Jeff "Cheezy" Morgan - Patterns of Automation

Jeff Morgan: Hello, everybody. I am super psyched to be here at the Automation Guild and to give this talk "Patterns of Automation". Just a little bit about myself. My name is Cheezy. Some folks actually know me by my nickname Jeff Morgan, but most people just call me Cheezy. I work for a company called Tango and as you can see, I dance the tango, as well. What do I do? Well, I work mostly in the area of continuous delivery, continuous deployment. The teams I work with every time we commit code, it goes directly to production. So, that's what I do. I help companies learn how to do that and take them all the way through. Many people believe that's largely about DevOps and DevOps is a big piece of that, but it's also about the way we write code, the way we test code. Those have to radically change, because we've got to get to a point where we don't develop and then test after the fact, but instead, we can know each moment that we have a high quality software. So, it's sort of a little bit of a shift left.

This specific talk, I'm giving to you, because there's a lot of things I've seen in working with dozens of teams out there. A lot of problems that I've seen with automation code and a lot of really good patterns that I've seen that help us structure our code in a great way. So, you're going to see some coding. Yeah, you're going to see me do some coding, mistakes and all. But, that's okay. I don't worry about that too much. So, what are these patterns? Well, these patterns are just things that I have seen, and others in the industry have seen, and observed that help us produce a higher quality product, and it's something that we can talk about amongst ourselves. I can mention this pattern or that pattern, and if you're familiar with it, you know exactly what I'm talking about.

I'm going to skip past this fairly quickly and get straight to the first pattern I'm going to talk about. It's this pattern called specification by example. Some people call it BDD. There's other terms that are used, but we'll just use "Specification by example". When we're building software, there really are three things that we tend to create. I work exclusively in a [agile 00:02:19] fashion, so one of the things that we build is user stories that has some acceptance criteria that's along the way that tell us what we should be building, and how we'll know when we're finished. Another thing is, as we build the actual coded self with lots of lots of unit tests, and the final thing is we build some automated higher level tests, as well.

To be quite honest, all three of these things should completely reflect each other. In other words, the specification should be reflected in the code and all of the tests along the way, as well. The problem that we have, is that when lots of different people work on different pieces of this, they tend to get out of sync with each other, and we tend to have some duplication. Especially, between the acceptance criteria and the tests, because the way I see it is that for each high level business objective as defined in a user story, we actually needed an

automated test that first of all, validates that it made it into the system, and secondarily, that it works as it was specified.

Whenever we start to think about what are the differences between these, though, for me, the difference between a requirement and a test comes down to data. Let's take a quick look at an example of that. Let's take a simple example that we're trying to build a calculate, and we're starting with the addition function. So, we might say that the requirement is that addition would equal the sum of the numbers that we put in place, and whenever we would actually turn that into a test, we would have to actually provide real numbers. So, my test would have to say something 3+4=7 and 3+-1=2. As a good tester, I would start putting boundary conditions. I'll try to test adding a zero. I might come up with a really, really, really large number and test it. But, as you can see, all I'm doing is taking the requirement, producing an example, and adding data to build my test. That is essence is what specification by example is. It's all about building examples that define how the requirements are carried out. Furthermore, the other interesting aspect of it is that we try to make those examples the requirements.

So, I could ramble on and on about it, but realistically, it's much better if you see it in the code. So, I'm going to switch over real quickly, and write some codes that you can sort of see this a little bit right now.

Okay. So, here we are looking at the code. Specification by example. So, what you're seeing here is something called Gerkin and as you can see, it's plain English that kind of describes the requirements for what we're about to build. This first one is talking about a thank you message. When I complete the adoption of a puppy, then I should see "Thank you for adopting a puppy!" Again, this is the specification. And, what you see now are the steps. These are generated by the tool, and they all basically say pending right now, so, as you can see, I haven't done any of the coding, yet. Let's get started.

I'm going to start by going out to the page and so, here we go. And, now that I'm on the page, I want to navigate through all of the pages, and complete an adoption. I've got a button to click here and another button, and buttons everywhere, right? So, just click this one. And, one more button, and that gets me to the checkout page. And, on this checkout page, I've got a form that I need to fill out. So, what are the ID's? Yes, okay. I could use some coffee or caffeine right now. But, that's okay. By the way, this isn't a live site, so I'm not going to be receiving puppies at my house by placing this order. And, the dropdown. Okay. And, one more button to click. And, now, I need to validate that I can see the thank you page, so I'm going to use this ... Just look anywhere on the page and see that the text should be anywhere, and I think that does it. So, let's run this first test, and let's see what happens. Wow! What do you know? I got it right.

Okay. Let's go to the next text. I'm just going to copy this down, because that much of it is the same when I checkout without a name. So, the challenge I have

hear is that either the name or the address, I've got to set it to a blank field. So, I'm just going to start by creating these variables. I'm going to try to set them to the right value based on if it's a name or address. So, if the field is equal to a name, I want to make it blank. So, I'll set it to a blank string, and if not, I'll set it Cheezy. The same with the address. So, if the field is an address, I want to set it to blank, otherwise I'll set it to one, two, three. Now, I just need to use these variables when I'm setting the value on the page. So, there it goes.

And, now, error message. So, there is a div on this page that kind of contains the errors, and it has an ID of error explanations. So, I'm going to get ahold of that div first. Inside of that div, there is an unordered list, and I want to grab the text of that, and I want to put that in as error. So, I'm expecting that error to equal the error message that I passed out. And, let's run that and see what we get. Yep, that one worked. Let's run it for address and see. Okay. Looks good. I've got this final test. And again, I need to navigate over to the checkout page. So, I'm just going to grab this and copy it down. I don't suggest you copy/paste, by the way. I'm the only one allowed to do that. So, now I want to grab the options from the page.

Now that I have those options, I want to collect the text from each of those. So, let's do that. And, now I've got an array with the text. Let me now walk through the values that I'm passing in. And, I'm going to expect that array of the text to include each of those things that I'm passing in. So, let's do that to include and that hash, pay by is the key that I use. So, let's run that. Okay. As you could see, we have all those tests working. And, that's all there is to specification by example.

Well, it looks like we got through that. Do you notice that in Gerkin, or in my specification, I didn't say "Click" or I didn't say that I "Entered this text" or "That text"? What I found is that it's important in the specification by example to talk about the business rules. I'm not really talking about the implementation, because the implementation can be a little messy. In this specific case that we looked at, I didn't say which puppy did I select. I didn't say things like having to navigate to these other screens. Instead, I simply said "When I complete the form and leave a name blank, then I should get an error". So, I didn't even talk about any of the details. So, that's kind of an important thing to keep in mind as you do this.

So, we're on to the next pattern and there's no coding involved in this. Basically, what this is is a practice acceptance test driven development I think is a good design pattern that takes the specification by example and puts a workflow around it. So, let's talk about it a little bit. So, in agile teams, they try to do something like this where they like to say hey, I'm going to write some code and then as soon as I write the code, somebody is going to be running to test it. The theory goes that while somebody's testing that, the developer can continue on to story two, and they could start to write the code for that. When they're done, someone magically is going to be ready to test story two, and everything just works perfect, right? Except, it really doesn't, because if an issue or a defect is

found while working the first story, guess what? That second story needs to be put on hold so the developer can come back and finish fixing story one, and deliver it back. But, don't worry, he's going to finally get back to story two, and everything's going to be great, except sometimes it doesn't go as planned.

So, sometimes people keep bouncing back and forth between fixing defects and trying to get more work done. Now, what I call this, is I like to call this development ping pong. It's kind of where a developer pings some code into a test environment and the very first time, the tester finds a defect in just a few seconds, and that gets ponged back, and so on and so on. You know, you've probably played this game many, many, many times. Your typical workflow in agile goes something like where a user story IE., the ball, that's what the orange thing is, goes to development when they're done. They kind of punt it over to test. When they're done, it goes over to review. The product owner looks at it. Of course, every time the product owner absolutely loves the user story, and eventually it goes to done.

The challenge that ping pong game that we just talked about where things keep going back and forth between development and tests. For me, in accepted test [inaudible 00:14:29] development, the first thing that we do is we completely get rid of that test phase. There is no more test. In fact, I think personally having the test phase is sort of an evil thing, because what it is does is it starts to build this idea that development and testing are actually two different things. In fact, they are not. In fact, they are one thing. Building it and building the quality, and then validating the quality are all one activity. So, therefore, whenever the user story comes into the team, the product owners already kind of written some Gerkin and they sort of think that it's right. They've gotten it 75-80% right, maybe. Who knows.

One of the very first things that we do then, before we start writing code is we have this thing called the three amigos, and that's where the developers and testers that are going to work on it sit down and talk with the product owner. They kind of read through that Gerkin that's been put together and they make any adjustments or changes that need to be happening there. Of course, even though they're all wearing sombreros, they're also wearing their hat for their role. For example, the test might be saying things like thinking about boundary conditions that exist in the software. So, they might say "Well, what if this occurs, or have you thought about this?" And, sometimes, people haven't. So, what do we do? Well, we create another scenario or whatever it might be. Basically, there's some collaboration at work there. For his questions, if the project owner is there, we just have it. If the tester finds a defect, then we fix it.

In fact, I don't condone and my teams I work with do not have defect tracking tools. So, we follow a zero defect policy, which basically means any time in any circumstance that a defect is found, it's addressed immediately. It's kind of like the pipeline. You know, you've all heard that your build process, your build pipeline, if it stops, the team stops, and takes a look at it, and figures out what's wrong and fixes it. It becomes the highest priority. We treat defects the same

way. If a defect is found, we stop, we fix it, we address it. As a result, we don't have any defects in our backlogs, so we don't need a defect tracking tool. Lots of collaboration going on here. Eventually, the user story goes over to review. Whenever we say we're done, it is because a developer and tester have built this together, collaborated on a lot of the development, and testing activities, and have said "To the best of our abilities, we don't know of any defects." That's kind of the chain that's there.

So, how really does this work? What we're trying to do is instead of code it and then test it, we're trying to do those two things at exactly the same time. So, what happens? Well, the developer ... Usually on the teams that I work with, they're doing TDD, and they're almost always doing that with a pair. Sometimes, we're doing it in a mob where a whole team or half the team is all together in front of a large monitor or TV screen, or something like that. The tester is actually automating those acceptance tests that we saw earlier. They're building up those BDD scripts and those get checked in. Once they get checked in, the developer's job is to actually check those out, and to run them on the machine, and to work to actually make those tests pass. The other thing that's going on in this work is that often the tester is doing some exploratory testing as pieces of the application itself are completed or as the developer is wrapping up, or getting close to wrapping up. Since, sometimes, the tester and the developer get together and do some exploratory testing.

So, in this world, other types of testing are shared by everybody. Things like visual inspection of the app or UI as we're building it. Things like accessibility. These sort of things don't belong to the tester, they don't belong to the developer. They belong to both of them, or, actually the whole team. So, imagine that the developer finally has got that last acceptance test to pass, they've checked that code in, it's going down the pipeline, they might say to the tester "Hey, what's left on this story?" Because, he's not done. There is no done done. The tester might say "You know, I haven't gotten around to looking at this from an accessibility standpoint." The developer says "It's okay, I'll pick it up" and he'll fire up Jaws and play the screen back, or whatever it might be so the tester and developer are talking together to make sure that everything that needs to be tested is handled. The way that they do this is yes, they have to talk with each other. They kind of share this shared goal of preventing defects. So, that is the essence of what acceptance test driven development is for me.

The next design pattern we're going to look at is page objects. You may have heard about those before. So, this one is a coding thing. So, yes. I've got to get back on the keyboard here in just a second. So, we know for a fact that the application is going to change. In fact, most people in development, that's their job to bring about that change. So, the challenge is how do we make it so that whenever it changes, it's easy for us to maintain the code. Yes, we know that the tests will break, how is it to fix? I like to say that it's okay if a lot of tests break, but it's not okay if I have to go to a lot of places to change it.

Now, if you're following this ATDD practice that we talked about before, whenever we make a change, we actually start by modifying the test or adding a new test. We kind of drive from there. But, there are some cases where a change we weren't aware of and it comes in, and it breaks a lot of tests. What we need to do is learn from the development community, where they've come up with these really good design patterns that allow them to separate out things like model view controller and there are several others that are out there where we learn where each thing in our test suite, or in our automation code, has a place and one place only.

With that, I guess that's enough talking about it. Let me go out and break some code. Let me pop over to do that right now. Okay, so let's start off where we left off. As you can see, I've got pages created for the homepage, the details page, and the shopping cart page already. I didn't want to bore you with a lot of the details, but I did want you to see it. I'm going to start by creating a new page object for the checkout page. I'm going to include the page object gem here, and all I need to do at this point is just start to define the different elements that are on that page. So, I've got this text field. The text area, the address, and I've got another text field, where we have the email. Let's put that in. The select list, or the pay type dropdown, and the button that's at the bottom. Set it in place. Order ...

And, with that, I'm going to kind of jump over now and ... Let me make sure. Yeah, okay. I'm going to jump over and update my steps now. Some of these are going to be using the existing page objects. So, I'm going to start by just visiting the homepage. Let me get rid of all of this for this next one. You can see that I'm basically going to be calling methods on the page objects that are in turn going to do what's necessary to complete that task. I literally am extracting that away inside of the page object, so out here, I don't know what's happening, what I'm interacting with. I'm just simply calling methods in the page as hidden, everything away.

For the checkout, I've got several things to do. So, I'm going to use a slightly different form that I'm passing with lock. I'm going to go ahead and set all of the values that I need. Again, this looks like I'm just setting some properties. The page object itself has abstracted away the details of what's going on there. So, let's select something from the dropdown, and let's click that button. Okay, great. Now, here, I'm going to use a sense variable current page that this page object jump keeps track of. So, I'm going to basically see all the current page text. I think with that, I think I've got the first test converted over to using page objects. Let's run it just to be sure. Let's see what we come up with. And, it worked. Excellent.

Let's go back to the second test. I want to delete all of this and I want to copy this down here a little bit. I still have to handle setting one of these two blank. I'm going to use a little bit of Ruby. I'm going to actually send a message by doing that dynamically goading it out using whatever the field is and say what the field equals. So, if that's a name, it's going to call the name equals method,

et cetera. Now I'm setting it up to blank. For the error, I want to go over to my page object and put here the knowledge of how to find those errors. So, I'm going to again, declare that div and I'm going to call it error_explanation. Then, for the unordered list, I'm going to use a little bit different notation. I'm going to pass the block, and here, I'm going to say start by looking inside of that div, the error container element, that div. Inside of there, find the unordered list, and that's it. With that, I can now go back over here and I want to get rid of that. So, now I want to say on my checkout page, .errors, and that should return the text for me. It's going to run these tests. I've got to make sure that they're fine. And, that one's good. Let's run the address. Excellent.

I have one more to go and I'm checking out. I just basically need to get to the checkout page. So, we'll use this to do that. Okay. So, how do I want to go about this? Delete the old. I'm going to add a new method to my checkout page. It's called payment options. Here, I'm just going to return that array of text. So, I've got the select list, pay_type. So, I want to get the options from that and I want to collect the test from each of those. Now, I should just simply be able to get that value returned from the page object. Let's run it. And, as you could see, that test worked, as well. The page object really helps us hide the details, abstract away the implementation from the HTML.

Okay. Here we are back over in slide land again, away from the code, so. Yes. So, let's get on to the next design pattern. This is another coding design pattern. It's called Default Data. Let's talk about this a little bit. Our tests tend to need a lot of data. In this specific case, I'm talking about the data that we use to drag through the front-end. So, in other words, the data that we use to fill out forms or to make decisions around what things are selected, or whatever it might be. But, the fact is that, the majority of the data that we have to provide for any specific test doesn't matter. In other words, of all the data that we have to provide, it usually is a very small subset of that data that actually changes the outcome of our test. So, this is the design pattern that I first kind of discovered when I was working a large data warehousing project, where it was an ETL type project where we were reading from a really large Oracle data store, making some manipulation and then writing it ultimately into another Oracle data warehouse, as well.

When I first got there, the way that it was being tested was that the testers would run a half a million records through it or so over night, every night. They would come in the next morning and they would try to go out and say "Okay, I want to test this thing" and they would go out and run some queries to try to find a record that matched that. And, they'd find "Oh, look, I found one", and then they would kind of go over to the source to make sure that the thing that's supposed to happen actually did happen. It was all manual. It was incredibly slow. It was very laborious and it was very error prone. When I started, one of the first things I wanted to do was to get some automation around it and I quickly learned that the data source, or where we had to set up the data, was over 120 tables. The complexity was just enormous, but over time, I discovered that whenever we're talking about specific business rules triggering inside the

middle of this pipeline, each of those business rules actually only looked at a very small subset of the data.

So, in that case, the majority of the data that was in there, as long as it would flow through the system, it didn't really matter. So, we were able to create a little test harness in about a week or so that would basically blast data across all those tables. It had some reasonable defaults, but the tester was able to say "Take this subset, blast the reasonable defaults across. But, in this specific table, these three columns need to be this exact value and this other table, these two columns need to be this exact value. It worked pretty well. It just so happened that the next company that I went to work for after that, where I went to consult, I was working on a web app, and what I quickly learned was exactly the same thing. So much of the data that we had to enter didn't really matter.

Now, if we go back to the puppy app that we've been playing with so far, so far the tests that I have, it doesn't matter which puppy I select. The first test where we're going all the way through to see the thank you message, again, it doesn't matter what value I put in the name field, what value I put in the address, et cetera. If you think about the second and third test, the only thing that really matters there is that the field that I'm testing is empty. It doesn't matter what's in the other tests. So, this is the idea behind default data. I think it's time to go out and take a look at it. With that, I'm going to jump from slide land over to code land, and we're going to write some code for this again.

Okay. So, let's see how this default data works. So, we're starting where we left off again. As you can see, I don't have any of the default data to find so far, so I'm going to start by creating a new file that's going to pull my default data or at least the knowledge of it, and let's just call it checkout, because that's sort of what we're doing here. I'm going to start by just putting the same values that we've been using so far. I've declared checkout page, I'm setting my name, and my address, the email. Now, the pay type. That's all I want to do at this time. We'll come back to this in just a moment.

So, I want to jump over to my checkout page and I'm going to start by including the Gem data magic, which is going to do some magic for us. I'm now going to add a new method called checkout. Let's start just like this. I'm going to use some built-in methods from the page object Gem that know how to work with data magic where I'm basically going to say "Populate the page with data for checkout page." That will get all of that data and automatically populate it once I'm finished. I just need to click the place order button. With that, let's go over to our steps. Let's go back to the steps for that first test and let's clear it up, or clean it up a bit. Here's where the checkout is, I can get rid of all of this code, and just simply say "Checkout". That should take care of it. Let's run the test and see if I'm right. It looks like something is happening here. Yes, something is definitely happening. I'm just going to let it fail, so I can kind of take a look to see what's going on. Yes, it failed. I forgot to specify which data magic file I wanted to use. I'm going to put an annotation on this test to tell it to use that checkout file that we defined.

Now, it should work. Yes, okay. As you saw, it populated that page properly. I think we're good there. Let's go back to this next one. Here, as you remember, I've got to deal with setting them to an empty value. I'm going to go back to my checkout page and I'm going to pass some data that I want a default to empty and for data, I'm going to pass an additional parameter, which will actually merge that value with the data that it gets from data magic. Now, in my steps, I can simply say "Checkout", but now I'm going to tell it which values I want to override. It's going to be the value that it's to fill, and I want to set it just to an empty string. Now, let's try to run this test. Let's see. I made the same mistake again. Okay. That's fine. I definitely need to wake up here, but let's write this code. Actually, you know, every test here is going to need this, so I'm just going to put it at the feature level, and that will apply it to every single test that's here. Every test is going to use that same file. Now I can run it and I won't have to worry about remembering it next time.

Let's see what happens this time. Good. And, the address ... Good. Alright, it's all good. I've got one more to go. Although, I want to show you something. If the data truly is default, it shouldn't matter what values go into it. So, this data magic Gem has built into it the ability to say "Just randomize the data for me" and it has all these built-in methods to kind of deliver up things. For the pay type dropdown, I can't just say anything and have it randomize. I've got to have only the options that are in that dropdown. I'll just go ahead and mention also, that this gem is fully extendable, so you can very easily define your own types. I'm just going to put a little sleep here, so that when I run it, you can actually see what data it's putting in there, because I think it's kind of interesting to see. Let's run it. Look at that, huh? Nice. Let me run it again, just so you can see that. Each time, it's bringing wholly unique data. The internet is a little slow at the moment. Oh well. And, there you have it. That's really all there is to the default data. Let me go take this sleep out of here, so that I don't have that going forward. And, okay. So, there we have it.

Now, let's talk about the next pattern, and the next pattern is test data management. I think this is one of the largest things that our industry still needs to tackle right now. Let me give some context here. So many people have lots of tests that they run. They might kick them off overnight and then they come in the next day and look at all the ones that have failed, and ask themselves "Did this fail, because there's some problem with the application? Or, did they fail, because somebody messed with my data?" So many places, they put lots of data in shared environments where lots of teams are left for themselves to try to fend out to find the data that they need to use, and they tend to step on each other. So, test data management is all about getting that right. Test data management is all about every time I request some data from the backend, the application requests the data, that it comes up in exactly the form that I need it. So, if I need an account with exactly $39 in the account, when I request it, it comes to $39. Even if my test reduces that by withdrawing some money, and the same test runs a couple minutes later, when it comes back it needs to come back with $39. In other words, precision. I always need to get exactly what I want.

I believe this is something that we haven't solved well in the industry and I think that we need to do a much better job. I've got a couple of little design patterns I'm going to talk about. The first one is what I think is the best pattern where our test begins. As soon as the test begins, it actually goes out and creates the data that we need for that test, that test only. That test executes. As soon as that test finishes, it removes the test data for that test and continues. Another option that I've seen people use is where all of the data for all of the tests is inserted upfront. All the tests are executed and then at the end, all of them are removed. All the data is removed. Now, there are trade-offs between these two. The reason I say that the first one is better is because it tends to require fewer environments. The second one, whenever we're doing these large loads and these large cleanups, tend to step on it so that it's hard to have one environment where lots of lots of teams could be running their tests together.

Secondly, the first one, if there's a problem, it's much easier to try to trace it, because the data that the test needs is there with the test itself. The second option, if we build something out like that, and we have an issue, we have to usually go to another system or another place to try to know and understand what data was in there. This tends to be a whole lot easier to build something custom that fits in the first one. Now, what do we do if we have a case where we've got a backend that we have no access to whatsoever? There's a couple of classic examples of this. One of them is if we're making a call to third-party services. Let's say there's a company that provides that some service that our company uses, and we make a request to there, but clearly they're not going to allow us to modify or write data into their test environment. So, somehow, we're going to have to either accept the fact that we're not sure what we're going to get. We're not sure if their service is going to be up or down. And, last but not least, the data's gone completely volatile.

Another example that you see in some large companies is where there might be some backend that's owned by a different group and they're not going to allow you to access it. The traditional one is a mainframe group where they believe that their job is to kind of protect the company from you as a tester. So, they're not going to let you write data directly into it. Another use case might be where we've built a front-end, and the service that we need to call isn't quite available. The backend is not quite there, yet. In all these cases, using service virtualization tends to be the right approach. What these are, is these are just fake services, if you will, that stand in and deliver up the data that our test needs, whenever the backend is totally volatile.

Now, something that I find that is really, really, really important is to not repeat the data. What I mean by that is, do not use the same account repeatedly across all of your tests, because you have some tests that are going to make changes to that, and it just becomes quite difficult. What I find is that using different data across different tests tends to make things a little bit simpler whenever we run into situations where we have issues. Let's take another example that goes along with this. What happens if we have a test that has to start with an account in a specific shape, and that test makes some changes and leaves it in a second

shape? And, we have another test later that expects something in that second shape. Well, we shouldn't try to reuse that data. Again, the challenge is as we scale up our tests, we usually parallelize these significantly. So, we really lose control of the order in which these run. So, if we're using the same data, we're undoubtedly going to have multiple tests that try to use the same data at the same time, or the order is going to be in an order that we don't expect.

The next design pattern that I want to talk to you about is one that called wrap navigation. Think about it, all of our tests, a very significant portion of our tests have to navigate through pages. They have to start at some place and navigate to the point where we're actually beginning our tests. This is something that's very, very common. What I tend to not like to do is pepper this idea throughout my code. I like to kind of extract that into one place and have only one place where I have to go if that changes. In other words, if the path or what I have to do to get to the new page changes, I want to go to one place. And, it also goes a long way to simplify our coding cleaning it up. So, let's switch out of slide land and go over to code land. Let's take a look at what this looks like.

Okay. So, routes. Let's start by creating a new file and in this file, I'm going to define my routes which simply is a hash, requires an entry of default, because if you don't specify which routes you use, it's going to by default, use a route called default. Isn't that clever? All I'm doing now is just defining a series of class names and methods to invoke. It's going to have the knowledge to walk through that, to new up those classes and to cull the methods that are there. Let me put these in here. There's shopping cart and the last one is checkout. In this case, I'm only creating one for default. I can have as many routes in here as I want. If I wanted to provide a parameter for the method, I could do it simply like this. But, in our case, we don't need to do that so I'm just going to bypass that right now. Let's go to the step definitions. Let's start with the first test here. Basically, really, what I want to do is to navigate out to that checkout page. In fact, this duplicates the route. All I need to now say is "Navigate all", because it will go through that entire route.

And, one a cleanup, okay? Let's take a look at the next one. So, here, I want to navigate out to that checkout page, but when I get there, I need to set it to blank values. So, I'm going to say navigate to the checkout page. And, let me run that one. Great. And, the address will work, 'cause we've only changed the routes. This next one, again, what we're doing is we're just simply navigating to the checkout page. If I can type it right, let's do that. And, let's run that one. Now, since I'm not specifying what route to use, it's using the default route. But, what if I wanted to use a different route? I would simply say "Navigate to this page" using whatever I named the other route. I would just simply say it just like this. Lots of flexibility going into this and that is all there is.

Now, let's go to the next one. To be quite honest, what I'm about to show you, I think is one of the most important take aways for you in this talk, because it's one that just about everybody gets wrong. I see very few companies who get this right. It's all about making sure that we test the right thing with the right

tool. Everybody has seen these pyramids before, but I'm going to quickly go through and talk about mine. I see it like this. I see that unit tests are at the bottom. These things called integration tests are at the middle. And, what I call user tests are at the top, which are the automated user tests and also exploratory testing. I'm going into each of these in just a moment and talk about them, but let's start with why do we talk about these three different places? What is it that I'm talking about? Well, to be quite honest, I want to take every single test and push it as far down that pyramid as humanly possible. The vast majority of the application should be test in unit tests.

Let me give some real world examples here of what I mean. These come from clients that I've worked with in the past or some that I work with right now. So, I started working with a team and one of the requirements that they had was that whenever you navigate to this specific screen that has a lot of financial data and it's kind of laid out in tables. If one of the dollar values was negative, that dollar value should be showed with a red font and it should have a special formatting. So, the tester had written some automation that logs in, navigates out to this page using an account that they knew had negative values, and they checked to make sure that the font was red, and they checked to make sure that the special formatting was applied. Then, of course, they had a second test that did the same thing, except with an account that they knew had a positive value there, and they checked to make sure that it did not have that red font, and that it didn't have that special formatting. What did I do when I started working with them? Well, we deleted both of those tests, 'cause those tests were not valid tests to be running through the UI.

You see, it was java script that was applying the style that turned it red. And, it was java script that was doing the formatting for the display. So, those tests needed to be written as unit tests. Taken this way, what you find is that we have far more unit tests than anything else, and in fact, those tests that are at the top of the period, there actually ends up being very few. Those tend to cast more larger tests that we can't test in the other two layers. Things that are larger business rules. Things that are more about the core behavior of the system. So, it is so, so important to get this right. What I find is that teams where the developers and testers don't work close together, the testers tend to try to automate everything up at this top level, and what they end up with is thousands and thousands of brittle, brittle tests.

The client that I'm currently working with, when I started there, they had over 40,000 tests that ran through the UI. We're now down to about 6,000, because we've been deleting the ones that shouldn't be there. I can tell you when this process is finished, we'll probably be at about 2,000 tests. Maybe even under that. And, this is a pretty sizable app. Let's talk about each of these. Unit tests, here's just some example of what an app might look like. You've got your user interface, you've got your services, you've got some backend data store, or something like that. A unit test literally is taking one of these little classes or one of these little small pieces of code, and testing an incredibly granular piece of functionality. For example, this dollar value is formatted properly when it's

negative. The same thing would happen in the services layer. These are the tests that test the majority of your application.

Now, these little classes, or these little code snippets don't work in isolation and sometimes they collaborate with other classes to do something that's a little bit larger. Now, we're still not talking end to end. These are what I call the integration tests and it's kind of how two, three, or four of these sort of work together to perform a little bit larger action. I like to pull those tests together, as well. And, last but not least, there are more of our through the UI tests, the ones that run end to end. So, of all the tests that we've been writing today, there's actually none of them that should have been written through the UI. Yes, I've written them through the UI so that I had examples to show you some of these design patterns, but every one of those could have been written as a unit test and test that behavior.

So, what are the patterns that we've talked about today? We started by talking about specification by example. We talked about acceptance test driven development. We talked about page object and how that works. We saw some code for that. We talked about default data and we saw code of how that works, as well. Test data management, route navigation, and writing the right tests at the right level of the pyramid. All of these are things that I use at just about every team that I work with. And, above all, keep your automation code clean, and please, please, please do not automate too much.

Thank you very much. I hope you got something out of this talk. If you'd like to reach out to me, here's my email address and my Twitter handle.