Thank you. We'll be right back. Thank you. I'm going to go get some food. I am Amanda. Hi, I'm Princess. Hi, I'm Alexis. Hi, I'm Lulu. Clean code Welcome welcome to episode 18 You didn't happen to bring a component for me to study, did you? Oh, wonderful. Anyway, do you remember the last episode, episode 17? In that episode, we discussed component coupling. Remember we began with the morning after syndrome? This is what happens to large teams. They start to step all over each other. principle. The acyclic dependencies principle says that components, when they depend upon each other, should not depend in a dependency cycle. After we talked about cycles, we started talking about stability. We defined stability as the impact that one component has upon another as it depends upon it, and we Components should depend upon each other in the direction of increasing stability. After that, we talked about the stable abstractions principle. You remember that. That was all about the components at the bottom of the dependency graph that were really stable and hard to change. The stable abstractions principle gives us an escape clause from all that stability and rigidity where they can conform to the open-closed principle. So, now in this episode, we're going to study how to partition a system into components. And the example we're going to use is the payroll case study from way back in episode 14. Remember that one? When we used all the solid principles to do the class design of payroll? Well, now we're going to break that into components. and we're going to figure out the best component partitioning for this problem. So I hope you're all packed and ready to go, because after we study some antimatter and a little bit of CP symmetry breaking, well then we're going to embark upon a journey of component case study! Ha! In 1898, a well-known British physicist by the name of Franz Arthur Friedrich Schuster published two whimsical letters in the journal Nature. They were meant as jokes. He talked about this mythical substance that he called antimatter. He discussed anti-atoms, anti-solar systems, and even an anti-universe. anti-matter met matter, the two would annihilate. He didn't have any evidence, or not even any kind of theory. He was just joking around. And I guess the moral of that story is, be careful what you joke about. In 1928, Paul Dirac was combining the equations of relativity with quantum mechanics The resultant equations allowed for the existence of an oppositely charged particle that was otherwise identical to an electron, that is, an anti-electron. And by cracky four years later in 1932, Carl Anderson detected one of them little buggers. He called it the positron for positive electron. Since those days we've come to realize that every particle of matter has a corresponding anti-particle which has the opposite charge and spin but is otherwise identical. We've even managed to create and store hundreds of neutral anti-hydrogen atoms for minutes at a time. We keep them in a magnetic bottle. anti-matter meets matter, well, it annihilates! Ha ha ha ha ha ha ha ha ha ha ha ha! Yeah, I mean, consider what happens when a hydrogen atom meets an anti-hydrogen atom. The hydrogen atom is surrounded by a negatively charged cloud of electrons. positrons. The two clouds attract each other and when they meet the electron and the positron annihilate each other in a furious burst of energy which leaves the two nuclei exposed. The nucleus of the hydrogen atom is a proton positively charged. The nucleus of the anti-hydrogen atom is an anti-proton negatively charged. The two attract each other and when they combine they larger burst of energy. Oh, so like, boom. Yeah, boom. Or rather, bang. Big bang. Because, you see, we believe that the universe began when equal amounts of matter and antimatter were created and then annihilated each other. Now, of course, that much annihilation would produce an immense amount of energy. And we see that energy. It's all over the sky. It's the cosmic microwave background. That's where all that energy came from. So we can see the results of that mammoth matter-antimatter annihilation in the sky. But that leaves us with an interesting puzzle. Indeed it does. For of all of that matter and antimatter annihilated, why are we here? Remember the experiment from the previous episode that allowed you to tell which side of the mirror you lived on? Recall that it was the spin of a beta decay neutrino that an experimenter on one side of the mirror would detect as spinning clockwise, but on the other side of the mirror they'd detect it as spinning counterclockwise? to live in an antimatter universe, then he would detect the beta decay neutrinos spinning clockwise. In other words, the two experimenters would give the same result. There would be no way to tell which side of the mirror either experimenter was on, or even if they were on different sides of the mirror. Indeed, given CP symmetry, an antimatter universe on the other side of the mirror would be absolutely identical to a matter universe on this side of the mirror, and no experiment could determine the difference. So here's the weird bit. In 1980, James Cronin and Val Fitch were goofing around with a particle called a kaon. because it decays into its own antiparticle, an anticaon. And anticaons decay into caons. So they kind of just bounce back and forth between the two states. What Cronin and Fitch realized is that the rate at which caons decay into anticaons is different from the rate at which anticaons decay into caons. the rates of k on decay can determine whether they are living in a matter world on this side of the mirror or an antimatter world on the other side of the mirror. And that means you can tell which side of the mirror you're on so that their CP symmetry don't work, do it! And so it was that when the matter and antimatter of the early universe underwent the vast convulsion of annihilation. A small asymmetry in the laws of physics allowed a tiny fraction of matter to persist. Us. Do you remember the design of the payroll system from back in episode 14? In this episode let's go get our bearings back and look at that design again. Oh yeah, we began with the use cases, didn't we? Yeah, that's right. We applied the single responsibility principle by mapping out the actors and operations for the add employee use case. that depicted the classes and data structures that implemented the operations of that use case. Yeah, but Uncle Bob, I'm a little confused because I thought I heard you say once that you really wouldn't draw those diagrams. Yeah, that's right. I mean, I drew them for you because in a video like this it makes it easy them anywhere near as formally. Oh I might go to a whiteboard with the team and scratch a few diagrams on the whiteboard to discuss it but I'm not going to do some big formal pile of diagrams that would be hideously wasteful. Yeah but like Uncle Bob like if you don't keep those diagrams like where's the design kept? Yeah, it's important that I don't mislead you here. You're going to see lots and lots of diagrams here, and you might get the idea that you're supposed to draw these diagrams well in advance of the code being written. Illogical. A flawed perspective. Right. So, while the diagrams I'm going to show you can be very useful in helping you partition a system into components, These kind of diagrams aren't drawn early. Typically, if they're drawn at all, they're drawn after a great deal of code has been written. After all, the component partitioning decisions we are about to make are real decisions, and therefore they had better be made on something just as real, code. I don't want to see you basing your component partitioning decisions on fluffy little diagrams better than castles in the air. Ten years ago in 2003 I wrote this book Agile Software Development Principles Patterns and Practices. The payroll case study that we're about to go through was fully documented and implemented in this book including all the component partitioning decisions. stuff until I had completely written and tested this application. It all worked before I started dividing it up into components. So the component level decisions I made in here were based on the real world, not castles in the air. The design I used in the book is not quite the same as the design you're going to see here. For example, in the book I did not use separate data structures the use case data in the use case objects which I called transactions. Nowadays I think externalizing that use case data into data transfer objects is a slightly cleaner solution. You may recall from episode 14 that we talked about quite a few use cases including add employee, change employee, sales receipt, add union service charge. We're going to assume that all these use cases follow the same design, which roughly means that the use case will accept as its input a request data structure and will produce an output in the form of a response data structure. Do you remember way back in episode 7 when we talked about architecture? I called these use case objects interactors. managed they managed entity objects business objects so here are the entities for the payroll design these are the objects that are controlled by the use case interactors these are the objects that contain the application independent business rules these are the business objects right there four strategy objects. The first strategy object is the pay disposition which determines where the employees pay will be sent. The second one is the pay type and that determines how much the employee will be paid. The third is the pay schedule which determines when the employee will be paid and the last is the union membership which determines how many union dues to deduct The derivatives of these strategy classes show us the rather large amount of degrees of freedom that these calculations have. I mean, paychecks can be delivered by mail, they can be held by the paymaster, they can be direct deposited, pay could be hourly, salaried or commissioned, paychecks could be delivered weekly, monthly or bi-weekly, and employees could be union members or not. Finally, we see that the hourly employees maintain a list of time cards, the commissioned employees maintain a list of sales receipts, and the union members maintain a list of service charges. Whoa man, I mean, can you stop and rewind and play that all back again? Because I'm kind of lost. Keep up, mister. UI of some kind and we'll probably assume it's the web. So that means we're going to have a whole bunch of controllers that activate the use case interactors by creating request data structures. And then the use case interactors will drive the views by creating response data structures. Now back in episode 14, we isolated the use case interactors and the entities from the controllers by creating builders, factories and presenters. We're going to have a lot of these builders, factories and presenters to manage here and we're going to hide them behind interfaces. We're also going to need a database and if you remember our architecture lesson from episode 7, you know that we need to put that database behind another boundary and access it through another there's our design controllers use builders to create request data structures that they pass to interactors that they acquired through factories the interactors manipulate entities and the database in order to create response data structures the response data structures are passed into presenters which create view model data structures which are then driven we going to partition this into components I know I know let's put the controllers and the views in the web component and the presenters the factories and builders in the UI component and the use cases and the interactors and their data structures in the use cases component and all the entities component. Yeah, yeah, yeah, that's right. Bogus, bogus, like there's a cycle between the web and the UI, man, and it's like way uncool. Gosh, I hate to pile on like this, but after all, all those entities, they're really concrete, aren't they? And all your use cases are depending right them? I mean, isn't that a violation of the stable abstractions principle? Oh, and for goodness sake, those use cases are volatile. Customers like to change their minds. And one of the things they like to change most frequently are use case implementations. And yet you've got the web and UI depending on these things that are likely to change for all kinds of different reasons. And that violates the common reuse principle. Suffice it to say that there's a number of problems with this partitioning. There's more, but we don't need to go into them now because I think the point has been made. So, does anybody else want to give this a try? How would you partition this system into components? like a really big believer in model view controller man because it's like the way coolest application framework for the web like ever so i'd probably split it up like this you know i'd have the controllers depending on the models because that's where like all the business rules are and then the process output thing. And you know, like, if you do that, then there's no cycles at all. Do you mean to tell me that you would have every single controller depending on every single use case? And then every use case depending on every single view? Can you see the transitive dependency of the controllers upon the controller depending on every single view this violates so many principles I can't even say them all in a single breath yeah every time you change the view you gotta rebuild everything that really sucks well I'm sorry to say but I have to agree because those views are really concrete aren't they and and like really depends upon them a lot don't they so isn't that a violation of the stable abstractions principle again yeah it pretty much looks like all you did was draw some lines on there to give us a rough model view controller split without any cycles it doesn't appear that you tried to make these components independently deployable remember the whole reason for having a component independent developability. So who's next? Who else wants to try and make an independently deployable component architecture? Oh, I'll go, I'll go. I had some ideas. First of all, there was an awful lot of red in there and so I thought a little bit of color would lighten the mood and I was disturbed by all those straight lines so I added some curls. Don't I put my logo up there in the corner Danny dotnet Excuse me Hmm, oh that'll look really nice on the wall of my wig one Alright, so let's start this over and this time let's focus on the principles. We'll start with the principles of cohesion. Do you remember this tension diagram? We're going to presume that our project is in the midst of active development and the bottom of that diagram, as far away from the Release-Reuse-Equivalence principle as possible, and as close to the Common Reuse and Common Closure principles as possible. We'll begin with the Common Closure principle. Which ones of these classes change for similar reasons, and which ones are closed against similar kinds of changes? with use cases. Each use case can be called by many different controllers. Therefore there is a locus of change around each controller and the factories, builders, and use case activators that they call. Okay, sure. But wouldn't the interactors have to change as well? I mean, if the controllers change, the use cases probably have to change too. might change without affecting its controller. Therefore the interactor is probably not part of the controller's locus of change since they are not closed together against the same kinds of changes. Oh now hold on! You're not saying that all the controllers, builders, factories and use case activators belong in the same component are you? Because if you are... at all a great big time polar like that would be worse than a barn stacked floor to ceiling with constipated cows no I reckon what we're gonna do is break them use cases up into families and then we'll divide all those factories and builders and controllers and use case activators amongst those use case families right putting all the controllers into the same component would violate common reuse principle because it would create a component that has way too much knowledge. Yeah, and the same logic applies to the view, the view model, and the presenter because these are all tightly cohesive around the presentation details of the view and they can all be split based on the are highly cohesive since the one builds the other. They're going to change for the same reasons, and so they belong together in the same component, although I'd maintain the use case family split. By the same token, the interactors and the use case responses are highly cohesive, and for the same reason, the one builds the other. I'd include in with them the factory impulse maintain the family boundaries. Entities have no external dependencies and so they should be grouped alone. They also fall into families unrelated to use case families. The entity families are liable to be complex because the structures of the entities are complex. The database gateways are are used by the interactors but are not cohesive with either the interactors or the entities. Therefore the database gateways belong in their own component but should be partitioned along the entity family lines. Yeah and them there database imples they're gonna be in their own component too and they's kin to the database gateway so they'll be split along the same So, now let's look at what we've got. The views all depend upon the interactors, and that's good because interactors are high level policy while views are low level detail. Interactors depend upon entities. Again, entities being application independent are higher level than interactors, which are application dependent. in the system. They don't depend on anything. Interactors and database implementations depend on them. And those interactors, why they're depending on the controllers and that's just peachy because those controllers are about as low level as the trough I slap my pigs in. Oh but Uncle Bob, Uncle Bob there's a request builders they depend upon the controllers that's a cycle uncle Bob a cycle oh heavens yeah a cycle I thought that was bad all right now don't get all hot and bothered I know this cycles bad but look we're not done yet all we've done so far the cohesion principles we still have to So let's tackle that cycle first. The problem is that we've mixed interfaces with concrete classes. Those controllers are concrete but everything else is abstract and if you think about it that violates the stable abstractions principle. We've actually got several problems others but this one is particularly bad because it leads to a cycle. I recommend resolving the cycle by splitting the controllers component along the abstract concrete boundary. That's a good idea. That gives us another component named use case boundary and of course it'll fit into still there. Like the request builders are depending on the boundaries and like vice versa. Who knew that cycles were so hard to get rid of? Well they can be, it's true, but this one is not too hard to deal with. All we have to do is move the use case request DTO into the boundary component like so. cool so now let's consider the stable dependencies principle are there any components here that ought to be easy to change that are dependent upon by things that are hard to change views should be easy to change and nothing depends upon them so there's no violation there and them controllers I mean they're changing all the time anyway. Every time somebody's grandmother discovers rest or soap or some other concerned thing, those controllers are going to be changing. But then there's nobody dependent upon them so I think that's just fine. The interactors will have to be changed frequently because they implement application specific business rules and those kinds of business rules are usually quite volatile. and therefore depend on something volatile. That's likely a problem. Yeah, it probably is. Strictly speaking, the views are less stable in the interactors, but only by a bit, so I think we should break that dependency. We can do that by adding the use case responders and the use case response DTOs to the use case boundaries component. Oh, would you please clean up that diagram? The shapes are all blurring together and making me quite dizzy. Indeed, I'm feeling rather ill myself. Okay, okay, okay. I get your point. I'll clean it up. Shazam! Okay, so now I've got it all reorganized into swim lanes. cross the red lines between the swim lanes going in the same direction so there can't be any cycles but let's look at the stability it's very clear that the instable controllers views and interactors depend upon the stable use case boundaries and since the use case boundaries are abstract well Okay, Uncle Bob. But check out that request builder swim lane. Do all those builder inputs really have to be alone like that? I mean, I'm kind of feeling sorry for him over there. Yeah, I was looking at that too. Those swim lanes make it pretty clear that the builder implementations could really be moved in with the interactors. So, shazam! There we go. The builder implementations are in with the interactors, and this makes sense because the interactors really depend very heavily on the format of those user request DTOs that are built by the builders. Yeah, yeah that looks better Uncle Bob, but I'm still pretty concerned about all those entities because they still look pretty concrete to me and there's a lot of incoming Stable abstractions, Principal Uncle Bob? Well, yeah, I think you're right about that. I mean, take a look at those entities. Those entities are actually instable for a whole bunch of other reasons. I mean, who knows what kinds of fields will be placed into them later or what new application independent business rules will be added. So, shazam! There. nice abstract interface. But I want you to take a careful look at those entity objects because they're not interfaces, are they? They're abstract classes. And they're abstract classes because some of the methods inside them implement the application-independent business rules, while other methods are abstract, the ones that are the data access methods. Those data access derivative and that entity implementation is actually created by the database implementation and so they might as well go together into the database implementation component cool like really like you know wow but you know i gotta say that those data transfer objects in the use case boundaries components they kind of bother me because like aren't Yeah, they really are. That's a good point. I suppose we could move them in with the interactors. Shazam! Ah, but that creates a cycle between the interactors and the use case boundaries. We don't want that. So, m'bash! I think we need to undo it. Oh yeah, like, bummer, We actually faced this issue back in episode 14 when we were discussing the request builders and we solved it by having the builder return a degenerate interface, an interface that has no declared methods. So, Shazam! There, that actually works pretty well doesn't it? Now the Okay, okay, that's cool, that's cool, but look man, there's still one of these data transfer objects left. They use case responses and it's all green and DTOE, like man, what are we going to do about that one? Well, Shazam! There, we'll just use an interface, a nice little interface that provides all the accessor the DTO so this is looking pretty good I mean there's no cycles we've got a bunch of concrete components that depend upon abstract components we've got a bunch of unstable components that depend upon stable components I mean from a high level this is looking pretty good Okay, Uncle Bob, this is great. But you know what I don't get? I don't get what all these families are. I mean, what are all these families, Uncle Bob? Ah, yes, the families. Okay, well, let's start with the entities. Here's that diagram we drew of the entities a little bit earlier. Now let's see if we can partition those classes into a set of components. Here is a very reasonable partitioning. The employee component is primarily abstract and very stable and all dependencies come inwards towards it. The disposition, membership, pay type, and schedule components are all concrete and they all depend on the employee component. Yeah, that's true, Elberdo, buddy. But you forgot one thing. You forgot about cohesion, son. there that disposition component that's full of classes those classes ain't used together they ain't closed together they ain't nothing together and if you were to use one of them like the male class why you'd have the other two tagging along behind you like coyotes behind a chuck wagon that's too much knowledge albert too much knowledge it violates that common reuse principle all the other components. So, SHAZAM! We're gonna have to separate them. Now, I know this looks like every class is in its own component, and frankly a lot of them are, but I want you to take special note of the hourly, commissioned and member components, because they contain subordinate classes. As time goes by and more features are added to this system, we might acquire more such subordinate classes. Anyway, these are the component families for the entities. There's 12 of them and that's a lot but well this system's got a lot of little options and each one of those options probably needs its own independently deployable component. So now let's from episode 14. We can partition these use cases into families by applying the same principles of cohesion and coupling. Eh, Shazam! And now you can see a potential partitioning. We see the CRUD use cases in one family, the ADD use cases in another family, the PAY use case stands alone. I think this is a pretty good split. go again Albert you done forgot all about cohesion you look at those ad use cases you think they're all related because they've got ad in their name son are you related to everybody named Albert course you ain't course you ain't those use cases ain't got nothing to do with each other Albert they're all just piled together into that one component and it's too much knowledge Albert Yeah, he's right again, so, Shazam! There we go, all the ads are in their own use cases and now we've got five use case families. So let's pick one of them and let's draw it out, let's draw all the components out for the Add Time Card use case family. which creates the AddTimeCard request and then passes it to the AddTimeCard Interactor. The AddTimeCard Interactor invokes the employee gateway to access the employee and time card and hourly entities. And then it builds the AddTimeCard response. by the Ag Timecard Presenter. The Ag Timecard Presenter builds the Ag Timecard View model and then passes it to the Ag Timecard View, and that's the whole story. Now, Albert, I think the wheels in your head are a little bit squeaky today, ain't they? I mean, that was all fine and dandy and everything, but you kind of glossed over all those factories and builders, didn't you? factories and builders you think we got? You remember back in episode 14 how many there were? There was one of each, Albert, just one of each. We didn't have a whole bunch of them, so there's something wrong with that diagram of yours, ain't there? Right again. And yeah, we did discuss this back in episode 14, and one of the ways we discussed There, now we've taken the boundary component and we've split it into a requester and a responder. The requester is not broken down by use case family but the responder is. Inside the requester we see the factory and builder. The factory has a make method that takes an argument that tells us which use case interactor to build and the builder takes an argument that specifies which request to build. Well now son you went and forgot about something else! Look at them factory and builder interfaces you only got one of them now but you got loads of implementations split by use case family that don't make no sense son you're gonna have to figure out some way to have one implementation of each that probably means you're gonna need some kind of registration scheme don't it? Shazam! Alright, so now we've taken the factories and builders and we've made them concrete, we've gotten rid of all the implementations from the interactors components, we've denoted that the factories and builders have to accept registrations, and I imagine those registrations will come from Maine. Wow, lots of adjustments and changes, kind of like when I moved in Is it always like that, Uncle Bob? Yeah, it pretty much is. You know, we could keep on tweaking this for a while because there's no such thing as a perfect model. But I think we've done enough. This is pretty good the way it is. There is one thing that disturbs me a little bit, and that is that we wound up with a lot more components than I thought we were going to. Really? How many? So, look at this diagram. all the components without any of the classes. There's two singular components, main and requester, and the rest are all split amongst the families. There's four components that are split amongst the five different use case families, and there's three other components that are split amongst the twelve entity families, and that of course makes a grand total of 58. Whoa, that's Yeah, it is, and that's a bit disturbing. We could decrease the number by merging all the controllers and views together, and that would knock the number down to 49. We could reduce it by a hell of a lot more by just merging all the entities together. But I'm reluctant to do that, because when you merge components together like that, you independent deployability. I mean, look at the freedom that this current plan gives us. If we want to deploy a system with just the CRUD use cases, we can. If we want to deploy a system with just the salaried employees, we can. These components are like a set of tinker toys. code and that's power. Alright, now see here, son. That's just too many components. I'm telling you, when the pigsty gets full, it's time to get in there and slaughter yourself some pigs! Okay, okay, you're right. As this project matures and we approach deployment, we should probably reduce the component count by popping some of those component families. But and I'd like to preserve the tinker toy nature of this project just a little bit longer. All right, you've made your point. I see the value in all of these independently deployable components. But what about those metrics from the last episode? I want to see the numbers. Show me the numbers. Why? I mean, look at the diagram. of 0 or 1, there's nothing in between, and most of them have A-metrics between 0 and 1 with nothing in between. So what we've got here is a bunch of adults and teenagers, there's really nothing to evaluate here. Well, what about the requester? That's not a perfectly abstract component, no is it? Okay, okay, you got me. This one is a little bit strange. A metric of 0.5 and that means its D metric is 0.5 and that makes it something of an oddball. And I suppose we could fix this by putting an interface implementation split in there and pulling out a new component. But I'm not sure why I'd do that. I think I'd just rather live with the oddball nature of this component. So, anyway, that's the component case study. Fun, huh? So what did you think? We're not worthy! We're not worthy! In any case, in this episode we saw how to apply the principles of component cohesion system into an independently deployable component structure yeah and we also learned that keeping this cohesion and coupling thing in balance that's a might tricky because you get the coupling just right and you've blown the cohesion all the partition but then you get the cohesion just right and suddenly you got cycles so it's a trick keeping it all just right ain't it are significant. The ability to compose systems from subsets of independently deployable components is a power that is not likely to be discarded. Okay, sure. But do you really do all this up front planning like that? I mean, it looks like a lot of work. No, not really. Like I said before, classes around between components as the code starts to come together moving classes between components is one of the goals of a good refactoring discipline and with that I will bid you adieu because well the dogs are ready for their walk and besides we're done with the case study and we're also done with the whole series on solid component principles so what should we talk about a load of topics in the backlog. Remember there's that advanced test-driven development thing I've been promising you, there's acceptance testing, there's a whole load of design patterns to talk about, there's professionalism, there's functional programming, but I have been promising you that advanced test-driven development episode, so you're not gonna wanna miss the next exciting episode of Clean Code, test-driven development. Come on you dogs, let's get going. Hey dogs, hey. Let's go for a walk. It's a heck of an interesting day out here. We'll be right back. We'll be right back. We'll be right back. Microsoft would make a box that no one knows how to open. You're not allowed to open it if you don't have a license. What is the license code? Oh They have to get fancy The can, the can, the can, the can, the can, the can. Split by boogal boogal boogal boogal. It's that time again. Seven years, you know. Come here. Enough. Enough. Con los terroristas. Con los terroristas