Hi, I'm Uncle Bob, and this is Clean Code. In this episode In this episode Mmm By Uncle Bob Enjoy Bye Welcome, welcome to Episode 9, The Single Responsibility Principle. This is the second in our series on the solid principles. Come on in, come on in. In the previous episode, we laid the foundations for the solid principles. The software told us that the source code was the design. We showed that these economics imply that software should be designed and built iteratively, without huge upfront planning, and that the design should be constantly cleaned and improved. We discussed design smells like rigidity, fragility, immobility, viscosity, and needless complexity. rot, and we showed how code with an inverted dependency structure tends not to rot. We defined object-oriented design as the inversion of key dependencies that isolate high-level policy from low-level details. Finally, we talked about dependency management and the role that the SOLID principles play in keeping source code dependencies under control. once we talk about general relativity, we're going to dive right into the single responsibility principle. We'll learn what we mean by the term responsibility as it pertains to software modules, and we will come to understand that responsibilities have profound long-term effects on the maintainability and flexibility of software. We'll learn why it's harmful to system structure, many responsibilities within the same module. We'll discuss what it means to have a single responsibility and why that should be the goal for every class in your system. I'll show you several techniques for splitting a class that has multiple responsibilities so that it conforms to the single responsibility principle. And then we'll look at a case study that walks us through all these exposes and explains the single responsibility principle in a real live example. So y'all, hold on to your hats, buckle up your boots, keep your feet in your stirrups and hang on, because we're going to stampede into the single responsibility principle. Yee-haw! In 1905, Einstein showed that Galileo's principle of relativity could be safely reconciled with Maxwell's equations of electrodynamics if we assume that the speed of light is constant to all frames of reference. His special theory of relativity showed that there was no mechanical or electrical measurements you could make that would allow you to determine your absolute velocity through space. that the very concept of an absolute velocity, other than that of light, was meaningless. But his theory had one huge flaw. It assumed that all frames of references were inertial, that is, they moved at a fixed velocity. There was no acceleration. But in a universe filled with gravitational fields, no such frame of reference exists. The special theory of relativity Interesting, but it doesn't apply anywhere in this universe. This bothered Einstein a lot. But then later, in 1907, while still working at the patent office, he had what he later described was his happiest thought. Imagine Galileo's ship again. You're inside the cabin and you can't see out. You've got no way to measure the motion of that ship. you feel gravity. You can drop items and see that they're accelerated downwards at 9.8 meters per second squared. But does that gravity actually exist? Or is the vessel within which you stand being accelerated upwards at 9.8 meters per second squared? This was Einstein's great insight. There is no experiment you can perform, no measurement you can make, you whether you're in a gravitational field or simply being accelerated in the opposite direction. The two situations are exactly equivalent. Indeed, Einstein called this the principle of equivalence. Now let's go back to the special theory for a moment. Einstein expressed the special theory of relativity as a set of coordinate transformations between each other. If you were in one frame of reference and you wanted to know what someone else in another frame of reference would measure, you would simply apply the transformations. And as we saw in the last episode, measurements of time and length do not agree between the two frames of reference. mixed those two concepts. He came up with the concept of space-time, and he used space-time as the coordinate system for the two frames of reference, because measurements in space-time do agree between the two frames of reference. In 1907, the problem that Einstein faced was whether his principle of equivalence, the more general form of relativity, could be expressed of coordinate transformations. It took Einstein eight years. But by 1915, he had finally found the coordinate transformations that incorporated gravity and acceleration into relativity. But whereas the special theory cast those transformations upon a nice, flat, simple Euclidean coordinate system, the only way Einstein could make the general transformations abandon Euclid altogether and assume that spacetime was curved. The upshot of all this is that gravity is equivalent to a curvature in spacetime. Mass warps space around it. Bodies that move through that warped space travel in curved trajectories, giving them the appearance of acceleration. Gravity is a curvature in spacetime. think that curvature is subtle. It's not. See that curved path? That is the straightest line this object could follow in this part of spacetime. That is the curvature of spacetime in this vicinity. Turns out, around here, near the surface of the earth, spacetime is the paths that objects will follow inside that space-time, and the slower time will flow for those objects as well. Is space-time really curved? All we can really say about that is that the predictions of general relativity have been verified over and over and over again. From the strange time dilation effects that occur deep in gravitational fields, to the dragged around rotating bodies and to the way light bends around massive objects. It all works and to the best of our ability to measure it, it works exactly the way Einstein said it should. General relativity is one of the most reliable theories we have. But that leads us to a really nasty problem. We're not going to talk about that just now. The Single Responsibility Principle is about functions and modules, and it says that the best modules are those that have just one responsibility. So what is a responsibility? methods calculate pay save and describe employee the first method is pretty obvious it simply calculates the pay for the employee the second method save stores the fields of the employee on some kind of database the third method Another process will incorporate that string into a report about all employees. How many responsibilities does this class have? You're clever. You can make a reason to guess that the answer is three, right? I mean, there's three methods, so there's three responsibilities. Sure. So let's make this a bit trickier. This method searches through the employee database and returns an instance of the employee that matches the ID. Now how many responsibilities are there? Of course the answer is still three because the new method is in the same family as save. They're both database functions. Similarly, if I added methods like calculate deductions and calculate taxes, we'd consider family as calculatePay, so the number of responsibilities wouldn't change. So now let's say I add a new method named summarizeHoursWorked. This method returns a string that gets incorporated into a report similar to describeEmployee. Is this a new responsibility? Which family does this method belong to? calculate pay family or does it belong to a different family altogether? The answer to that question is who is the audience for that method or rather who are the users who will request changes to that method? it talks about are the responsibilities that our classes and functions have to those users, specifically users who will request changes to the software. So a responsibility can be viewed as a source of change because the people served by that responsibility will certainly request changes to the software. Who are the people who are the sources of change for the employee class? calculates pay, it's the lawyers, managers, and accountants who define the payroll policy. For the family of functions that deal with the database, the sources of change are the people who specify the schema and platform of the database. In most companies, these would be DBAs. In some companies, they'd be architects. make changes to those reports are the consumers of those reports, the clerks and accountants in the operations organization who monitor the payroll process. These three different groups of people have very different needs and expectations, and yet the employee class, as written, is responsible to each and every one of them. So the responsibilities that the single responsibility principle is talking about are the responsibilities responsibilities that your software has to all the different groups of people that it serves. Of course, people tend to wear multiple hats. In a small company, for example, policy, architecture, and operations might all be handled by one person. As the company grows, other people will step into those roles. The allocation of people to roles will change. To avoid coupling the structure of our software to such messy vagaries, we simply separate the users of our software from the roles that they play. When users play certain roles, we call them actors. Responsibilities are tied to actors, not to individuals. So for our employee class, there are three actors. We'll call them policy, architecture, and operations. Whenever the needs of an actor change, the family of functions that serves that actor will also have to change. So a responsibility is a family of functions that serves one particular actor. needs of that actor change, it becomes a source of change for that family of functions. The actor for a responsibility is the single source of change for that responsibility. Our users usually expect to get value from the software that we make them. Usually that or make them money. And so often they pay us money in exchange for that value. There are two different values of software. I call them the primary and secondary values. We're going to deal with the secondary value first because that's the one that most people think of first. The secondary value of software is its behavior. If the software does what the users need bugs, crashes, or delays, then the secondary value is high. This secondary value of software is achieved when the current software meets the current needs of the current user. But the needs of the users change, and they change frequently, and so the behavior of the software gets out of sync with the user's current needs, and the secondary value of If our software is to have a reasonable lifetime, the business must be able to keep the secondary value of that software very high by rapidly changing and enhancing that software to keep up with the customer's constantly changing needs. The ability of software systems to tolerate and facilitate such ongoing change is the The primary value of software is that it is soft. Imagine a system that currently meets the user's needs, but is really hard to change. The secondary value of that system is high, but the primary value is low. Unfortunately, this is a really common situation. Such systems are initially profitable because they meet the user's needs, but the cost of keeping pace with the user's changing needs is so high that profitability rapidly decreases. So now imagine a system that has a high primary value but a low secondary value. Such a system is disappointing to the users at first because it doesn't meet all their needs right away. But because the primary value is high and the system is flexible, because more and more of their needs are met. And so, at the end, profitability increases. Sustainable profitability is therefore tied to the primary value of software, and that's why it's primary. In short, if the software is easy to change, it's good for the business. If the software is hard to change, it's bad for the business. Therefore, it is the first responsibility of software developers to keep the primary value of software high. Perhaps you thought your job was to get the software to work, and of course it is. But that's just your secondary job. Your primary job is to give the software a shape and structure that makes it easy to change. Where does the secondary value of software come from? What is it that meets the needs of the users? It is of course the responsibilities that we mentioned in the previous section. Those responsibilities are families of functions that serve the needs of particular actors. by writing the corresponding functions in those modules. And it is in this allocation of functions to modules that much of the primary value of software is achieved. We can make our software much easier to maintain and enhance by carefully choosing which modules to put our functions in. As a simple example, let's revisit the Employee class from the previous segment. This class is a single module with three responsibilities. So let's say that the needs of both the Policy and Architecture actors are changing. The Policy actors need some business rules to change, Let's also say that Bob is the developer who is most familiar with the business rules. And let's say that Bill is the developer who is most familiar with the database schema. So clearly Bob is going to be the one who changes the business rules. And just as clearly it will be Bill who makes the changes to the schema. But remember, policy and architecture are both combined into the employee class. and Bill are going to be changing the same module. This means that when Bill or Bob check their code back into source code control there's likely to be a collision, possibly even a merge. This is really unfortunate. Bill and Bob ought to be able to work separately on such different responsibilities. But because the two responsibilities are together in the same module, the employee class, Bill interfere with each other as they're making those changes and that makes those changes difficult to do. This reduces the primary value of the software. This employee class really knows a lot. It knows about business rules, it knows about databases, it knows about reports and formatting. It's got a responsibilities. Each of these responsibilities will cause the employee class to use other classes in the system. For example, the database responsibility will force the employee class to use the database API.

The reporting functionality will force the employee class to know about the string APIs and many And the business rule functionality will likely require the employee class to use several lower level calculation engines. This means that the employee class has a huge fan out, and that fan out makes the employee sensitive to changes in the lower parts of the system. What's more, there are other classes above employee that depend upon employee, and they dependencies. In general, it's always a good idea to restrict the fan out of a class. And one good way to do that is to minimize the number of responsibilities in that class. So let's say that the operations actor needs a new report. Since the old report's already sitting there in the employee class, we might as well put the new report in there too. But the employee class also contains the policy and architecture responsibilities. Those responsibilities haven't changed at all, and yet the module that houses them must be changed in order to add the new report. The modification date of the employee class is going to change. And that means, in languages like Java, even though they don't use the new report. Think about it. The classes that call the business rules for calculating pay are going to have to be recompiled and redeployed because somebody changed a report. That means the business rules have been coupled to the reports. What's more, the classes that call the database functions of employee will also have to be recompiled and redeployed. reports. All the actors served by a module will be affected by any change to that module, even changes that those actors don't care about. The co-location of responsibilities couples the actors. Once two responsibilities are coupled by accident, other couplings tend to over time. Developers start to share resources between the two responsibilities simply because they appear in the same module. For example, there might be some function in the operations responsibility that does something similar to a function in the policy responsibility. Maybe they compute some total. If the policy and operations responsibilities are together thinkers will almost certainly call the operations total method from the policy responsibility, and that will create a much tighter coupling between those two responsibilities. But responsibilities change at different times and for different reasons. The coincidence between policy and operations is likely to be short-lived, and that means Policy will eventually change the function that computes the total, and that change will inadvertently break something in operations. So now the system is beginning to exhibit the symptom of fragility. And remember how customers and managers react to fragility. When a small change to one part of the system affects and breaks other parts of the system, can draw only one conclusion. They must believe that the software developers have lost control of their product and don't know what the hell they're doing. reason to change, one and only one responsibility. Another way to say this is that we gather together the things that change for the same reasons and we separate the things that change for different reasons. When we design a system, we are careful to understand who the actors are. Then we identify such that each module has one and only one responsibility. For example, we do not mix SQL and HTML in the same module. It's simple. We do not put business rules into JSPs. We do not implement business rules in stored procedures. and messages with business rules. We do not mix data access and control code with business rules. We keep all these responsibilities separate from each other. We don't put them in the same function, the same class, or the same source file. We do this because these responsibilities change at different times and of different actors back in episode 7 and 5 we learned about boundaries we learned that applications should be kept separate from UI code and database code and other framework code why the single responsibility principle when we keep our responsibilities separate we can change them with the software such that those responsibilities can become plugins to the rest of the application. You cannot have plugins unless you separate. Consider this code. Is there a single responsibility violation or not? And if so, where is it? © BF-WATCH TV 2021 The code performs two functions. First, it creates a list of available directions. Second, it builds a message that it sends to the user about those available directions. Those are two different responsibilities. If the language of the program were to change from English to Spanish, for example, then the second responsibility would change, If, on the other hand, we change what it means for a direction to become available, then the first responsibility will change, but the second will not. Finally, if users don't happen to like the way that messages are punctuated or phrased, then the second responsibility will change, but the first will not. responsibility principle. The two responsibilities should be moved into separate source files, so that changes to one responsibility don't affect the other. Now look at this code. Does it violate the single responsibility principle? This is a nice little recursive algorithm that follows the flight of an arrow from cavern to cavern until it reaches the end. to follow that arrow and report where it winds up. Or does it? It also terminates the game. What does following the flight of the arrow have to do with terminating the game? One of them is mechanics, the other is policy. Clearly these are two separate responsibilities, and they should be separated into different functions, possibly into separate source files. Here's another one. to the Single Responsibility Principle. The function draws the cell as a green rectangle on the screen. It also translates the coordinates into window coordinates by multiplying by cell size. Should the function that knows how to draw the shape and color of a cell also know how to translate coordinates? To answer that question to think about the actors. Is there an actor somewhere out there who cares about the color and state of a cell, but doesn't care about the window? If there's not, then there's probably no single responsibility violation. On the other hand, I have this feeling that a function of this form ought to exist. This function takes cell coordinates and cell state and renders it appropriately. of greenness, rectangleness, and window coordinates from the more abstract concepts of cell coordinates and cell state. Are there actors out there somewhere that care about the state and coordinates of a cell but don't care about the color, shape, and window size? Probably. So I suspect there's a single responsibility principle violation lurking here somewhere. more subtle than the others. To fix this, I'd separate the code that knows about cell coordinates and cell states from the code that knows about shapes, colors, and window coordinates. I'd move them into separate source files so that when one is changed, the other is not affected. One more, and this one's a subtle one. You think that's a single responsibility violation. I hate this kind of stuff. The verbose message function creates log messages if the verbose flag is set. This kind of stuff clutters up the code, making it much harder to understand. Just look how much better this code would be without all those log statements. Is logging a responsibility? Is there an actor served by the logging responsibility? And if so, how do you decouple the logging responsibility from the other responsibility in that function? Clearly logging is its own responsibility. And just as clearly, there's an actor for it. It's not these two dogs. Usually it's you or somebody in operations, the person who reads the log files. separate them. At first, this kind of separation might seem challenging. After all, how do you separate logging statements from the code that's being logged? But it's not really all that hard. Remember episode 3? By extracting till we drop, we can move all those log statements into functions that are well positioned to be separated. We can also clean things up a bit. message statements into functions that do nothing except what is being logged about. This is really nice. We've separated the logging statements and the logged code from the code that's not being logged. Now all that remains is to move those functions into classes and then create a base class that doesn't know how to log and a derivative that does. Notice how the executive base knows That's really nice. Notice also how the logging executive derivative simply defers to the executive base class and surrounds all the calls with logging. So its sole responsibility is logging. And that's also really nice. So we've separated the logging itself from the statements that are the subject of the logging. We've separated those responsibilities. In all the other cases of SRP violation, we moved the two responsibilities into separate source files because we wanted to make sure that if one responsibility changed, the other was unaffected. We could separate executive and logging executive into separate source files, but I think that might be too much separation in this case. After all, logging messages just aren't the kind of thing that change that often. executive and logging executive will ever be separately deployed. So leaving the two responsibilities together in the same source file, but moving them into separate functions and inner classes is probably all the separation we really need in this case. The two responsibilities are significant enough to require separation into functions, but As you can see, conformance to the single responsibility principle occurs at many different levels within the code. Sometimes it causes us to pull code apart into separate source files. Sometimes it causes us to pull classes and functions apart into sub-functions and inner classes. In an upcoming episode, we'll see how sometimes it causes us to pull whole responsibilities and create physical locations in the code where single responsibilities exist. Those physical locations may be functions, classes, source files, modules, or even higher level structures. As we climb this ladder from the small to the large, we'll be merging small responsibilities into ever larger ones. functions, we merge into single classes. Responsibilities that we keep in separate classes, we merge up into single modules, and so on up the ladder. Let's look again at the employee class. In this diagram, you can clearly see that the three actors and the three responsibilities are in a single class. How can we separate those responsibilities? In languages like Java, C-sharp, and C++, the typical OO strategy for this kind of decoupling is to split the class into an interface and an implementation.

This is an example of the dependency inversion principle, Yeah, we will. That's right. We will. We will. We will. That's right. That's right. That's right. Cut. While this certainly decouples the actors from the implementation of the responsibilities, it doesn't decouple the actors from one another, because they all depend on the same interface. It also doesn't decouple the implementations of the responsibilities from one another, because they're all still bound together in a single class. A change to just one of those responsibilities is going to affect all the actors and all their implementations. So although we've achieved some separation, in most cases we're going to want to do better. One of the most obvious ways to split these three responsibilities up is to break the employee class up into three different classes. data and the policy functions behind in the employee class. I pulled out all the database functions into the gateway class and I pulled out all the reporting functions and put it into the reporter class. Now the actors depend on three separate classes and the implementations of those responsibilities are strongly separated. If there's a change to any one of those But this isn't a perfect solution either. There's a transitive dependency from the reporter and the gateway to the employee, and this means that the actors aren't perfectly decoupled. Any change to the employee could potentially impact all of the actors. Second, the concept of employee has been split into three different pieces. So now programmers who are interested in the save or have to hunt for where those functions are. We can make it easier for programmers to find functions by using the facade design pattern. We'll be talking more about this pattern in an upcoming episode. To use the facade pattern we simply put all three function families back together into a hiding those implementations from the actors. This makes it pretty easy for programmers to find functions because they're all in the facade. It also keeps the implementations of the responsibilities very separate since they're all in different classes which are probably in different source files. However, the actors are coupled to each other again. Any change to just one of the functions can affect all of the actors. We can turn this around and use interface segregation, which we'll be studying in more depth in an upcoming episode. To segregate the interfaces, we simply create three interface classes, one for each responsibility, and then we implement those three interfaces with a single class at the bottom. This keeps the actors entirely decoupled, depends on its own interface, and those interfaces aren't related. However, it reintroduces the hunting problem, because now developers are going to have to hunt for the interface they need. It also leaves the implementations of those responsibilities coupled, since they're all held in the same class. I'll bet you thought I was problem, didn't you? I'll bet you thought that with a wave of my wand, I could make all those problems disappear. Welcome to engineering. Welcome to the world of perpetual trade-offs. Perfect solutions are for mathematicians and fiction writers. The rest of us who live in the real world, we're stuck managing a plethora of mutually exclusive forces. coupling the responsibilities in one way or another. Or we can completely decouple the responsibilities and leave the functions a little harder to find. There isn't always a perfect solution. In fact, most of the time the solution simply balance the forces without resolving them completely. Perhaps you're not worried about how hard it is for the programmers to find the functions. Perhaps you don't think it'll be that hard. Or maybe you know your team know the team is familiar enough with the code to find the functions they need. In that case, you can push the separation to the limit. Or perhaps you're working on an API that hundreds of other programmers are going to use, programmers that you don't know. So you want to keep that API real simple and real easy to use. In that case, maybe you're willing to endure the extra coupling of a facade just to keep the API And that kind of trade-off is exactly what engineering is about. And that is why that software, although it's a craft and an art, is also an engineering discipline. The solid principles and the other principles and patterns that we'll be studying in this video series are all bound together by the same forces and constraints and trade-offs. They're all part of the larger discipline. of software engineering. So before we end, let's take a look at a simple case study. Do you know the game Mastermind? This is a two-player puzzle solving game in which the first player, the code maker, creates a four-letter code out of the letters A, B, C, D, E, or F. the code breaker, tries to guess that code by offering a sequence of guesses. After each one of those guesses, the code maker offers specific clues. Okay, so let's say that I'm the code maker and I make up a code DFCA. You're the code breaker, so you have to guess what my code is. And your first guess is ABCD. As the code maker, I have to score your guess. And the score that I give it is minus minus plus. The minus sign means that you guessed a correct letter, but it's in the wrong place, like A and D. The plus sign means that you guessed a correct letter, and it's in the right place, like C. I've written a little Java program that plays this game with you. code breaker. You give it clues, and it responds to your clues with more guesses. Let's see how this works. Okay, let's try to play this game. First we'll invent a code. Let's say F-E-A-D. Okay, I've just written that down so that we can refer to it. to score A, A, A, A. So that's the game's first guess for A's. And I can give it a score. It's a plus sign because one of those A's is in the right place. So now the game says, okay, score A, B, B, B. And, well, I've got to give that a minus sign because the A is the only correct letter and it's in the wrong place. Okay, I'll give that another minus sign, aren't I? Now it's D, D, A, D. Well, the first two Ds don't matter. The last A is a plus sign, and the last D is a plus sign. Now it's D, E, A, E. Hmm. place. The E is a plus sign. It's in the right place. So is the A. And the final E doesn't matter. F, D, A. Plus sign for the F. Minus sign for the D. Plus sign for the A. And minus It took seven tries and it managed to get the whole thing done. And that's how you play the game of Mastermind. Which parts of this program should be separated from one another? And who are the actors? One actor, we'll call him the game designer, is responsible for the messages, is responsible for the text of the messages, the formatting of the messages, the content of the messages, the language of the messages, and whether those messages appear on the web, the console, a thick client, or some other venue altogether. Another actor, we'll call him the strategist, is responsible for choosing the algorithm that guesses the code. randomly or perhaps this algorithm would simply choose all the codes in sequence or perhaps that guessing algorithm would be a little more intelligent than that and actually look at the clues yet another actor in this system is the person who determines the flow of the game this is the person who would decide whether or not we were doing nothing more than guesses and responses or would decide that it would be best to insert a betting round after every guess. This person might decide that we should offer advertisements of other people's products after every third or fifth round. He might decide, for example, that the player should be taunted whenever he makes a mistake. Maybe there should be level ups or badges. We're going to call this These are the three actors who will want to make changes to their particular areas of concern without affecting the other actors. So these are the three responsibilities that we need to keep separate in our source code. The customer responsibility defines the game of Mastermind. This is the responsibility that houses most of the use cases. So if you want to know how to play the game go talk to the customer actor he's the one who knows so the customer responsibility is the architectural center of the application now remember what we learned back in episodes 5 and 7 about boundaries and architecture the architecture of this system should have the application at its center the other responsibilities should be separated from it by boundaries and all should cross those boundaries, pointing inwards towards the application. In this diagram, we see the three packages, or namespaces, that serve the three responsibilities. The GamePlay package serves the customer responsibility, the GameInterface package serves the game designer responsibility, and the Strategy package serves the strategist responsibility. direction of the dependencies. They all point towards the gameplay package, the package that supports the customer responsibility. This comports well with our notion of good architecture because the application is in the center and the other responsibilities plug into it. The design of the program follows this architecture nicely. The game engine class and its minions drives the overall flow of the game. It serves the customer actor. The game console communicates with the player. It specifies the formats and spellings of all the messages. It also interprets the responses from the player. It serves the game designer actor. The remembering guest checker provides the strategy for choosing the next most appropriate guest. actor. Note the direction of the dependencies between the classes. Whenever one of those dependencies crosses the boundary, it does so pointing towards one of the classes in the gameplay package, which supports the customer responsibility. And now for your homework. There's a URL on your screen. we just studied. I suggest you download it, study it, including all the test cases, and then go back over all the old episodes to see if I was following my own rules. You should find, if I've done my job well, that each class serves one and only one of the three actors, and therefore each class has one and only one responsibility. By the way, I presented this to you in waterfall order, didn't I? I mean, I started with the actors, and then I showed you the package diagram, and then the class diagram, and then I referred to the code. So that's like pure waterfall, isn't it? You think that's the process I used? You think that's the order I built it in? Ha! Hardly! I used an old trick of David Parnas instead. He said, We will never find a process that allows us to define software in a perfectly rational way. The good news is, we can fake it. And boy oh boy did I ever fake it. step rational way. What a load! What I really did was to start writing tests and then getting them to pass. I got a function working that figured out how to score a code. I got another function working that figured out how to generate guesses. And I hopped around like that from function to function until a design started to emerge. Then I put that design together, driving the whole thing with got the whole game working. Then I looked at the architecture. The tests had driven me to create about 80% of the design that I showed you, and the tests had also gone a long way towards helping me identify the three responsibilities. Unit tests, as it turns out, tend to align with actors. separate. This isn't always true, but I have found that it is very often true. But the tests didn't lead me to a perfect design. Once I got the whole thing working, then I did quite a bit of analysis and cleanup. Initially I found some misplaced responsibilities and also a few dependency cycles. We'll be talking So I refactored mercilessly. I kept all the tests passing, but I also moved bits of code around. I ripped classes apart. I changed the names of things and the structures of things. I made lots of changes. It was only after that that I identified the three actors, and so I pulled the classes apart into three distinct packages. all those pretty diagrams for you. Because the best time to draw pretty diagrams and do system documentation is when you're done. So, wow. We've covered a lot of ground here. We've gone from highfalutin philosophies to looking at low-level code. We even threw a little bit of general relativity in for the boot we learned that classes have responsibilities to their users and those users can be classified as actors by the roles that they play we learned that a responsibility is a family of functions that serves one particular actor we learned about the two values of software so that it can continue to meet its requirements throughout its lifetime. We learned that carefully allocating responsibilities to classes and modules is one of the ways we keep the primary value of software high. We found that when modules contain more than one responsibility, the system tends to become fragile due to unintended interactions between those responsibilities. We looked at several examples of code that violated the single responsibility principle. And we looked at several ways that that code could be improved. We learned about several solutions to violations to the single responsibility principle. We studied separations, facades, and interface segregation. We realized that none of these solutions were perfect. Finally, we looked at the mastermind case study, and you learned a little about how to fake a rational design process. So yeah, we've covered a lot of ground in this episode. And if your brain isn't fried, well, mine is, so we're going to stop this here. But we've got so much more to talk about. There's four more of the solid principles to talk about, then the component principles after that, and you're not gonna wanna miss the next exciting episode of Clean Code, episode 10, the open-close principle. Come on you dogs, let's go! Hurr, dogs, hurr! Time for a walk, let's go! Somebody open the door for us! Hurr, hurr! We discuss design principles like virginity, sterility, imbecility, and all the other ilities out there. Cut! Note the direction of the dependencies. Alright, you ready? Yeah. Action. The single… And so now, in this episode, after we talk about general relativity, we're going to dig right into the single responsibility principle. Excuse me. Cut. if I've done my job well, that each class serves one and only one of the three actors, and therefore each class has one and only one responsibility. In the Twilight Zone. Cut. Hey, you wanna see something cool? This was really neat, I just saw this yesterday. Whoa. There's one hell. The Oh, my God. Look at that. Storbritannia Kjell Krona . . . . It surrounds the calls with logging. That's right. That's exactly what it does. It's not usually this bad. Yes, it is. No, it's not this bad. Yes, it is. There are always those that I get completely snuck in. All right. Action.