Hi, I'm Uncle Bob and this is Clean Code. Welcome to Episode 5, Form. we learned about function structure. We learned that function should have no more than three arguments, none of which should be Boolean. We learned about how to organize the methods within our classes using the step-down rule. We learned a lot about switch statements. We learned why they weren't OO and we learned why they impede independent deployability. We also talked and about functional programming. We learned about the techniques of command-query separation and tell-don't-ask. And finally, we learned a lot about error handling. We learned how to use exceptions and how to employ the special case pattern. Now, in this episode, we're going to talk about issues of form. For example, comments. How many comments do you think you should write? Where should you put them? When should you write them? And should you have a tool like CheckStyle and force the placement of comments? How long should a file be? How wide should a line be? How can you use whitespace to lay out your code properly? And what about indentation? What is a class? and what does it expose you to? Should the variables in your class all be private? Should you write getters and setters for all of them? What's the difference between a class and a data structure? What would you use a data structure for? Does a data structure have methods? What things does a data structure protect you from and what risks does it expose you to? What are boundaries in software projects? How should those boundaries be crossed? For example, databases usually live in a different partition from the application. So should the application know about the database schema? Should the tables in the database correspond to classes or domain objects in the application? and other partitions like the database. These are the questions that we're going to be addressing in this episode of Clean Code. So if you're ready, hold on to your hats because we're about to deal with the question of form. Six billion years from now, our sun will begin to die. As befits a stellar entity, Saul's death throes will be long, violent, quite beautiful, and very lethal to the four rocky worlds that are nestled within its warmth. indeed if they are to save her from Saul's agonal cries but before we talk about the death of Saul let's first talk about his birth six billion years ago when the universe was just half as old as it is now a giant star blew itself to smithereens the shockwave from that event eventually slammed into a cloud of spurring it to collapse. New stars were formed out of that collision. Our son, Saul, was just one of many brethren. Like any child, Saul awoke with a squall. He cried forth a huge stellar wind that cleared away all the dust and gas of the cloud that spawned him. Saul's natal breath. After that, Saul settled down into a calm and consistent lifetime of nuclear burning, converting hydrogen into helium. The helium accumulated gradually in his core and, to compensate, Saul grew both brighter and hotter. In fact, Saul is 20% hotter today than he was when hotter and hotter. In a few hundred million years he'll be so hot that the oceans will start to evaporate. The earth won't be able to tolerate it. If we're still here by then, perhaps we will launch some giant shades into orbit to block the solar radiation and keep the oceans from drying out. If we're gone and no one has taken our place, then the earth will dry out, it will wither, and it the Sun will live on getting warmer and hotter with every passing year eventually the icy moons of Jupiter will begin to thaw and Mars while Mars will grow warm like summertime perhaps some of us will go there for a while to live in relative comfort but the helium ash in the core of the Sun will continue to grow eventually all of the hydrogen in the very center of the core will be nothing but helium. Some hydrogen around the outside of that helium will continue to burn in a shell ever increasing the pressures and temperatures that the helium experiences. It takes a lot to get helium to fuse, but once a certain line is crossed, the helium responds with a vengeance. One day, six billion years from now, that line will be crossed oxygen. But there's a quirk of quantum physics known as degeneracy that will prevent the energy released by that fusion from increasing the pressure in the core. And that means the core won't be able to expand and cool, so the temperatures will skyrocket and with them the rate of helium fusion leading to a runaway reaction known as the helium flash. In just a few seconds, the energy output of the core will skyrocket by a factor of about 100 billion. The temperatures will soar to the point where degeneracy is lost and normal pressure can exert itself. That pressure will cause the core to expand, and as it expands it will cool to a mere 100 million Kelvin where stable fusion of helium into carbon and oxygen can commence. The energy produced by this catastrophe and the subsequent helium fusion is enormous. It will heat the sun and the sun will grow by a factor of about a hundred. It will devour Mercury, gobble up Venus, and it will just brush the surface of the Earth. Unless we move the planet maybe out to Jupiter or Saturn, the Earth will melt, then vaporize. bones. For a billion years, the sun will continue to grow and then shrink and then grow and then shrink, oscillating in a tortured dance of death. With every pulsation, it will shed away some of its mass into space, surrounding itself with its own funeral shroud. But that dance of death will final oscillation and then the Sun will begin to collapse for a while the heat of that collapse will cause it to glow in the ultraviolet for 10,000 years it will glow so brightly in the ultraviolet that the shroud around it will begin to glow and will become one of the most beautiful sights you can see in the sky a glorious planetary nebula a funeral pyre befitting the majesty of the that gave us life. But the star will cool and the UV will fade. The shroud will dissipate into the vacuum of space, leaving behind the corpse of Sol, a cinder of carbon and oxygen, a white dwarf star glowing with the heat of its formation but gradually cooling. For a billion years, it will be a brilliant gravestone, Of course, a white dwarf star generates no heat of its own. So after billions more years, it will cool down to the temperature of space itself. It will have become a frigid, degenerate ball of carbon and oxygen, an ignoble fate for something so great as a star. Interesting. We're going to have to talk about that in another episode. Excuse me. It's time for us to have a frank conversation about comments in our code. What's their purpose? When should we write them? In every organization above a certain size, there eventually emerges the bureaucratic urge to create a document. This document specifies, dictates, and mandates what every jot and tittle of the code ought to look like. That document is called the coding standard. in a coding standard. I think every team should have one, and every developer on the team should follow it. But I don't believe the coding standard should be written down in a separate document. I think the coding standard should be clearly visible inside the code itself. The code should be the coding standard. Indeed, if you're forced to write a separate document, it's probably because your code is not a good showcase for your standards. is implicit in our code or written down in a standalone document, it often tells us that we must document every variable, function, class, and every block of code with some kind of a comment. This is silly, officious nonsense that leads to code clutter and dilution. When comments are that frequent in our code, we tend to Hmm. When programmers are forced to write comments, when comments are mandated, then programmers write them because they have to, not because they need to. Those comments become vacant, vacuous, vapid skeletons with no meat on their bones. They become something to ignore. Our eyes flit over the comments in search of real code. We might even tell our IDEs to color them in some kind of muted gray so that they're easier to ignore. When comments are common, they become like the boy who cried wolf. They become worthless. Comments should be rare. They should be reserved for those special cases where programmer attention is really necessary. colored bright red so that readers can't miss seeing them. And every programmer who reads them should be grateful and relieved that those comments are there. But you don't have to take my word for this. Let's hear what Kernighan and Plogger said back in 1978 in the Elements of Programming Style, second edition. I'll read the first paragraph from chapter 8, documentation. Get a load the best documentation for a computer program is a clean structure it also helps if the code is well formatted with good mnemonic identifiers and labels if any are needed and a smattering of enlightening comments flowcharts and program descriptions are of secondary importance the only reliable is simple. Whenever there are multiple representations of a program, the chance for discrepancy exists. If the code is in error, artistic flowcharts and detailed comments are to no avail. Only by reading the code can the programmer know for sure what the program does. write code that expresses its intent so well that it doesn't need comments. If we adopt that goal, then every comment we write is a failure. Is it possible to express intent in code? Some people think that it's not. Some people think that code is just the wrong mechanism for expressing intent. Some languages, like assembly language, aren't expressive at all. Comments in these languages are absolutely essential. You can't understand the code without them. Even languages like Pascal and Fortran and to some extent even C are so syntactically challenged that writing expressive code without comments is very difficult. early 90s quite a bit of code was written in severely resource constrained environments when cycle times are long and memory is in short supply then it can be very difficult to be expressive the resource constraints dominate the structure of the code but our modern languages like Java C sharp and Ruby are remarkably expressive I mean we've got a whole bunch of interesting syntax elements we can use like classes and nested classes and namespaces, enums, limitless names. These are very powerful things that we can use to express our intent. And our processors are so fast and memory is so cheap that any minor inefficiencies created as ignored. Nowadays there is no excuse for not making your code as expressive as possible. And that means that every comment you write is a failure to express yourself well in code.

Every comment is a failure. Don't get me wrong, I do write comments. It's just that I don't congratulate myself for doing it. I express myself well in code. Oh, I try to figure out ways to avoid writing comments. I'll try and refactor the code any way I can to make the code more expressive. But in the end, there are times when I fail. I can't figure out how to make the code express itself well and cleanly. Why am I so down on comments? Well, I don't like clutter and I don't like lies. And comments tend to be both. It's very difficult for comments to remain truthful for any length of time. The author was probably trying to tell the truth when he wrote the comment, at least in so much as he understood the truth. into misinformation and lies you might think that this is just a discipline problem that if you could just get the programmers to work a little bit harder they would keep those comments up to date but that's not really the issue I mean comments don't rot just because programmers When you change a line of code in some module in order to fix a bug or or add a new feature, how do you know that there's not some comment somewhere else in the code that you just invalidated? I mean that comment could be at the top of the class you just changed, it could be in another file in the same module, it could even be in one of the users far away. There's no good way to know. And so Comments must rot. They gradually become lies that do more harm than good. Oh, not all comments are bad. I mean, sometimes comments can be pretty darn useful. Sometimes they're required. Let's look at a few cases where some comments are actually worthwhile. to be there no choice so jam them up at the top of the file better yet write some script that automatically puts them up at the top of every source file and automatically updates the dates better yet get an ide that does it for you informative comments okay there are some informative comments that are pretty useful for example if you've got a comment that describes or some other weird arcane syntax, maybe some sequel or something that you need to describe, fine, fine. Comments like that can be worth their weight in gold, especially when you consider that comments don't weigh anything. Clarifications and Explanations of Intent When you must write a comment to explain your intent, it means that you have failed. it is best not to leave the code unexplained therefore write the comment and then do appropriate penance warnings of consequences yes it's true a comment can be used to warn you about bad things to do comments? Am I reading that right? What? Who? What nitwit put to do comments into this script? I thought this was supposed to be a show for professionals for cripes sake. Holy cripe. Cut, cut, cut. Get this off the script. Public API documentation. Nothing can be quite so useful as a well documented public API. If you're using a tool like Java doc or some other automated documentation tool, you should certainly write nice documents for your public APIs. Of course, you should also make sure that your function signatures are self-explanatory. And keep in mind, the best public API documentation is the documentation you don't have to write. Don't talk to yourself in the comments of your code. Don't write about the lyrics to the song you're listening to, or your life situation, or how your co-workers have bad breath, or they write the system the wrong way. We don't want to hear it. Just shut up and code. Redundant explanations. it adds something new. Don't just restate the code. Don't tell me the algorithm if the algorithm is perfectly clear from the code. Please don't tell me what a variable holds if it's perfectly obvious from the name of the variable. Remember the DRY principle. Don't repeat yourself, not even in a If you have a build system that uses a tool like Checkstyle to mandate stupid and redundant comments, then turn it off. If it's controlled by someone else who refuses to turn off the requirement for stupid and redundant comments, then gather your friends together, paint your faces blue, make cards go to their desk and start chanting in unison, all we are saying is give code a chance. All we are saying is give code a chance. Get them to turn that stupid check off. What I'm about to say may seem a little bit silly, but I have to say it just because I've seen so much code that has it. or misleading either fix it or delete it I know that sounds obvious but I've read an awful lot of code where the programmers were apparently not willing to do that get rid of misleading erroneous comments please journal comments you do have a source code control system right well then use it Don't clutter your code with a bunch of comments that are better handled by the source code control system. If you've got a big load of journal comments in your source code, delete them and don't add any more of them. Nobody cares that JPG turned an int to a double on March 21st, 1992. Noise comments. come for you in the cold and dark of night. Position Markers and Big Banner Comments it are never read. Nothing shouts ignore me more than a big banner comment. Why? Because we so often see big banner comments that call attention to things that we'd rather not pay attention to. For example, a great big banner comment that says, here are the default constructors. 80s when that made sense I mean we used to put comments at the end of every closing brace to let you know what block was being ended but nowadays our IDEs are just too good at it they're the ones that make sure that we're closing our braces and our parentheses appropriately so there's really no need for these closing brace comments anymore and please don't use them Is it really necessary for you to put your name on each line of code? You know, JJQ did this? The source code control system will remember this. You don't have to put your initials on every line of code. If we want to figure out who to blame for something, we can go to the source code control system. You don't have to sign every line of code you write. Kilroy was here. HTML in comments. Yeah, I know, we want our documentation tool to automatically print a nice document, and so it's kind of tempting to put HTML into the comments so that certain words are italicized and certain words are bold. But look, this documentation is in the code, most readable. HTML in the code obscures the comments in the one place where they should be most readable. Please don't put HTML in comments in code. Non-local information. Comments that talk about parts of the code that are far away will rot and rot quickly because the code to know that that comment exists. So when you write a comment don't talk about code that's far away. If you must write a comment make sure that comment sits right next to the code it describes. is good about software. When you see commented out code, you must delete it, expunge it from the source code. Don't read it, don't touch it, don't try to understand it. Simply cut it out of the source code. Remember, you won't actually be using it because you do have a source code control system, don't you? Have you ever seen commented out code functions that are so old they don't exist anymore, or variables whose names have changed, or they've been moved to a different place in the code, maybe a different class. This code is so old it's got no chance of ever executing again. Every time I see commented-out code, I delete it on the spot. I don't read it, I don't try and understand why it's there, existence and you should do the same. Remember, the code's not really going to be lost. Your source code control system is going to remember it. So really, you don't have to hold on to it. There's no point in having it there. No one's going to miss it. It doesn't need to be sitting there in the code getting less and less relevant with every passing day. Rather than using comments to describe what your code does, learn how to use explanatory variables and names. Make your code read like well-written prose by choosing the correct parts of speech for your names and composing readable sentences in your code. episode called Lycral Numbers. And the functions that we're scrolling past right now are tests. Notice that these tests are filled with explanatory functions. You don't see the typical assert equals, assert true, assert false littering the tests. Instead, you see a set of assertions you haven't watched the screencast, but you can still tell that those are facts. Does not converge, is palindrome, is not palindrome, reversed. These are all explanatory functions inside the tests that make the tests much more readable. You can see the reversed function here is defined. It calls assert equals finally, but I tend to bury all the asserts in these nice little helper application code itself. And you'll see that it's also full of explanatory functions and explanatory variables. Nice functions like converges at iteration, private functions like converge. These are nice little utilities. Notice the functions are relatively small, not a lot going on inside them. You can see some explanatory variables there, the context of this application. Another nice explanatory method is palindrome. So you can see that in this screencast we're using plenty of nice explanatory variables and functions. discipline is important. I hate reading code where someone has strewn white space around with reckless abandon. White space carries information, and so we should use it with the same level of care as any other structure in our source code. Remember, when someone first looks at our code, we want them to be impressed by the attention to detail, struck by how orderly it is, convinced work and the very first thing they're going to see when they look at our code is the formatting. Formatting is about communication and communication is the first order of business for every programmer. Remember getting your code to communicate is even more important than getting your code to work. How big should some data that you might find interesting. First of all, note that the vertical scale is logarithmic, so small changes have huge consequences. In this box plot, we have seven programs represented. The vertical lines represent the extremes in the file size. The boxes in the middle represent the standard deviation, and at the center of each box is the mean Notice how similar JUnit and Fitness are. Both of them have maximum file sizes of about 500 lines. Both of them have average file sizes of about 50 or 60 lines. And both of them have most of their files in between about 30 and 100 lines. The striking thing about that is that the two projects are so vastly different in size. Fitness, at the time of that measurement, was about 8 times larger. That implies that project size and file size aren't related to each other. That big projects do not imply big files. Take a look at JUnit and JDepend. These two projects are roughly the same size, and yet the average file length of JDepend is twice that of JUnit. size are not strongly correlated. On the other hand, take a look at Ant and Tomcat. These projects are much larger than the others, and they do seem to show some correlation between file size and project size. The average file size inside of Tomcat is 200 lines, and it's not at all uncommon to see a file of 500 lines. after all? Well, perhaps, but there's something else you should know. The projects on the left side, they were all written with test-driven development as an overriding discipline. The further to the right you go, the less that's the case. In any case, fitness shows that where most of the files are less than one or two hundred lines and that the maximum file size is about five hundred lines it seems to me that this is a reasonable and desirable standard now look we've all seen large files that have become the dumping ground for stuff that we haven't taken Keep your file sizes small. Use blank lines to separate things that should be separated, like methods. And don't get into the habit of banging the return key a whole bunch of times. Be disciplined in your use of blank spaces. Decide how many lines should be used to separate methods from each other and stick to that decision. use one line between methods. I also use one line to separate methods from variables. And if there are more than one kind of variable, like public constants and private variables, I'll use a blank line in between them too. Inside a function, I will use a blank line to separate variable declarations from the rest of the executable code. I'll also put and while loops separating them from the code that follows of course when you keep your functions as small as I do you don't need very many of those blank lines so let's take a look at how we deal with blank lines in the screencast of the lycra numbers that's associated with this episode you can purchase it separately if you'd like we're gonna deal with some blank line issues here For example, way up the top, there's some lines there that probably ought to not be there. The imports are separated by blank lines. I'd like to group them together because they're related, obviously. Scan down a little bit, see if there's any blank lines there. That one's a nice one, separates two different topics. Let's see, there's a wasted blank line. Let's get rid of it. I don't want that. Now let's switch over to the application side. Looks like a bunch of nice little small functions. Not an awful lot of blank lineage in here. But as we scroll down, we'll see some opportunity. Notice that there's some variables there, like end digits and R digits and last index. And then there's a nice blank line separating the for loop from the return statement. And it looks like the same thing there. We've got a couple of variables. Blank line. We could use a blank line in between those two. That makes it a little nicer. Separate the return from the loop. Ow! other should be grouped together. If the variables are used together, then group them together without blank lines between them. The general rule is pretty simple. Things that are related to each other should be vertically close to each other. The distance between them is a measure of how closely related they are. Things that aren't related to each other should be How long should a line of code be? I have a very simple rule about this. You never have to scroll right to see it. Horizontal scroll bars are demon spawn. Manage your line lengths so that you never have to scroll to the right. Of course, screens are getting wider all the time, and pixels are getting smaller too, cram 200 characters on a line, clearly that would be bad. Oh yes, that would be bad. So, let's look at some simple statistics. This is a histogram that shows the frequency of line lengths in the same seven applications that we looked at before. Once again, the vertical axis is logarithmic. The curves are almost identical. These programs are using the exact same frequency distribution of line lengths. It's as though the frequency of line lengths is some kind of constant of the universe. Tiny lines predominate at first. These are probably blank lines and braces and tiny little assignment statements. There's a local minimum at about 10 characters. And then after there's this nice wide hump between about 25 and 45 characters this is where the vast majority of the lines congregate after that the frequency falls off very rapidly remember this is a log scale I think a reasonable set of conclusions we could draw from this is that first of all we longer than about 80 characters. You think that's a coincidence? I think it's a reasonable rule to suggest that lines should be less than a hundred or a hundred and twenty characters. You shouldn't get there very often by the way. But if you do, our screens probably are good enough nowadays that they won't make a horizontal scroll bar. Remember, never make your readers scroll right. indentation oh no you don't don't get me into that you guys sit down and figure that out for yourselves and you leave me the hell out of it so look I'll just say this I don't care what indentation style you use I don't care if you use tabs or spaces use spaces and I don't care whether your four or eight characters wide. Use two. And I don't care where you put your braces. Use the K&R style. What I do care about is that everybody on the team use the exact same style. The code that comes out of a team should look like the team wrote it. I shouldn't be able to tell who You guys need to sit down and decide which indentation style to use and then be grown-ups about it. Everybody in the team just uses that style. And I don't want to see people reformatting the code to their personal style every time they check it out and then re-reformatting it back to the team style when they check it in. I also don't want the reformatter run automatically on each check-in. with the source code control system, this is just sloppy. Choose a style and then use it. Now, I don't mind if you use the IDE to reformat bits of your code, as long as you reformat it in the team style. Maybe your IDE's got a configuration file that describes how to apply indents. That's good. Make sure that every team member uses the same configuration file. IDE, do so in small snippets. Don't reformat the whole file, because that wreaks havoc with the source code control system and makes merges a nightmare. You are using a distributed source code control system like Git, aren't you? What is a class? class by writing private variables and then you manipulate those private variables with public functions so from the outside looking in a class appears to have no variables at all it's just a bag of functions from the outside looking in you can't see those variables so from the outside looking in a class it's also true that an object appears to have no variables. To say this differently, from the outside looking in, an object appears to have no observable state. No observable state? You might object to this observation by pointing out that most classes have getters and setters, or in some languages, mutable properties. respond to that objection like this. If you take the private variables of your class and you expose them to the outside world through getters and setters, or mutable properties, then you've got a bad design. After all, why would you make variables private and then just expose them through a set of getters and setters? Remember our previous episode where we talked about the discipline of If an object has no observable state, then although it's easy to tell the object to do something, it doesn't make a lot of sense to ask it anything. An object that follows the discipline of tell-don't-ask will probably not have very many getters. And if it doesn't have many getters, there's probably not much point in making it have any setters either. The methods of a class manipulate the variables of that class. The more variables within a class that a method manipulates, the more cohesive that method is. A maximally cohesive method manipulates every variable inside the class. A maximally cohesive class is composed of nothing but maximally cohesive methods. Getters and setters are not very cohesive because they only manipulate a single variable each. The more getters and setters a class has, the less cohesive that class is. So does that mean that a class should never have getters and setters? Dogmatic rules like that seldom apply in engineering disciplines. I certainly write getters and setters from time to time, In those instances where I choose to have a getter, I don't simply expose the variable. I try to abstract out the information being retrieved. Here, let me show you what I mean. So let's say that we've got a class named car. It represents an automobile. And let's say that of gas and this variable holds the fuel level of the car. Let's further suggest that we want to expose the contents of that variable to the outside world. What should we call that getter? Well, we could call it get gallons of gas, but that exposes an awful lot of the implementation of our class. that holds the gallons of gas. There are a number of problems with being so specific. For example, let's say we wanted to create a derivative called diesel car. It would of course inherit the method get gallons of gas, but that's not really right, is it? I mean, diesel cars don't run on gasoline. a derivative called electric car. We've got a method named get gallons of gas in the base. That derivative, electric car, just doesn't fit with that base. When derivatives don't fit with a base, something's wrong with the base. Often that's because the base exposed some implementation that it shouldn't have. For example, instead of creating get gallons of gas, we could have used method named getPercentFuelRemaining. This method would have worked just as well with a car or an electric car or a diesel car. In fact, it would even work with Doc Brown's plutonium driven nuclear car. Remember Back to the Future? Can you smell the polymorphism coming? That's the advantage to hiding your internal variables. the more opportunity you have to make polymorphic classes. That car class with the get gallons of gasoline method could not be polymorphic to electric cars and nuclear cars. But once we abstracted out all that implementation, then suddenly polymorphism became possible. As we learned in the last episode, polymorphism is the key to independent deployability To be more specific, polymorphism allows us to protect client code, like car driver, from changes to the implementation of server code, like car. It's pretty easy to write a car driver that can drive any derivative of car without knowing or caring what that derivative is. drive Doc Brown's DeLorean. And so, from the outside looking in, a class is nothing but a set of methods. Those methods may operate on data, but they tell you nothing about how that data is implemented. These methods are not getters and setters. They hide the data, they don't expose it. But maybe you're dubious. Maybe you're wondering about classes like Employee that clearly have methods like GetName and GetAddress that expose the data within them. Maybe you think I'm just being academic or dogmatic or a little too extreme. Okay, but let me ask you a question. What do you think the difference between a class and a data structure is? A data structure is kind of the opposite of a class. A data structure has a whole bunch of data variables that are public and virtually no functions. Do you see the difference here between a class and a data structure? Classes have private variables but public functions. It might be going a bit too far to say that data structures have no methods. Data structures can have methods, but typically they're simple things like getters or setters or little navigation aids. The methods of a data structure manipulate individual variables. They don't manipulate cohesive groups of variables the way the methods of a class do. expose implementation. They don't hide it and they don't abstract it. You can't tell a data structure to do anything. All you can do is ask it questions. The software that manipulates a data structure is the antithesis of tell don't ask. You can tell an object to do something generic like print itself on the screen and it will appropriate to its type. It will polymorphically dispatch to the appropriate method. But you can't ask a data structure to do anything generic. The only thing you can do with a data structure is ask it very specific questions. If you want to print a data structure on the screen, then you have to ask it for its type and then dispatch to the appropriate function yourself, probably with a switch statement. Data structures and switch statements are related in the same way that classes and polymorphism are related. When you see a switch statement, you can be pretty sure there's a data structure lurking somewhere behind it. In the last episode, I told you that we didn't like switch statements and that they weren't object-oriented. The last part of that is perfectly true. They're not object-oriented. dislike switch statements quite as much as I implied. You see, data structures and switch statements offer a protection just the way objects do, but it's a very different kind of protection. Remember in the last segment we said that the polymorphism of objects protects your client code from new kinds of server code. In other words, objects protect you from new types. in the last episode, switch statements expose us to changes in types. Whenever a new type is added, all the switch statements have to be modified and all the client code is impacted, breaking independent deployability. But objects aren't immune to all kinds of changes. What happens if you add a method to a base class? When you do that, all the clients of the base class are affected and all the derivatives of the have to be recompiled and redeployed. When you add a method to a base class, you break independent deployability. Data structures and switch statements, on the other hand, are immune to the addition of new functions. When you add a new function, all you have to do is add a new switch. Nothing else needs to change. Nothing else even needs to be recompiled. that represent shapes, data structures named circle, square, triangle, rectangle, that kind of thing. And let's also say that I have a set of functions that operate on those data structures by taking lists of them. So I've got a function named drawAllShapes that will iterate through a list of shapes and draw them all. I've got another function called eraseAllShapes, one called rotate all shapes another one called drag all shapes if you looked inside those functions you'd expect to see switch statements selecting amongst the various shape types but what if I add a new function like rotate all shapes of course you'd expect it to have a switch statement in it but adding that it have an effect on any of the existing data structures. So adding a new function to data structures and switch statements does not break independent deployability. Notice how these two schemes are the opposite of each other. Classes protect us against new types but expose us to new methods. Data structures protect us against new methods but expose us to new types. question. Is there any way to get protection from both new methods and new types? This is called the expression problem, and there are good solutions to it. We're going to explore some of those solutions in a future episode in the series on design patterns. For now, the key to independent deployability is to know which form to use and when. We use classes and objects when it's types that are more likely to be added. We use data structures and switch statements when it's methods that are more likely to be added. In the last episode, we talked about how to separate main from the rest of the application. side and the application on the other. We also talked about how all the source code dependencies should cross that boundary going in one direction away from main and towards the application. This makes main a plug-in to the rest of the application and this is very good advice but it's a specific The code on one side of a boundary is very different from the code on the other side of the boundary. The boundary that separates main from the application is just one example of such a boundary. Two other examples are the boundary that separates views from models and the boundary that separates the database from domain objects. abstract. For example, main is concrete whereas the rest of the application is abstract from main's point of view. Whenever you have a boundary crossing like this you want all the source code dependencies pointing away from the concrete side towards the abstract side. For example, consider the boundary between the domain objects and the database. The database is concrete, the Therefore, you want all the source code dependencies pointing away from the database towards the domain objects. In other words, the database depends on the domain. The domain does not depend on the database. For years and years, design experts have been telling us that we should design our applications so that they are separated from our databases by some kind of interface layer. advice. The last thing we want to see is a bunch of SQL code smeared through our application code. We especially don't want to see SQL code in our views. That's a capital offense and I will find you. Keep your SQL sequestered nicely into the layer between the application and the database. It should be clear that depending on the database. What's often not well understood, however, is that the application should not be depending on the layer. Remember, the application is abstract, the layer is concrete, so the layer should be depending on the application. For some of you this may seem absurd at first, but remember, object oriented design allows inverting the flow of control. That means that the application can still call the database layer even though it doesn't know that the database layer exists. But if the application doesn't depend on the layer, then the application can have no knowledge about the database at all. It doesn't know about the table names or the column names But consider this. A database table is a data structure. It has exposed data and no methods. That means it's the opposite of a class. Database tables are so concrete, they never have any chance to be polymorphic. And so now we come to the age-old, very famous issue of the impedance mismatch between relational databases and object-oriented programming. The mismatch is a mismatch because a database row is not an object. It's the opposite of an object. It's a data structure. Your databases do not contain domain objects. They don't contain business objects. They don't contain any kind of object at all. contain data structures and you cannot force a data structure to be an object. But what about object relational mappers like Hibernate? I think these tools are lovely. I think they serve a wonderful purpose but they are not truly object relational mappers because there is no direct mapping between a database row and an object. One is a data structure the other is an really are our relational table to data structure mappers hibernate for example is a wonderful tool for taking a data structure in the database and moving it into a data structure in memory what we'd like on the application side of the boundary our domain objects the methods of those domain objects will be business rules now it turns out that when you gather a bunch of business rules into domain objects, then you get classes that surprisingly don't look very much like the database tables or the database schema. This is the true impedance mismatch between relational databases and object-oriented design, and it has a very simple explanation. Most databases are designed for the enterprise and not for a particular application. the security and performance of that enterprise. Specific applications might desire to have different schemas, but instead they must conform to the combined needs of the enterprise as a whole. On the application side of the boundary, we can separate ourselves from the enterprise schema by actually designing the objects we'd like to use. and hidden data. This will make the application much more natural and easy to understand. Rather than manipulating table rows, we'll be manipulating business objects. It's the responsibility of the database layer to convert back and forth between the data structures that live in the database and the business objects that the application wants to use. Furthermore, the layer's got to do this without letting the application know because remember all the source code dependencies must cross the boundary from the database side towards the application side that means that on the application side of the boundary we'd like to see a set of interfaces that declare data access methods we would like our business objects to use those interfaces to access the data they need on the other side of the boundary we would classes that derive from those interfaces and implement the data access methods by interrogating data structures that have been fetched out of the database. We could use an ORM tool like Hibernate to fetch those data structures out of the database. And so, nothing up my sleeve, the layer depends on the application using a bunch of inheritance And that's exactly what we wanted. Did you notice the separation between data structures and classes? This is pretty typical of boundary crossings. Often the more concrete side will be composed of data structures that use switch statements. Of course we saw this before in the previous episode. Remember the main partition was composed of concrete factories that used switch statements? Views are concrete, and so the dependencies that cross the boundary between the application and the views should point towards the abstract domain objects and away from the views. The views should know about the application. The application should know nothing about the views. This rule of boundaries is fundamental to good object-oriented design. It's fundamental to good software design. boundary should point towards the abstractions and away from the concrete side. This is just one aspect of a principle called the dependency inversion principle, which we're going to be studying quite a bit in future episodes. Oh good, glad you're here. Come on in, come on in. a lot of stuff to talk over. Comments. Every time you write a comment you've failed. Oh you may have to write them every once in a while but keep them rare. I want you working hard to find ways not to write those comments. Comments should never be mandatory. And remember this, commented out code is an abomination that must be destroyed before it spreads. White space discipline is important. Don't be sloppy with your blank lines. Know when you're going to use them and stick to that reason. Be consistent. Remember, horizontal scroll bars will not be tolerated, so keep your line widths under control. They should average about 40 characters or so and should never exceed 120. And keep your file sizes under control too. Average file size should be less than about a hundred lines and should almost never exceed 500. Classes are bags of functions that hide the implementation of the data they manipulate. Use tell don't ask to avoid writing getters and setters. When getters are absolutely necessary make sure you hide the implementation of the data by abstracting it. And use classes to protect yourself from new types but not new functions. Data structures are bags of data with no cohesive methods. Don't put business rules into data structures. Manipulate them with switch statements if you must, and use them to protect yourself from new functions, but not new types. Boundaries crisscross our applications, separating things that are concrete from things that are abstract. dependencies cross those boundaries pointing away from the concrete stuff towards the abstract stuff remember that domain objects and database tables are not the same thing and aren't even strongly related separate your domain objects from your database by putting a layer in between them and remember that that layer should depend upward on the application and downward on the database the and should definitely not know about the database. And so we've come to the end of yet another fun-packed episode of Clean Code. But we haven't even scratched the surface yet. I mean, what about design principles and design patterns and unit testing? And then there's things like behavior-driven development and acceptance testing and the whole topic of agile methods. episode of clean code episode 6 test driven development my dog let's go yeah there you go we will fit the shroud will dissipate into the vacuum space leaving behind Switch statements and... You are using a distributed source code. It was polymorphous. Blah, blah, blah, blah, blah, blah, blah. Oh, man, this is awful. Okay. Oh my god. Oh my god. The boundary that separates Maine from the application. Okay. Nah. That was silly. Okay. New kinds of implementation code. Blah, blah, blah, blah, blah, blah. Software systems are criss-crossed by boundaries and... Hello? Hello? Did you come in to record with me? Come on. Come on. Yes, that's the green animal. Okay. That is the green animal. Yes, I'm right. Okay. Yes, yes, yes. I'm right.