Hi, I'm Uncle Bob, and this is Clean Code. In this episode... By Uncle Bob. Uncle Bob. Enjoy. Uncle Bob. Welcome, welcome to Clean Code, Episode 11. The Liskov Substitution Principle. Come on in. This is our fourth episode in the series of the solid principles. Can I take your hat? Remember last time we talked about Bertrand Meyer's open-close principle? That was Bertrand Meyer's remarkable insight that a module could be both open for extension and closed for modification. We figured out that it would be possible to add new features to a system by adding new code and not changing any old code. I told you that the open-close principle was the moral center of system architecture, that good architects and designers will strive to make their systems open for extension but closed for modification. And then we looked at a system that just broke all the rules and that exhibited all those horrible design smells of rigidity, fragility, and immobility. that was perfectly conformant, or so we thought. It was simple, it was beautiful, it was elegant, and it was a lie. Then we found that in order to perfectly conform to the open-close principle, you have to have perfect foresight. If you're going to make a system that is infinitely extensible, you've got to be able to perfectly predict the future. But we didn't give up hope. realized that if we used a process that was iterative, if we relied on lots of feedback, if we delivered often, and if we used lots and lots of refactoring, then we could well enough conform to the open-close principle. Now, in this episode, after we talk a little bit about wave-particle duality, we're going subtyping, the Liskov substitution principle. So first, we're going to take a stroll through the history and theory of types, from their mathematical underpinnings to their application in computers. Now, I see all you ruby, groovy Python enclosure programmers thinking that, no, this doesn't apply to me. But it does. It applies to you. whether the compiler checks those types or not. Next we'll talk about subtypes. We'll discuss Barbara Liskoff's critical insight into the definition of subtypes. We'll talk about why that definition is important and the implications of that definition for static and dynamic languages. the definitive case for the Liskov Substitution Principle, and we'll see what can go wrong when you use inheritance and subtyping intuitively instead of according to Liskov's definition. This will lead us to some heuristics, examples, and rules that will help us detect and solve violations of the Liskov Substitution Principle. Finally, we'll look at a larger case study to create substitution violations that you don't recognize right away. And then we'll see the terrible damage that's done by such violations. And in the end, we'll look at some simple ways to either avoid or repair that damage. So you put your thinking caps on and sharpen your pencils, because we're about to submerse ourselves in the fascinating world of the Liskov Substitution Principle. As we learned last time, in 1900 Max Planck discovered that although light moves in Maxwell's waves, it's also absorbed and emitted in discrete amounts of energy. Einstein went on to show that this discreteness in the light waves was an attribute of the but not of the atoms that were emitting and absorbing them. That is, the light waves were organized into discrete quantities of energy, which Einstein called light quanta, and we now call photons. So somehow, light is both a wave and a particle. But we should be clear about what the word particle means. hard little nugget. Instead it means discrete and indivisible.

A photon of light is a bundle of energy that cannot be split apart. It always equals Planck's constant times its frequency. What does it mean for a particle to have a Light travels in waves. It carries its energy in waves. But when it's forced to deposit that energy on some surface or some entity, it always does so in discrete amounts at discrete point-like locations. And that discrete amount and that discrete location is the photon. That's clear, isn't it? Maybe an example would help. This is my green laser pen. I use it when I'm giving talks. It's very bright. This laser pen is a photon gun. It fires a stream of photons in a very narrow beam. The number of photons it fires is huge, but discrete. and they travel in a wave towards the wall. When that wave strikes the wall, the wave deposits its energies upon the atoms of the wall in discrete amounts upon discrete atoms. Those atoms then reflect those photons back outwards towards our eyes, and we see the green spot on the wall. on this laser pen that would adjust the rate at which the photons were fired. Maybe I could turn that knob all the way down so that it fired one photon a second, kinda like that. And then the blinks you would see on the wall would be one photon each, depositing its energy at a particular discrete location with a very particular discrete energy. Now let's crank the firing rate of this laser back up to trillions of photons per second and we'll shine it through something that's got a bunch of tiny little slits in it, like my comb. What should we see on the wall? Presumably the beam will pass through the slits and strike the wall and so we should see one green spot on the wall. see the shadow of the slits. We can illustrate this by using this flashlight and a paper plate with two big holes in it. We can still see the flashlight beam, but only the part that's getting through the two holes. But now let's go back to my laser, and instead of a comb, we'll use this diffraction grating, in it. 13,500 per inch. We'll shine that laser through this again and we get stripes, spots. Look at that. And of course, that's exactly what you'd expect if the light were traveling in waves. You see, the waves pass through the slits in the diffraction grating. They break up into a bunch for each slit. Those little waves move towards the wall, but as they do so, they interfere with each other, resulting in this characteristic pattern of wave interference. That, my friends, is an interference pattern. And if I take the slits away, I get one nice little spot on the How cool is this? So now let's crank our laser pen down so that it's only firing one photon per second. What do we see? Well, we see one flash per second all right, but the location of those flashes is strange. They seem to be all over the screen. But if we accumulate all those little flashes over an hour or so, we find something really startling. We see the interference pattern. Each photon travels as a wave. That wave passes through the slits and breaks into many waves, which then interfere with themselves. with a discrete energy. Where will the photon land? There's no way to know. It's completely random. But the odds are that the photon's going to land where the interference pattern is bright. The odds are very low that the photon will land where the interference pattern is dim. Did you hear what I just said? I said odds. Odds. that determines where a photon will land? Odds! Odds in the shape of an interference pattern. Apparently the wave of a photon is a wave of odds, a wave of chance. A light wave is not a wave through the ether, it's a wave of probabilities. my head hurt.

And so these waves of probability propagate through space and interfere with each other, leaving an interference pattern of probability on the screen. The position of the photon is undetermined until that photon strikes the wall and deposits its energy. The position of the photon is uncertain. And the principle behind that uncertainty will be our topic for next time. Alright, now I want you to be a little bit patient, because what you're about to hear is going to sound like another science lecture. But it's not. to the Liskov substitution principle. In the second half of the 19th century, a German mathematician by the name of Friedrich Ludwig Gottlob Frieg decided to treat mathematics the way Euclid had treated geometry. His goal was to derive all of mathematics from a few simple postulates. a paper whose title in English was a formal language for pure thought modeled on arithmetic it was a masterpiece Frigga had created a formalism that allowed him to describe all of mathematics integers fractions functions Not only could Friege describe mathematics with his formalism, he could describe virtually anything. This formalism of his resolved a whole series of problems that had plagued logicians and mathematicians for years. Twenty-three years later Bertrand Russell discovered the first chink in Friege's formalism. He showed that it was possible to create paradoxical statements using statements that could be proved neither false nor true. Russell's paradox looked like this. It can be described with this simple question. Does the set of all sets that don't contain themselves contain itself? So, now you should hit pause and then play that over again. And then hit pause again play it over again and keep doing that until you understand it. Or perhaps you should look at it this way. If your mother only cooks for those who don't cook for themselves, who cooks for your mother? Or, if you want it to be really simple, this statement is false. Clearly, the question cannot be answered. In fact, the question is caught in a logical loop. There's no resolution to that loop. If you were to write a program to solve it, that program would loop forever, recursing infinitely, and would blow the stack. intuitively intuited that the solution to the problem was to restrict the types of things that could be operated upon by functions. It soon became clear that the loops could be prevented by expressing types in a hierarchy that had no cycles. This work was carried forward into the 20th century by a whole host of mathematicians, including Russell. quote from the work of Kurt Gödel in 1944. Yes, yes, yes, yes, yes. By the theory of simple types, I mean the doctrine which says that the objects of thought are divided into types, namely individuals, properties of individuals, relations between hierarchy for extensions, and that sentences of the form, A has the property phi, or B bears relation R to C, etc., are meaningless if A, B, C, R, and phi are not of types fitting together. Now, if you squint enough, that sounds like the definition of objects and function signatures in Java. a hint of inheritance and polymorphism in there. Now guess who was working in the field of mathematical logic at the time? It was Alan Turing of course. The year was 1936. He had just finished his paper introducing the Turing machine. Have you read this book? The Annotated Turing by Charles Petzold. It's a fascinating book, and if you haven't read it yet, you should. It will convince you that Alan Turing had worked out many of the principles of programming, even though that was never his intent. As the century of progress progressed, the overlap between the theory of types and the practice of computing continued to grow. Fortran was designed in 1953, paper and only one year after I was born. Fortran had a pretty rudimentary type system but what types there were were strongly enforced. For example you weren't allowed to mix integers and real numbers in the same expression. COBOL allowed very fancy data types to be created but didn't from very humble beginnings to become the progenitor of the type systems in Pascal, Modula, and even C++. In short, the type systems of modern languages can trace their origins back to the type theory of Frieg, Russell, and Gödel. And the operation of modern digital computers use a logical structure that Frieg would recognize. Fascinating! Consider an integer. What's inside it? Is it 16 bits? 32 bits? 64 bits? And does it use 1's complement math or 2's complement math? Where's the sign bit? And might it perhaps be binary coded decimal? to the integer that represents 2 results in the integer that represents 3, you don't care. It doesn't matter what's inside a type. All that really matters are the operations that can be performed on that type. So a type is really just a bag of operations. Do you care what the internal structure of a floating point number is? Or do you just care that 2.0 is 0.5. Again, we don't really care what's inside a type. We only care what a type can do. We care about the functions and operations that it provides to us. So, what is a type? From the outside looking in, a type is nothing but a bag of operations. Oh, there may be data within it, existence of that data is hidden behind those operations. And that, of course, is exactly what a class is. From the outside looking in, a class is nothing but a bunch of methods. And the data within is private and hidden behind those methods. Does this statement in C define a type? Clearly not, because it doesn't define a set of operations. All it defines is a data structure. For example, we could define distance, translation, and rotation operations as follows. Now we've got a type, especially if we hide the definition of the data structure and prevent anyone from manipulating it unless they use the operations that we defined. Of course, in order to do that, we'll have to supply some getters and setters. This was really a pretty common style of C programming during the 70s and 80s. I was a programmer during that time, writing lots of C, and I remember that we also used another clever trick. type is identical to the point type in its first two elements. And this means that we can cast a described point to a point and then pass it into any operations of point without any danger. So what's the relationship between point and described point? Clearly, any function that takes a point into. But the reverse isn't true. Here's a function that takes a described point and you can't pass a point into that. This means that the relationship between the two types is asymmetrical. Described points can be used as points but points cannot be used as described points. And this asymmetry of usability defines the relationship. That is the subtype relationship. Described point is a subtype of point. This notion of subtypes had been trying to emerge from languages since the times of Fortran. In some sense, for example, an integer is a subtype of a real number. And Fortran made some concessions to this. loading point numbers. There were other attempts to deal with subtypes in languages such as ALGOL. The writings of the day referred to things like discriminated unions and other structures that approximated subtypes. But it was in Simula 67 that the notion of a subtype really came to the fore, because and inheritance appeared in a way that we'd recognize today. Indeed, the C++ inheritance scheme and all those that derived from it can be traced directly back to that language. It wasn't until 1988 that a formal definition of subtype was created. of the ACM and was authored by Barbara Liskoff. I didn't learn about this definition for another four years. It was in 1992 and I happened to be sitting in the San Jose airport reading Jim Copleen's wonderful book Advanced C++ Programming Styles and Idioms and in that book I read these words. If, for each object O1 of type S, there is an object O2 of type T, such that for all programs P defined in terms of T, the behavior of P is unchanged when O1 is substituted for O2, then S is a subtype of T. but eventually it sunk in. What Liskov was describing was that asymmetry of usability that we talked about in the previous segment. Subtypes can be used as their parent types. Or to say this a different way, imagine that you've got a bunch of users, we'll call them you, and they can use a type that we will call And then let's say that we've got a subtype of T, which we will call S. That means that all the users, U, should be able to use S. Without knowing it. That may seem like an obvious restatement of basic polymorphism. But as obvious as it may be, as we'll see in the next segment, consistently get wrong. By the way, as I sat in that San Jose airport reading Coplin's book, I noticed that he called Liskov's idea the Liskov Substitution Principle. And that got me thinking. principle. It had a certain ring to it. It sounded like the Pauli exclusion principle or the Heisenberg uncertainty principle, which, by the way, we're going to be studying in the next episode. And that gave me an idea. It made me think, well, maybe there are other software principles out there like this one. When I read of the idea that eventually grew into the solid principles. Notice that I used the inheritance arrow in the UML diagram to describe this principle. But is inheritance a necessary element? Well, in languages like C++, Java, and C Sharp it certainly is because those statically typed languages on inheritance to provide polymorphism. The only way that you can polymorphically deploy a method call in those languages is if the caller invokes the method on an instance of what it believes to be a base class but is really an instance of a class derived through inheritance. But in dynamically typed languages, the rule is different. Instead of invoking methods, we say that we send messages. The effect is the same, but the connotation is helpful. In a dynamic language, if you want to call a method, I mean send a message to an object, the compiler doesn't know the type of that object. So the language has to wait until runtime to determine whether or not you can call that method. or not you can send that message. If the object can respond to that message, then the method is invoked. Otherwise, a runtime error occurs. So no, no inheritance is necessary, so long as the two objects respond to the same message they can be used polymorphically. This is often called duck typing. and quacks like a duck and swims like a duck. I call that bird a duck. Rubber ducky, you're the one. You make bath time so much fun. I don't remember the rest of it. Rubber ducky, I'm simply in love with you. Now remember what a type is. A type is a bag of methods. a type, it too is a bag of methods. And since that subtype must be substitutable for its parent, it must have the same methods in it that the parent has, even if they have different implementations. In statically typed languages, we achieve this by using inheritance. In dynamically typed languages, we achieve it by writing the same methods into the subtype. But in of the subtype is that it's substitutable for its parent. So, from this point on in the episode, we're going to use that inheritance arrow to mean subtype. When the Liskov substitution principle is violated, the result is usually a refused That term is one of the code smells that appears in Martin Fowler's wonderful book, Refactoring. If you haven't read this book, you should, and don't let the 1999 date scare you off. This book is as relevant today as it was back then. I remember reading it when attending hockey practices that my son was doing, and although the parents sitting next to me might have thought that I was going on out on the ice? Often as not, my cheers were related to what I was reading in this book. A refused bequest occurs in dynamically typed languages when you send a message to an object and the object doesn't have the corresponding method. Most languages will cause some kind of exception to get thrown. So for example, in Ruby, let's say that I that I create an instance of this class and I try to send it the f message. It will of course throw a no method error. That's a refused bequest. A more subtle form of refused bequest occurs when a method in a subtype does something that the client of the supertype does not expect. that users of the base class don't expect. Or, the derived class might cause a side effect that the users of the base class don't expect. Clearly, that kind of refused bequest can occur in both static and dynamic languages. Let's look at a classic case. Let's say that we have a class named Rectangle that's used by some program. This class has instance variables such as height, width, and top left point. It also has methods such as area and perimeter. And of course it has the normal array of getters using this rectangle class for years. There are instances of rectangles getting passed around all over the place. Now let's say that we discover a new requirement for squares and we decide that in order to meet that requirement we need a new class named square. Square, between the rectangle and the square. Clearly, a square is a rectangle. And we all know what those magic words, is a, mean, don't we? It means that in a statically typed language, we'd use inheritance. And in a dynamically typed language, we'd use duct typing. And in either case, square would be a subtype of rectangle. inheritance we can already see a problem rectangles got two variables height and width square is going to inherit both of them which means square is going to be too large now look memories cheap maybe it doesn't bother you that instances of square have one extra variable but it ought to because it means there's But never mind that. I mean, maybe we just don't care about the extra memory. It turns out that Square is inheriting something even stranger than that extra memory field. Square is inheriting setHeight and setWidth. What does Square want with a setHeight and setWidth method? What it really wants is a setSide method. to it, but it would still be inheriting set width and set height from rectangle. And that's just strange. Now you Ruby programmers may think you just dodged a bullet, because you don't have to use inheritance to inherit the rectangle into the square. And you're right! is a duck type. It's going to have to walk, swim, and quack like a rectangle. It will be passed around the system through dozens of instances as though it were a rectangle. And that means it's going to have to have set width and set height operations. So we're stuck with these methods that don't make a lot of sense for a square. But at least consistent with the behavior of a square by overriding them to preserve squareness. In the Java case, we can override the two methods to set both the height and the width. In the Ruby case, we can override both methods to set the side. And so now, though some of the methods seem out of place, square, and the rectangle behaves like a rectangle. So, all is well. Well, maybe everything isn't all well. Imagine that there's a module in our system that calls set width on what it believes is a rectangle. Does that module have the right to expect Of course it does. It's using a rectangle. And when you set the width of a rectangle, the height doesn't change. But if a square gets passed into this module, then when the module changes the height, the width will change. And that's something this program didn't expect. What happens when you call a function and it does something you don't expect? We've got a name for this. We call it undefined behavior. Do you know what the definition of undefined behavior is? Undefined behavior means that it's going to work perfectly in your laptop and in your development environment but as soon as you ship it to the customer Or if you'd rather, it means you're going to spend weeks debugging some problem that occurs once per day or so. Or if you'd rather, it means that you'll have to dig through megabytes of stack traces trying to figure out why the heap got corrupted a billion instructions ago. someone passed you a square when you were expecting a rectangle. What will you do? How will you fix it? You know how you're going to solve this, don't you? You're going to put an if statement in. An if statement with an instance of. You're going to ask that rectangle square. And that hangs a dependency from the midst of our program onto the new class square and that is an open-close principle violation. Every refused request, every violation of the Liskov substitution principle is a latent violation of the open-close principle because in order to repair the damage the refused bequest. We're going to have to add if statements and hang dependencies upon subtypes. And when the open-close principle is violated, well, you remember what happens then, don't you? The software gets rigid and fragile. And then managers and customers will come to the conclusion and don't know what the hell they are doing! So what's the solution? How can we represent both squares and rectangles in our software? Some folks have suggested that in static languages we should invert the inheritance relationship and have rectangle derived from square. this solves one of the problems, sort of. It means that we won't have too many variables in the rectangle. The name of the variable in square is going to be side, but a rectangle needs two variables, so what should we call the other one? Other side? But the real problem square has the right to expect that when he sets the side to 5, the area will be 25. If you passed him a rectangle, that wouldn't be true, and so there'd still be a refused bequest. There'd still be a violation of the Liskov substitution principle. Another common solution to this problem is to make the squares and the rectangles immutable. The problem goes away. Well, much of the problem goes away, not all of it. I mean, the square still has one variable too many, and the names height and width are still going to be prominent in the square. So making the types immutable doesn't solve everything. The best way to avoid all these difficulties is to treat square and rectangle as completely different types, never try to pass a square into a function that expects a rectangle. I'm sure you think that's nuts. I mean, isn't a square a subtype of a rectangle? Isn't a square a rectangle? Doesn't the is-a relationship hold? Of course it does. A square is a rectangle. A square is a bona fide, absolute, perfectly about it. The problem is that this is not a rectangle. It's a piece of code and that piece of code represents a rectangle but it is not a rectangle. Here's the thing about representatives. For example, imagine two people who are getting divorced. Each one of them has a lawyer which represents them. It's very unlikely that those two lawyers are themselves getting divorced, because the representatives of things do not share the relationships of the things they represent. So while geometrically a square is a rectangle, The software representatives of those concepts don't share the subtype relationship. This may bother you at some deep level. Perhaps you think the whole point behind OO is to create models of the real world. And if you can't model the ISA relationship, well then, what's the point? But no real world principle has been violated here. relationships of the objects they represent is a real-world principle. It's important to remember that when you create models in software, the things in your models are mere representatives. Here's another example. Consider the type integer, and remember that the internal representation of integer is not important. Well consider the class real number. It is undeniably true that every integer is a subtype of a real number. You can pass integers into any function that accepts a real number. So every integer is a real number. It's also true that every real number is a complex number. A function that can manipulate a complex number can also manipulate a real number. But every complex number holds two real numbers inside it. One for the imaginary part and one for the real part. This makes a really pretty UML diagram. And the UML diagram makes perfect sense until you convert it to code. Once you convert it to code, it becomes nonsensical just like Russell's Paradox. This is a refused bequest of the first order. You can't even create an instance of these objects without blowing the stack. So there's a perfectly good example of a real-world model that makes perfect sense in the real world, but makes no sense at all inside the computer. And why? Because that integer class is not actually an integer. It represents one. The real number class is not a real number. It represents one. The complex class represents a complex number. And the relationships between objects are not shared by the representatives of those objects. These examples have all been pretty academic. example and we'll see just how damaging a violation of the Liskov substitution principle can be. However, before we leave the realm of the academic examples, there's one more I should share with you, and it's actually much more pragmatic than the others. You're not going to like it. In fact, you'll struggle to try and prove it wrong, but you'll fail. S is a subtype of T. A list of S is not a subtype of a list of T. To make this concrete, let's suppose that circle is a subtype of shape. Then if we create a list of circle, we cannot pass that list into a function that expects a list of shapes. because that function may put a square in the list. Here, look at this code. Notice that the function g creates a list of circles, and then it passes that list of circles into the function f. But the function f takes as its argument a list of shapes, and then in the body of f, it puts a square in the list. Fortunately, the Java compiler makes this an error, because a list of circles cannot be passed as a list of shapes. This is another example of the principle of representatives. A list of shapes represents a group of shapes. The list of shapes is not a shape in and of itself. And the representatives of things do not share the relationships of the things themselves. On a side note, the composite pattern adds a lovely little wrinkle to this issue. We'll be looking at that in an upcoming episode. This principle can be generalized further. Given that S is a subtype of T, then the generic class P is not automatically a subtype of the generic class P. Hwyl. We'll be right back. We'll be right back. We'll see you next time. 23 years later Bertrand Mussel Bertrand Mussel Bertrand Mussel I grabbed my eyebrow Alright, alright, oh We call it undefined behavior Do you know what unbehame define your means? Do you know what unbehame define your means? The knife jams. It jams on you and you can't pull it out. Is a real world principle. And now I have no idea what else to say because I've forgotten my lines. Now it should be clear that if you carefully express your pre and post conditions then someone will flush the toilet right while you're talking. And you get plumbing type safety. There will be no 2 a.m. Frank Pone calls. I wonder who Frank Pone is. Frank Pone, Frank Pone, calling Frank Pone. In the end, the only way to avoid the exception is to put the dreaded if instance of statement in.