**Adapter Design Pattern**

Introduction

Hi, this is Bryan Hansen, and in this module we're going to look at the adapter design pattern. The adapter pattern is a great pattern for connecting new code to legacy code without having to change the working contract that was produced from the legacy code originally.

Plug Adapter

Just to get it out the way, it seems that everybody wants to describe the adapter pattern as a concept of a plug adapter. We have a device that needs to plug in to an outlet, and we can use an adapter to make that connection. This is a fairly accurate description, but there are some twists to the adapter pattern that the typical plug example doesn't cover, and frankly it isn't a software example. One specific thing that I don't like about this example is that we usually only look for or need one adapter with the plug example, and with software we could have multiple adapters. So there's one variation there that you should keep in mind that the plug adapter doesn't really demonstrate very well.

Concepts

We already talked about the notion of a plug adapter, but let's dive into more detail about the concepts surrounding the adapter pattern. We would choose this pattern when we were wanting to have a client talk to an existing interface. This is usually the case when one portion of our system is a legacy app or module that we don't want to, or can't possibly change. It effectively is translating requests from the client the code that we are adapting to, basically a client to talking to a legacy app or an adaptee that we have used an adapter to talk to. Examples of this in the Java API are the collections API, specifically the usage of the arrays to lists conversion. Arrays were original, or you could classify as a legacy API, and lists were the newer part of the collections API introduced later. There are methods in the collections API to adapt arrays to lists. Another example in the Java API are the stream classes surrounding I/O. Almost all of the stream classes have adapters to work with other streams or readers.

Design Considerations

The adapter is very client centric. It is typically implemented to adapt or integrate a new client to legacy components. Oftentimes it is implemented to an interface, but it doesn't necessarily have to be. It can simply just be a new class. The adaptee can be the new portion of the code too, it typically isn't, but that is one portion of the this pattern that people often don't thing about or overlook. Looking at the UML, you can see we have a client with a specific method that wants to do something. The legacy API doesn't support it, or we don't want to modify the client to work with it in that manner, so we will add an adapter that will carry out that operation for us, and finally integrate with the client to do what we want to get completed in our legacy operation.

Example: Arrays.asList()

Here's an example of the Arrays.asList method that is an adapter to convert an array of something into a list from the collections API. There are a few things to note of this. First, it is an adapter because it is just adapting functionality rather than adding or decorating the class. That is a separate pattern that we will cover later called the decorator pattern if you want to add more functionality to your legacy API. The other thing to note is that in adapting, the Java API makes good use of generics and returns the list as the correct object type without us having to specify it. I will say that one thing that I don't like about this example, and others that I've seen is that it only shows one adapter in its use, we're just going to convert an array to a list, we're not converting other things to a list, so there's not multiple adapter types. Let's look at this example in live code.

Demo: Arrays.asList()

So here's a small demo of the adapter pattern in use through the Arrays.asList method. You can see here we have an array of integers that we are going to go ahead and pass into the Arrays.asList method that will then return us a list of integers. And we go ahead and print out the arrayOfInts and the listOfInts just to show you the different functionality that's going on there. We go ahead and run this code, and you can see it will return our java.lang.Integer address of our arrays down here, and then our list of integers here, so it's gone ahead and adapted this array into our list. So back behind we don't really see what's going on, but it's converted that for us, or adapted that integer array for us behind the scenes. Pretty simple example, the thing I don't like about this example as I mentioned earlier is this is one method, or one adapter of whatever array type we're passing in, so I don't feel like it's the best example. We're going to go ahead and create our own example that really demonstrates the effectiveness or flexibility of the adapter pattern in our code.

Exercise - Create Adapter

Now that we've seen an everyday example that exists in the Java API, let's go ahead and create a little more concrete example that really shows some of the strengths of the adapter pattern. We're going to go ahead and walk through some code that I've already built just to help set up the demo that we're going to run through of the adapter pattern, then we're going to go ahead and create an adapter, demonstrate how to not let this become a decorator, and then we're going to add another adapter in there just to really solidify the concepts that we've learned.

Demo: Adapter

For our example, we're going to go through and build a list of employees that we obtained from two different data sources, one of which would be a database and another, which is LDAP. You can see from our end users perspective, we're just getting a list from somewhere and printing that list out. So let's go ahead and open up our client and we begin to see from this example what problems we're going to face. So I have a list of employees that is just basically built from whatever sources we can compile it from, and we an Employee interface. This Employee interface is real simple, we are looking for getId, getFirstName, LastName, and Email address, and then we have an EmployeeDB object, which naturally implements this Employee interface, and it has getters for get id, get firstName, lastName, and email. Pretty straightforward. So if we look at our client and if run this right now, it'll run just fine. You can see it will print out an ID of 1234, First name: John, Last name: Wick, and an Email of john@wick.com. Now where the problem comes into play is when we want to do this same thing using our LDAP object. So we have an LDAP object over here, and you'll notice that the naming convention for LDAP for ID, first name, last name, and email address is different. Cn is our ID, surname would be our first name, or our last name rather, and givenName is our first name, and then mail would equate to our email address, but you'll notice we don't implement any interface at all. Now you're saying I've got this code right here, I could go ahead and change this, well, yes, but that's not in the nature of what this pattern is built around, and a lot of times we can't change this legacy code. Rather, let's go ahead and implement an adapter to do this. So let's go back to our client, and if I try to create this, this will just illustrate further that we can't do that, I can't create an employee LDAP and then assign it to an employee. Rather what we're going to do is go ahead and create a new instance of this and then use an adapter to go ahead and store that in our list. So for now I've gone ahead and created an EmployeeLdap instance and gone ahead and put in here chewie and Solo and Han and han@solo.com, and now we need to go ahead and create an adapter to add that to our list. So I'm going to go ahead and write a little bit of code here for employees.add, and it's going to look for an employee here, so I'm going to say new EmployeeAdapterLdap, and I'm going to make this take an employeeFromLdap instance as its constructor and save this. Now this class doesn't exist yet, this is what we're going to write, we're going to write this adapter. So let's go ahead and click on this and say Create class EmployeeAdapterLdap. And it's already trying to be smart for us here and it says oh, well, I'm going to go ahead and make the interface that you want to implement the employee and name everything for us correctly. So let's go ahead and click Finish here, and it has some stubbed out methods in here for us, which is everything that we need from the Employee interface, but now let's go ahead and start implementing some functionality here. So I'm going to say private EmployeeLdap, and we'll just call this instance, and go ahead and save that, and we want to make our constructor that comes in and brings our instance in for us, we'll say public EmployeeAdapterLdap, and we'll go ahead and pass in here EmployeeLdap, and we'll say instance. And then we can go ahead and set those together. So we can say this.instance is equal to instance. Pretty straightforward what we're doing here. We're going to create an instance and this is going to wrap this instance, or adapt it to whatever we're trying to do. Now all these getters that are stubbed out with TODOs here, we could just say return, and we'll say instance.getCn, because that is what our ID is from Ldap. And our first name, we can go in here and say return instance.getGivenName, and then for our LastName, return instance.getSurname, and for our Email we'll say return instance.getMail. Now let's go ahead and save this. Now we've built our adapter, this honestly is it. We pass in a private instance, we go ahead and wrap that and call the methods we want to tie in to that corresponding interface. Now to point out the key points of this pattern, EmployeeLdap did not implement the interface that we wanted to, the client doesn't need to know about the gory details of what went on there, it really wasn't that gory, we just created an adapter that wrapped it, and it went ahead and added pieces in here to map to the fields that we wanted to call based off what the fields were in the object. But let's go ahead and look at our client, it's added, it's happy, so we've added our employeeFromLdap, if we run this again we'll expect to see two calls down below, or two results from our list that gets returned, and sure enough, we see ID 1234, First name: John, Last name: Wick, Email: john@wick.com, but then we also see this com.Pluralsight.adapter.EmployeeAdapterLdap. Now you can argue that if I start adding fields in here this now becomes a decorator pattern. So if I wanted to come down here and say public String toString, and then go ahead and implement out my toString where I say, well your ID, let's say return, let's say ID just like the string below was doing for us, and say + instance.getCn, and then subsequently build out the rest of our toString just like this, if I run it again you'll see that it now is starting to build it out just like the previous example did from the database. Some people argue that that's turning this into a decorator and not an adapter, because it really is changing the toString, or what would be represented of that object historically. It's really somewhat of a semantic, but I guess if you wanted to be a purist you wouldn't add this toString method in here, but I believe in making it match the output of the other object, we're adapting this object to make it, I don't classify this as a decorator because it's returning the same interface. If our toString from our other object represents this string and ours is representing something different, modifying that toString or changing that toString in our adapter to make that represent the same thing still seems like a nice way of using an adapter to me.

Demo: Second Adapter

To help solidify the concepts of an adapter in our code, I wanted to go ahead and add another adapter to our project, so I went ahead and inside of our EmployeeClient added an EmployeeCSV object that we can pull information from, so it's just a comma-separated value string, we're passing in 567,Sherlock,Holmes and sherlock@holmes.com. Now a few things different in here is that the EmployeeCSV object has an integer for an id, the firstname and lastname are all lowercase, and emailAddress is CamelCase. I just pass in a string and tokenize it based off of the values passed in, really not important to what the object is doing or what the pattern is really doing, but I wanted to point out the naming convention has changed and the id object type has changed, as well as we're not implementing any interface in here. So if I go back to my EmployeeClient and save this, we're going to do the same thing that we did with the LDAP instance, and let's go ahead and create a new EmployeeAdapter that's going to going to wrap our instance of our object. And we'll go ahead and say Create class and it's going to already implement the Employee interface for us, which is what we want to have happen because that EmployeeClient is all expecting to just look at that basic interface. I'm going to start off this object by doing the same thing, by saying private EmployeeCSV instance, and then add a constructor for it, public EmployeeAdapterCSV, and pass in that instance. So we'll say EmployeeCSV instance, and then go ahead and set that instance equal to the other object, this.instance is equal to instance. Now that we have this all in here, we can go ahead and change a few things around, and this is going to be very similar to what we did with the LDAP instance, but I wanted to point out a few things. We can say return instance.getId. Now our ID from our CSV object is an integer, not a string, and that's what it's expecting, so this is one of those cases where we can change the object type that we're returning, and I'm just going to do a real poor man's string by adding a + "" on there to convert it to a string. But you get the idea that we are building this string instance to return to our client that's expecting a string rather than an integer ID. Also, if we go down here to FirstName and LastName, we can see quickly that it's expecting a CamelCase and our method was get firstname all lowercase, same as our lastname. We'll say return instance.getLastname and go ahead and save that. And then finally Email, I just pointed out in this that the instance we have is using getEmailAddress rather than Email, so naming conventions, object types, the different things that we want to adapt it to. Now if we go back to our EmployeeClient, you can see that it says everything is compiling fine. And finally we can go ahead and run it and see if this will adapt to our object, which it did. You can see we have our EmployeeAdapterCSV out there. One thing to note when using the pattern, typically we create an instance and it holds or wraps one instance every time we want to represent this. So if we want to adapt an employeeFromLdap to it, we go ahead and create a n EmployeeAdapter. If we want to adapt an employeeFromCSV to it, we create an instance of the EmployeeAdapterCSV, and it's a new instance every time because it is holding a reference to the instance that we want to adapt.

Pitfalls

What are some of the pitfalls of the adapter? Well, not a lot actually. Don't overcomplicate them. Typically an adapter would provide multiple types of adapters, that doesn't mean to say that if you only have one adapter that it isn't an example of the adapter pattern, but more to point out that you may or may not be using the pattern correctly. The adapter is also used to make things work together. If you are adding functionality to your legacy code through the adapter, then you should probably be considering the decorator or some other type of structural pattern.

Contrast to Other Patterns

To contrast the adapter pattern against another one, let's compare it with the bridge. The adapter makes things work after they were designed, basically dealing with legacy code. The adapter is almost always retrofitted to make unrelated classes work together. It is essentially created to provide a different interface to our legacy code than was originally intended. The bridge on the other hand was designed up front to let abstraction and implementation vary independently. It is built in advanced so that we can provide a layer of abstraction and let both systems be flexible while we are implementing and creating them. Both the bridge and the adapter are meant to adapt multiple disparate systems and work in concert with one another.

Summary

Let's briefly recap the things that we've learned while implementing the adapter pattern. It is a simple solution to a very descript problem, quite easy to implement, you saw how quickly we implemented two different adapter patterns inside of our code. It is typically used to just integrate with legacy code that we can't or don't want to change, and usually this method or pattern will contain multiple adapters. So one common problem I see with people implementing the adapter pattern is that they create one adapter and then walk away from it. Don't be shy to look at other uses that you can utilize and extend your application with multiple adapters to interface with those legacy APIs.