of trust that the author knew and pass the right Boolean in. Passing in two Booleans is even worse. A function that takes two Booleans does four things. Besides, what order do they come in? Have you ever stared at a function call that passed in two Booleans, true, false, and wonder what the heck the two of them mean and what order they're coming in? Two Booleans will cause a double take indeed. How do you feel about output arguments? Oh, I hate them. I hate them. I know in some C libraries and if you're working with Microsoft libraries, you really don't have a choice but to use them. But every time I do, I feel bad. I feel icky about it. But in Java and Ruby and Clojure, I rarely have the need. I don't think I ever have the need. And I like it that way. I dislike output arguments. don't expect data to be coming out of a function into an argument. When your readers see an argument they pretty much expect that the data in that argument is going into the function not coming out. If the data comes out it's going to confuse them. They're going to do a double take and we don't like double takes. So if you've got to pass some data out of How many times have you studied a function call, looking at some argument, wondering why the heck that argument was going in, only to finally realize that the argument wasn't going in, that in fact it was there to receive some output from the function? Output arguments cause double takes. They aren't really expected, so I avoid them where possible. passing null into a function or writing a function that expects a null to be passed in is almost as bad as passing a Boolean into a function. In fact, it might actually be worse because it's not at all obvious that there are two possible states. But there are. There's a behavior for the non-null case and there's a behavior for the null case. functions, one that takes the non-null argument and the other that doesn't take that argument at all. Don't use null as a pseudo-bullion. Besides, I don't like defensive programming. I hate littering my code with null checks and error checks. I'd hate to think that my teammates were carelessly allowing nulls to slip through into my functions. Defensive programming is a smell. I kind of trust my tests. Defensive programming leads to offensive code. It's just a waste of time. Defensive programming means that you don't trust your team or your

team's unit tests. If you are constantly checking your input arguments for null, it means that your unit tests are not preventing you from passing those nulls. Of course, in public APIs, I'm going to be programming defensively, because who what people are going to be passing into my functions. But for code within my system and written by my teams, I have a very different policy. That policy is that the best defense is a really good offense, and a good offense is a good suite of tests. Long ago in the galaxy of C++, we had a rule. the scissors rule. It said that you should organize your class such that all the public members were up at the top and all the private members were down at the bottom. And then you could take a listing and cut that listing between the public and the private parts and then you could hand the public part of the listing to your users. G sharp and Java. But for some reason the early Java programmers adopted a different rule. They took the private stuff, the stuff that we're trying to hide behind a wall of privacy and they put it at the very top of the class. Now look, I'm not here to fight that convention. In fact, I follow that convention. I follow it because it is the convention and might be a sub-optimal convention than it is to confuse everybody by doing something unexpected. No, I am here to introduce another rule called the step-down rule. This rule will help us impose some order on our methods despite the silly rule of having all the privates at the top. When an author writes a magazine article or a newspaper article, they follow a simple rule. details go at the bottom. For example, take a look at this article. It starts with a headline, there's a synopsis paragraph, after that, which kind of outlines the whole article, and then after that, every paragraph goes into more and more and more detail. There are two benefits behind a rule like this. The first is that an editor who needs to fit the article into a magazine few paragraphs without missing any of the essence of the article. The second reason is that readers can start at the top, read downwards until they get bored, and then stop. If at all possible, I like to follow the same rule in my software. Big method at the top that we'll call other methods lower. The public methods are really telling me what I can do with it, so I like to group those and then the public stuff should be at the top the big important ideas are up at the top and then as we go down I want the to increase in detail I want the private things below the public's and then the private methods that are called by the first private methods and then those called by the second private methods I don't want any backwards references I want all the function calls to point because as you read downwards, the functions step down one level of abstraction at a time. The functions at the top are very high level. The functions at the bottom are very low level and very detailed. And there's this stepping down one level of abstraction at a time. So here is a class that I have chosen from the fitness application to demonstrate the step down rule and I have made some modifications to this class to accentuate the step down rule those modifications unfortunately cannot persist because the IDE would not allow it and besides it violates Java convention you see the the private variables here are at the top as per Java rule but that is not the convention so I also have some public variables which are even further at the top that seems appropriate to me following that is a constructor and following that is the sole public method of this class I've chosen a class with one public

method to highlight the step down rule and then at how do you deal with more than one public method. Notice that this public method named serve calls three functions, try process instructions, close, and close and closing. The try process instructions is a child of serve. It's a kind of sub-function of serve. put it here and I have indented it below serve. Obviously the IDE won't tolerate this. Once I do a reformat it will undo this indentation. I am a little bit annoyed by that. It would be nice if this indentation could persist. Even better, it would be nice if Java would allow me to have inner functions, a truly block structured language like ALGOL. reality that we live in. I have maintained this indentation so that we can see the step-down rule, but then I will undo it for you and you can see how the end result looks. Try process instructions calls initialize, and it also calls process one set of instructions. Here's initialize. It's a child of try process instructions, which doesn't seem to call anything this class it does the lowest level work here process one set of instructions which was called up here does call other instructions it calls getting instructions from client and process the instructions get instructions from client is a child of process one set of instructions and it does the lowest And process the instructions is called by process one set of instructions there. And it has some children as well. It calls execute instructions and then send results to client which you see here. And then finally we have the last two functions that were called by the top level function close and close and closing. that those were called all the way up here. So that shows the step down rule. Every function calls children function, which then call their own children function, and we order them in the order that they are called and in the order of their hierarchy. order that they are called. There are no backwards references. No function down below calls a function that is up above. How would we deal with another public function? Should the public functions all go at the top? Well, in Java, because there's no way to preserve this nice I don't do this. In Java, what I will usually do is start the next public function at the end of the hierarchy of the first public function. I find this a bit annoying, because I would like all the public functions up at the top. What I would really like is for my IDE to know that these are child functions the one public function and then I could see all the public functions at the top and expand them if I wanted to. So the way I work right now in Java is that I will put the public function followed by all its privates, followed by all their privates, followed by all their privates until I completely exhaust that hierarchy and then I would put the next public function I can understand if you don't want to do that it it's it's a bit of a dilemma for me if you wanted to put all of the Publix up front and then start the expansion of the trees I can understand that it's not the choice I have made at the moment now I'm going to undo all that lovely Notice how much information is now lost. You can't tell where these functions belong. I wish we could. I wish there were some gesture we could do in our IDEs, at least, if not our language, that would allow us to say that one function is a child of another. You might think that, well, if you've got functions that are children of another, doesn't case no it doesn't these these little functions all share the same variables they are truly child procedures not child classes so that's the step-down rule all the functions are in the proper order you can easily read from the top to the serve

function then to the try process instruct try process One last thing to note here. Notice the way we have constructed the try blocks. Notice how the try block simply calls a single function. We're going to talk about that in an upcoming segment. So this was something that was burned into my brain a long time ago at Abbot Mentor. They would always say that switch statements are a missed opportunity to use polymorphism. Uh, yeah, I just don't. I don't think of it as a rule, I just don't use them very often anymore. I wouldn't say I have a rule about it, I generally don't use them. Why do we hate switch statements so much? What's behind that loathing? Some people will tell you that switch statements aren't OO, they're not object oriented. That's not a very satisfying answer. I mean, why aren't they object oriented? And is it important that they're not object oriented? And is it really true that we don't like switch statements? Yes, it really is true that we don't like switch statements. And the reason does have to do with the fact that they're not OO. aren't they, OO? And what's so great about OO anyway? In a future episode we're going to discuss a final and unequivocal definition of OO. For the moment I'll just say that one of the big benefits of OO is the ability to manage dependencies. Imagine, for example, that we have two modules, that calls a function in module B. That means that there is a dependency from module A to module B. Now that dependency has two components. Clearly, at runtime, module A depends on module B. So there's a runtime dependency. But there's another dependency. The source code of module A in languages like Java, must contain some statement like import or include or using that refers to module B. It is this source code dependency that we're interested in. What if you want to deploy module A and module B separately? For example, what if module B is a plugin to module A? If module A has a source code dependency on module B, then the two cannot be deployed separately. can't even be compiled separately. Any change made to module B will force a recompile and redeployment of module A. OO allows us to do something spectacular. We can invert that source code dependency and yet leave the run-dime dependency alone. Watch how we do this. We remove that original dependency and we insert a polymorphic interface. Module A depends on the interface, module B derives from the interface. This changes the source code dependency. The source code dependency now points that way, at least from B's point of view. So B's dependency points against the runtime dependency. The source code dependency the flow of control. And so this allows module A and B to be deployed separately. Module B can plug into A. In fact, there can be many different module Bs that plug into A, all of which do interesting things, and A's source code has no knowledge of them at all. And that's just one of the things that's so good about OO, independent deployability. is the antithesis of independent deployability. Each case in the switch statement is likely to have a dependency outwards on an external module. When there are many cases, there will be many such outgoing dependencies. We call this the fan-out problem. In a switch statement, the source code dependencies point in the same direction as the flow of control. on all the downstream modules. If any of these downstream modules is changed, there is an impact on the switch and anything that depends on the switch. And that means that if this is changed, then switch

and the application that depends on it will probably have to be recompiled and redeployed. In other words, the switch statement creates a knot of dependencies that makes virtually impossible. When we see a switch statement, we've got one of two options. The first option is to invert the dependencies with polymorphism the way we did in the last episode during the video store example. The other option is to move the switch statement to a place where it can't do any harm. As we saw in the video store example, it's really not all that difficult to replace a The trick is to take the argument of the switch, which is some kind of type code, and replace it with an abstract base class that has a method that corresponds to the operation being performed by the switch. Then each of the cases of the switch becomes a derived class that implements that method to do what the case did. Notice that the runtime dependency, the flow of control, remains the same. still flows downhill towards the case methods. But now the source codependencies have been inverted, because the case methods live in subtypes, and those subtypes depend upwards on the abstract base class. So the only thing we have to do now is figure out when and how to create those instances. Typically we do that in some kind of factory. In every application you write, you should be able to draw a line through the module diagram that separates the core application functionality from the low-level details. On the left-hand side here we have the application partition. This is where most of the code of the application lives, configuration data, and on this side we've got the main program. So I call this partition the main partition. Now the application partition is usually subdivided into a bunch of different modules. But the main partition should be kept small and subdivision should be limited. The dependencies between these two partitions should point in one direction and one direction only. the line pointing towards the application. The main partition should depend on the application. The application should have no dependencies backwards towards main. In essence, main is a plug-in to the application. This is a technique called dependency injection and there are many frameworks that can help with it. Indeed, some developers overuse those frameworks. They build instead of depending on good partitioning. The trick to dependency injection? Carefully define and maintain your partitioning. Go ahead and use the framework, but only inject a few entry points from main into the rest of the application, and then let main do the rest of the work with factories and strategies. partition usually do no harm. So long as all the cases stay down here in the main partition, and so long as all the dependencies cross the line pointing towards the application, then nothing that happens in the application can affect the main partition, and therefore the main partition remains independently deployable, and the switch statements here wind up doing no harm. So switch statements down in main are safe, to be an independently deployable plugin. All the source code dependencies point in the right direction. And in fact, this is true of any independently deployable module. Switch statements in those modules are safe so long as the source code dependencies all point in the right direction. If your application is composed of independently deployable plugins, and it should be, then all of those plugins and the central core shouldn't know anything about the plugins at all. All the source code dependencies should point inwards from the plugins

towards the core. No software dependency should point outwards away from the core and towards the plugins. Switch statements in plugins are harmless so long as the dependencies point in the right direction and the plugins are isolated. But what if the plugins have plugins of their own? well-written applications, the line between application and plug-in blurs. Oh, there's still a main partition that's independently deployable. But the other modules of the system form a kind of jigsaw puzzle of modules that plug into each other. So for example, module A plugs into module B, module B plugs into module C, and module C plugs back into module A. not isolated plugins and so we have to be careful about how we deploy switch statements inside them the goal of all this partitioning is to create a system that is composed of independently deployable modules of course lots of systems are not independently deployed indeed most of the time we just gather up all the modules into a single file and ship it as one unit but that doesn't we don't want independent deployability. You see, a system that is independently deployable is also independently developable. When we have a plug-in structure, then the development teams working on those pluggable modules can work independently because the plug-in structure makes it much less likely that the teams will interfere with each other. On the other hand, ruins the independent developability. Now the teams are going to be colliding with each other. We'll have much more to say about this in an upcoming episode. For now, just remember, switch statements break the plug-in structure that we desire for our applications, and they introduce significant impedance between otherwise independent development teams. statements have the same fan-out problem that switch statements do. So make sure you replace most of them with polymorphic dispatch and move the rest of them into some safe independently deployable module. Over the last 50 years we have seen a number of revolutions in the software these have been fads that have come and gone, but three in particular have been persistent and profound. The functional, structured, and object-oriented paradigms. In the last segment, we took a very brief look at the object-oriented paradigm. In this segment, we're going to take an equally brief look at the functional and structured paradigms. Future episodes are going to deal with these topics in much more detail. Functional programming was the first of the paradigms to be invented. We can trace its birth to 1957 with the advent of Lisp. Ironically, it was the last of the paradigms to become popular. What is functional programming? So you can't change the state of variables. So there's no side effects. Pure fun. So functional programming is this new age fad. I was a new age fad. It's a lot of fun. Functional programming can seem pretty radical at first. If you've never looked into it, I think you're going to be surprised. And you might even be skeptical that writing programs like this is even possible. You see, the functional paradigm tells us to write programs without any assignment statements. No assignment statements? absurd and unreasonable. But in fact, it's perfectly possible, and it's even easy once you get the hang of it. Instead of setting a bunch of values into variables, you pass those values as arguments into functions. Instead of looping over a set of variables, you recurse through a set of function arguments. mathematical function. You're calling functions that always return the same value back, if you've given the same

input. Right. The value of a function depends only on its input arguments and not on any other state in the system. So every time you call that function with the same inputs, you get the exact same output. There are no side effects. And that leads us to the interesting topic When a function changes a variable that outlives the function call, for example when it changes the state of an instance variable, then that function has a side effect. And that side effect might change the behavior of that function or some other function the next time it's called. This kind of scary action at a distance makes programs difficult to understand and Often side effect functions come in pairs. Set, get, open, close, and of course, new and delete. Now you know why I called these things a persistent source of error. The functions have to be called in order. Open must be called before close, new must be called before delete. A temporal coupling is when you depend on one thing happening before or after another thing. Like, any time you're using a database, you're coupled to the fact that you've opened up a database connection and then you're going to do your stuff and then you're going to close the database connection. So things happening in a particular order. Temporal coupling is when the order of operations will matter. I think it's like the mind's note. Ah, like spy. Right. temporal coupling your couple temporal lobes I think okay you've certainly faced the issues of temporal coupling before everybody has you've got some system failure and the reason the system is failing is because two functions got called out of order and usually it's not as simple as open and close I mean you can look at open and close and see that there's a temporal coupling between hidden, they're part of the background. You look at the two functions that must be called in a certain order and you can't explain why they have to be called in that order. It's just that the system fails if you don't. Can you eliminate temporal couplings? Often you can. Imagine this case here. It's the case of open and close. The problem with open is that it has a side effect. this way. Open would take as one of its arguments a file command. Then open would open the file, it would execute the file command, and then it would close the file. This leaves the system in the same state that it was before and eliminates the temporal coupling. This is a technique called you can resolve the temporal coupling by passing in a block. You hide the second function inside the first, and then you pass a command into the first, and the first one will do the open or do the first function, then it'll execute the block, then it'll do the close of the second function. This guarantees that everything stays consistent, that the functions are always executed in the right order, and that there's limited side effects. In the end, we can't get rid of all side effects. And in fact, we don't want to. I mean, we need to be able to change files. We need to be able to update databases. We need to be able to generate output. All these things are side effects, and they're desirable side effects. So our goal is not to eliminate side effects. Our goal is to impose discipline upon where and how those side effects happen. One very successful discipline for managing side effects is to create a strong separation between commands and queries. In this context, a command changes the state of the system. It has a side effect. A query does not. A query returns the value of a computation or the state of the system. A command changes the state of the system and returns nothing. The obvious cases are

getters and setters. Getters are queries. You don't expect them to ever change the state of the system. If you called a function called getAmount, you would expect to be able to call it any time you wanted. You'd be horrified if you found out later that getAmount actually changed the state of the system. By the same token, setAmount clearly has side effects. It'll be involved in temporal couplings. But, what do you expect set amount to return? You could invent something, of course, but there's no obvious and clear value that you'd expect. Command query separation formalizes this. It says that functions that change state should not return values. Functions that return values should not change state. This may sound a bit silly and arbitrary at first, fact some distinct advantages. The advantages are that it's easy to recognize whether or not a function has side effects. A function that returns a value does not have side effects. It's a query. A function that returns void has side effects. Let's look at this the other way has created a new logged in user but the authorizer passed it back to you what are you supposed to do with it are you supposed to manage it are you now smack in the middle of a big temporal coupling are you the one who's supposed to call log out on this user at some point don't you think the authorizer should be keeping control of the user don't you think the authorizer should be managing Why did the authorizer pass the user back to you? Why did the login function return the user? Maybe login returned the user so that it could return null if login failed. We are often tempted to return some kind of error code like this from our commands if they fail to change state. But it's better instead to throw an exception, Now of course multiple threads complicate things. Imagine for example that we've got a function that changes the state of the system but also returns the previous state so that we can save it and restore it later. We don't want to separate this into two calls the way command query separation would ask us to because we don't want a separate thread to sneak in between those two calls. the open and close problem. And if you think about it, you can resolve it the same way by passing a block. So don't confuse your readers. Functions that return values should not change state. And functions that change state can throw exceptions, return values. An extreme form of command query separation tells us to avoid queries altogether. How often have you seen code that looks like this? Wouldn't this be better if we used an exception more like this? And wouldn't that actually be better if we just let the user object deal with the problem After all, it's the user object that knows whether it's logged in or not. That state belongs to the user object. Why would we take that state out and make decisions for the user here? We should let the user deal with the problem. That last was an example of tell don't ask. What's tell don't ask? That you should be telling other objects to do the work and not just asking them. kept up on these politics lately. Simply stated, tell don't ask is a rule that advises us to tell other objects what to do, but not to ask objects what their state is. We don't ever want to ask an object for its state and then make decisions on that object's behalf. The object knows its own state and can make its own decisions, thank you. then maybe we wouldn't need so many query functions. And that could be a good thing, because query functions can get out of control pretty quickly. I'm sure you've seen long chains of functions that look like this. We call

these train wrecks, because they look like boxcars coupled together. They are a clear violation of tell, don't ask, because we ask, then ask, then ask, and ask we tell anything. Wouldn't this be better? Like this. Now we've gotten rid of the problem. We've told O to do something, and O's got to figure out how to get it to Z. O probably doesn't know where Z is either, but O knows somebody they can tell to figure it out, so the call to do something will propagate outwards until it eventually gets to Z. long chains of queries, violate something called the Law of Demeter. This law tells us that it's a bad idea for single functions to know the entire navigation structure of the system. Consider how much knowledge this line of code has. It knows that O has an X, that X has a Y, that Y has a Z, and that Z can do something. one line of code to have and it couples the function that contains it to too much of the whole system. We don't want our functions to know about the whole system. Individual functions should have a very limited amount of knowledge. We don't want a function to be able to walk the whole configuration database. We don't want a function to be able to wend its way through the is tell our neighboring objects what we need to have done and depend upon them to propagate that message outwards to the appropriate destinations. The Law of Demeter formalizes tell-don't-ask with this set of rules. You may call methods on objects only if those objects were passed as arguments, were created locally to your method, of your method or are globals. In particular, you may not call methods of an object if that object was returned by a previous method call. We don't want these chains of methods. Following this rule is hard. In fact, it's even been called the suggestion of Demeter because it's so hard. But the benefits ought to be obvious. Any function that follows this that tells instead of asks is decoupled from its surroundings. Biological systems are an extreme example of tell-don't-ask. Cells do not ask each other questions. They tell each other what to do. What that means is that you are an example of a tell-don't-ask system. Within you, the law of Demeter prevails. So maybe this law isn't so hard to follow after all. Our last stop on this exploration of paradigms is going to be with the youngest of the three, structured programming. That might surprise you. Structured programming, the youngest? But yeah, in fact, that's right. Functional programming is the oldest. It got its start And although some of the ideas for structured programming can be traced as far back as 1940, it really didn't come to its fruition until Dijkstra published his seminal paper, Go To Considered Harmful, in 1967. I find it pretty amazing. All three of those paradigms were invented within a 10-year period over 50 years ago. have been no new persistent paradigms invented. Structured programming says that all algorithms should be composed out of three basic operations, sequence, selection, iteration. Sequence is simply the arrangement of two blocks in time. The exit of the first block feeds the entry of the second. the flow of control into two pathways. Each of those pathways contains a block. One of those blocks gets executed, then the two pathways rejoin, and there's a single exit. Iteration is simply the repeated execution of a block until some Boolean expression is satisfied, at which point the iteration exits. Dijkstra showed that if you composed your algorithms out of these three about them sequentially. That's because each of the structures' correctness does not depend on any of the others. He also showed that so long

as you didn't violate the recursive block structure with unconstrained go-tos, you could construct proofs of correctness. Constructing proofs is not usually our goal. But think about this. If it's possible to construct a proof that your code is understand your code. A provable system is an understandable system. The recursive block structure of sequence selection and iteration is due to the fact that all three structures are self-similar. They all have a single entrance at the top and a single exit at the bottom. So into algorithms, those algorithms continue to have a single entrance at the top and a single exit at the bottom. And as you compose those algorithms into modules, those modules continue to have a single entrance at the top and a single exit at the bottom. And as you compose those all parts of algorithms, all modules, all systems share this attribute. A single entrance at the top and a single exit at the bottom. Does the single entry, single exit rule mean that you can't have multiple returns from a function? No, not at all. where the exit is, so there's no violation of structure whatever. On the other hand, there is a small problem with returning from the middle of a loop. Mid loop returns add an unexpressed and indirect exit condition, and while this doesn't exactly violate structure, it does make the loop a lot more complicated. What about breaks and continues inside of loops? problem at all because the continue just becomes a no-op down to the bottom of the loop and then the loop does the next iteration. So it doesn't violate structure at all. Break on the other hand is a bit more problematic. Just like returning early from a loop, a break creates an indirect and unexpressed exit condition. A labeled break is even worse because it exits from more than one loop at a time. constructs don't explicitly violate structure, but they do make the loop constructs a lot more complicated. So do you think this matters? Yeah, it matters. Anything you do that makes your code harder to sequentially reason about matters an awful lot. Remember that your first responsibility is to your readers. It is more important for you to make your code understandable than it is for you to So for that reason, I usually avoid mid-loop breaks and returns. Oh, I'm not talking about simple little loops like this that scan through an array and do an early return. I'm talking about complicated loops like this one. All those early breaks out of the loop make that loop really hard to understand. Oh yeah, keep this in mind too. If you keep your functions as small as we told you to back in episode 3, then it'll be pretty tough not to follow structured programming. After all, a four-line function always has a single entry and a single exit, and it's really, really hard to get them to have mid-loop breaks and returns. that error handling is important, but if it obscures logic, it's wrong. He's dead right about this. So how do we avoid obscuring logic? How do we handle our errors? What's the strategy we should employ? We're going to look at this problem by actually handling some errors. Let's sit down and write a stack We begin by creating a package named stack, then a file named stack test. We open up a dummy test just to prove that we can execute. We can, so we delete that test. Now we're going to write the test that creates the stack. We will use the name myStack to avoid collision with the library stack. Once we get that to compile, then we can change the name from myStack back to stack. should run, of course. Then what we'll do is we will check that the stack is

empty. We will have to implement the isEmpty function, of course. And then we've got to fiddle around with some scopes and imports and so forth. But eventually we get the test to run and fail. And we can make it pass by assigning the empty flag to true so that the stack defaults a little bit better, newly created stack should be empty. That passes, of course. And let's also check that the size is zero. That's easy to do, we have to create the getSize function, then a little bit more scoping gunk, and we'll eventually get that to pass. Excellent. Now let's make sure that we're manipulating the size variable correctly let's try pushing first, and we'll make sure that when we push the size gets incremented properly. We're going to have to refactor that stack variable. We'll do that in just a minute. There's our push. We create the function. Then we check to make sure that the size is 1. This should fail, of course, but we can make it pass by incrementing the size in push. Lovely. And let's also just check that the stack is not And of course that fails, but we can make it pass by replacing that empty flag with size equals zero and That passes how nice now. Let's make sure that the size is decremented in pop So after one push and one pop the stack should be empty again once again, we're going to refactor that stack variable make sure that it gets initialized in setup and That cleans up the tests a bit. Now we're going to run the tests. We will push and we will pop. And, ooh, got to write that pop function. There we go. And now the stack should be empty again. And, of course, this will fail, but we can make it pass by decrementing the size in pop. Excellent. It's always best if you write your error handling code before you write the rest of the code. That way you don't paint yourself into an implementation that can't handle errors well. In our case, we're worried about two errors, basically, underflow and overflow. Now we're going to do the overflow test. When we push past the limit, the stack should overflow. What does it mean to push past the limit? Well, we're going to have to set a limit. So we'll do that in the constructor of the stack. We'll make the limit 2. And then we're going to change the constructor to a factory method. We will make that a static method inside the stack class itself. the constructors properly. We will move that up there. We will implement that constructor that takes a capacity. And make it private so that it can't be called outside. Isn't that nice? Okay. Now we've got a stack with a capacity. So let's push it past the limit. That would be three pushes in a two-size stack, of course. And then, well... At this point, we could have push and pop return an error code of some kind. For example, push could return false. Pop could return null. But that reminds me too much of the horror scene of the 70s and the 80s when every function's return value was usurped by an error code. certainly don't want to go there. So I think it would be better if we threw an exception. I think what I should do here is expect an exception. So and what expectation should I expect? So what exception should we throw here? I threw stack.overflow but I suppose I could thrown subscript out of range exception or buffer overflow exception or even illegal state exception but I don't like reusing canned exceptions from the Java library when I throw an exception I want it to be scoped to the class that throws it and I want it to be named with as much specific information as possible going to expect my own exception. I like to scope my exceptions inside the class that I'm creating.

There's my exception, and it should derive from... The experiment with checked exceptions is over, and it failed. It failed in C++, didn't even try. The problem with checked exceptions is that it creates a reverse dependency, a dependency up the inheritance hierarchy instead of down. If you have a method in a derived class that suddenly throws a new exception, then you've got to go change the declaration in the base class. And you've got to recompile and redeploy all the users of that base class. This violates independent deployability. principle which we'll talk about in an upcoming episode and in general it's just a pain in the posterior so I don't want you using checked exceptions anymore from now on in Java derive your exceptions from runtime exception so let's extend runtime exception and now of course that fails because it doesn't We'll have to save the capacity from the constructor, and then in the push function, we will just compare the size to capacity. If the size is equal to capacity, then we throw the overflow exception, and that should make it pass. Let's also do underflow, because that's easy to do. We can check to make sure that if you pop an empty stack, it throws an underflow. We just pop an empty stack right there. we will create that exception, we will derive it from runtime exception, that should fail, then we will check the size in pop, and that should make it pass. Lovely. So what message do you think I should put in this exception? I generally don't put much. And in fact, that's the way I like it. I think the best case for an exception is to not have a message at all. I like my exceptions to be so precise that no message is needed. Remember, the best comment is the comment you don't have to write. Now that doesn't mean I don't sometimes put messages in my exceptions. Sometimes I do. Sometimes there's some little tidbit of information that's useful to pass along. and context and scope of the exception to do most of the work. All right, so now let's actually make this stack look like a stack. So let's try pushing a 1 and then making sure that a 1 gets popped. That's easy to do. And, of course, that will fail because, well, we're not popping anything at the moment. We're going to return a minus 1 from pop. Yeah, it's going to fail. But it's pretty simple to make that pass. We will transform that negative 1 into a variable named element, and then we will just initialize that element in the push function. So now we essentially have a stack of one element. And that pass is just fine. So the next test is the obvious one. in reverse order, 2 and 1. Simple test to write. Push 1, push 2, pop 2, pop 1. And that will, of course, fail. But it's very easy to make this pass. We convert the scalar element into an array named elements. We initialize that array in the constructor with the capacity. functions and indexing the array appropriately, fortunately we already have the size to help us and it does just what we want. We'll do that in both the push and the pop, stealing the incrementer, sticking it right in the index and voila you have your lovely passing tests. What should it do? Well, let's have it throw an illegal capacity exception. We will make that stack with a negative one, and we'll expect the illegal capacity. We'll create that exception, and of course derive it from runtime exception. And then it's a simple matter of putting that check near the constructor. How about that? Aha! We've got another error. It's possible someone might just construct a stack with zero size. a stack created with zero size exception, but we really don't want to proliferate

exceptions, do we? And besides, is this really an error, or is there a reasonable definition for the behavior of a zero size stack? does have a very well-defined behavior. If you call push, it overflows. If you call pop, it underflows. If you ask for the size, it returns zero. And so on. Now, we could probably implement this by adding the appropriate if statements to the stack code to make sure it behaved properly if the size is zero. But there's probably a better way. to return a degenerate stack that simply behaves appropriately for the zero-sized case. Okay, so let's start building that null object pattern. Sometimes this is called the special case pattern. When creating a stack with zero capacity, any push should overflow, of course. While this is easy to write, we're going to expect the overflow exception. going to create a stack of zero capacity using our factory method and then we're going to push and that should throw an overflow exception and of course it does because we're already checking whether or not the size is equal to capacity and since they're both equal to zero so what we're going to do now is split this class into we're going to extract an interface out of it and we will be left a stack interface and a derived class named bounded stack. This is a simple refactoring, extract interface, ka-bam, and it will happen. After a little fiddling, of course. There we go. There's our bounded stack, there's our stack, isn't that pretty? Now the magic is going to happen in here. Let's just check what our tests look like. Our tests are now using the stack still, the interface, but creating the bounded stack. And you notice it's creating the bounded stack everywhere that make is called, but the interface we're using in the stack test is the stack. So now in that factory method, we're going to do the magic. If the capacity is 0, we're going to return a new anonymous inner class based on the interface, to do exactly what you'd expect of a zero capacity stack. Pushing will overflow, popping will underflow, and we have a stack that is automatically zero capacity without any if statements. How lovely. Now we're going to take that anonymous inner and we're going to pull it out as a zero capacity stack, up all that space in the factory method. That still passes. So now, let's write a new feature. The top of the stack. We'd like to be able to peak at the top of the stack. So when one is pushed, one should be on top. We'll push a one, and we will expect one to be the top of the stack. We have to implement the top function, of course. And when we do so, it will put it in the interface first. And then we need to implement the the bounded stack to return elements of size minus one. That's perfect. And then we will also implement it in the zero capacity stack to, goodness, well, we'll just return a negative one for the time being. Lovely, that passes. So now let's write a new test. When stack is empty, Hmm, top. The top function returns the value that's at the top of the stack, but what should it return if the stack is empty? We could return a null, of course, but nobody who's calling the top function would expect it to return null. And nulls, especially when they're unexpected, have this tendency to slip and slide through the system pointer exception. So I think it would be better to throw a new exception entirely, a stack empty exception. So let's throw an empty exception here. We're going to ask for the top of the stack on an empty stack. We will expect an empty. We have to create that empty exception, of course, and it will derive from runtime. Notice I'm making it static. These exceptions

should all have been static. Oh, that fails, of course, because we're not doing it. So, okay, we're going to implement the function so that if the size is zero, or rather if it's empty, then we'll just throw empty. And that should pass. Yep, and notice up there there's another size equals zero we can replace with is empty, so we will. And that passes as well. Good. zero capacity stack. That'll just throw empty. Good. And let's just write a special test for that. With a zero capacity stack any pop should just throw empty. Any top should just throw empty. So we're going to make ourselves a zero capacity stack and then we'll just call top on it and it should throw that Expected clause. Yeah, works just fine. So now let's write a new feature, a find feature. Given a stack with 1 and 2 on it, find the 1. So here's the 1 and the 2. Now what we're going to be finding is the index of an element, starting with the top of stack, which is 0, and then climbing up. the stack. We have to create the find function, of course. So we'll create it, add it to the interface. That means we're going to have to add it to the two implementations a little bit later, of course. All right. And let's just make sure that we can find the top of the stack as well. So there's the top of the stack there. Lovely. Finding the two should be a zero. Yes. Now we're We'll do it in the bounded stack first, and we'll just stub it out. We'll also implement it in the zero capacity stack, and once again, we'll just stub it out. And that should make the tests fail. And it does. So let's see if we can implement the find algorithm. It's not too hard. We're just going to loop through the elements of the stack. We'll start at the top of the stack and work down. So we'll be decrementing through the stack, and we'll check to see if the element is equal to what we're searching for, and if it is, then we will return the computed index of the found element. Gee, I hope that expression is correct. I suppose our tests will tell us if they are or not. Then we're going to return a something. Oops, that doesn't work. That still doesn't work. Ah, yes, that's because I did not compute the index correctly. Okay, now it's working. Lovely. So, now let's check to see what happens when it's not found. What should we do? is a method that searches the stack from the top to the bottom, and it returns the index of the first element it finds that matches the input argument. So it returns an integer. But what should it return if it doesn't find the argument? We could throw an exception. But that would be strange, because we expect find to fail from time to time, and we usually reserve exceptions for What we need is a value that we can return from find that means not found. The Java convention is to return minus one. For example, string index of. But minus one is still an integer. It might get used in a computation. I think it's better to return null. Null is a value that means nothing. Minus one is not nothing. we're going to return a null, aren't we? Yeah, all right. We're going to return a null. And, jeez, you know, we're going to have to muck around with more of these asserts and scoping them and everything. But, okay, we can do that. And, yeah, tests that fail. Holy crike. So, I guess we'll, oh, jeez, we're going to return a capital integer. We're going to be boxing. Well, all right, we'll box the doggone things and we'll return null. Great, okay, null, return null. I just, you know, I'm just not sure that I can go on with this. This has just been going on and on, and I don't know, I just don't know. Look, look, look, I mean, we're screwing around with the tests

now. Look at that, we're trying to unbox things in the tests. I don't know, I don't know. So frustrating. Oh, crap, now we're gonna undo that. Can't seem to make up our mind. Oh, no, he's going to do infal. He's doing infal. Oh, my God. He's doing infal in the test. Oh, no, he's mistyped it. Okay, he got it under control. Oh, he's taking them away. Oh, no. No, no, no. What's he going to do? What's he going to do? Oh, cripe, he's creating variables. Variables, not variables. Oh, God, he's going to do it. Explanatory variables. And he's going to change their type. Oh, God. I just don't know. I think it's time for me to put this away now. Oh, God. One last thing. I have a rule when it comes to try blocks. If the word try appears in a function, it must be the very first word in that function after any variable declarations. single line, a function call. And the close and finally blocks are the very last thing in the function. Nothing follows them. And remember, a function is supposed to do one thing. Error handling is one thing. A function should either do something or it should handle errors. But it shouldn't do both. Come children, come children, I feel I must summarize now. Gather round, gather round. We've learned that function signatures should be small, that they should have few arguments, maybe three or fewer, yes, and that you should not pass booleans or nulls into functions. And please, please, no output arguments, no output arguments, no. We learn to organize our methods according to the step-down principle. That's right, where we have our public methods at the top, and then the methods below them that are called by them, and then so on down the chain. We don't want any backwards references, remember? We learned about switch statements, didn't we? Yes, we did. We learned that switch statements violate the OO paradigm, and independent developability. That's correct. That's right. We learned that we should replace them with polymorphism where possible or at least move them down into safe, independently deployable modules like Bane, where they can live happily, perhaps inside of a factory. That's right. We learned about some paradigms. We learned about the functional paradigm and about the structured paradigm. about temporal couplings, the horrors of them, but how to solve them by passing a block. We also learned about command and query separation and, oh, tell no task. That's right, tell no task. Oh, remember the law of Demeter children and the dangers of too much knowledge in single lines of code. They don't want our functions to know the whole structure of the system. No, no we don't, no. And we learned about structured programming. Remember that? Remember the structured programming? All that single entry and single exit stuff we talked about? Oh yes, we learned about that. And why breaks and mid-loop returns cause complications we'd rather not have to face? Oh no, we don't. We spent some time on error handling, didn't we? Yes, we did. We talked about error handling and how we should consider our error handling first part of the implementation. Oh, we said it's better to have exceptions than error codes. We don't want to be returning error codes, do we? No, no. We'd like to throw exceptions where possible. And remember, those exceptions should be scoped to our classes. Oh, yes, they should. Scope them into your classes. Don't reuse those horrible things like illegal argument exceptions and so forth. are better than checked exceptions. That's right. We don't want to have checked exceptions

anymore. We learned that the null object pattern and the special case patterns are often better than errors in the first place. And we also learned something else. The null value is not an error. Now, go children, go in peace, and consider these things wisely. so that was quite a ride and I hope you enjoyed yourself and maybe even learn something but there's a lot more for us to be talking about we aren't even close to being done I mean what about comments what about code formatting and what's the difference between data structures and objects and then there's the whole topic of tests and test driven development and object oriented principles don't miss the next exciting episode of Clean Code, episode five, Form. Come on dog, let's go. Yeah, yeah, good dog, good dog, let's go. Yeah, that's a good dog, good dog, good dog. Hey, Vicky? you done Inside of We learned that the null object and special case patterns are often better than passing the... Holy shit. Did I just blow that whole thing? Wow. Plugins. We took a very brief look at the object oriented paradigm in the... Blah blah blah blah... Paragrime, later, paradigm... Blah blah blah blah blah... Is this a video? Nikki? You have the camera off, don't you? No. Is Nikki here? No.