Hi, I'm Uncle Bob and this is Clean Code. Welcome! Welcome to episode 8 of Clean Code. The first in a series of episodes about the solid principles. Come on in! Come on in! In episode 7 we discussed use cases, architecture, and high level design. Many of the SOLID principles were silent players in that episode. We used them, but we didn't name them. In that episode we learned that architecture is the shape that a system adopts in order to meet its use cases. We learned what use cases are and what they're not. the system. We learned that while model view controller may be an excellent architecture for user interfaces, it's not a particularly good application architecture and should not be visible at the highest levels of the system. We learned that user interfaces, databases, and frameworks are details to be hidden. They're not the central abstractions of our architecture. We should think of them Plugins that can be quickly and easily changed. We learned that we can achieve these goals by creating boundaries that separate the application from external details like the database, the user interface, or frameworks. And then we manage the source code dependencies that cross those boundaries so that they all cross in a single direction pointing towards the application. details should depend on high-level policies. High-level policies should never depend upon details. That statement is actually one of the solid principles, the dependency inversion principle, which we'll be learning about in an upcoming episode. Finally, we learned that successful architectures allow us to defer decisions about user interfaces, databases, and frameworks for as long as possible. the number of decisions not made. In this episode, we're going to lay the foundation for the solid principles by studying the old issue of code rot that we explored back in episode one. But this time, we're going to do it in a lot more detail. We'll talk about why code rots, the form that rot takes, and the mechanisms behind that rot. rot and the cost of that rot to the project and to the enterprise as a whole. We'll talk about what software design is and the best ways to represent it. We'll discuss the roles that UML and source code play in the proper expression of software design. We'll identify a set of design smells, symptoms of bad design, that all developers should We'll do a deep dive into the history of object orientation. We'll create an unambiguous definition of what object-oriented means. And we'll show how OO is really all about managing dependencies. Finally, we're going to study dependency management. And we'll look at how the SOLID principles help us keep all the source code dependencies in a software system under control. So, lock and load, brothers and sisters, because we're about to storm the outer gates of the solid principles. In 1905, Einstein showed that Galileo's ship of relativity could safely sail on Maxwell's that the speed of light is constant to all frames of reference. But what precisely does that mean? It means that no matter how fast you are moving relative to the source of a light wave, you will measure the speed of that light wave relative to you at 299,792,458 meters per second. measure the speed of a light beam as it passes you. Let's say you also have a flashlight. Take that flashlight, point it at your device, and it will measure the speed of that light at 299,792,458 meters per second. Now, get onto some railroad tracks. Stand in front of a train as it's barreling down to measure the speed of the light from its headlamp as it's coming towards you. And your device will read 299,792,458 meters per second. Now

jump out of the way as the train passes, then get back on the tracks and measure the speed of the tail light as that train runs away from you at 100 meters per second. 799,792,458 meters per second. You can repeat this experiment as often as you like. You can do it in a car, you can do it in a jet plane or a rocket ship. You can use the Earth whipping around the Sun or the Sun plowing through the galaxy. It doesn't matter. Every light beam you measure, regardless of how fast the source of that light beam is moving, will move relative to you. at 299,792,458 meters per second. And every other observer who measures those light rays, regardless of how fast they're moving relative to you, will measure them moving at 299,792,458 meters per second relative to them. ray passes you it's going at 299 million 792 thousand 458 meters per second no matter what now imagine that I have a clock this clock is constructed out of a tube with mirrors at either end there's a beam of light bouncing back and forth between the two mirrors the clock ticks every time the beam of light hits one of then the clock ticks at a rate of once per nanosecond. Imagine that both you and I have such a clock. You are in one spaceship and I'm in another. We're both moving fast. You're moving towards me. From my point of view, I'm stationary. I look at my clock and it's ticking at one tick per nanosecond as usual. But your moving relative to me. The light beam must move at an angle in order to hit the two mirrors. That means the light beam is moving a longer distance. And so your clock ticks slower than mine. You don't see that at all. From your point of view, you're stationary and I'm moving. You look at your clock and you see the light bouncing at one tick per nanosecond. But when taking longer than one nanosecond per tick. From my point of view, your clock, and therefore your time, are running slow. From your point of view, my clock and my time are running slow. How slow? Well, by applying the Pythagorean theorem, it's pretty easy to show that I will see your time running slower by a factor of the square root one minus your velocity squared divided by the speed of light squared. This is a factor often called tau. As you fly overhead, I hold up a foot-long ruler parallel to your trajectory. I count the number of ticks that occur on my clock as the nose of your ship passes from one end of the ruler to the other. You do the same, counting the ticks on your clock, and therefore you believe my ruler is shorter than it actually is by a factor of tau. As we pass each other, our rear stabilizer fins bump into each other ever so slightly, giving us both a nudge in a perpendicular direction. Momentum should be conserved, so when I look at your ship, I should measure your perpendicular velocity equal to mine. running slowly, I see your perpendicular velocity is too slow by a factor of tau. The only conclusion I can come to is that your mass has increased by an identical factor. Remarkable isn't it? All these counterintuitive effects simply by assuming that the speed of light is constant to all frames of reference? And these effects have been experimentally accuracy. The world is truly a very bizarre place. And we haven't even talked about general relativity yet. In 1992, Jack Reeves published a landmark paper entitled, What is Software Design? This Principles, Patterns, and Practices books. You can find that paper there or in the URL that's on your screen. In this paper, Jack makes this truly beautiful point. Question. What do engineers produce? Answer. Engineers produce documents that specify how to build

products. blueprints, building diagrams that specify how to build a building. Electronics engineers produce documents, circuit diagrams that specify how to build a circuit board. Mechanical engineers produce documents, mechanical drawings that specify how to build machines. So what in the software world would qualify as such an engineering document? The only document produced by software engineers that is detailed enough to fully specify a software product is the source code. Okay, I hear you out there. You're saying that the source code is the product. No, it's not. The running program is the true product. The binary executable is the true product. derives. Let's look at this differently. If I had an automated factory capable of building houses, then the input to that factory would be the architects blueprints. If I had an automated factory capable of building circuit boards, then the input to that factory would be the diagrams an automated factory capable of building mechanical components, then the input to that factory would be the drawings created by the mechanical engineers. And it turns out that I do have an automated factory that can build a software product. It's called a compiler, and the input to that factory is source code. Any other documents you might produce that are preliminary to the source code are just that, preliminary. They are not the design. If you draw UML diagrams to help you organize your thoughts, these diagrams are not the design. They're just preliminary diagrams that help you create the real design. There's nothing wrong with drawing them. They can be very useful. But the diagrams are not the design. The source code is the design. And this leads us to a very interesting conclusion. When we build a house, we spend a great deal of time up front designing it, because the cost of designing it is far less expensive than the cost of building it. When we build circuit boards, we spend a lot of time up front designing them, of building and mass producing them. When we build mechanical components like gears and levers and things like that, we spend a lot of time on the mechanical design because design is cheap compared to cutting dies and milling metal. And in all three of these cases, the cost of correcting errors after design is complete front on design in order to minimize the cost of building. But in software, the reverse is true. It's far cheaper to build the product than it is to design it. And fixing errors prior to release, that's very cheap too. In fact, even after release, the cost of fixing errors is far cheaper than changing the foundations of a house. I can compile a million line application in seconds. I can test it in minutes. And fixing problems within it will take a matter of hours. So for software, the cost of building is cheap. On the other hand, the cost of designing that software is actually very high. Software developers make pretty good salaries. And the amount of code they write per hour or per day is really small. completely inverted. The cost of design is expensive whereas the cost of building is cheap, and this inversion of costs means that the strategy for building software is entirely different from the strategy for building a house. What if the cost of building a house was tiny? What if And what if every change you made to that house cost $100 and took an hour? How would you go about building such a house? Would you hire an architect and pay him a small fortune to create the entire design of the house and then build the whole house at once? Of course you wouldn't. to sketch a couple of rooms on the back of a napkin and

build them and see what they look like. Then you'd walk through those rooms looking for things that you didn't like and you'd make a list of those things and then you'd spend another hundred dollars and another hour fixing those things. And of course you'd continue to do that for several more days. You'd continually be tweaking adjusting the relationships between them. Eventually you would evolve the building into a structure you thought you could live in and then you'd move in. But of course you wouldn't stop there. Every day you'd be finding things about the house you didn't like or you'd think of new rooms that you needed to add. You'd make a list of these things and then at the end of every week you'd spend another And you'd never stop doing that. Oh, you might slow down to one change a week, or one change a month, or even one change a year, but you'd never stop fiddling around with the design of that house. That's software for you. It's crazy to spend a lot of time on upfront design when you can get something working quickly and then evolve it into a system that meets your needs. And therein lies the rub. evolve the design of a system, there's no guarantee whatever that you'll design it well. It's easy to evolve a design into something that works. Unfortunately it's also easy to make it hard to modify, hard to maintain, unstable, crashable. Some people call this a big ball of mud. eliminates fear and allows us to keep our code constantly clean. So to avoid that big ball of mud, we need to practice test-driven development and apply lots of effort to keep our code continuously clean. The problem is that in order to clean our designs, we need to be able to recognize when those designs are going bad. We need to know what bad design smells like. What does bad design smell like? What are the symptoms of bad design? And what are the situations that a good designer should avoid? Back in episode one, we studied the design smells of rigidity, fragility, and immobility. We're going to look at those smells again now in more detail. on top of it. Rigidity is the tendency of a system to be hard to change. What makes a system hard to change? A system is hard to change when the cost of making a change is high. For example, if by then that system is rigid. Now let's say that you have a system that requires three hours to build and test. Let's also say that the most minor change to the most insignificant subsystem within that system requires you to do a three hour build and test. What makes that system rigid? Two things. First, it takes a long time to do a test and build, and secondly, it's just a tiny change that forces a total rebuild. If we could reduce the build and test time dramatically, we could make the system much less rigid and much easier to change. If we could find a way to restructure the system, that when you changed it, you didn't have to rebuild and retest the whole thing, then changes would be a lot easier to make and the system would be much less rigid. We'll talk about long-running tests in another episode. In general, however, when the tests take a long time to run, it's a good indication that the developers have been careless. Long build times are a function of coupling. This is especially true in C++, where the build time is proportional to the number of coupled modules squared. But again, this is something we're going to be talking about in an upcoming episode. When small changes force rebuilds, it's also a symptom of high coupling. When modules are coupled, tiny little changes cause the whole system to be rebuilt. Therefore, one of our design goals is to manage

the dependencies between modules to ensure that when one module is changed, the others remain unaffected. A system is fragile when a small change to one module causes other unrelated modules to misbehave. Imagine the software that controls an automobile. That software would be fragile if when you fixed a bug to the radio, affected the electric windows. These kinds of long-distance behavioral dependencies are very scary, especially to managers and customers who view them as indications of significant incompetence. After all, if every time the developers fix a bug or add a new feature, something completely can come to is that these developers have lost control of their software and don't know what the hell they're doing. The more this happens, the more uneasy managers and customers become. In the end, they'll simply freeze development and official rigidity will set in. Long distance sensitivity like this is always caused by strange couplings and dependencies snaking across the system. The solution is to manage the dependencies between the modules and isolate them from each other. A system is immobile when its internal components cannot be easily extracted and reused in new environments. typical username password login module. If you can't quickly extract that login module and use it in an entirely different system, then that module is immobile. It can't be moved. Immobility is caused by couplings and dependencies in the modules of the system. For example, let's say that I've got a login module that used a particular database schema interface scheme. I would not be able to reuse that login module in a different system if it had a different database schema and a different user interface scheme. That login module would be immobile. The strategy for avoiding immobility is precisely the kind of architecture we explored in application from the database, the UI, and the frameworks. A system is viscous when necessary operations like building and testing are difficult to perform and take a long time to execute. A development environment in which check-ins, checkouts, is viscous because the cost of those essential operations is high. System designs, in which new features must be added across multiple layers of the system, dealing with multiple transport mechanisms, serializations, marshallings, hydrations, this is always viscous because even the simplest change is costly to make. of viscosity is always the same irresponsible tolerance developers tolerate conditions they know to be bad and do nothing to correct them the cost of those bad behaviors is coupling tight coupling makes systems hard to build hard to test and hard to change it is that tight coupling that high. The cure for viscosity is to attack the symptoms by decoupling the modules and then managing the dependencies that remain. A real common issue in software design discussions is how to deal with the future. Should we design our system all the future requirements of the system. In other words, should we put the hooks in for future extensions, or not? Systems that carry a lot of anticipatory design are needlessly complex. Each hook, each extension point, is another weight added to the system that the developers must carry in the present. code and you think it's hard and expensive to change, then you're going to litter that code with all kinds of anticipatory design elements so that you don't have to change the design later. If, on the other hand, you follow the advice given in episode 6 and maintain a comprehensive suite of tests, then you won't be afraid to change the code. of anticipatory elements. Your

designs will be simpler, easier to maintain, and they won't be needlessly complex. Needless complexity often leads to tight coupling because we anticipate the future need for relationships between modules that are not currently related. the software becomes now. The solution, of course, is to keep your design focused on the current suite of requirements, while maintaining a comprehensive suite of tests that reduces your fear of changing the design later. Of course, nobody starts out to design a system that smells bad. decisions that are motivated by carelessness, fear, and false expedience. The greater the mess, the harder it is to make progress. Everything gets more and more difficult. And the greater the temptation to take the kind of shortcuts that increase the mess. Let's see how this happens. Code rot. Monday morning, your boss calls you into a conference room. And then he says... So, I called you in here because I've got a new project for you. Okay. What I'd like you to do is write a program that copies characters from the keyboard to the printer. Hmm. All right. Anything else? No. No, that's about it. How long do you think that's going to take you? I think this is about six lines of code. Three weeks. Excellent. Let's get started. Three weeks is the minimum estimate where you work. If anybody gives an estimate less than three weeks, he's taken out back and soundly beaten by the other programmers. The first thing you do is draw a diagram because, as you know, all programmers draw diagrams before they write code. The copy module contains all the high-level policy. It contains the main loop that gets characters from the keyboard reader and sends them to the printer writer. It also recognizes exits. This diagram looks good and you're about to write the code that matches it but your boss walks in with a new guy that you've got to orient today and you got to show them the ropes and it takes all day long so that's gonna pretty much eat up Monday. Tuesday you write the code it looks like this it's the six lines you had in your head a simple loop that terminates on end of file and otherwise You're about to compile it when you realize you're late for a quality meeting that's going to take all day. Wednesday, you compile the code, and it compiles right away, too. That's a good thing, because right after that, the field service manager rushes into your cubicle with a horrible bug in the field, and you're going to have to go fix it. It's going to take you all day. code and it works first time you run it too. Good thing because your boss comes in and hauls you into some horrible cross-functional meeting that's gonna take all day long. Friday. No meetings, no bugs, no interruptions and it's a good thing too because it takes all day long to get this code into the source code control system. Whoa! You're done with two weeks to spare! But don't let your boss so that you're done early. You better keep busy with other stuff and then you can release it on time. You win awards for this software. Hundreds of other programmers start to use it in their systems. It's so successful, those six lines of code may be the most successful lines of code ever written at your company. and says... So, you know that program you wrote, that copy program? Yep. That was great work. And you're going to see our appreciation in your next salary review. So, now what we'd like is for it to read from the paper tape reader. That's it? Just read from the paper tape reader? Sometimes. Sometimes from the keyboard, sometimes from the paper tape reader. Okay. take you to do. Hmm, this sounds like a

Boolean and an if statement. Three weeks. Good, let's get started on that. So now you modify the diagram to show the new dependency upon the paper tape reader. So how should you modify this program? You could pass a Boolean into other hundreds of programmers that used your function are going to have to recompile and retest if you do that. So they'll come to your cubicle with clubs. No, it's probably better to use a global. It's a simple idea. If someone wants to copy from the paper tape reader, they'll just set the GPT flag variable to true, and then they'll call copy. They'd better remember to clear that flag when they're done. Otherwise, the You can cover your butt with an appropriate comment, like so. Remember to clear. Okay, so now we have to make this work. The best feature of the C family of languages is the ternary operator. It allows you to put whole if statements into a single expression. So we'll just insert it into the program like so, and voila! A few months later, your boss asks to see you again. So you know that copy program you wrote? Yeah. Sometimes we'd like it to write to the paper tape punch. Hmm, I've got a design pen for this now. I know how to solve it. Three weeks. Great! One more addition to the diagram, to the copy module. The change to the code is simple. Just one more global right here. You can reuse the comment. And now you can add another ternary operator just like so. There, that'll work. Ship it. coming back to you with more and more changes. He'll want to read from the optical character reader and write to the voice synthesizer. There'll be no end to it. And so that module will grow and rot and fester and degrade. A few years from now, it'll be time to polish off your resume and leave that mess to somebody else. Of course, it didn't have to be this way. We could have written the code like this. Yes, this looks just like the original six lines, but there's a small difference. Instead of reading from the keyboard reader and writing to the printer writer, we're reading from getchar and writing to putchar. writes to standard output which defaults to the printer. So this version does exactly what the previous version did. However, standard input and standard output can be redirected to other devices like the paper tape reader and the paper tape punch. This means that when your boss says, sometimes we need it to read from the paper tape reader you've got an option. You could say, three weeks. Or you could tell him that it already does read from the paper tape reader. This code differs from the previous code by two words, and yet those two words somehow prevent the code from rotting when you add new devices. In fact, adding new devices, Just what's so special about these two words? How have they so completely changed the maintenance characteristics of this module and utterly stopped this code from rotting? To understand why those two words are so important, let's look at the diagrams again. Here's the diagram of the first solution. Look at the direction of those arrows. The module that contains the high-level policy depends on the low-level details. And when we added new devices like the paper tape reader and the paper tape punch, we had to add new dependencies to the copy module. So the fan-out of the copy module grew with each change. But now look at the diagram for the new version. Copy depends upon getchar and putchar, but does not depend upon the keyboard and the printer. What is it that fills the gap between those two? It turns out that getchar and putchar

are part of a Unix abstraction known as file. This abstraction is represented by a data structure that, among other things, contains a table of five pointers to functions. read, write, and seek. The I.O. drivers for the keyboard, printer, paper tape reader, and paper tape punch implement those five functions. So when you redirect standard input and standard output, what you're really doing is loading those five functions into the file abstraction. Now, look at these two diagrams side by side. Notice the inversion of the dependencies. In the first version of the code, the dependencies point in the same direction as the flow of control. But in the new version, the dependencies oppose the flow of control. This inversion of dependencies prevents the system from rotting because it stops the fan out of the copy module from growing. doesn't need to be modified because all of its outgoing dependencies terminate at the file abstraction. New devices can be added ad nauseum without affecting the copy program one little whit. Now consider this. Those five functions in the file data structure are exactly equivalent to C++ V tables used to implement virtual functions. They're logically equivalent to used in Java, C-sharp, Python, Ruby, Smalltalk, and every other OO language. This means that getchar and putchar are logically equivalent to polymorphic methods on a class name file. So the new version of copy is really an object-oriented program. OO language. But that's not really important. You don't need an OO language to write an OO program. All you really need to do is to invert key dependencies by using dynamic polymorphism. To make this point clearer, take a look at this diagram, which is the logical equivalent of the getchar-putchar solution, and yet it's clearly an object-oriented program. File is an interface two I.O. drivers for the keyboard and the printer, and it's used by the copy algorithm. And here's the code. Again, it's semantically identical to the get-char-put-char solution, but it's written in an OO language. Notice the inversion of the dependencies. The keyboard and the printer derivatives of the reader and writer interfaces depend in a direction What is OO? In 1966, two Norwegian computer scientists, Ole Johan Dahl and Christian Nygaard, were fiddling around with the ALGOL 60 compiler. They took a critical data structure, the function called stack frame, and they moved it from the stack to the heap. They had invented the first OO language, Simula 67. Dahl and Nygaard invented the method call syntax that we're so familiar with, O dot F of X. But is this syntax really the essence of OO? Is O dot F of X really so different from F of O and X? Any Python programmer can tell you it's not. imbued Simula 67 with dynamic polymorphism. This gives the statement O dot f of x an interesting new interpretation. The caller does not know which implementation of f will really be invoked, so the caller has been decoupled from the function that gets called. But it was Alan Kay who gave us the most effective metaphor. OO is about passing messages. over how that message is going to be interpreted. You don't know where it's going to wind up. You can only hope that the receiver of the message reacts appropriately. Thus, the sender does not depend upon the recipient, nor does the recipient depend upon the sender. Both of them depend upon the message, which is an abstraction. The dependency opposes the flow of control, It's often said that OO is about modeling the real world within your software. There's truth to this, but in fact

there's nothing special about OO that allows it. Programming is about modeling the real world within your software. It's often said that OO is inheritance, encapsulation, and polymorphism. polymorphism and encapsulation to write programs that rot every bit as well as that copy program rotted. Encapsulation, inheritance, and polymorphism are mechanisms within OO, but they are not its essential quality. The essential quality of OO, the thing that makes it different from other paradigms and the thing that makes it useful, is the ability to invert key dependencies, In the end, object-oriented programming design is all about dependency management. Over the years I've assembled eleven principles of object-oriented design. Each of these principles involves an aspect of dependency management. Indeed, we could call them dependency management principles. The first five principles control the relationships and operations between classes. They're called the SOLID principles because their names form the acronym SOLID. These five principles describe the way that classes in an object-oriented design relate to one another. They're all about the dependencies between those classes and the motivations for creating The next three principles are called the Principles of Component Cohesion. They describe the forces that cause classes to be grouped into independently deployable components. The last three principles are the Principles of Component Coupling. These principles describe the forces that govern the dependencies between components. which describes how we use OO to build applications out of classes and compose them into independently deployable components with high cohesion and low coupling. The next several episodes will investigate these principles in extreme detail. We'll look at them from all sides and we'll investigate case studies that apply them. the principles of object-oriented design to create software applications with robust designs and architectures that don't smell and don't rot. So let's review. In this episode, we've laid the foundation for the solid principles of We discussed Jack Reeves' remarkable insight that, unlike most other industries, software is expensive to design, but cheap to build. We showed that this implies that software should be designed and built iteratively, without huge up-front planning. We talked about rigidity, fragility, immobility, viscosity, and needless complexity. We watched some code rot, and we saw how the design of that code promoted that rot. We also learned that designs that have an inverted dependency structure, where the dependencies oppose the flow of control, tend not to rot. history of OO, and then we created a definition that was independent of the more mechanical definitions of OO, such as polymorphism, encapsulation, and inheritance. In our definition, an object-oriented design is one in which key dependencies have been inverted in order to talked about dependency management and the role that the solid principles play in keeping the source code dependencies in a software application under control. So that's it. I hope you learned something. I hope you had fun. But boy, do we have a lot more stuff to talk about. We've got to talk about all the other design principles and then a whole load of design patterns. We've got to talk about practices like continuous integration. driven development session I promised you. You're not going to want to miss the next exciting episode of Clean Code, episode 9, the single responsibility principle. Mighty dogs, let's go! Let's go dogs! Out you go! Out you go! For a lot... I don't know. Oh no! So,

you know that coffee program you wrote a few months ago? That was a great job. This is going down. Okay. Micah, Micah, Micah, Micah, nuclear explosions have happened. The world is just caving in. It's got to come, come, come. then the more the better need some more of my brandy Welcome, welcome to episode 8 of Clean Code. The first in a series of episodes about the solid principles and jet planes. Thank you.