You I'm going to go get some food. I'm going to drink a lot of water. Hi, I'm Amanda. Hi, I'm Princess. Hi, I'm Alexis. Hi, I'm Luna. Welcome to another episode of Clean Code. Clean Code. Clean Code. Clean Code. Your master's in the business. To begin a journey of... In this episode... Episode. Episode. By Uncle Bob. Uncle Bob. Uncle Bob. Enjoy! Welcome, welcome to episode 15, Solid Components. Here, let me take your hat. You remember the previous episodes, episodes 8 through 14? design. Those principles included the single responsibility principle, the open-closed principle, the Liskov substitution principle, the interface segregation principle, and the dependency inversion principle. These were all principles about class design. They told us what methods should go into classes and how the classes should depend upon each other. Now, in this series larger scale entities made up of many classes. We're going to study the three principles of component cohesion, which tell us what classes ought to go into a component. We're also going to study the three principles of component coupling, which tell us how components ought to be related to each other. And we'll see the eerie connection between the solid principles of class design and the that. We're going to have to study white dwarf stars and then we're going to learn what components are and the roles they play in design. Ten billion years from now the corpse of the Sun will be a white dwarf star. An after the violence of its death throes. As I told you back in episode 5, the burned-out remnant of our once great Sun will be roughly the size of the earth and will retain about half the Sun's original mass. It'll be so dense that a teaspoon of this material will weigh several tons. But fueled its exuberant youth. The nuclear reactions that supported it against its own furious gravity ceased eons ago. But that leaves us with a question. What fights that gravity now? After all, the gravity is still there, relentlessly trying to pull that white dwarf in on itself. Back when it was a true star, it resisting it now? Something must be, otherwise it would collapse. You might be thinking that the answer is obvious. It's made of carbon. Carbon is solid. It's the solid carbon that's resisting all that gravity. But if that's what you were thinking, you're wrong. Solidity is merely a manifestation of the electromagnetic force. For example, it is balls that makes them feel solid. OK, then, you say. It's the repulsion of the electrons in the outer shells of the carbon atoms that resists the gravitational onslaught. But no. You see, the carbon atoms in a white dwarf star have no electrons of their own. The temperatures at the time that the white dwarf star normal atoms to exist. Indeed, this is a characteristic that the white dwarf inherited from its parent star. The temperatures in the core are so high that the electrons and protons are moving too fast to stick together. The result is that the electrons form a kind of electronic fluid that flows between between the nuclei of the carbon atoms in the white dwarf. These electrons form a single system, and within a single system they must obey a rule called the Pauli Exclusion Principle. In 1925, Wolfgang Pauli, in an attempt to explain why the electrons within atoms organize Exclusion Principle, which says that two electrons in the same system must not be in the same state. The state of an electron can only differ in two ways, spin and energy. There are only two spins, left and right. But there are many different possible energies. Since the

electronic fluid will arrange themselves into 10 to the 57th different energy levels. 10 to the 57th? That's a big number! Some of those electrons must have very high energies. The higher the energy of an electron, the faster that electron moves. Since there are so many electrons in a white dwarf star, some of those electrons must be moving very fast indeed. Curiosity creates a pressure and that pressure resists gravity. Isn't that weird? I mean it's not the heat that resists the crush of gravity. It's the requirement that all the electrons have to be in a different energy state. Cool. The pressure of electrons trapped in different energy states by the Pauli Exclusion Principle degeneracy. Indeed, the matter within a white dwarf star is said to be in a degenerate state. And that would be the end of our story, except for one small detail. Though the carbon in a white dwarf star is degenerate, it's still carbon. And carbon under pressure forms something that men life it will become an earth-sized diamond in the sky. So just what is a component? Well to answer that question I'm gonna have to tell you a story. I began my programming career in the late 60s using machines like PDP-8s. We an assembler. At first, we just told the assembler where the program was supposed to start. We'd put this origin statement into the source code. You can see it here, it's got a star in front of it. And that told the assembler where the program was supposed to start. The assembler would generate the absolute binary code, and then we'd load it into memory and run it. A highly logical, if somewhat inflexible approach. As long as programs were small, less than a thousand lines or so, that inflexibility didn't matter so much, it was manageable. But over time, as programs grew in size, it became a problem. Big programs usually include subroutine libraries. And that's the problem, because where are those subroutine libraries loaded? Let's say that we've got a program that starts at location 200. Let's say that we also have a subroutine library that begins at location 1200. That gives us about 1,000 locations for our program. What if your program is longer than that? You could always continue the program at address 2,000. subroutine library is longer than a thousand instructions. The more subroutine libraries you use, the more complex this issue becomes. I believe it could become rather annoying. Yeah, that would be an understatement. Of course there was a simple solution to this. We could keep all the subroutine libraries as source code and just append that source code to the source code of our programs. Remember, this was in the days of paper, tape, and punch cards. The computers were slow, compiling took a long time. It took a long time just to read in all those cards from the card reader. And besides, who wants to take a 5,000 card subroutine library and put it on the end of a 300 card program? The source code solution just wasn't very practical. code without resolving the addresses. So that's exactly what we did. We changed the assemblers that they would emit relative addresses instead of absolute addresses and then we changed the loaders so that you could tell them where to begin loading and they would add that value to all the relative addresses. This was great. binary decks and then we could stack those binary decks up on top of each other and just load them. Uh, well, almost. Well, tarnation, son! How in the name of horse apples are you gonna call a subroutine if you don't know where that subroutine's loaded? Yeah, that was the problem. When everything's

absolute, you can just call the subroutine But when everything's relative, you don't know where anything is. Don't those subroutines have names? Call them by their names. And that's exactly what we did. We changed the assemblers so that they would emit external references for any name they couldn't resolve, and external definitions for any name they had defined. that it would link together the external references to the external definitions. This worked great, except for one thing. The more subroutine libraries we created, the longer the linking process took. I worked on some systems where it took over 45 minutes to link all the libraries together, and that's just too long to wait when all you want to do is load and run a program. process of linking by creating a new program called the linker which resolved all the names adjusted all the relative addresses and then emitted a single quick to load executable mmm and that is how we came to create executable programs from independently relocatable modules linked together by leaders This set the stage for the explosion of libraries and languages that took place in the 70s. By the early 80s, we had moved from paper, tape, and punch cards to disk. We kept everything on disk. We kept our source files on disk. We kept our relocatable files on disk, our subroutine libraries, and all our executables. files that we would run through the linkers and link with subroutine libraries in order to create executable binaries that loaded very quickly. Compiling a source code module into a relocatable binary usually didn't take more than a minute or two. But linking hundreds of modules together along with dozens of subroutine libraries was often process requiring a half an hour or more. And so it was the executable files that we deployed. And then in the late 80s and early 90s we started using object-oriented languages. And with OO came the potential for frameworks. Applications call subroutines. Frameworks call applications. This inversion of calling order is important because to get it you have to use the dependency inversion principle. The flow of control from the framework to the application can't be in the same direction as the source code dependency which goes from the application to the framework. The source code of the framework must never depend on the application. And so there are two kinds of library. Framework libraries, from which control flows into the application, and subroutine libraries, into which control will flow from the application. And yet, the source code dependencies between the application and these two libraries remain the same. but the libraries know nothing at all about the application. And we still had to link them libraries together using them cods aren't slow as molars as linkers. By the late 90s, memory had become very plentiful and processor speeds had just gone through the roof. shrink dramatically. Indeed, linker execution times diminished until they were nearly as fast as load times. As a result, we started seeing new kinds of libraries. Shared libraries in UNIX, dynamically linked libraries, DLLs, in Windows. You didn't have to link with these libraries, at least not right away. would just link them for you. And that trend continues right up to the current day. Nowadays we hardly ever use a separate linking step to link our executables together. We just compile our source code into jar files or DLLs and then let the loader do all the linking and relocating for us. And this leads us to the definition of a component. A component is nothing more than an a gem, a jar file, or a shared library in

Unix. What does it mean to be independently deployable? It simply means that a change to one does not cause others to be recompiled or redeployed. For example, subroutine libraries are independently deployable, because I can change the application without recompiling or redeploying the subroutine libraries. is true of framework libraries. Changes to the application don't require that the framework component be recompiled or redeployed. The reason for the independent deployability is the one-way relationship. Applications depend on frameworks and subroutine libraries. Frameworks and subroutine libraries do not depend upon applications. And so the key to creating independently deployable is to manage dependencies. DLLs and jar files that depend willy-nilly all over the place cannot be independently deployed and are therefore not useful components. But why is independent deployability so important? Well, wouldn't it be nice if we could make a change to our system or one jar file instead of having to recompile and redeploy the whole system? Indeed, you should be able to hot-swap those components without bringing the system down. And wouldn't it be nice if the teams in your organization could work on their own components independently without interfering with each other? development. And that's what this Clean Code series on components is all about. It's about designing systems of components that enjoy the maximum amount of independence. What you have just heard was the opening statement by the counsel for the defense. You are the judge, jury, and executioner. You will hear the a little jaunt through the problem of making coffee. And so here is the design of the hardware of the Mark IV coffee maker. I will be presenting to you the individual elements, their purposes, and the way they work. horse hockey and just tell you the APRs that you need to go. We begin with the boiler. Water is entered into the boiler through some means we don't care about right now. Notice the heating element at the bottom. This heating element is under software control and it will be turned on when we wish to boil the water to make coffee. set boiler function with a true and when you need to turn the boiler off you're gonna call it with a false you got that son there is a sensor at the bottom of the boilers that detects the presence of water it will tell you whether the boiler is empty or not now what he means is that you can call this here get boiler water in that there boiler. If it returns false, then they ain't. Notice the siphon hose that goes from the bottom of the boiler up out and over the coffee filter holder. Then you turn on the boiler heating element. It boils the water in the boiler. Steam builds up inside the boiler and There ain't nothing you need to do about this, because once you turn on that concern boiler, well then that hot water is just gonna spray out all over those coffee grounds. All you got to do is wait for that to happen. There's nothing else for you to do. As the hot water pours out over the coffee grounds, it leaches the coffee compounds out of the coffee grounds, forming coffee. Then the coffee of the filter and collects in the pot. Boy I'll tell you that fella over there he just yep yep yep yep all the time he's just a windbag in it cuz there ain't nothing here for you to do. That coffee is gonna collect in that pot and all you gotta do is wait for it. Notice the warmer plate that the pot sits upon. There's a heating element in the plate that's under the software control. We to keep the coffee warm. Now I bet you're just dying to

know how you're gonna turn that warmer on aren't you? Well you do that by calling this here set warmer function with a true and when you want to turn the warmer off well you call that same there function with a false. There is a pressure transducer at the The software can read this transducer to determine the status of the coffee pot. Trans-ducer-schmans-fus-ser. It don't matter one donkey's snout dribble what you call it, son. All you gotta know is this. You call this here get plate function, it's gonna return to you a one or two or a three. A one means there ain't no pot on that plate, son. ain't got no coffee in it and a three that means there's a pot on their plate and there's coffee in at their pot. Now let us return to the boiler. There is a pressure relief valve up at the top of the boiler. It's under software control. When you open this valve it relieves the steam pressure from the boiler and sometimes these kind-signed executives get so doggone impatient for their coffee that they're just gonna grab that pot and pour themselves a cup no matter what's coming out of the spout so what you gotta do son is this you gotta call this set valve function with a true that's gonna open the valve and stop that coffee from pouring out all over the place making a mess bigger than button and this light. You can read the status of the button from the software and the software can turn the light on or off. This is how the user controls the coffee making process. Now what he means is that you don't start brewing coffee until this here get button function returns a true and then when that boiler goes empty well then you call this set light with a true that'll And then when an empty pot goes back on the plate, well then you call set light with a false to turn that light off. All right there young'uns, you understand? You capiche, you savvy? I mean this here ain't rocket science, so now you go design that software to control this here coffee maker. Now pause this video and take 30 minutes or so that will control the coffee maker through this API. Am I to understand that you've summoned me for the purpose of designing this trivial toy application? Why should I waste my time on such nonsense? This is a simple exercise for the benefit of our audience. It will not take much of your time and your cooperation would be Oh, very well, since you insist. The accepted technique for decomposing requirements into object-oriented designs is through noun-verb analysis. Indeed, I have heard of this. Yes, well, we enumerate the nouns in the requirements and these nouns become candidate objects for our design. maker, warmer plate, boiler, valve, heating element, sensor, button, and light. Logical. Clearly the coffee maker noun represents the central object in our design. It is here that the high level intelligence of the system resides. It is the coffee maker object that knows how to sensors and two kinds of heating elements. The obvious conclusion is that both have base classes. There must be a heating element base with derivatives, one for the boiler element and one for the plate element. And there's a sensor base class, one for the boiler sensor and one for the plate sensor. I believe I see where this is headed. Well, the rest is The boiler sensor, the boiler element, and the valve both belong to the boiler. The boiler, the warmer, the light, and the button all belong to the coffee maker. I see. As I said before, trivial. A problem unworthy of my prodigious talents. So now if that will be all, I shall take my leave of you, sir. Holder, please. moment if you'd be so kind perhaps you can

explain one or two points that I find puzzling oh very well if it will hasten the end of this dissipation what message will the coffee makers send to the boiler in order to open the valve clearly it will send the open valve message as I thought and so what message boiler send to the valve object? Are you being purposely obtuse, sir? The boiler will send the open message to the valve. Indeed, yes, so it would. And so how then will the valve object implement that message? You try my patience, sir, you and pass it a true indeed sir indeed but since the coffee maker already knows it wants to open the valve why doesn't it simply call the appropriate api function why insert all these extra layers like the boiler and the valve the question is absurd sir this is an object-oriented As astute an observation as I have seen. But I have another question. I note that the heating element base class has two derivatives, but no direct users. Correct, sir. That base class forms the abstract foundation for many different kinds of heating elements. Such abstraction is the very nature of object-oriented design. is clearly impressive. But is there any code inside that base class? Of course, sir. It contains the abstract methods turn on and turn off. And yet the boiler contains the boiler element, the warmer contains the warmer element, neither of them make polymorphic calls to the base class. So why is the base class there? The base class represents the commonality between the two different derivatives. Are you so ignorant of object-oriented design that you cannot see this? Apparently. But then there is the sensor base class. What methods does it contain? This line of questioning grows tedious beyond my ability to endure it, sir. It contains one abstract method. And what, precisely, does that function return? It returns the sensor status! Precisely what type does it return? An integer? An enum? A string? Don't trouble me with trivialities, sir. Such details are beneath me. I am the architect, not the programmer. It's abundantly clear that you are not a programmer. Quite. Will you grant me, sir, that the caller of GetSensorStatus must know the type of the sensor he is calling, and therefore the call cannot be polymorphic? Of course, sir. How could it be otherwise? Will you also grant me that the majority of dependencies in your design point at concrete classes? That is plainly obvious, sir. What is your point? Have you ever heard of the term... solid? Eh? Eh? Solid, you say? Whatever do you mean? It is simply a term I thought you might have recognized. Pay it no further mind. Allow me, however, to point out that if you removed every class from that diagram except for the coffee maker, the logic inside the coffee maker would remain substantially unchanged. A few names would change and that is all. Every decision, every bit of policy, every meaningful line of executable code remains in the coffee maker class. In short, your design is farcical. It is not a design at all. All you have done is to meaninglessly fill space with empty partitions in the name of object-oriented design. But I am the architect! Ugh, a rumpf! Let's try this design again. And this time let's apply the solid principles to the design. The first step in any application design is to apply the single responsibility principle. to properly partition the components, you need to determine who the actors are so you can separate the responsibilities. So who are the actors in this problem? Which groups of people will request changes to this system? It seems clear that one of the actors is the person who decides to make coffee in the first place.

Right, let's call this actor the brewer. This is the actor that's interested in the user interface and this actor will be requesting changes to the button and the light. I like my coffee now. I refuse to wait for the pot to fill up. Okay, we'll call this actor the now drinker. so he can remove the pot from the plate when the coffee's still brewing. And I like my coffee hot! Ha ha ha ha ha ha ha ha ha! Uh, sure. We're going to call this actor the hot drinker because this actor is interested in the temperature of the brewed coffee. Now son, you got yourself three actors. And you remember that they're single responsibility principle, don't you? So how many modules you think you're gonna need? Well, at least three, right? By the single responsibility principle, there ought to be at least one module for each actor. The module for the brewer actor is pretty easy to understand. It's the one that controls the button and the light. Uncle Bob, you have tripped the concretion alert. When designing the high level policy of a system, it is customary to avoid mentioning the low level details of that system. Ah, yes, good point, good point. at all about the low level Mark IV coffee maker implementation. We're going to avoid words like boiler, valve, heating element, button, light, and sensor. Instead, we'll describe the abstract purpose of these components. So as I was saying, the module responsible to the brewer actor is responsible for controlling all the communications with the user. module. The module responsible to the now drinker must allow that actor to get a coffee while coffee is still being brewed. We don't want a mess all over the place. We're not allowed to use the B and the V words here so we'll call that module the hot water source. An interesting choice of words It might as well be based on an infinite supply of water, a microwave heater, and a pump. Both are software-controllable sources of hot water. The module responsible to the hot drinker must collect the coffee and keep it hot. Now, we're not allowed to use either of the P words here, I have to say that containment vessel is an appropriate word. I like my coffee strong enough to kill a Denebian slime devil. Okay, so now we've got our three modules. the hot water source and the containment vessel. And now what we've got to do is figure out what the relationship between all these modules is. To do this we're going to go up and down the abstraction layers like a yo-yo. We're going to start with the yo-yo up because all of our modules are abstract. They don't know anything about the details of the Mark IV coffee maker. of those modules we're gonna throw the yo-yo down and we're gonna look at how the mark for coffee maker behaves we'll use the mark for as a test case for the behaviors of the abstract modules we start with the yo-yo up we're at the abstract level the brewer actor wants to start brewing coffee but now we throw And down in the Mark IV that means that we've got to somehow detect that a button has been pushed. But we're not allowed to know about buttons, so the yo-yo comes back up. And all we know is that the brewer actor sends the start message to the UI. What should the UI do with this message? Well the obvious answer is that it should tell the hot water source to start. No, you twit! have to determine whether or not we're already brewing. Second, you've got to know whether there's water in the boiler. Third, you've got to find out if there's an empty pot on the plate and only then can you tell the hot water source to start. Ah yes, good point, good point. The UI is gonna have to remember whether or

not we already told it to start brewing and if we're not already brewing then source and the containment vessel if they're ready and if they're both ready then it can tell the hot water source to start. This simple analysis has allowed us to see that the UI and the hot water source both have a start method. It's also allowed us to see that the hot water source and the containment vessel both have an are you ready method and finally it's clear that the UI depends So, what does the hot water source do when are you ready is called? Well, in the Mark IV case, it's going to make sure that the boiler's got water in it. And by the same token, the containment vessel in the Mark IV case is going to make sure that there's an empty pot on the plate. When start is called on the hot water source, the Mark IV implementation needs to close the valve heating element. And then the threat of control bubbles all the way back out to the brewer. What happens to the threat of control after that? Well, we don't know. Maybe there's some kind of operating system working behind the scenes. We'll deal with that later. The next event our Mark IV coffee maker has to deal with is when coffee starts to drip into the pot the sensor on the warmer plate is going to switch to pot not empty of course it's the containment vessel that detects this but then in the mark for the only action is to turn on the warmer plate and since that all happens down in the implementation well there's nothing for us to do at the abstract level the next event occurs when the now drinker removes the partially brewed detected by the containment vessel but in this instance other modules must be brought into play right but in the mark for case when the pots removed what we need to do is open the valve but we're not allowed to think about valves and pots at the abstract level so instead we'll have the containment vessel tell the hot water source to suspend the flow by the same token when the has returned to the plate, the valve must be closed again. At the abstract level, this means that the containment vessel will send the resume message to the hot water source. This nonsense of people taking the pot on and off the plate just because they can't be bothered to wait until the coffee is ready will continue until the boiler is empty. At We'll call that message done. The question is, who sends it? Clearly in the Mark IV case, it's the boiler that would detect this event. But the boiler is represented by the hot water source. But what if, in a different kind of coffee maker, it were the containment vessel that detected that it was full? vessel that would have to send the done message. Which module sends the message is irrelevant. What matters is that the message gets sent. This is a matter for the low-level implementation to work out and has nothing to do with the high-level policy. Once the boiler is empty, people are still going to be taking the pot while and we don't want the containment vessel to be sending send and resume messages to the hot water source during that time so if the hot water source sends the done message to the UI it should also send it to the containment vessel to let it know that brewing has completed at some point an empty pot will be returned to the plate and that's when the light should be turned that detects that it's got no more coffee to deliver so it'll send the finished message to the UI and that's the high-level design of our coffee maker there are three objects separated by responsibilities and they collaborate to brew coffee what you see here is the high-level policy and lights and heating elements and

valves and all that stuff. Pure policy. Now that we have the high-level policy, we need to implement the low-level details. This is where the open-close principle comes in. We begin with the start message sent from the brewer actor to the UI. Who really sends this message? Clearly, some function must call the getButton function and check for true. Right. So, let's say that there's a derivative of the UI. We'll call it m4UI. And let's give this derivative a poll method. Now, when the poll method is called, it will call the getButton function of the API. what? It's true. It should start the brewing process. Sure, sure, but it's the UI base class that starts the brewing process. Remember, we had the UI send those messages. Are you ready to the hot water source and to the containment vessel? So we need some function in the UI base class to do that. Let's call that function start. And we'll have the pull method of the M4 UI call the start message in the base class. Remember to make that method protected. Ah, good point, good point. The only class that calls that start method in the UI is the M4 UI derivative. So start should be protected. gonna deal with all those events there bucko you're gonna create yourself a derivative for each class and give each one of them a pole method well I don't see why not it seems perfectly reasonable to me each of the three base classes has its own derivative each of those derivatives has a pole method that pole method calls protected methods in the base classes when the events that So when the pot is removed from the plate, it'll be the M4 containment vessel that actually detects that by calling the getPlate function. And then it will call the protected method on the containment vessel base class named stopFlow. Boiler in the API and then it'll call the at capacity protected method of the hot water source base class. But master who would put the pole method on all these objects? Why Maine of course grasshopper. Maine will sit in a hard loop calling the pole method of all three objects. And that pretty much wraps which deal with the high-level policy of making coffee, each with its own responsibility. We've got the three derivatives that implement all the details of the Mark IV without making any policy decisions. And we've got Maine that sits in a hard loop, divvying out calls to bowl. Logical, flawlessly logical. An intriguing and apparently well-reasoned design. The evidence before the court is incontrovertible. There's no need for the jury to retire. You may... Here is the design of our coffee maker. Now, watch this neat trick. See that red line? Notice it contains all of the high-level policy classes of the coffee maker, and all dependencies that cross that red line cross, going inwards in accordance with the dependency inversion principle. The classes inside that red line make our first component. Ja, und wat a fein komponent it is! high-level policy and none of the low-level details. I think this component could be the heart of many, many different coffee machines. Consider for example a coffee maker that has an infinite supply of water heated with a small fusion reactor. The water is pumped over coffee grounds and the coffee is collected into a tank with a spigot at the bottom. coffee hot black yes the voice recognition unit would be the UI the fusion generator and the pump would be the hot water source and the tank and spigot would be the containment vessel this would work out quite well why hell boys I think this component could be used to make hot chocolate or chicken still right that's kind of the point the m4 derivatives are plugins to this

component this component is completely isolated from them and knows nothing about them better yet we could make all three of those derivatives their own component they're all independent and that would allow us to do something cool and couple that to an M6 containment vessel. I mean, think of the possibilities. Complete interchangeability. Yes, interchangeability. Independent deployability. The physical separation of high level policy from low level detail. These are the benefits of good component design. You know you want it now, you know you do. But what are the rules? How do you know what classes should go into a component? And how do you know how to manage the dependencies between those components? What are the principles of design that you should use? And that is the topic of this series on the component principles. three principles of component cohesion. These principles tell us which classes belong together in a component and which classes should be kept in separate components. We'll continue this series by discussing the three principles of component coupling. These principles tell us how to manage the dependencies between components and what direction those dependencies should take. And we'll a case study that'll show how to use all these principles together in concert. And so the evidence has been presented and the closing arguments have been made. And now you, the jury, must deliberate. You're not going to want to miss the next exciting episode of Clean Code, episode 16, Component Cohesion. Done. guitar solo Thank you. I'm going to go get some food. Let's do two more. Okay. That's good.