Hi, I'm Uncle Bob and this is Clean Code. So how do you know if you're violating the Liskov Substitution Principle? not setting yourself up for a refused bequest and the horrible open-closed violation that follows from it. Here are a few simple heuristics you can follow that you'll find helpful. First, remember that if the base class does something, the derived class must do it too and it must do it in a way that does not violate the expectations of the callers. that you cannot take expected behaviors away from a subtype. A subtype can do more than its parent type, but it can never do less. This means that whenever you have a derived class that has degenerate functions in it, functions that don't do anything, then you've possibly got a Liskov substitution principle violation, especially if those functions were Now look here children, not all degenerate implementations are a violation, so you can't make this a hard and fast rule, but you should all be suspicious whenever you see a degenerate implementation. A more blatant clue occurs when a derived function is written to unconditionally with the Liskov substitution principle, because the author of the derived function clearly doesn't want you to call it. But how are you supposed to know that you're not supposed to call that function when all you know is that you've got the base class? How are you going to stop that exception from being thrown? In the end, about all you can do is use the dreaded if instance of statement. Another indication that the Liskov substitution principle being violated is the presence of the dreaded if instance of statement. This isn't always a violation. There are sometimes good reasons to check the type of an object, but it's very suspicious. When is it safe to check the type of an instance? There's only one rule. Why would we bother to check the type if we already know it? Only if the compiler has forgotten it. For example, consider this function that adds a new time card to a list of time cards within an hourly employee object. Hourly employee derives from employee Our function takes the ID of an hourly employee and uses it to fetch that hourly employee from the employee gateway. But the gateway returns the hourly employee as the type employee. But we know that this ID must be for an hourly employee. If not, then the function makes no sense. know to be true. This use of if instance of is harmless. It added no new dependencies to the function. This function already needed to know about hourly employee and so no new information has been added. The if instance of simply reasserted information that the compiler had lost. of becomes a problem when there's a chance for an else if instance of. Remember the square rectangle problem from the previous segment? We added an if instance of statement in that problem because we were in a module that was using a rectangle, but somebody passed us a square. We weren't just asserting that we had a rectangle. We were questioning whether or not the object was a rectangle or a square. And that begs the question, what else might that rectangle be? Are we going to have other derivatives of rectangle in the future? For example, are we going to have 2 to 1 rectangle? 3 to 1 rectangle? This kind of structure is called a type case, and clearly it can become a significant problem. It should be replaced with polymorphic dispatch. In a future episode, we'll talk about the visitor pattern, and how you can use it to safely remove type cases without polluting your hierarchies with irrelevant methods. If you see a type case, or anything that looks like it might become a type case, it's very likely a

violation of the Liskov substitution principle that's driving it. As we noted before, statically typed languages achieve subtyping through the use of inheritance. And subtyping is what gives us the open-close principle, for architectures defined by boundaries. But inheritance is the strongest relationship that can exist between two types. When you derive from a class, you inherit everything that's inside that class, including all of its dependencies. The design smell of rigidity is caused by unrestrained accumulations of dependencies, Inheritance breeds rigidity. We'll see more about this when we study the dependency inversion principle in an upcoming episode. For now, it ought to be clear to you that when you inherit a base class, you drag along all the baggage that's inside that base class. Now what this means is that the relationship that gives us flexibility also makes us rigid. All that open-close principle stuff that depends on subtypings, we get that from a relationship that also gives us inflexible structures. This isn't the case for dynamic languages like Ruby or Python or Smalltalk because they don't rely on inheritance for subtyping and that means that they can create flexible structures This is one of the reasons that dynamic languages like Groovy and Ruby and Python have become so popular lately. Using those languages, it's possible to conform to the open-close principle and build highly flexible structures without experiencing increased rigidity. But dynamic languages have a different problem. Java compiler would catch at compile time. This is why languages like Java, C Sharp, and C++ are considered to be type safe. If you make a type error in Java, the program won't compile. If you make a type error in Ruby, the program crashes. That's a huge difference, and it's proven to be a significant disincentive Indeed, for years programmers chose the rigidity of type safe languages over the flexibility of dynamic languages because they simply could not tolerate the risk that their programs would crash for reasons that were so easily detected by a type safe compiler. is the topic we studied back in episode 6, test-driven development. This discipline was invented and refined by programmers of dynamic languages, and for obvious reasons. When you don't have a type checker, you do your own type checking. However, when test-driven development was eventually adopted by the programmers of static languages like Java, themselves an obvious question. If unit tests protect you from type errors, why do you need a language that imposes rigidity on you to do it? And that is how the rebirth of dynamically typed languages began. But type-safe compilers do not find all type errors. the type errors that can be found through static analysis. For example, the Liskov substitution principle violation that we saw in the square and rectangle problem, that can't be found by a Java compiler or a C sharp compiler or a C plus plus compiler using static analysis. The reason for this is that types are Remember that a type is a bag of methods. And while it's true that the signatures of those methods are part of the definition of the type, the behavior of those methods is also part of the definition of the type. It was in the 80s that Bertrand Meyer set about to address this issue. The book that he wrote, Object-Oriented Software Construction, on resolving this issue. The technique he used he called design by contract. The idea actually derives from the work of C.A.R. Hoare and the type theories of Frigg, Russell, Gödel, et al. Every type has certain invariants which can be stated as Boolean expressions that must

always be true. and width of a square must always be equal. Every function in a class can be surrounded by preconditions and postconditions. Preconditions are Boolean expressions that must be true before the function can be called. Postconditions are Boolean expressions that must be true when the function returns. If you carefully express your pre and post conditions, then you'll be providing precisely the functions necessary to do dynamic type checking. If you encounter a problem at runtime, it will be discovered instantly by your pre or your post conditions and pinpointed for you so you can quickly fix it. Meyer actually wrote the language Eiffel around this concept. conditions and invariants, and the language would call those functions for you at just the right times, and combine them with just the right logical operations to ensure dynamic type safety. Were it not for test-driven development, design by contract might have taken hold. But as it happens, test-driven development is a far more general, if perhaps Nowadays, it's test-driven development and not designed by contract. That is the primary means of preventing dynamic type errors. Consider a system in which a very old suite of programs called the file movers using modems. The FileMover programs use the interface named Modem to communicate with all the different kinds of modems, including the Haze modems, the U.S. Robotics modems, and even the old Zoom modems. The modem interface has methods like Dial, Hang Up, Send, and Receive. The FileMover programs call dial to connect to the systems between which the file will be moved. Then they call send and receive to enact the file transfer protocol. And when they're done moving the file, they call hang up. The FileMover programs are very fragile. Anytime anybody touches one for any reason, they break in a whole bunch of places. to touch the FileMover programs. Official rigidity has set in. However, a new requirement has come to the fore. Some of the nodes in our network have decided they need to move away from dial-up modems and start using dedicated modems instead. Dedicated modems are modems that you don't need to dial. They're permanently connected. Another group of users in our company has been using dedicated modems for years for a whole range of other applications. This group is called the dead users. The dead users use the dedicated modem interface, which has send and receive methods. The FileMover programs have never used dedicated modems before, change. So now the FileMover programs are going to have to use dedicated modems. Management has recently heard about the open-close principle, and they like the idea. They like the idea that they can extend the behavior of FileMover without modifying it. They see this as a way to continue their policy of official rigidity. They like the idea that we rate of. So we've been charged with getting the old FileMover programs working with the new dead modem interface without touching any of the old FileMover programs. Now the easiest way to get this to work is to simply derive the dead modem interface from modem. we could implement the hang up and dial methods to do nothing. The file movers will still call them, of course, but those calls will be ignored. And we'll just stick dummy phone numbers in the database for the dedicated network nodes. Yeah, those dead users are certainly gonna be angry because we're changing the dead modem interface. And that's gonna force them to recompile and redeploy all their applications. Well, that's

too bad and really sad for them, but it's just one rebuild and redeploy and then we're done. And everything works. Management declares us to be geniuses. We're all promoted and given big raises. The dead users hate us. But otherwise, life is good. But as the weeks go by, the users start to report more and more strange behaviors. About once per day, one of the moved files gets corrupted. Its file length is one character too long. At first, we think this is some kind of hardware problem. But as the weeks go by, we realize that the only files that ever get corrupted modems. After a few months the situation has gotten to be critical. Every customer who uses a dedicated modem has begun to complain about this issue. Management is up in arms. They're deeply concerned and they've told us that we'd better get to the bottom of this as quickly as possible. We suspect that the fault lies friends the dead users and we ask them if they're sure that their modem driver actually works this doesn't help our relations with the dead users at all they calmly tell us that they've been using this driver for years and they've never had any problems with it we point out that we added no new executable code from the file movers into the dedicated modem driver. So the problem's gotta be in that driver. The dead users glower at us, but they agree to take a look. The next day, they come back to us with an explanation. There's nothing wrong with their modem driver, they tell us. The problem is a race condition in the file mover programs. It turns out that the FileMover programs were not written to expect the modems to receive characters until after Dial had been called. But the dedicated modems can receive characters at any time. So every once in a while, an unsolicited character sneaks in, and the FileMover programs don't realize that this character was received before they called Dial. us that we're going to have to change the FileMover programs to ignore characters until dial is called. They're right of course, but that leaves us with a real dilemma, because now we're going to have to change those FileMover programs, and we were told not to do that. What are we going to do? What are we going to do? But then we get a really clever idea. into that dedicated modem class. That flag would not allow characters to be received until you called dial, and it would stop them again once you called hang up. So we head to the lab to try this out. We force the race condition to happen. We show that before the fix, the file movers do in fact add extra characters every once in a while. But after the fix, no such additions occur. We've got the solution. It works. So now all we have to do is tell the dead users that they have to call dial and hang up. I don't think they're going to be very pleased about that. You want us to what? You want us to call dial? What phone number do you want us to them that this is a simple fix, it's no big deal, and besides, management has mandated that no one is allowed to change the FileMover programs. We beg, we plead, we promise to buy lunch for them for a year, but they don't want to have lunch with us. They don't want to see us again. They don't want to have anything to do with us. their door again. And it all works perfectly. There's no more file corruptions, there's no more problems of any kind. And after a while, everybody kind of relaxes and things settle down and go back to normal. And even those annoying 2am prank phone calls aren't happening quite so often anymore. And that makes them realize that they can't use 10 digit phone numbers anymore. Clearly this

means that the dial method of the modem interface is going to have to change. It can't take a 10 digit array anymore. It's going to have to be changed to take a variable length string. This means that the FileMover programs are going to have to change. So instead they're hiring some very expensive consultants. We are told to make the changes to the modem interface, the modem derivatives, and the dedicated modem class. I think they took the news quite well. is a subtle violation of the Liskov substitution principle. There is no way to make the dedicated modem subtype perfectly substitutable. The dedicated modem can only be made to work with the file movers if it violates the expectations of the dead users. It can only be made to work with the dead users if it violates the expectation of the file movers. there's no way to remove the refused bequest. The first hint, of course, was the degenerate implementation of the dial and hang up methods. Those degenerate methods overrode other methods that had real behavior, and when we removed that real behavior, we set the stage for the refused bequest. class into the battle between the dead users and the file movers, they created a long distance dependency that resulted in a fragile and rigid system. Some of you have already guessed that one of the solutions to this problem is to use an adapter, which is a pattern by the way that we're going to be studying in a future Rather than derive the dedicated modem from modem, we could derive an adapter from modem and then delegate just the send and receive methods to the dedicated modem. That adapter will contain the Boolean flag that protects the FileMover programs from the race condition. Notice that the adapter has no effect at all on the dedicated modem class. we're never going to have to talk to those dead users, and our phone's not going to be ringing at 2 a.m. All the refused bequests have been eliminated. All the subtypes are perfectly substitutable. There is no Liskov substitution principle violation. Now, you might complain that this is an ugly hack. And you're right. It is an ugly hack. But look at the direction of the dependencies. Notice that they all point away from the ugly hack. Nobody in the system knows that that ugly hack exists. Except of course for main. Remember way back in episode 4 we talked about how main should live in a separate module that's behind a boundary and that all source code dependencies that cross that boundary should point away from main? The dead modem adapter is only known to main. The rest of the system has no idea that it exists. If you need an ugly hack, make sure you isolate it from the rest of the system by pointing all dependencies away from it. Whoa, what a ride! And I don't know about you, but my neurons are like fried. We sure have covered a lot of ground, haven't we? We took a pleasant little stroll through the history and theory of types, from its mathematical beginnings to its application in computer science. And then we looked at Barbara Liskoff's critical insight into the definition of subtyping, that James Coplin called the Liskov Substitution Principle. We walked through the old square rectangle dilemma, and we saw what can go wrong when you violate the rule of representatives. We talked about type checking, and then about design by contract. And then we looked at some heuristics that can help you find and solve of the Liskov Substitution Principle. Finally, we learned how to be prank called at 2 a.m. by an angry bunch of programmers whom we've abused by violating the Liskov Substitution Principle.

Alright, well, that's the end. It's over. And I hope you learned something. I hope you enjoyed yourself. But boy, do we have a lot more to talk about. And there's two solid principles left. And then there's the component principles. bunch of design patterns there's agile practices there's advanced test-driven development you're not gonna want to miss the next exciting episode of clean code episode 12 the interface segregation principle all right you girls you dogs let's go come on come on dogs girls ha ha grandchildren are the dessert of life Hwyl! Hwyl! Hwyl! Hwyl! Hwyl! Hwyl! Hwyl! Hwyl! Hwyl! Hwyl! Hwyl! Hwyl! Hwyl! Hwyl! We'll be right back. We'll be right back. We'll see you next time.