Hi, I'm Uncle Bob and this is Clean Code. Welcome to Clean Code, Episode 7. Architecture, use cases, and high-level design. Come on in. we learned about test-driven development. We learned that code rots because we're afraid to clean it, and that the only thing that can cure that fear is a comprehensive suite of tests. We learned that only a suite of tests that you trust with your life can eliminate that fear, so you should treat that suite of tests like it was a parachute. We learned that the discipline of test-driven development the only sure way to generate such a trustworthy suite of tests. We learned the three laws of test-driven development. One, you can't write any production code until you first write a failing unit test. Two, you can't write more of a unit test than is sufficient to fail and not compiling is failing. Three, you can't write more production code than is sufficient to pass the currently able to reduce your debugging time, generate reliable low-level documentation, decouple your design, but most of all, you'll eliminate the fear of change, and that means you can clean your code. We answered some of the common objections to test-driven development, and then we talked about how test-driven development was similar to the common accounting practice of double-entry bookkeeping. enough about their code to do double entry bookkeeping, shouldn't we? We're not done with test-driven development, not by a long shot. We've got to talk about statism, mockism, acceptance testing, a whole bunch of stuff. We've got plenty of episodes coming up on that. But now we're going to take a look at another important topic, architecture. What the devil is it? Who does it? And what's it for? I am the architect. Architecture is the discipline of laying the foundation of an entire software system. It is composed of the highest level decisions. Decisions that must be made first. Decisions that must survive the lifetime of the software. that we're going to be addressing in this episode. Architecture is not just the foundation, architecture is the whole enchilada. Architecture includes the grand shape of the system and the teeniest low-level design. It includes the most abstract of the module interfaces and the most is that programmers are all architects in one form or another, and that architects who don't code aren't really architects at all. There's something else. So, after the science lesson, we'll begin by learning what architecture is and what architecture does. We'll discover that architecture is the shape that the system takes to remain flexible and maintainable. We're going to learn about Ivar Jakobson's idea of use cases. We'll learn what use cases are and what they aren't, how they relate to user stories, and why they're so important to driving the shape of our applications. We'll learn about Model-View-Controller and how it makes a good low-level user interface partitioning good high level application architecture. It doesn't split the system in the right ways. We'll also learn about Jacobson's partitioning, the boundary control and entity partitioning, that splits the application architecture much more nicely and can be used with Model-View Controller to make a very good system structure. We'll review how we shape the system by isolating the core from the periphery and crossing boundaries appropriately. Finally, we'll study the horrid disease of framework binding and we'll find ways to treat and even cure this frightful scourge. So now I want you to bend over, grab your ankles, and kiss your old architecture goodbye because we're about to slam into

the world of architecture use cases and high-level Relativity. When you think of relativity, you think of this guy, don't you? Albert Einstein. He's the guy who developed the theory of relativity, right? Actually not. The story of relativity begins with someone much older than Einstein. begins with Galileo. You see, Galileo studied the way things moved. He slid blocks down inclined planes. He threw things and dropped things and rolled things, and he measured how they moved, and he observed how they responded to forces. it caused him to notice something, something puzzling, something profound. He described it this way. Imagine you're in a ship at sea. You're inside a cabin, but you can't see out of this cabin, it's completely enclosed. Inside the cabin there are butterflies, birds flying around, and a bottle is suspended from Imagine also that even though the wind is blowing and the ship is under sail, that the sea is perfectly flat. The ship is not rocking or rolling in any way. It's just moving forward at a constant velocity. There is no observation you can make, neither of the butterflies, nor of the birds, nor of the drops of water dripping from the bottle, that will tell you how fast that ship is sailing. cannot measure your velocity. The laws of physics are independent of velocity. The only velocity you can ever measure is velocity relative to something else. The very concept of an absolute velocity, including the absolute velocity of zero, is meaningless. This eventually became known as the principle of relativity. And It's a daily demonstration. The Earth spins at a thousand miles an hour at the equator, but the people who live on the equator don't seem to notice. The Earth flies around the Sun at 67,000 miles an hour, but on a lazy summer day while we're lying on the beach, that velocity doesn't impress us much. The Sun races around the center of our galaxy at 1.3 million miles per hour while we sit on our couch and watch Star Trek reruns. The principle of relativity was described by Galileo in 1632. It tells us that the laws of physics are identical, regardless of your velocity. There's no experiment you can perform, no physical set of things that you can do absolute velocity. The only kind of velocity you can know or have is velocity that is relative to some external frame of reference. Then, in the 1800s, people like Gauss and Faraday and Ampere were studying electrical and magnetic phenomenon, and they realized something very strange. They of electrodynamic charge was a velocity. Imagine their surprise, their excitement, their concern. Here was the simple ratio of two physically measurable quantities and it produced an absolute velocity, a velocity without an external frame of reference. Galileo would have thrown them overboard. In 1856, Wilhelm Edvard Weber and his associate, Rudolf Karlsruhe, performed a set of delicate and careful experiments in order to measure that ratio. What they found was startling. That ratio, that velocity that had no frame of reference, was three times ten to the eighth meters per second, the speed of light. named it C. What a dilemma! How could a physical experiment produce an absolute velocity? Something really had to be wrong. Five years later, in 1861, James Clerk Maxwell, possibly the greatest theoretician the world has ever known, produced four simple equations. Those equations showed that electromagnetism in waves. What's more, they showed that the speed of those waves was C. This gave physicists the answer they were looking for. After all, if electromagnetism moved in waves, something's got to be waving.

The velocity, C, didn't have to be absolute after all. It could be relative to the waving medium. On top of this, physicists already that light traveled at sea and that light was affected by electromagnetism. So what else could light be but propagating electromagnetic waves? For this reason, physicists named the waving substance the Luminiferous Ether. This strange substance, massless, frictionless, must permeate all of space. that spins on its axis and swings around the sun in its orbit and races around the center of the galaxy must be continuously plowing through this eerie ether. And that motion through the ether should be detectable. Many scientists tried to detect that motion. They built careful and exquisite apparatus. They conducted many, many experiments, but they all failed. Then, in 1887, Albert Michelson and Edward Morley conducted the definitive experiment. They built the most sensitive apparatus. They conducted the most careful and delicate of experiments. But it was to no avail. No one could detect a relative motion through the ether. It's hard for us today to realize how confounding this was. to scientists at the time. Either they had to find this mysterious substance that was the frame of reference for C, or they had to completely abandon Galileo's principle of relativity. It must have seemed to them as though Galileo's ship was sinking in Maxwell's waves. Then, in 1905, Albert Einstein published a small and simple paper entitled of moving bodies in which he showed that Galileo's ship could safely sail on Maxwell's waves if the speed of light remained constant to all frames of reference. But that's a topic for another time. As I was explaining earlier, the architecture of a system is the set of irrevocable early decisions that lays the foundation for the entire system and its development. For example, allow me to describe a system that I recently architected. It was a vast enterprise system, and I chose after much deep thought and consideration. as the development language. I chose Eclipse with many of the RAG plugins as the development environment. Then for frameworks I chose Spring, Tomcat and Hybrid. I also chose the MySQL database Those aren't architectural decisions. Those aren't even close to being architectural decisions. I mean, that's like saying that the architecture of a house is hammers, nails, saws, and wood. Architecture is not about tools and building materials. Architecture is about usage. Imagine the architecture of a library. and you see a grand entrance, a reception area suitable for a check-in, check-out clerk, some little areas off to the side where people can sit and read or have small discussions, maybe some conference rooms, hither and yon, and then chamber after chamber and gallery after gallery capable of holding bookshelves. scream library at you. Or imagine the architecture of a church. There'd be an obvious entrance leading to a gathering area. That gathering area would be fairly large. It would be surrounded by some offices, maybe some little conference rooms, things like that. But there would also be one or more grand entrances into a large chamber. That chamber would be suitable for holding hundreds of people was all focused at the front where the altar would be. The architecture of the church would scream church at you. Indeed, no matter what kind of building you looked at, the architecture of that building, if it was well designed, would scream its intent at you. That's what a good architecture is. A good architecture, whether it's the architecture of a building is all about how the system is being used. A good

architecture screams use cases. When you look at a software system and all you can see is the model view control or structure of a web system, then the architecture of that system is hiding the use cases and exposing the delivery mechanism. We don't want to see the delivery mechanism. We want to see the use cases. More importantly, we don't want the use cases coupled to the delivery mechanism. Indeed, we want the separation between the UI and the use cases to be very strong. So strong, in fact, that they can be deployed independently of each other. We don't want the use cases to know anything at all about the delivery mechanism. the database, the frameworks and tools, the service layers. We want all those decisions to be completely independent of the use cases. The use cases should stand alone. Decisions about databases and UI and service layers can and should be deferred. of a good architecture. A good architecture allows you to postpone decisions about frameworks and web servers and UI and all of that stuff. A good architect knows how to keep options open for as long as possible. A good architecture maximizes the number of decisions not made. utter rubbish rubbish rubbish you say let me tell you the story of fitness as you probably know fitness is a wiki and a wiki is a database of pages in 2002 when micah and i first started the fitness project we chose the mysql database we'd need to get the database running and develop the schema before we did anything else. But then it occurred to us that this wasn't immediately necessary. We focused on all that code that would translate the wiki text into HTML. We realized that all that parsing code and all that translation code could be written without using the database. translation system working. He had implemented it around a central abstraction called wiki page. This wiki page abstraction held on to the wiki text and it had a to HTML method that would translate the wiki text into HTML. The wiki page abstraction also had some abstract methods for reading and writing the database, but we overrode them in a mock wiki page to do nothing. Then it came time for the wiki to deal with more than one page. Surely that would be a good opportunity to fire up the database and develop a schema to deal with those pages. But we quickly realized that we could avoid the database, at least for the time being, in a hash table in memory. We called this new derivative in memory page. We didn't know it back then, but this kind of derivative is called a test double. A test double is an object that stands in for another for the purpose of facilitating test driven development. So by using that in memory page test double, we were able to get the whole system working. feature after feature after feature. We could create pages, link other pages to them, get all that wiki text working nicely, we could create tests, we could create whole suites of tests, and we could run them. The only thing we couldn't do is save any of the pages to disk. Eventually, we just ran out of things to implement that didn't require persistence of some kind. needed some kind of database persistence. So finally we decided we just had to fire up the database, invent a schema, and figure out how to read and write wiki pages out of the database tables. But Michael Feathers was there at the time, and he reminded us that we didn't really have to do that. He suggested that instead we write a new test double, of a database without the complexity. This test double would allow us to continue writing tests that depended upon persistence without the need to

implement the database. He suggested that the new test double should write the wiki pages to flat files. He even wrote the test double for us. We could save pages, we could write and run tests, we could create whole suites of acceptance tests and run them over and over and over again. And with every new test came new features that made fitness even more and more useful. In the end we were even using it in courses and in demonstrations. And then, one day, it occurred to us. The first release of Fitness was done. We didn't need the database. That flat file system was good enough. We had deferred our first major architectural decision right off the end of the universe. put in place for the purpose of testing turned out to be good enough. I could stop the story right here and just make the point that even a decision as critically important as the database and the schema can be deferred for a very, very long time, perhaps even beyond the project lifetime. But I'm not going to stop the story because there's more to tell. One of our customers came to us and said they really needed a MySQL implementation because they had a corporate policy that all software artifacts had to be stored in a real database. Yeah, that's a silly policy, but you can't fight City Hall. So we showed the customer how we had put together the test doubles for mock wiki page and in-memory page system page and we suggested that he tried to develop the my sequel page he came back the next day had the whole thing working he'd made this class called my sequel page we used to ship that class with fitness just as a plug-in in case anybody wanted to use it nobody did fascinating indeed the lesson here Many architects consider the database to be the core abstraction, the central organizing principle. They find it unthinkable that you'd do anything at all until the database was working and the schema was in place. Yet, in the fitness project, the central organizing principle was the wiki page and all the business objects that surrounded it. defer and shim in at the last minute. Good architectures are not composed of tools and frameworks. Good architectures allow you to defer the decisions about tools and frameworks, like the UI, like the web server, and even like the dependency injection framework. How do you defer those decisions? decouples you from them and makes them irrelevant. How do you decouple from tools, frameworks and databases? You focus your architecture on the use cases, not on the software environment. I mean all the business value is really in the way the system is delivered and not what the system does. You know, a simple CRUD system, create, read, update, delete, that kind of system. And the real business value is in how that CRUD is delivered. Well, that's fair enough. Such systems are pretty common. The question is, does that change anything? about its use cases or should it be about the UI? That's the wrong question to ask. The real question is, since the UI is so critical, should it be coupled to the rest of the application? And the answer to that is clearly not. If the UI is critical, we don't want its architecture polluted by the concerns of the rest of the application. Another way to look at this is the separation Consider for a moment just the use cases and the UI. Let's say that we've separated these two into independently deployable and independently developable components, such that the UI is a plug-in to the use cases. Let's also say that the estimate for developing the use cases is $50,000, Now the business can compare those two values and ask itself a very interesting question. Why is

the UI so much more expensive than the use cases? And should that be the case? Maybe the UI shouldn't be more expensive than the use cases. Maybe there is a less expensive way to do the UI. Now it's possible that the business would look at those numbers and decide that the that they don't mind at all that it costs three times what the use cases cost. On the other hand, they might look in horror at those numbers and quickly backtrack to a more appropriately sized UI. The point is that without that separation, there's no way for the business to tell whether the cost of the UI or the cost of the use cases have the appropriate ratio. allows the business to measure the cost of each and then compare that cost to the corresponding business value. And this isn't just limited to the use cases in the UI. All the system components can be isolated in this way. The database, the web server, the various frameworks. And once they're isolated, you can subject them to the same kind of cost versus value analysis. So, by focusing the architecture of our application upon its use cases, we can defer decisions about the UI, the database, or other system components. This deferral allows us to keep our options open for as long as possible, and that means we'll be able to change our minds if we need to, perhaps many times during the course of cost. It also creates a strong separation between the system components which allows the business to compare their cost to their business value thereby supporting prudent decision-making. I think we've established that use cases are pretty important so that leaves us with a question what the devil are they? If I showed you the software architecture of an accounting system delivered over the web, what's the first thing you'd notice about that software? Would you notice that it was an accounting system or would you notice that it was a web system? Clearly the fact that this So when you look at the architecture of this system, it should scream accounting at you. And it should barely mention web. But most web-based systems do exactly the opposite. They scream web at you and barely mention their true business intent. Consider the model view controller architecture that's so prevalent amongst web systems nowadays. strongly web related. The models are completely subservient to the controllers. And the views in the controllers are so tightly coupled to the models that they have a tremendous influence over the structure of those models. To make matters worse, many developers mistakenly believe that the models are the objects that represent business rules. the central organizing principle, and relegate the business rules to annoying details. Ladies and gentlemen, it is the web that is the detail. The web is nothing more than a delivery mechanism. When you look at the architecture of an accounting system, you should see accounting In fact, you should be able to completely change the delivery mechanism without changing the architecture of the system. Given two versions of the same system, one delivered on the web and one delivered over a console app, the architectures of those two systems should be identical. In other words, when looking at the architecture of a system, you should not be able to tell if it is web delivered or not. the high-level architecture of the system. The web delivery mechanism is a detail. In 1992, just a year after Tim Berners-Lee put up the first ever website, a friend of mine wrote this book. His name was Ivar Jakobson and in this book he resolved the delivery mechanism problem understand the ways that users interact with the system in a delivery

independent way. In other words, we describe how a user interacts with the system without using web-related words like link, button, click, or page. Instead, we use words and concepts that don't imply a delivery interaction descriptions, use cases. Notice the title of his book, the subtitle, A Use Case Driven Approach. Jakobson's idea was that the development of applications should be driven by these delivery independent use cases. In other words, it's the use cases that form the central organizing principles and the abstractions around which the system is built. at the architecture of a use case driven system, you see the use cases, not the delivery mechanism. What you see is the intent of the system. So what does such an architecture look like? Well before we can answer that, we're going A use case is nothing more and nothing less than a formal description of how a user interacts with the system in order to achieve a specific goal. For example, let's say that the goal was to create an order within an order processing system. Then the use case might look like this. This use case doesn't mention anything about screens, buttons, fields, pulldowns, pop-ups, or anything that might be related to the web. Instead, it talks about the data and the commands that go into the system and the way the system responds. This is important. independent use cases. Notice also that the response of the use case, the order ID, can be completely hidden by the delivery mechanism. The human order clerk never needs to see it. Instead, the delivery mechanism could pop up a nice friendly screen asking the user to add items for the interpretation of input data and the generation of output data. This means that we could create an object that implements that use case. Of course there are details that make use cases much more complicated than the scenario I just showed you. For example, in this use case, what would happen if there were validation failures on one or more of the entered fields? of the use case. It's called the primary course because it shows you what happens if nothing goes wrong. But what if something does go wrong? That's handled by an exceptional course or an extension course of the use case. Again, notice how these extensions are simple modifications to the high level algorithm of the original use case. It'd be pretty simple to integrate these extensions We don't have time in this video to go through all the nooks and crannies of use cases. That's a video for another time. In the meantime, I suggest you take a look at this book, Alastair Coburn's Writing Effective Use Cases. There's nobody better than Alastair to flesh out the details of use cases for you. And if you're confused about the difference between user stories and use cases, don't be. the use case, they eventually evolve into the full-fledged use case. If you'd like a really good description of that process, get a copy of this book, User Stories Applied by Mike Cohn. When it comes to user stories, Mike is the undisputed master. Now look again at the algorithm of the primary course of this use case. It mentions other business objects like customer defines the use case and clearly it has business rules in it but those business rules don't belong in the customer and order business objects so the question is where do those business rules belong what kind of object should we put them in and where would that use case object fit in our system architecture as we create more and more use cases we're going to discover more and more business more use case algorithms. This leaves us with a problem.

How do we partition our system in such a way that these use cases become the central organizing principle? At this point you might be trying to fit this into the model view controller structure. not what we're doing. I'll show you how Model View Controller fits into this later. The objects we're talking about now are at a higher architectural level than Model View Controller. In his book, Jakobsen recognized that architectures like this have three fundamental kinds of objects. Business objects, which he called entities, user interface objects, which he called boundaries, which he called controls but that we will call interactors to avoid confusion with model view controller. Entity objects are repositories of application independent business rules. The methods on entity objects perform functions that are valid in any of the applications that the entity object can be used in. For example, consider a product object. would be useful to an order entry system, an order fulfillment system, an inventory management system, or even an online catalog. In Jakobson's view, the methods of that product object should be useful to all those applications. More to the point, that product object would have no methods that were specific to any of those applications. would go into one of the interactor objects. Use cases are application specific. Use cases are also implemented by interactor objects. Therefore, interactor objects are also application specific. And that means that any application specific business rule belongs inside of an item. These use cases belong to the order entry system. Therefore, the interactor objects that implement those use cases are specific to the order entry system. The interactors achieve their goals with application specific logic that calls the application agnostic logic within the entities. For example, the create order interactor invokes both the constructor and the get ID of the order entity. Clearly, these two methods are application agnostic. It's the interactor that knows how to call those methods to achieve the goal of the use case. One of the jobs of a use case is to accept input data from the user and to deliver output data back to the user. This is the job of the third kind of object, Boundary objects isolate the use cases from the delivery mechanism and provide a communications pathway between the two. If you've got an MVC system or a console system or a thick client system all of that delivery stuff is on the far side of the boundary. The use cases on the other wraps it up into a nice neat canonical form, ships it through the boundary to the interactors. The interactors then invoke their application specific business rules, they go on to manipulate the entity objects and their application agnostic business rules, finally they gather the result data together, wrap it up into a nice neat canonical form, and ship it back through the boundary to the delivery mechanism. And so the business rules, both the application-specific and the application-agnostic ones, are strongly decoupled from the delivery mechanism. In the case of a web-delivered system, the whole web framework hangs off the side of the application architecture the way your appendix hangs off the side of your large intestine. And that's the way it should be. web or on a console or through a thick client or even as a set of web services. The delivery mechanism should hang off the side of the primary architecture like an appendix. This then is how we partition a system. We describe the behavior of the system in terms of use cases. we capture the application agnostic behavior into entity objects controlled by those interactors

then we hang the UI off to the side like a big appendix using boundary objects that communicate to the interactors and the thing about appendices is they need to be easy to remove from our system so that they're easy to excise. That isolation is what we're talking about next. To keep the delivery mechanism appendix strongly isolated from the rest of the architecture, we need to strictly control those source code dependencies so that they cross the boundary in just one direction. So let's say that the delivery mechanism is the web. Okay then, the web server, the web framework, all the web trappings live on the delivery side of the boundary. URLs, routing, HTML, CSS, JavaScript, all on the delivery side of the boundary. In fact the model view controller of the boundary. Models? MVC? Wait a minute! Aren't the models of MVC business objects? Aren't they supposed to be on the other side of the boundary? Heavens no! But that's because the model view controller that we know and love today is quite a bit different from the invented in 1979 by Tryggve Rynskeog. In those early days, the model really was something closely related to a business object. But nowadays, in modern model view controller frameworks, the models are something much more complicated. Nowadays it's naive to think that complex views can be derived from simple business objects. whole collaboration of business object. In fact, most views are a significant portion of a use case. So, nowadays the model in a model view controller framework should draw its data from the collaboration of entities and interactors that implement the use cases. That model may be a single object, but it's not a business object. In fact, those models are seldom more than simple structures that are passed back and forth across the boundary. Think of it this way. In the web version of Model-View-Controller, the web server receives an HTTP request. It runs this request through its routing mechanism in order to select a controller. The controller then parses through the HTTP request, extracting the request data from it and putting it into a nice, simple little data structure. That data structure is really simple. It doesn't have any of the dictionaries or hash maps that are usually associated with web servers and web frameworks. All the web related data tokens and data identifiers and formats have been removed. If you were to look at it in isolation, you wouldn't be able to tell that any part of it came from the web. It's just a pure and simple data structure. and it's shipped across the boundary to an interactor. The interactor orchestrates the magic of converting the model of the request into the model of the response. It implements the use case, it coordinates the dance between all the entity objects, it gathers together all the result data, and places it into a pure data structure, the model of the response. structure has no trappings of the web. All the data is there, but you can't tell it's for the web just by looking at it. The interactor then passes the response model back through the boundary to a presenter object in the delivery framework. That presenter object converts the data elements in the response model to a format more suitable for the web. The final presentation data structure is This renders it in HTML. Some of you are going to recognize this as the good old model view presenter pattern. Now notice the structure of the boundary. It's composed of two separate sets of interfaces. The first set of interfaces is used by the controllers but implemented by the interactors. It accepts request model data structures. interfaces is used

by the interactors but implemented by the presenters. It accepts response model data structures. Those interfaces are the boundary. They belong on the application side and they are part of the application architecture. The delivery mechanism depends upon them, even implements them. Now, if you're using a web framework like JSF or Struts or Spring or Rails, then I want you to notice something about these entity and interactor objects. They don't know anything at all about the framework. And that means you can develop them and you can test them without the web server running. You can run all your tests without the web server running. plain old objects. In fact, you could write and test all the interactor and entity objects long before you had to make a web server decision. You could get the whole application working before you decided on a web server. You could defer that web server decision for a very, very long time. In fact, even really late in the project, if you decided that the web was just the wrong it wouldn't be too late to change your mind. And isn't that what a good architecture should allow? How does the database fit into all of this? Are your entity objects persistent? database? If you're using Rails, do your entity objects derive from active record? If you're using Hibernate, is Hibernate fetching your entity objects from the database? Probably not. Remember back in episode 5 we talked about the impedance mismatch between relational databases and objects? Remember we said that databases contain data structures and not That's likely the situation here. The schema of the database will very likely contain data that will be used both by the interactor objects and by the entity objects. What's more, the way that data is arranged in the database is probably not the way either the interactors or the business objects would like to see it. Therefore, you will likely need to provide a boundary layer from your entity and interactor objects, probably using the design mechanism we talked about back in episode 5, namely a set of interfaces on the application side implemented by the database side with all the dependencies pointing towards the application. Generally, it's the interactor objects that know when to open, commit, or roll back a transaction. through the appropriate layer. So it's the interactor objects that use the database abstractions on the application side. The implementation of those abstractions on the database side construct entity objects and return them back to the interactors so that they can use them to fulfill their use cases. And so there you have it, an architecture that isolates the delivery mechanism and use cases an architecture that makes those use cases the primary abstraction and the central organizing principle what a concept oh and just imagine how easy it's going to be to do test-driven development on all that application code it's going to make the tests really trivial to conceive of and to write. What's more, all that isolation makes those tests run really fast. I mean, you don't need the database, you don't need the web server, your tests can just scream! Better still, this architecture allows you to independently deploy the application. in its own DLL or JAR file or GEM or what have you. If you were to change the UI or the database or the web server, you might not have to redeploy the application. And on top of all that, if in the end you decide you don't like that GUI framework, well then the task to replace it is definable, quantifiable, and has a minimum impact on the application structure. Oh, and if you think that

this kind of architecture is just too much work, that it'll slow you down, I suggest you talk to your mother about that. She'll tell you. If you put everything away in nicely organized places instead of munging it all together and shoving it under the bed, you'll spend a lot less time looking for what you need. Nicely organized architectures make you go faster, where everything goes, stuff's not all tangled together in a knot of dependencies, and it's very easy to stay out of other people's way. I wrote this book, Agile Software Development Principles, Patterns and Practices, back in 2003. Surprised the heck out of me, but I won the Jolt Award for it. That was cool. I didn't even know there was an award ceremony, and then all of a sudden they called my name and gave me a blue bottle. Cool. Anyway, this book contains discussions on the solid principles of object-oriented design, which we're going to be doing quite a few episodes on. It also talks about design patterns, which we're going to be doing even more episodes on. And it talks about agile practices, It's kind of a compendium of all the great stuff that's happened in software since about 1990 to the year 2000. In this book, there's a case study showing the kind of architecture we've been talking about. We're going to do a quick walk through that case study right now. The application is a payroll system. What should you see when you look at the architecture of a payroll system? You should see payroll. Do you think you should see web? Do you think you should see dependency injection? Do you think you should see model view controller or hibernate or database or spring? No, you shouldn't see any of those things. When you look at the architecture of a payroll system, you should see payroll. And that's exactly what we're going to see here. The payroll system is pretty simple. say about it. Some of our employees work hourly and they get paid an hourly rate and like they have to submit time cards at the end of every day so on Fridays we need to know whether or not they worked more than eight hours a day or 40 hours Some employees work in sales and they get a commission rate plus base salary and every week they have to submit their sales receipts. On the first and the third Friday then they get paid but it's a base rate plus their commission. The rest of the employees, they're exempt and on the last working day of the month then they just get their regular salary. ways to get paid like they can pick up their paycheck by the paymaster or we could mail it to them or it could just be direct deposited into their account this next one is really tricky so I've got notes some employees belong to a union and the union deducts their dues out of their paychecks ah and they might Anyway, these charges are submitted to the payroll system at the end of the pay period and it will be deducted out of the paychecks of the next week. So this is a web-based system and during the day all of our clerks will make changes to the database and they'll do stuff like add employees or modify assistant employees and sales receipts and service charges and all that kind of thing we want it to be a web 2.0 system with a drag-and-drop feature and all that cool Ajax stuff we've heard about Ajax and think it's really cool and think it'll give us a competitive edge architecture we heard about sawa and really think it's the way to go we also heard like that there's like a standard database schema and a service protocol that is developed for payroll systems developed by ISO SIPA and we think it's really important to conform and be ISO SIPA compliant Oh, and let me tell

you, we need to have these done by next month. And we've heard about the agile practices and methods that they make you go faster. So we want you to use those, okay? Oh my God, I thought we sent these people off on the first spaceship. Okay, never mind that last bit. episode about how to say no, how to do estimates, and the real purpose for agile methods? In the meantime, let's just analyze the meaningful requirements. First of all, we're going to ignore all that stuff about Web 2.0 and Ajax and service-oriented architectures and database schemas and goofy standards We're going to treat all that stuff as things we can safely defer. It will be the goal of our architecture to make all those things irrelevant, which means they'll be easy to add later if necessary. We begin, of course, with the use cases. The first one we're going to look at is add employee. Here it is in schematic form. Notice that the use case has three variations, types. Notice also that there's a lot of data associated with the use case. The employee's ID, his name, his address, payment numbers, employee type, and notice that the employee type is encoded into letters H, S, and C for hourly, salaried, and commissioned. Notice that this use case doesn't or even I-S-O-S-I-P-P-A, ISO-SIPA. The use case seems to be agnostic about all this stuff. As a matter of fact, if you squint at it really hard, I bet you could imagine that that use case could be applied for punch cards, or even some strange console application. as possible about delivery mechanisms, frameworks, databases, standards bodies, and all that architecturally irrelevant stuff. So we separate the things that matter from the things that don't. This use case lends itself pretty nicely to a little bit of object-oriented design. The use case comes in three variations. It's that captured the generic parts of all three of those variations, and then three derivatives that captured the differences. So here is the design for our first use case. Notice that I put it in a class named Transaction. The word Transaction just means Interactor, I used them as synonyms. So this is the design for our first Interactor. entities represent business rules or business objects. So what kind of business objects should we have for this application? Well here's a proposal. Maybe we should have an employee base class and three derivatives, one for each of the three employee types. Nice isn't it? So the next use case we're going to look at is change employee and whoa it's kind of complicated. isn't it? I mean, look at all the things that can change. You can change the employee's name, his address, his payment numbers, the employee type, where you send his paycheck, his union stuff. Wait, did I say type? You can change the employee's type? We better go back and look at that object model again. Yeah, that's not going to work. They're going to want to change hourly and salaried employees into commissioned employees. In most OO languages, it's not easy to change the type of an object once you've constructed it. So maybe we need to look at a different abstraction. Okay, that's better. Now the employee doesn't have three types. Now the employee holds a payment classification of which there are three varieties, hourly, salaried, and commissioned. place to hang the time cards and the sales receipts and it also gives us a hint about how we might manage the payment method and the union affiliation. Do we really trust that all the hourly employees are going to be paid every Friday from now until eternity and that all the commissioned employees will be paid every

other Friday that long as well or might it be change. Indeed it might. So let's complete the entity model by using the same strategy for the payment schedule. Notice that it was consideration of the use cases that led us to the reasoning that resulted in this model. Had we not carefully considered the change employee use case, it's possible we would have us to a nasty hack later on. Indeed, early use case analysis would seem to be essential for understanding the business object model. Now we can look more closely at the interactors that implement the addEmployee use case. object, but it's the three derivatives of the addEmployee transaction that supply the schedules. This seems wholly appropriate. We don't want the entities to know that hourly employees are paid weekly or that salaried employees are paid monthly. It's the use cases that know this. Look where that database lives. It hangs off the side of the Interactor base class. really nice I mean none of the entities knows anything at all about the database it's the interactor that controls the database and uses it as needed it's the interactor that knows about database transactions queries connections and all that database II kind of stuff if we continue this use case analysis as I did we'll start to see a lovely little abstraction peeking out through the details. Take a look at this pseudocode. This pseudocode tells us the truth. For every employee, if today is the day that employee should be paid, then calculate his pay and deliver the pay. This is just the fundamental truth. And notice that it is devoid of all details. web, no mention of frameworks, no mention of hourly, commissioned, salaried, weekly, bi-weekly, monthly, all of that stuff is gone. There's something beautiful about the fact that this code stands alone, independent, and says the absolute truth. Better yet, this code is independent. independent of the rest of the system. There's something beautiful about the fact that this code is both true and independent. It's the core abstraction of the system. It is the utter truth, and it's stated in one place without any obscuring details or any nagging dependencies. If you want to see it in more detail, take a look in this book. Or if you're a.NET guy, you can take a look in the C Sharp edition. Notice that in the end, we're going to wind up with a set of interactor objects that manipulate a set of simple business entities and control persistence through a basic database abstraction. Nowhere have we mentioned service-oriented architecture, schemas, standards, or any of those architecturally irrelevant things. The architecture of this system is about payroll, not about those other distractions. And yet, we could take this architecture, wrap it in a fancy Web 2.0 sheath, bind it to MySQL, and couple it all together with a service-oriented architecture, and not change And notice, we haven't even mentioned struts, or spring, or rails, or hibernate, or any of those frameworks. If we need to, we can always add them later. In short, we have not been flummoxed by frameworks, saddled by standards, corralled by communications protocols, detained by databases, or wrangled by web servers. tangling influences. And that's what architecture is all about. Bah! Stuff and nonsense! Utter balderdash, sir! And so we've come to the end of yet another episode of Clean Code. In this episode we learned that architectures are not based on tools and frameworks. On the contrary, good architectures allow you to defer the decisions about tools and frameworks for a very long time. In fact, a good architecture maximizes the number of decisions that good system

architectures hide the delivery mechanism rather than exposing it. For example, if you look at the shape of a system, you should not be able to tell that it's a web system. We learned that a good architecture allows the business to separate the cost of the use cases from the cost of the UI and other system components, allowing the business to compare that cost to their corresponding business values. that the use cases of the system should be the primary abstractions and the central organizing principles of the system architecture. That when you look at that architecture, you should see the intent of the system and not the user interface. We learned how to use Jacobson's Interactor, Entity, and Boundary partition to create use case oriented architectures. We learned that interactors encapsulate use cases, entities isolate business objects, and boundaries isolate us from the user interface. Skol! We learned that to achieve the details of that isolation, we can create a boundary of interfaces that separate the application from the delivery side. We also learned that Model-View-Controller makes a good structure for the delivery side, to communicate across that boundary. We learned that databases should be separated from entity and interactor objects using the layering technique that we studied way back in episode 5. We also discussed that it is the interactor objects that are responsible for orchestrating database access. Finally, we examined a simple case study to solidify all these ideas. should talk about. Would you build a house on a pole, sir? Deferring critical foundational decisions will lead you to create a big ball of mud. You will have not laid the foundation properly and your entire system will rest on a rickety, undefined, unstable pole. leaving you to maintain a formless pile of wreckage. Good luck to you, sir. I hear this argument a lot, and frankly, it's nonsense. Web frameworks are not foundations. Databases are not foundations. They're tools. There's nothing foundational about them. The true foundation of a system lies in its use cases. To lay that foundation, you ask yourself, must this system do? But you ask it in a delivery agnostic way. Then you create the abstractions that support those delivery agnostic use cases. This does not lead to a big ball of mud. On the contrary, it leads to a system whose structure is based on its intent and not on some architect's architect. You've no idea what I do or who I am. Many companies establish a high-level technical role that they call architect. Sometimes this role is more political and administrative than technical. Often to defend management's budgets and schedules, and to make framework procurement decisions. While these functions are necessary, they're hardly architectural in nature. Other companies will promote senior programmers and senior developers to the status of architect. They might not expect these architects to code. In fact, usually they don't want them to code. lead, someone who will set the shape of interfaces and high-level designs, maybe they might go so far as to do some code reviews. Both of these definitions carry the assumption that architects don't code, that somehow coding is too low level for someone of the stature of an architect. Well that's baloney. become irrelevant. Architects who make high-level designs and never code them never lie in the beds that they make for the developers. If you are an architect and you want to be effective in that role, then you write code. You should work right alongside the programmers who are lying in the bed that you made for them. the troops in

the trenches with them so you can see what the real architectural problems are. You don't have to code a hundred percent of the time. You don't even have to code 50% of the time. But you do have to code some of the time and when you code you should code well. So that's it. I hope you enjoyed yourself. I hope you learned something. But boy we have a lot more to talk about. I mean, there's all the design principles and design patterns and agile methods and you're not gonna wanna miss the next exciting episode of Clean Code, episode eight, the solid principles, part one. Come on, dogs, let's go out. Yeah, dogs, yeah, let's go, yeah, let's go. We'll be right back. We'll be right back. We'll be right back. We'll see you next time.