Hi, I'm Uncle Bob, and this is Clean Code. episode by Uncle Bob. This is the third in our series on the Solid Principles. In the last episode, we learned about the Single Responsibility Principle. We learned that classes have responsibilities to their users, and that those users can be classified into actors based on the roles that they play. We learned that a responsibility was a family of functions that served a particular actor. two different values of software. We learned that the secondary value of software is the behavior that meets the customer's needs, but that the primary value of software is the flexibility that allows that software to continue to meet the customer's needs throughout its lifetime. We learned that one of the ways in which we keep the primary value of software high is by carefully allocating functions to classes and modules. contain more than one responsibility, then the system can become fragile due to interactions between those responsibilities. We looked at several examples of code that had varying degrees of single responsibility violations and we also looked at how that code could be improved. We studied several general solutions to single responsibility violations such as separations, None of these solutions is perfect. Welcome to engineering. Finally, we looked at the mastermind case study, and then I showed you a little bit about faking a rational development process. Now in this episode, we're going to talk about the open-close principle. We'll discuss Bertrand Meyer and what he meant by open and closed. code of a system to be open for extension but closed for modification. We'll look at some code that violates the open-close principle and we'll show how that violation causes all the design smells that we saw back in episode 8. Then we'll change that design to conform to the open-close principle and we'll see how that conformance eliminates those design smells. shatter your hopes and dreams by showing you that the promise of the open closed principle is actually a big lie and then we'll resurrect those hopes and dreams by describing a development process that restores the truth behind the promise of the open closed principle so you all set yourselves on down now and get yourselves Open up one great big can of worms here called the open close principle. Gr-rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
campfire coal or the sun glows because of its own heat, the distribution of the color and the intensity always follows this curve. This curve always has the same shape. Oh, at higher temperatures it gets taller and it moves leftward towards the blues, and at lower temperatures it gets shorter and it moves rightward towards the reds. But the And the shape of that curve is pretty interesting. It's an x squared curve on the low frequency red side, but it's an e to the minus x curve on the high frequency blue side. It's strange enough that a single process would be described by two different equations, but it's that hump in the middle that's the really interesting part. century. It wasn't until 1890 that our measurement technology finally got good enough to map out just the high frequency side of the curve. The low frequencies remained a mystery for several more years. In 1896, Wilhelm Wien came up with an e to the minus x based formula But as measurement technology improved and the lower frequencies came more and more into view, it became clear that Wien's approximation became less and less accurate the more towards the red side of the spectrum

we went. Wien had a friend whose name was Maxwell Plank and Maxwell Plank decided that he'd try to tackle the problem. He played around with ideas but never really got anywhere. Then in 1900 he decided to try a trick that he'd read about in a paper written by Ludwig Boltzmann in 1877. Boltzmann you see was trying to calculate the entropy of a gas and he used a technique common gas had an energy that was an integral multiple of some constant. No fractions of that constant were allowed within his calculations. In short, Boltzmann assumed that the energy of the gas molecules was quantized instead of continuous. Boltzmann reasoned that if he made the constant of quantization small enough, then he could in a gas. This worked pretty well for Boltzmann. It gave him a decent approximation and that approximation helped him later to completely eliminate all the quantization and use continuous energies instead. Planck hoped for the same kind of help so he applied a variation of Boltzmann's trick. or absorbed in integral multiples of some constant proportional to the frequency of that radiation. Or, to state it mathematically, E is equal to nVH, where E is the energy of the radiation, n is some integer, V is the frequency of the radiation, and H is the constant of proportionality. Nowadays, we know that as Planck's constant. This formula fit. The quantization was a compelling explanation for why the low part of the graph followed an x squared curve, but the highs cut off with an e to the x decay. The closeness of the fit was quite disturbing to Planck, because Planck did not believe that energy was quantized. He thought it was continuous. And the fact that his formula fit so well caused him to wonder whether or not it was the mechanisms of absorption and emission that were quantized. forth or swallowed energy in discrete chunks had more to do with the atoms than with the energy itself. It was Einstein in 1905 who changed all of that. This was Einstein's golden year, the same year he published his special theory of relativity. The paper that changed everything was entitled, On a Heuristic Viewpoint on the Production of light. In this paper, Einstein coined the term light quantum and he suggested that energy itself was quantized. What an incredible thought. Energy, the motive force of the universe, comes in discrete little packets that can't be further subdivided. This thought is so radical that even Planck didn't until 1908. That incredible idea, which won Einstein the 1921 Nobel Prize, changed everything. Without that idea, there could not have been a transistor, the fundamental unit of our electronic empire. What would our lives be like without transistors? That thought that came from Einstein's own mind has sown the seeds of destruction for his other great theories, the theories of relativity. Because relativity and quantum mechanics are about as incompatible as two theories can be. One demands that the universe is a continuous curvature of space and time. The other demands that the universe is made up of pixels. or the other or both are going to have to change. But we'll leave that discussion for another time. In 1988, Bertrand Meyer wrote this classic book, Object Oriented Software Construction. Nowadays we would view some of the ideas in here as a bit quaint and disagree with, but for the most part there's no doubt at all that this book was a milestone in the development of object-oriented theory. In that book, Meyer coined one of the most important principles known to software design, a principle that is at the moral center of

software architecture, the open-closed principle. but closed for modification. What the hell does that mean? It sounds like an oxymoron, doesn't it? How can something be both open and closed? When Meyer says he wants a module to be open for extension, what he means is that it should be very simple to change the behavior of that module. But closed for modification means that the source code shouldn't change. So, Meyer wants it to be easy to change the behavior of a module without having to change the source code of that module. But this still seems like an oxymoron, doesn't it? I mean, how can you extend the behavior of a module without modifying it? Ah, but of course there is a way, and we've already seen it. Remember how we implemented the high-level copy policy up in the copy module, but we put the keyboard and printer down in low-level modules? We arranged the dependencies in that application so that we could change the lower-level details without even recompiling the copy module. That meant that we could change the behavior of the copy module to add new devices, for example, module at all, which is exactly what Meyer demands. That's pretty cool. What trick did we use to make that work for the copy module? What magic did we apply? Our incantation was abstraction and inversion. We inserted an abstract interface between the copy module and the devices. dependencies so that the copy module did not depend on the devices. Instead, both the copy module and the devices depended on a new abstraction that we called file. And that's how you conform to the open-close principle. Whenever you have a module whose behavior you would like to extend without modifying it, you separate the extensible behavior behind an abstract the dependencies around. Let's try this. Let's suppose that we're writing a point of sale system. Part of the checkout algorithm might look like this. We loop through all the items, getting the price of each item, and then adding the price and the item to the receipt. Once we're done looping through cash payment and then we add that payment to the receipt as well. This works great for cash, but we also have to extend this algorithm to use credit cards. Now we could use an if statement like this, but that violates the open-close principle because we've modified the algorithm in order to extend it. Or we can conform to the open-close principle by separating the extensible with an abstraction, like so. In this case, payment method is an abstract interface which has two concrete derivatives, cash payment method and credit payment method. Notice how the dependencies have been turned around. The implementations of the payment methods now depend upon the abstraction, and our checkout algorithm has become independent of those implementations. Now we can extend the checkout module with new payment methods without the need to modify the checkout module. Indeed, in languages like C-sharp or C++ or Java, we don't even have to recompile the checkout module. We have truly conformed to the open-close principle. a system that's open for extension but closed for modification, it means that every time you add a new feature, you'll do so by adding new code, not by changing old code. I do believe that bears repeating. If you design your systems in conformance to the open-close principle, then when you modify them, you can do so by adding new code, not by changing any old code. That's quite a thought, isn't it? I mean, if the old code doesn't ever have to get modified,

then it can't rot. You could write your modules once, nice and clean, and never have to worry about somebody else messing them up. It also makes sense from the customer's point of view. When they think about adding a new feature, they think they're adding a new feature. They don't think So from their point of view, it's strange that we would have to modify a bunch of code in order to add a new feature. So code that conforms to the open-close principle also conforms to the way customers think about software. Is this possible? Can you really design a system that conforms to the open-close principle and never needs Theoretically, yes, it's possible, but it's hard, very hard, so hard in fact that often it's not practical. First, there's the problem of main. Remember back in episodes 4 and 5 how we said that main should be separated from the rest of the application by moving it across a boundary and causing all the dependencies that cross boundary to point away from main if you follow that rule then main just can't conform to the open-close principle because it's responsible for loading resources and strategies and factories and stuff like that into the application if ever those strategies and factories and resources have to change main is going to have to change there are tricks you can use to get around that issue dependency injection frameworks. But the cost is usually larger than the benefit, and in the end all you really do is move the problem, you don't solve it. But never mind Maine. There's a far more serious problem called the crystal ball problem that makes it a virtual certainty that we cannot completely conform to the open-close principle. Of course not! It may be difficult to get entire systems to conform to the open-close principle, but it's not at all difficult to get functions or classes or small components to conform. It turns out that the difficulty in conforming to the open-close principle is a matter of size. Small things like functions, classes, and small components can be made to conform very easily. in large systems where we start to run into trouble. Let's look at some of that trouble now. In a 60 minute video, it's tough to explore the problems of a large system. So we're going to play a little game here. I'm going to show you a small system one. The code I'm going to show you is smelly. It's a mess. And behind that mess are a bunch of open-closed violations that make this code very difficult to manage. The truth is that the code I'm going to show you is so small that it'd be pretty easy to fix. But I want you to pretend that it's actually part of a much larger, interconnected corporate accounting We're going to focus in on just one function in this large interconnected accounting system. It's the function that prints expense reports. Here it is. That's quite a mess packed into that 26 lines, isn't it? I think to understand that mess, we should probably be looking at the unit tests. And yes, there are unit tests. remember back in episode 6 we discussed how unit tests provide unambiguous documentation the first test shows us that when you print an empty report whatever that is then what you get is a set of totals at the bottom and a dated header at the top and I suppose that makes some sense the second test gives empty meant in that first test. This test adds an expense to the expense report. So presumably an empty expense report is one in which no expenses have been added. In this test we've added the dinner expense. We can see the dinner showing up on the report and we can see it reflected in the totals. So all this makes perfect sense. The third test adds two

meals to the report, We can see these meals on the report itself, and we can also see their amounts reflected in the totals. The fourth test adds two meals and a car rental, and we see all three of those expenses printed on the report, as we might expect. But now the totals have diverged. The meal total counts only the breakfast and the dinner, whereas the final total counts all the expenses. us something interesting. It would appear that breakfasts over $10 and dinners over $50 are marked with an X. The name of the test implies something about overages, so perhaps the accountants desire some visual cue for expenses that are over some particular limit. Now if you think those tests are well structured, I'd like you to think again, because They're based on a huge violation of the single responsibility principle because they test the business rules through the user interface. And that's a huge no-no. Remember that episode I promised you on advanced test-driven development? We're going to talk more about that there. Now let's look at that code again. This is where the header gets printed. Next, there's a loop that prints all the expenses. And I'm just itching to extract that out into a method named print expenses. Finally, there's a bit of code down at the bottom that prints out all the totals. And you know what I'm thinking about them, don't you? You see all those divide Actually, that divide by 100 occurs two more times in this function. You think that might be an opportunity to perhaps extract a method? And what do you think we should call that method? How about, hmm, pennies to dollars? Now let's dive right inside that loop. First, we check to see if the expense is a meal. And if it is, we add its value to the meal total. Then we use this horrible switch statement over here to convert the expense type into a string of the corresponding name. We use the string TILT to represent an unknown type code. We never expect to see the string TILT printed, do we? As a young software developer of 24 years, I worked at a company where we tested the quality of telephone lines. up on the console. Deep within the software that generated that report, we would emit the word tilt whenever we ran across impossible situations. Of course, we never expected the customers to see the word tilt. Nevertheless, we received field service calls with relative frequency. What does tilt mean? Finally, we print the expense. And you can see the format. It's simple enough. There's a placeholder for the overage indicator, the expense type, and the expense amount. The overage indicator is calculated with this horrible ternary operator that compares the expense type and amount against these hard-coded overage limits. The result is either an x or a space. And finally, we do the pennies to dollars trick on the amount of the expense. We've already figured out what the expense class is, but here it is just in case you were wondering. It's nothing but a dumb data structure with an enum for the type and a constructor for convenience. Clearly this code is a mess. That function is too large. It's got embedded constants within it. There's duplication, and it's loaded with single responsibility violations because we mixed business rules with messages and formatting. But there's something very insidious hiding beneath that mess. And that's the problem with messes. They hide the rotten structures that lie beneath. What is that rotten structure? This module violates the open-close principle flagrantly. If I want to extend the business rules, I've got to reach into

this module and fiddle with it. If I want to change the messages and formatting, I've got to modify the inside of this module. This module is closed for extension and open for modification. For example, what if we wanted to add a new kind of meal type, like a lunch or a snack? What's the first line of code that would have to change? Right, it'd be the enum in the expense class. We'd have to add the new type there. modules in this large, interconnected accounting system depend upon the expense class? It's probably a lot of them. In fact, let's stipulate that it is a lot of them. Alright then, how many of those modules depend upon lunch? Probably not very many at all care about lunch, and yet they'll all be impacted because we have to extend that enumeration. Every module that redeployed. And if there's a database representation for expense, the scheme is going to have to change. And if the expense instances are transmitted to services, then the service interfaces are going to have to change. Changing something that is as core to the application as the expense class can be very burdensome to a large system. In fact, the developers may consider it so indirect hack like maybe adding a lunch flag to a meal attributes map. What design smell is this? It's rigidity. Even though the design change is small, the impact is huge, so the system resists change. It's rigid. for lunch are just so important that we go ahead and make them regardless of how difficult they are. The next problem we're going to face is even worse because that problem is going to be visible to our customers and our users and our managers. Look at that switch statement that selects the name or the conditional expression that looks at the meals or that ternary operator that looks for overages. accounting system how many other such switch statements and conditional expressions on the expense type are scattered throughout the system consider that this system has to deal with taxation reimbursement invoicing contracts and many other complications that have to do with expenses so we can expect to find lots of switch statements and conditional expressions that on the expense type. How will we find them all? And when we do, how will we know how to modify them for lunches? How likely is it that we'll miss one? How likely is it that we'll incorrectly modify one? What design smell is this? It's fragility. Fragility. Imagine how awful it would the taxation calculations. Of all the design smells out there, fragility is the one we want to avoid the most. Why? Because when systems are fragile, managers and customers come to the conclusion that the developers have lost control and don't know what the hell they are doing. But the problem is even a fundamental change in approach. Their original business model had been to sell a fully featured, end-to-end, enterprise-ready, absolutely complete accounting system to their customers. It could be configured within limits, but it otherwise had all the features anybody needed. Now they've realized that their target market includes a bunch of small options in the system. So they've decided to break the system up into components. They simply plan to give away the base component, the engine that has no features, and then they'll charge for all the little plug-ins. Plug-ins for the meal expenses, plug-ins for special lunchtime expenses, plug-ins for car rental expenses, and so on and so forth. what they want us to do is to create a set of gems or DLLs or jar files that we can deploy and sell independently. Unfortunately we're going to have to tell them that

the current structure of the system makes this impossible. We cannot cut the expense report printing function up into components because it's conditional statements that depend directly on all the things that management wants us to deploy separately. Look at this fanout diagram. It shows the print report function and all the little bits of functionality that management would like us to deploy separately. Notice those arrows. They show how the print report function depends on that functionality with contains relationships. all that functionality. Even if we extracted out all those functions, the arrows would still point in the same direction. The print report function would still depend on those extracted functions, and so we still couldn't independently deploy them. This is the design smell of immobility. There's functionality out there that we would like to deploy separately and conditionally, but we can't do that because of the way this system is put together. As it currently stands, deploying this system is an all or nothing proposition. Rigidity, fragility, immobility, not to mention just a plain old mess. This code is pretty bad indeed. So what are we going to do to fix it? Are you ready for the lie? Of course there is a way that we could have solved all these problems. If only the programmers had known about good object-oriented design principles. If only they had known about the open-closed principle. Here, let me show you how it could have been. experience the clean, cool, soothing flow of clean code as it was meant to be. We begin with the expense class. Oh, the simplicity, the elegance, the serenity and confidence. This class simply holds the value of the expense and provides simple abstractions for the business rules such as is meal and is overage. Dinner expense is nothing more than a finely tuned derivative. It implements the is-meal and is-overage abstractions appropriately, without fuss, without muss, just simple function implementations. And so too are car rental expense and breakfast expense implemented. They're almost too simple to talk about. And then there's the expense report class. No strings, no formatting, nothing but the totaling functions that capture all the business rules. This class is the epitome of single responsibility principle decoupling. Likewise, there is the expense reporter class, which is responsible for all the messages magical constants. Nothing but simple lexical manipulations. It is the ultimate in single responsibilities. I'd like to draw your attention to something down here in the expense reporter. Notice how it gets the name of the expense from the namer. That namer holds a reference to the expense namer abstraction. What a marvel of minimalism! What a paragon of abstraction! carefully crafted little method that separates the expense reporter and everybody else who wants to know the names from the names themselves and where are those names they're here in the expense report namer class oh the rye panache the flare the sheer ill-lond of hiding this gritty type check in such Isn't it a delight to the eye and to the mind, a fusion of rigorous engineering and superlative artistry? And behold how this marvel of software science, this masterpiece of abstraction behaves when we posit the addition of a new meal expense, the lunch expense. Does the expense class or any of its derivatives require modification? They require neither recompilation nor even redeployment. They are oblivious to the change. So too are the expense report and the expense reporter class, oblivious to this change.

No hand must move mouse, no fingers to press keys, no mind must ponder the complications of the change, because no change is required. the lunch expense requires only that we create the lunch expense class itself. There's also a small modification to the expense report namer class where we have to add an if-else statement. Perhaps you think this is a problem, a chink in our lovely edifice, a ding in our beautiful structure, but no, look at the architecture diagram. The expense derivatives and the expense report namer point inwards towards the application. Those parts of the system that are open for extension and closed for modification live on one side of the boundary and they are protected by abstractions. Those parts of the system that must change implement those abstractions from the other side of the boundary. This design is nowhere near as smelly and fragile as the design in the previous segment. We don't have to go hunting for any switch case statements or if-else statements. We don't have to wade through lots of conditional logic looking to see whether or not we can put our new feature in. All we have to do is add the new derivative and add the name to the expense report namer class. You just get the idea that you actually do know what you're doing. But it's better than that! Oh yes it is! It's much, much better than that! Do you remember that new business model that our managers wanted to try? They wanted to give away the base component and then treat all the features as plug-ins that customers could buy. All the things they want to give away are on the far side of the line. All the things they want to charge for are on the other side of the line. The plug-in structure that we need has come for free. How perfect is this? How incredibly suave, how brilliant and bold, how inseparable from the moral center of system architecture. responsibility principle and creating the abstractions that enable the open closed principle we've solved all the problems and we've created a simple elegant beautiful practical and moral design Wow I'm going to go to bed. now you just wait one cotton-picking minute I know you're really smart and all so I just got one little question for you I know that hyper whammy doodle design here is certainly gonna help man a lunch expense off grant you that but what if them our customers over there decide what for every meal expense over $20 on weekends. What you gonna do about that, Smarty? Are you gonna tell me you can add that feature without modifying any existing code there, buck-hole? Yeah, that's what I thought. Looky here, you don't even have a date expense class. How the hell are you going to know if it transpired on a weekend? Uhhhh... And another thing you ain't got is an abstract method telling you whether or not that expense is transportation related. Uhhhh... ... And you're also going to have to change that expense reporter class just to add you. So that whole spiel you just unloaded on us about that var open closed principle was just a load of dingoes kidneys wasn't it? Okay okay now wait a minute wait a minute nobody told me about any new feature that checks for transportation related expenses If I had known, then I could have conformed to the open-close principle. I could have arranged it such that the new feature could be extended without modification. If only I had known. But you didn't know, did you? And how were you supposed to know something like that? Are you trying to tell me that this here open-close thingamawhat'sit only works if

you already If you gotta be able to predict the future, it ain't gonna be much good to you, son. Now lookie here, Sonny. I don't know who taught you to program. But seems to me your education might be just a little bit short of a full one, if you know what I mean. If I've learned one thing over the years, Sonny, it's that customers do the unexpected. freedom mat but those customers will find the one thing you didn't think of and that's the thing they're gonna change on you he's right of course it's easy to sign abstractions that protect you from future changes if you know what those future changes are going to be but I don't have that kind of a crystal ball any ability to change the one thing you forgot to protect yourself from. This is the dirty little secret about object-oriented design and the open-close principle that people don't like to talk about. It only protects you from change if you can predict the future. What do we do about this? How can a principle be useful if it depends upon presence, the ability to see the future? What good is the discipline of OO if it requires perfect foreknowledge? Over the last 30 years, the software industry has struggled to address this issue. As part of that struggle, we identified two major approaches for the crystal ball. The first is to think really hard. You carefully consider the customer and the problem domain. You create a domain model that anticipates the customer's needs and desires. You adorn that domain model with abstractions making them open for extension but closed for modification. And then you continue in that vein until you have thought up everything that could possibly change. We call this approach Big Design Up Front, or BDUF. And the problem with BDUF is that it creates large, top-heavy, complex designs aren't necessary. As useful and powerful as abstractions are, they're also expensive. They create indirections. They disconnect those things that we think of as being connected. They invert dependencies. And they make it hard to follow from cause to effect. useful to us also make them costly. When we need those abstractions, then the cost is bearable. But when those abstractions are anticipatory and not currently needed, well then the cost can be overwhelming. The cost of maintaining the large, over-engineered designs created by big design up front is often prohibitive. I've seen many development teams hobbled to near immobility designs that attempted to anticipate their customers' future actions. And remember, customers seem to have an uncanny ability to change the one thing you forgot to protect yourself from. And when they do, changing an over-engineered design will be a lot harder than changing a simple design. design is both pragmatic and reactive. The best way to demonstrate that is with a metaphor. Imagine you're part of a squad of soldiers pinned down by enemy fire. You're hunkered down in a foxhole with your buddies while bullets whiz by overhead. If you could focus your fire on the enemy you could probably break out and win the battle. The problem is you don't know where You don't know what direction he's firing from. And if you stand up to look around to find him, he'll cut you down before you've got a chance to focus your aim. And so the sergeant makes an executive decision. He says, Johnson, stand up. And now you know the direction that the bullets are coming from. Agile design is like that. You do the simplest thing you possibly can. and then the customer starts shooting at it with change requests. And then you know what kind of

change requests are likely. One of the best predictors of change is past change. Once you know that something is likely to change, you can protect yourself from that kind of change in the future. So instead of trying to out-think the customer by predicting absolutely every change that customer might make abstractions that would protect you from all those changes. What you can do instead is simply wait for the customer to make a change and then invent the abstraction that will protect you from any further occurrence of that change in the future. Agile designers deliver something simple every week or so. And when customers make changes, those Agile designers refactor the code, of change easy to make in the future. Thus the system becomes open for extension, but closed for future modification. Of course in practice we live somewhere between these two extremes. We avoid big design up front, but we also avoid no design up front. and creating a decoupled domain model up front. But we err on the side of the small and the simple. Our goal is to establish the basic shape of the system and not to think through every single little detail. If you overthink the problem, you'll create a lot of unnecessary abstractions that are very expensive to maintain. often and refactoring based on changes that customers make. In fact, this is where the open-close principle really shines. But doing this without a simple domain model will often leave you with a undirected, chaotic structure. Let's say that you're part of a team of 10 developers. And you've been given a new project, a Greenfield project, months. What design process best conforms to the open-close principle? The team should spend a week, possibly two, scoping out the initial requirements and coming up with a simple architecture and domain model. The requirements should not be precise and the domain model should not be detailed. The team might write some code at this time, but the focus is on the initial requirements and the architecture, not on implementing features. Some teams call this iteration zero. In a future episode we'll discuss the structure and form of these initial requirements. For now we'll simply call them user stories. These user stories, and the architecture that surrounds them, will not be correct in iteration zero. In fact, they don't need to be correct right now, because all we really need in iteration 0 is to establish a way forward. Then the team should begin to work in one- or two-week iterations. The goal of each iteration is to get something executable in front of the users or their appropriate proxies. When users see something execute, they start thinking of changes, and those changes will be the basis of the abstractions that the team uses in order to conform to the open-close principle. We'll discuss the process for these iterations in an upcoming episode. For now, just think of them as design and programming time. Each iteration should begin with a simple design session, where the team members look at the changes that have been requested, and then figure out how to apply the open-close principle Then the developer should refactor the code and add new features with those architectural changes in mind. They should follow the discipline of test-driven development that we learned back in Episode 6 and keep their code clean and easy to change. Each iteration should see a growing conformance to the open-closed principle. Each iteration should clarify and intensify the boundaries that describe the architecture and the managed dependencies that cross those boundaries. Every iteration should

see more code that is open to the expected kinds of extensions and yet closed for modification. If you follow these guidelines, you'll be able to create systems that are very flexible, robust, and mobile. However, this is engineering and not magic. There's no way to perfectly conform to the open-close principle. You just can't think of everything. No matter how closely you follow the rules, no matter how careful you are, eventually the customers are going to think of some change that will force major modifications throughout the structure of the entire system. That's just not feasible. Your goal is to minimize that pain. And that's what an open-close compliant design will do for you. Remember, the open-close principle is the moral center of system architecture. And while moral perfection may not ever be attainable, it's certainly worth striving for. Alright, so let's recap. In this episode we started by talking about Bertrand Meyer and his brilliant insight that a module can be both open for extension and closed for modification. This implies that you can create systems in which new features are added by adding new code as opposed to changing old code. principle is at the moral center of system architecture. I'd like you to consider that statement very carefully. To the extent that a system is not open for extension and closed for modification, that system is immoral. We looked at one system that flagrantly violated the open-close principle, and we saw how that violation led to all those awful design smells like rigidity, fragility and immobility. And then we looked at a solution that perfectly conformed to the open closed principle. Or so we thought. It was simple. It was elegant. It was beautiful. And it was a lie! Then we saw that in order to perfectly conform to the open closed principle, for in order to create a system that is fully open to extension and closed to all modification one must be able to perfectly predict the future but then we found that all hope was not in fact lost that by using an iterative process with lots of feedback and refactoring we could in fact develop systems that of a bar. And so we've come to the end of yet another episode and boy we sure covered a lot of ground didn't we? I mean my head is spinning and I've got to go for a walk to clear my head but we've still got so much to talk about there's three solid principles left and then there's all the component level principles and a raft of design patterns there's advanced test-driven development there's the whole suite of agile practices you're not gonna want to miss exciting episode of Clean Code, episode 11, the Liskov Substitution Principle. Come on you dogs! Time to go out. Does the expense class or any of its derivatives require modification? Not at all! Not at all, I say! Ha! Good. Action! I'm going to go get some food. I'm going to go to the bathroom.