Introduction and Prerequisites
Version Check

Introduction

[Autogenerated] So you've just been in an interview and someone asked you to describe a design pattern. Or maybe you were just describing a problem to a co worker. And they said, It sounds like you were reinventing the will. They might have even said that. It sounds like you're describing the decorator or a proxy pattern. If this sounds like a situation that you've been in, then this courses for you. Our focus in this courses on presenting design patterns described in the Gang of Four in an example driven way while using Java to do so. Hi, I'm Brian Hansen and welcome to structural design patterns using java.

Why Learn Patterns?

[Autogenerated] you might ask yourself why Design patterns air important? I first learned patterns as a means of communicating a problem to another developer. You may very well already know how to solve a particular problem, and it might follow a structure of a pattern. But it's better to have a common vocabulary that you can explain to someone of what the problem is. Patterns are an abstract topic. It isn't something that you look at the concept and then have it memorized from there some patterns or more easily applied to a particular problem set than another. Revisiting pattern materials is always a great idea, whether it be this course or a book or something else, it is amazing how your perception of a pattern will change. After gaining more and more experience with it, you may come away with a much different understanding of how the pattern works after you've applied it. Ah, lot of people have used single tens or maybe a decorator, and the thing they have used that with that design patterns, they'll find out that that's all they know, and there's much more than just Singleton's or decorators or adapters out there

Pattern Classifications

[Autogenerated] the Gang of four has patterns broken out into three groups the groups Air Creation, ALS, structural and behavioral. This course is going to focus on patterns classified under the structural group or category. The other categories have been covered in separate courses, structural patterns, air focused on how you use or utilize objects. It could be for something like performance or re factoring or memory utilization, just to name a few concepts. Let's look at what we're going to cover as faras patterns are concerned in this structural group.

Which Patterns?

[Autogenerated] we're going to start off by looking at the adapter pattern, and then from there we're going to look at a bridge. The composite, which shouldn't be confused with composition, a decorator facade, flyweight and then finally, a proxy. Let's look at how we're going to learn this material.

How Do We Learn Them?

[Autogenerated] each pattern of this course will be in its own individual module and follow a structure of having an overview of what the pattern is. The concepts to consider when choosing the pattern. What the design implications of that pattern are a live example utilized from the job. A p I. A demo in which we coat our own pattern, the pitfalls associated with this pattern. And then we'll contrast it with another pattern that maybe kind of similar to what we're looking at, or something that really abstracts or contrast what we're doing with this particular pattern. And then we'll follow up with a brief summary.

Prerequisites

[Autogenerated] The prerequisites for this course are quite simple. I am doing all of my coding using Java seven or later, and I'm going to be running this from Spring STS, which is just a flavour of eclipse. Honestly, though, this should work in any I d you choose. Or you could even use the command line and compile if you want to go old school and do it that way.

Next

[Autogenerated] Now that we've seen what we're going to cover and how we're going to learn it, let's get started by diving into our first pattern. The first pattern we're going to look at is the adapter pattern, and it's a great one for just making some code work in a situation or a context in which it originally didn't have a fit for or an interface to tie into.

Adapter Design Pattern

Introduction

[Autogenerated] Hi, This is Brian 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

[Autogenerated] just to get it out of 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 into an outlet, and we can use an adapter to make that connection. This is a fairly accurate description, but there's 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

[Autogenerated] 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're wanting to have a client talk to an existing inter vase. 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 request from the client to the code that we're adapting to basically a client to talking to a legacy app or on adapt E that we have used an adapter to talk to. Examples of this in the job. A p i. R. The collections AP I specifically the usage of the arrays to lists conversion arrays were original, or you could classify as a legacy a p I and list where the newer part of the collections AP I introduced later their methods in the collections a p I to adapt a raise to lists. Another example in the job, a p i. R. The stream classes surrounding Io. Almost all of the stream classes have adapters to work with other streams or readers

Design Considerations

[Autogenerated] the adapter is very client centric. It is typically implemented to adapt or integrate a new client to legacy components. Often times it is implemented to an interface, but it doesn't necessarily have to be. It can simply just be a new class. The adapt E can be the new portion of the code to it typically isn't. But that is one portion of this pattern that people often don't think 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 ap I doesn't support it or we don't want to modify the client toe 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()

[Autogenerated] Here's an example of the arrays as lis method that is, an adapter to convert an array of something into a list from the collections, a p I. 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, a P I. The other thing to note is that in adapting the job AP, I 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 have 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 life code

Demo: Arrays.asList()

[Autogenerated] So here's a small demo of the adapter pattern and use through the arrays dot as lis method. You see here we have a array of intra jurors that we're going to go ahead and pass into the raised on as less method that will then return us a list of managers, and we go ahead and print out the array events and the list of managers 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 dot Laing imager address of our arrays down here and then our list of imagers here. So it's going 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 imager 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 gonna go ahead and create our own example that really demonstrates the effectiveness or flexibility of the adapter pattern in our code

Exercise - Create Adapter

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

Demo: Adapter

[Autogenerated] For our example, we're gonna go through and build a list of employees that we obtain from two different data sources, one of which would be a database and another which is held up. You 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 get to see what example from this example what problems we're going to face. I have a list of employees that is just basically built from whatever sources we can compile it from, and we have an employee interface. This employee interface is real simple. We were looking for Get I D get first name, last name and email address, and then we have an employee database object which naturally implements this employee interface. And it has getters. Forget I d get first name, last name and email. Pretty straightforward. So if you look at our client, if we run this right now, it'll run just fine. You can see it will print out an idea of 1234 1st name John Last name Wick and an email of Jonah quick dot com. Now where the problem comes into play is when we want to do this same thing using our l DAP objects. So we have an l DAP object over here, and you'll notice that the naming convention for Held up for I D first name, last name and email address is different. CN is our i d surname would be our first name and our last name rather and given name is our first name. And then Mel would equate to our email address, but you'll notice we don't implement any interface at all. Now you're saying I got this code right here. I could go 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 adapter to do this. 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 and employees held up in and sign it to an employee. Rather, what we're going to dio 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 employee, held up instance and gone ahead and put in here

Chewy and Solo and Han and han solo dot com. And now we need to go ahead and create an adapter to add that to our list. So gonna go ahead and write a little bit of code here for employees dot ad and it's gonna look for an employee here. So I'm gonna say new employees e adapter from filled out. And I'm gonna make this take an employee from L dap instance as its constructor and save this Now, this class doesn't exist yet. This is what we're gonna right. We're gonna write this adapter. So let's go ahead and click on this and say, Create class employees adapter held out, and it's already trying to be smart for us here. It says, Oh, well, I'm gonna go ahead and make the interface that you want to implement the employees and name everything for us correctly. So let's go 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 want to say private employees held up. We'll just call this instance and go ahead and save that. We want to make our constructor that comes in and brings our instance. And for us will say Public employees adapter held up and we'll go ahead and pass in here. Employees held out. Well, say instance, and then we can go ahead and set those two together. So you say So. This star 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 dio now. All these getters that air stubbed out with two DUIs here we could just say return and we'll say instance dot get CN because that is what our ideas from L dap and our first name Weaken go in here and say in return instance dot get given name and then for our last name. Return instance dot Get surname and for email will say return instance dot Get Mel. 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 method that we want to tie into that corresponding interface now to point out the key points of this pattern and play eld up did not implement the interface that we wanted. Teoh. The client doesn't need to know about the gory details of what went on there, and it really wasn't that gory. We just created an adapter that wrapped it and went ahead and added pieces in here to map to the fields that we wanted to call basis. What, based off what the fields were in the object. Let's go ahead and look at our client. It's added. It's happy. So we've added employees from eld up. If we run this again, we'll expect to see to two calls down below or two results from our list that gets returned. And sure enough, we see I d. 1234 1st name John Last name Wick. Email John Wick. But then we also see this calmed up through all site that adapter employees adapter L dap. Now you can argue that if I start adding fields in here, it this now becomes a decorator pattern. So if I wanted to come down here and say Public string to String and then go ahead and implement out my to string where I say, Well, your I D we return say I d just like the string below is doing for us and say plus instance dot get CNN and then subsequently build out the rest of our to string 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 to to the to string or what would be representative that object. Historically, it's really somewhat of a semantic, but if I guess if you wanted to be a purist, you wouldn't add this to string method in here. But I believe in making it match. The output of the other object were adapting this object to make it. I don't classify this as a decorator because it's returning the same interface if our to string from our other object represents this string and ours is representing something different modifying that to string or changing that to string 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

[Autogenerated] 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 employees, Client added, And an employee CSB object that we can pull information from. So it's just a comma separated value string were passing in 567 Sherlock Holmes and Sherlock at homes dot com. Now a few things different here is that the employees see SV object has an insecure for an I. D. First name and last name are all lower case and email addresses. Camel case. I just passing a string and token ISAT based off of the values passed in really not important to what the object is doing or what the patterns really doing. But I wanted to point out that naming convention has changed and the I D object type has changed as well as we're not implementing any interface in here. So if I go back to my employees client and save this one to do the same thing that we did with the L Dap instance, and let's go ahead and create a new employee adapter that's going to wrap our instance of our object. 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 employees client 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 employees see SV instance and then add a constructor for it. Public Employees adapter See SV and pass in that instance. So say employees see SV instance and then go ahead and set that instance equal to the other object. This stop instance is equal to instance, now that we have this all in here, we go ahead and change a few things around, and this is gonna be very similar to what we did with the old up instance. But I wanted to point out a few things we say a return instance dot get I d. Now our i d. From our in si SV object is an integer, not a string, and that's what it's suspecting. So this is one of those cases where we can change the object type that we're returning, and I'm just gonna do real poor man's string by adding a plus quote quote on their toe. 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 inter i D. Also, we go down here to first name and last name. We can see quickly that it's expecting a camel case, and we are. Method was get first name, all lower case say Mazar last name. I must return instance dot Get last name and go ahead and say that and then finally email. I just pointed out in this that the instance we have is using get email address rather than email. So naming conventions, object types, the different things that we want to adapt it to. Now, if you go back to our employees client, you see that it says everything's compiling fine. And finally, we'll go ahead and run it and see if this will adapt it to our object, which it did. You see, we have our employees adapter. See SV 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 for Impl, if we want to adapt and employees from eld up to it, we go ahead and create an employee adapter. If we wanna adapted employees from C S V to it, we create an instance of the employees adapter CSP, and it's a new instance every time cause it is holding a reference to the instance that we want to adapt.

Pitfalls

[Autogenerated] What are some of the pitfalls of the adapter? Well, not a lot, actually. Don't over complicate 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 to the adapter, then you should probably be considering the decorator or some other type of structural pattern.

Contrast to Other Patterns

[Autogenerated] 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 adapters, 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 friend to let abstraction and implementation very independently. It is built in advance so that we can provide a layer of a distraction 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

[Autogenerated] Let's briefly recap the things that we've learned while implementing the adapter pattern. It is a simple solution. Teoh A very descript problem. Quite easy to implement. You saw how quickly we have 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 AP eyes.

Bridge Design Pattern

Introduction

[Autogenerated] Hi. This is Brian Hansen. And in this module, we're going to look at the bridge design pattern. The bridge pattern is very similar to the adapter, with the main difference being that the bridge works with new code, whereas the adopter works with legacy code.

Concepts

[Autogenerated] the concepts surrounding why you would choose the bridge pattern or that it is meant to decouple abstraction and implementation. To do this, you utilize a few techniques, namely encapsulation, composition and inheritance. A key concept with the bridge pattern and why it is more than just inheritance is that changes in the abstraction won't affect the client. What is meant by this is that the client is unaware of the abstraction on the back end. This is important because this d couples the implementation from the contract or interface that the client is using. One of the key reasons for choosing this

pattern is that we know the details won't be right to begin with. This may sound a little strange at first, but the bridge allows for a level of indirection that we add into our application. If you aren't quite sure of what the end product of what you're building will be. The bridge is great for giving us flexibility without breaking things with change. Examples of this are drivers. We use drivers all the time and the bridges and the bridges in a lot of ways. Just a driver. A good example of this in the job a p i. R J D B C drivers. We have an interface that we work with and a driver that works with the underlying database.

Design Considerations

[Autogenerated] The design of the bridge is more complicated than the adapter. It will utilize interfaces and abstract classes. It also places an emphasis on composition over inheritance. But it is more than just composition. Your application is designed to expect change from both sides. Normally, the UML for this pattern would be on this slide as well, but it is too large and will be on the next diagram. The pieces of the diagram are an abstraction, implement or refined obstruction and a concrete implementer. Let's look at that diagram now.

UML Diagram

[Autogenerated] If you aren't very familiar with UML, don't not. The names of the objects in this diagram confuse you with the types of classes that they are. The obstruction in this case is an interface. This interface could be refined into our refined abstraction, and this is just a more specific implementation of that interface. From here we have our implementer, which is the hierarchy of our abstract. Classes are abstract. Classes will then be defined into a more concrete implementation, which will utilise composition of our various pieces of our bridge pattern to define those concrete implementations. There are a few more moving parts in this pattern, and we will look at more detail when we create our own pattern. But first, let's look at an example of this pattern in our everyday usage.

Example: JDBC

[Autogenerated] J. D. B C. Is an A P I for executing sequel statements. Classes that implement the interface R J D B C drivers and applications that rely on these drivers are abstractions that can work with any database for which a J BBC driver exists. The JBC Architecture de couples in abstraction from its implementation so that the to convey very independently, thus being an excellent example of a bridge pattern. Let's see this code in a live demo.

Demo: JDBC

[Autogenerated] in this demo, we're gonna go through and use the JBC AP I to go ahead and create a database table in a Derby database. So this examples pretty straightforward. In fact, there's a more detailed example of this in the creation all design patterns course under the single 10 that shows how to wrap all of this. So when they we only get one instance of that connection each time we want to utilize it. But in this example, we gonna go ahead and

register our driver, and then we're going to build our U R L, which actually builds our database for us. If you're familiar with Derby at all, it doesn't matter if you're not, then we can use that driver manager to get our connection. Now. The interesting thing about this is that our abstraction is that in this is the client. The client is abstracted away from the underlying details of what's going on, and the driver that we're using is also enabling us to use a different database with the same code or different code. With that same database. There's that bridge that goes between the two using the driver and the A P I that we have. So from there we can create our statement and then execute Our statement will so tell us that we've created a table. So let's go ahead and run this and I go ahead and execute this. You see that our table was created pretty straightforward example, but it's a good example of the J. D B C, a P I, and how it is a bridge to an underlying database. We could swap out our database with my SQL or sequel server Oracle. Any of those and her client could stay the same. Or our client can change without our underlying database having to worry about the details or whatever other configurations we need to weaken. Just utilize our driver to make that bridge between the two.

Exercise - Create Adapter

[Autogenerated] Now that we've seen a live demo of the bridge and use, let's go ahead and create our own gonna start off by looking at some code that deals with color and shape and then how to turn that into a color and shape bridge. I'm not a big fan of this example because it's not a real life example, in my opinion. But it does show and illustrate the exact problem we're trying to solve with the bridge in a very clear description. If we dive right into other problems, they're a little too complex where you kind of get lost in the details of what it is. So we're gonna go walk through the color and shape example and then implement that in some real world code. Once we've created our bridge, we're going to go ahead and implement another bridge just to help solidify that. So we're gonna look at color and shape done incorrectly color and shape using a bridge, create our own bridge and then cement those details in by adding another bridge onto that

Demo: Shape Without a Bridge

[Autogenerated] here is an example that illustrates the problem that we're trying to solve with the bridge pattern. So it's a simple, simple application. We have some shapes, a circle that we're gonna create. That's an instance of a blue circle, a square that is an instance of a red square. And then we're going to say apply color to this. Now, the problem with this is that it can't grow with us. It it has orthogonal problems. And so as we get in here, we can see we've got a shape class that is abstract, and this is what people think of when they're referring to all. I'm gonna use inheritance to solve this problem. Will Inheritance doesn't really solve this problem. Have a shape, have a circle that extends it. I have a square that extends that, and then we can start implementing our colored instances. So we have our Red square. We have our red circle. We have a blue circle on a blue square. The problem is, is in this Kotal run just fine. When we go to run our application, it'll run and it'll it'll apply our colors to us. You see, we have our blue color in our red color. But what happens when we want to say at a green square? Oh, well, now we got to go through and create a new class and say, Well, we want a green square and it's super class is going to be these square and we'll go

ahead and click Finish and it will go through and generate a method Forests will say, Oh, well, let's a player Colors will do system dot out dot print Lynn and say that we're going to be applying the green color. Okay, this isn't that hard, cause this is a very simple example. That's what I really don't like about this example is such a simple solution. But you see that for every color I want to add, I have to go add a green square and have to do the same thing that have to add a green circle. And now we have yellow. We're gonna add a yellow square in a yellow circle, and then we run into the bigger problem of what happens when we add another shape. Oh, so let's say we add rectangle. Well, now I got to go through an ad, a rectangle that's red, blue, green, yellow and it just keeps expanding and expanding from there. So just to show that this works, find the code Will will run just fine with us. Once we do a Green square run just fine for us, it was new Green Square and then weaken. Apply. That's we say, green square dot apply color, and I know it'll run just fine, you know, to run just fine. As we can see, all of our colors are blue color, red clover and green color, but it can't expand with us. It can't grow with us. So let's look at a different example about how we can solve this.

Demo: Shape With a Bridge

[Autogenerated] So here's a demo of that exact same shape problem. Using the bridge pattern you can see right off the bat that one major change here is our client has changed. We now have colors extracted into their own interface and class hierarchy and shapes air still implemented the same way. But now we've utilized composition. Let's talk about how that all works, so our shape object is still in abstract class. It still has an abstract void method for applying color, but now we take a color in the constructor to utilize through composition. Color is its own interface now, and we have the apply color method defined in the interface. So shape has apply. Color and color has the color that it's going to apply. It looks a little redundant there, but it's not now. As we create colors, we can look at our blue color. We say that we're applying the blue color and we look at one of our shapes such as our square. We can see that it is utilizing that composition to go ahead and apply the color. So let's run this. Make sure that it works. I want to do run as Java application, and you can see that we apply our blue color in our red color. But now let's show by adding another color or a shape to this. How it's not the issue of a North Ogle problem like we had before. So let's go ahead and add another color Here will say New Class and we want to do green and our interface is color. We'll go ahead and click finish and now it has our color for us here. So we'll say system dot out up print Lynn just for our basic example. And we're going to apply the green color. So we're playing our green color and save this. Everything works. Here we go back to our sample, and now if I want a green circle, we already have. Our are shaping our square here. Let's go ahead and create a new circle. Let's say shape and we'll dio a green circle equals new circle and I could just pass in green. Now we have to create an instance of our green here that hasn't been defined yet, which is easy enough to do. It will say color green equals new green and save that now in come down here and just to execute it. Make sure that runs right, will say green circle thought, apply color and run in. Now you can see that we have our blue, red and green colors being applied. But notice one thing we did not do in here. We did not change our circle object at all. It was abstracted out from the changes of the colors on the back end. Same thing. If I want to do a an instance of a green square now, I can also say shape Green Square equals new square. I can now pass in that same green color instance to it and apply that color down here. Green square, a play color and save it. Now, if

we run this, it'll also work as well. So we were able to now add changes at colors. We could add another shape to to really illustrate it. But you get the idea. The bridge is the way we're utilizing that composition and abstracting out the properties on this. Now, why I don't like this example is because typically and I'm not saying that there's not ever an instance, but typically we don't have a hierarchy of colors, colors, arm or oven attributes. But I do believe that this illustrates the problem very well of what the bridge is trying to solve, that I have something over on one side, this shape and something over on this other side. That's this color. And I want those two to be independent of one another of changes, and you saw that we were able to add a shape or at a color without having to go through and changed the other half of our application. So we're getting changes from both sides that can act independently of one another. And that's why I really like the bridge pattern when we're dealing with something that we're not quite aware of what our end details are going to be.

Demo: Movie Printer Bridge

[Autogenerated] this bridge pattern demo is a little bit larger than some of the other patterns that we have gone through. So I'm gonna go ahead and pre code some things to show you what we need to to create the pattern but not bore you with the details of watching me type. So you see inside of here I have a movie object that's already created and it's populated, and it's a simple Poggio. There's nothing about it doesn't inherit from anything or implement anything. Just has a classification runtime title in the year and here. Basic movie attributes. There's another object in here called detail, and that detail is just simply a name value pair. We have a label and a value associated with it that we just used to pass information between the bridge pieces that were that were going to build. Now, inside of the demo. There's some code that I have commented out, and this is to just help us illustrate what we're going to be building. So let's start off by creating our interface, which is new interface. Four matter now, this four matter is, representative represents the same thing as we were doing with the color and our previous demos. So this has a very simple method inside of here. We have string format, and it takes a string for the header and a list of the detail objects that we just covered. So we got her details and then that saved this and import our job, You till list. And now we have our four matter cloth interface. This this interface, like I said, is very similar to what we had with the color in her face. Now we're gonna go ahead and create our printer class, which is an abstract class. So we want to go ahead and right click and say new class. And this is the printer and it is abstract because any of our specific printers air going to extend this or implement the various aspects of this, they're gonna be our concrete implementations. So now we have a couple of methods we need in here, and I'm gonna type these ones out because this really is the heart of the bridge pattern. The rest of stuff I'm going to just copy and paste in, but we're going to have a method inside of here that is our print methods so se string print, And this is going to take the four matter instance in. Now, remember, this is our composition. So we have our instance of our four matter, and this returns four matter for matter dot format and it's gonna pass in our get header. And this method isn't defined yet and are get details. You could quickly see how you could add stuff for getting a footer or those various aspects inside of here for matter. Now, let's go ahead and create those too abstract methods inside of here. So I wanted to abstract protected list detail get details and this is what we're going to call and the implementation class to go ahead and build what it is we want to put inside of this body. And

then we need one more for the get headers. We're gonna do abstract protected string Get header. Now we have all the pieces in place for our bridge. We have our four matter, which is represented. Our color shape is the 1/2 for bridge that we're gonna acquire through composition. And then we have our printer that's actually going to do the work and combine those two together, but just like how he had with their colors, where we had a red square, a blue circle. Now we can have printers and for matters, stand independently of one another so we could have a Web printer or, natch, HTML printer as well as a print printer, a more concrete strings. Whatever we want to do, we can build upon those to do whatever we want and have these to change independently of one another. Now that we have those pieces put in place, let's build our print for matter and our movie printer. So we're gonna come over here and create a new class. We want to start off with the movie printer, and it is going to extend printer Click Finish, and you can see it's put our to are. Two instances are two methods in here already from our abstract class that we inherited from printer. So let's go ahead and store a movie of an instance of the movie in here. So I want to say private movie movie and we're going to pass this instance. It is through composition. We're going to pass this in and say public movie printer and take that instance of our movie in here. I'm typing this out on purpose because this is a key point. This is where the composition comes into place. So there stop movie equals movie. Now the details of getting the details and getting the header. I'm gonna go ahead and pace those in from some sample code that I have earlier. Let's grab the details there and go into our movie printer and replace these details here with that, and you can pause it and look and see what I'm doing. All I'm doing is just adding details, objects and and creating those name value pairs is we go and then I'm gonna do the same thing with Header and the headers. Really straightforward. I could have just type that, but it will work just fine. It's a movie printer and replace our header. Say that and now we have our movie printer built. Now we need to do the same thing with our four matter. On the other side's. We have 1/2 of our bridge done. Let's do the other half of the bridge now, which is our print for matter. They're gonna come over here and right click and say new class and do print four matter, and it's going to implement the four matter, not extend. The super class can implement the interface. That's the four matter interface. Quick finish. And now there's a basic method in here and that is just building our format. And all this is going to do is go through and basically print to our screen, just like our system out print Lynn was applying before to our screen. This is gonna build the string for us that we're going to return to our client. So already have that done in here is well, which is just a basic string builder. And you can see what we've got going on here. Paste this out. I go through, create an instance of a string builder and start a pending the header and the body details to what I cycle through each one of the details, build up to the string and then return that builder dot to string to the claim calling code were actually done. That is the bridge pattern. Now let's go back to our demo and start uncommon Amazing CR print four matter and our movie printer noticed that movie printer takes an instance of the movie and to store it. And then we can take and call movie, printer dot print and pass in our print four matter. So let's look at our movie printer, and we can see it has our It extends our printer method. And if you look our printer class, if you look in our printer class, there is our print method that takes an instance of the four matter notice. It doesn't know anything about the four matter that's being passed into it just the contract through the interface. Now, if we come back over here, we can see that movie printed up. Print is going to return, are printed material, and we will be able Teoh. Print that out to our screen. Let's go ahead and run this now and you'll see that it goes through and prints out action title John Wick and Year 2014 runtime. Two hours and 15 minutes, you'll notice that this knows nothing about the movie printer object itself, and the movie printer

object knows nothing of the print. Four matter. All of that stuff can change independently. Now this is a maybe a little bit more complex from something that you could just cycle through and grab those values out. But think about that in terms of grabbing a Web printer. So I want Teoh. We're gonna you know, in a business sense, I I want to print a catalogue which this is the example that we just did. But I also want to make these objects available to our website. So I want to create and html print four matter instead of just the print for matter that we have. That's actually what we're going to do in this next demo just to help solidify these concepts.

Demo: Movie Printer HTML Bridge

[Autogenerated] Now that we've seen a demo of the bridge pattern and how it worked, let's go ahead and implement another instance of our four matter to do an HTML four matter. You see, I've stubbed out what we want for the methods down here to call inside of our demo class. Let's go ahead and right, click on bridge and say New and we want to do a class. It's an HTML four matter, and it's going to implement the four matter interface Click finish, and you'll see that it just goes ahead and, like it did with our print for matter, create a basic method signature for us knowing. Save this, Let's open up. I have already created the body for this previously because it's just mundane code for me to type out and paste this in here, and you can see that it's very much like our print for matter. I've gone ahead and created an instance of a string builder and then Donna build out a pan and you can see that I've added on here the headers and different columns and then got into the body of it. And we you know, the sky's the limit with what you want to do here. But what I want to point out is all I had to do to add another formatting type is create one instance of this class, and now I can utilize it anywhere. Go ahead and say this. If I go back to my bridge demo, I now have my HTML four matter and notice that I didn't have to create a new instance of movie printer. I'm using the same movie printer that we had before and just passing in the different four matter and getting that material back. So now I can save this and run this and you'll see down below. Here that we have are printed format as well as our HTML format being printed out. So all I had to do to add another formatting type was go ahead and create an instance of the HTML four matter and passed that end to my movie printer, and I'm done now. I'll let you go ahead on your own and create a different type of printer. Maybe we have another media object of book, so you would just create an instance of the movie and change that you know to utilizes a book and then create a book printer and go through and build those details that you want to. Maybe you'd have page count. Maybe you would have the Dewey decimal number or whatever other features you want. I ESPN number those types of things that you would build into those that details object. You don't have to use those that detail object The way that I did hear this little class, it was just a convenience way for me to pass things through generically and and share that information from those objects from one to another. You can utilize that every want, but all you have to do to create the other half The bridge, like I said, was created a different type of object from movie, maybe book and then a book printer and our print for matter and our HTML four matter would already work. You could just grab those values and pass him in, and you're good to go

Pitfalls

[Autogenerated] what are some of the pitfalls of a bridge? It does increase complexity. Also, as we saw with the color and shape example, you need to look at the code and see what makes sense to abstract out. It can be conceptually difficult to plan. Your code needs to be fairly thought out, and it might not lend itself into an agile codas. You go scenario. It is definitely more than just good. Oh, principles. One comment that has often made about design patterns is that I use sound oo principles and I don't need to know patterns. Well, this is more than just inheritance and abstraction, and overall, it could just be a little confusing as to what goes where in your code.

Contrast to Other Patterns

[Autogenerated] to contrast the bridge pattern with another one. Let's go ahead and look at the adapter pattern. The bridge pattern is designed upfront. It's something we consciously go into our application with the mindset that we want to dio the abstraction and the implementation convey areas we saw with the demo that we did. We have our printer object in R four matter object, and we can change either side of those without breaking the other. It is definitely built in advance, just like it's designed up front. This is something we consciously build up front as well, and it is a little bit more complex. It adds a layer of complexity, which isn't necessarily a bad thing, but it is mawr complex than what we might just do out of the gate on our own. The adapter, on the other hand, works after the code is designed. It is intended for legacy applications. So it's something that we we try to tie in legacy code with a new application, and in doing so, it's typically retrofitted in. It's something that were working with, and we try and piece into our application later and really the ultimate design or point that we're trying to do with an adapter is just get a different interface for existing or legacy code. We're just trying to make this tie into something else that already exists and nothing more, nothing less. We're not trying to add any more functionality. Where with the bridge. We're trying to break that functionality apart so that we can make those change independently of one another.

Summary

[Autogenerated] to wrap up the bridge. Let's just quickly recap what we've covered. The bridge is designed for uncertainty. It can be complex and often times adds a layer of complexity to our application that we weren't planning on. It provides a level of flexibility, so we will take that complex city because it gives us flexibility. So it helps us in designing for the uncertainty of what we might do inside of our application and then to last, they just point out it is much more than composition. We do utilize composition. We also utilize inheritance through abstractions and interfaces Inside of our coats were really taking a lot of the principles of good oh design and wrapping them into this one pattern.

Composite Pattern

Introduction

[Autogenerated] Hi, This is Brian Hansen, and in this module, we're going to look at the composite design pattern. The composite pattern is a hierarchical, tight pattern that deals with tree structures of information.

Concepts

[Autogenerated] the concept surrounding why you would choose the composite pattern are that it is meant to treat components the same, whether it is part of your structure or the whole structure itself. This is done by configuring your objects into tree structures. Once your data is built this way, you can treat individual objects the same as a composite object and treating objects the same. We can apply operations or functions on both the individual and the composite and expect them to work the same way. Examples of this in the job a P I. R. Java, a WT component Almost everything from the Java Ada beauty library is built this way as well as JSF widgets. Now JSF isn't part of the core AP I, but almost everything in Java server faces is built around the same structure. Another useful example is rest ful Web services. In fact, the way we structure gets are almost always built with the composite structure in mind. So if you've never used a rest full service, don't worry about it. But there is a couple, of course, is out here on portal sites, specifically one on wrestle services using Jersey that talks about how we navigate and re cursed down through the directory structure. Using this composite model

Design Considerations

[Autogenerated] the design of the composite is that it is tree structured. The root of the tree starts with a component. Components are one of two things either a leaf or a composite of objects. The difference is that a leaf just has operations and a composite has all of the same operations available, but also knows about its child components. The pieces of the UML diagram are a component, a leaf and a composite. Let's look at that UML diagram now.

UML Diagram

[Autogenerated] This is the UML diagram for the composite pattern. The component class is the abstraction for all components, including composite ones. It declares the interface for objects in the composition. The component can also define methods for accessing the parent, but it isn't necessary to still use this pattern correctly. The leaf represents the leaf objects or nodes, in the composition. It should be noted that it also implements all of the components methods. The composite, though, represents a composite component or a component that has Children and implements methods to manipulate those Children. It also implements all of the components methods as well, but typically delegates the functionality to its Children. So a leaf and a composite have thesafeside functions in sight of them. A

composite just knows about what's Children, which Children it has and what it could do with those Children.

Example: Map

[Autogenerated] the map and other collections are an interesting implementation of the composite pattern. Much of the collections AP I have the option to add all or in this case, staying with the naming convention of a map. Put all of the attributes of one map into another. The key distinction here is that one element of a map is treated just the same as the entire map itself. Let's look at this in a live coating example now.

Demo: Map

[Autogenerated] here is theme app. Example that we were just previously describing in this example we're going through and building a list of person attributes and a list of group attributes and then combining them all in one to represent our security attributes using the put all method, which treats each map as a composite of the previous ones. What treats the individual object the same way? And if we run this application, you can see that it has eventually combined all of our rolls into one security map. Now I've done this. This is actual code that I have pulled from an application that I've written recently for a client that was representing their person group and final security roles inside their application. And although says this is a much simplified version of that, this is exactly the same type of thing that we did and a very good example of how the composite pattern can be used in other things rather than just map hierarchy structures of men. Ewing Systems, which is typically where you see the composite pattern being used

Exercise - Create Composite

[Autogenerated] Now that we've seen in everyday example of the composite pattern, let's go ahead and create our own. We're going to start off by creating a menu, menu item and menu component to represent our composite leaf and component of the UML diagram. There were going to create our composite pattern itself. We're finally going to wrap up with showing how you do the features not supported in all levels of the hierarchy because you don't have to implement all of the methods to use the composite as we discussed in the design criteria for implementing the composite pattern.

Demo: Composite Menu

[Autogenerated] to show this example. I have created a couple of stubbed out classes just to help us get things rolling. And first, let's start with the main method and say this main method. I just have a basic menu structure that we're building that has a main menu, and there's a safety menu item that is its own menu underneath that in a claims menu and then a personal claims menu and all that builds into a much larger composite menu. So we have a menu with sub menus and nose have individual items underneath them. So to facilitate this, I've started off with a menu component class. The menu component class is pretty

simple, has a name and the URL and a ray list of menu components, and you just see the basic features we have in here a getter for the name I get for the girl, and I've put an abstract to string method in here. We're gonna add a little bit more meat to this here in a minute, but that's the basic of this class right now. The menu object, which is the composite of our UML diagram, goes through and this will have stuff referring to the Children and how you can add and remove things to it as well as it has a name in the U. R L. A cell. So if you think about a fly out menu or a drop down menu that has a hierarchical structure, that's what we're building here. The other piece that will do this is we're going to also add a two string method to print out this menu structure just for display purposes, but also see about which pieces should know about the Children, which shouldn't end there. The last piece that we're going to implement is the menu item menu items. Not gonna change a whole bunch from this. It just has a name in the URL. But we're going to also add some to string method stuff in here to just help round out the object and show all the pieces. They work together. So let's get started with the menu component. As we go through this, you'll see these red squiggle ease underneath our objects. Here. Go away are red access, so start with the menu component. Now the menu component itself is quite simple, but it is the root or the start of our hierarchy. So what we want to do is add functionality to this that's going to be consumed by the Children of this object. Go through and I have some sample code already put in here. I'm gonna add this print method that I've already pre made and just paste this in the bottom of this object. Now, the print method here goes through and creates an instance of a string builder, which, by the way, is another design pattern that you can look at in the creation. All section of my other course talks about what the builder pattern is and how it works. But, ah, the builder pattern here. We're gonna go through and get the name of this ST of this menu item and upend the colon to it. And then the your l and return that object out so we can just see the menu hierarchy get built as we go Here we have our menu here. Now let's go ahead and move onto the menu class. We have our menu component. Let's move on to the menu class itself. Now we need to do two things here. We want to add stuff to navigate the Children. So on Adam even move method as well as we want to build out the two string in here Now the add and remove is quite simple. Since we are implementing an array list for our menu components, we're just going to put some functionality to navigate those. I like to do something a little bit different here. As we add things and remove them. I like to return the actual object that we're dealing with. You don't always have to do this, but it's not any more costly for you to do it, and I find that it makes it easier to make changes. Did you go down the road? If you want to know which object was removed or which object was added, you just returned their reference to it. And so I've got a add method in here that's going to just take those menu components from our parent class and add them to it. And then it returns that object at the end, and then I have the remove method that does pretty much the same thing now, for right now, I have the At Override commented out, and I'll explain that later. It's because of you don't have to implement these features for this to be a usable class, but we'll talk about why I've got those commented out here later. Now I still have a compilation Aaron here, and that's because I don't have the two string implemented yet that is required from our menu component class. So we have this to string method here. Our menu object is looking for its Let's go ahead and put a two string in there as well, and this one is fairly complex. We're gonna walk through what happens here, so it's open up the menu, object here and paste it in. Now the two string and hair has a couple of things going on, just like in the menu component. We have a stream builder. We also inside of our to string here have a string builder, but we call the print method from menu component. So this print method here is referring to the one inside of our menu component, and then we iterated through all of our

Children. Now that is one of the features of this pattern. As a composite, we know about our Children now. Each child objects structure can be handled the same way as its own leaf. And so we just hand off it. Do what it needs to and a pen that back, and you'll notice we're doing exactly that. When we get our next menu component, we go ahead and just a pen, that menu component to string to us and return it back. So we re cursed down through the entire tree structure and then walk back up. And that's what this method is effectively doing here. So we use this builder pattern and then navigate through our tree structure and finally return out. Now let's look at our menu item to see what we have left to do here. This one's really simple. The only thing we haven't done here is override the two strings. We're gonna grab that and paste that in and save that now our application is full functioning. Now let's recap the pieces that we did. We went ahead and added our print method inside of here. We went through an added are add and remove for our child or are composite structure. Here's we know about our Children and navigated down recursive Lee through our to string method by calling the menu component dot to string and having it walked down through anything and it's Children. And then our menu item. We added, A two string is out of this and this is just calling the print function up in the tree structure of our menu component, and the last thing we're going to do now is run our menu demo. Now you can see from our menu demo. We have our main menu getting build our safety menus, claims some menu, all of that stuff. We run this, we'll see that it prints out our main menu safety menu claims and personal claims menu as we go through and build our structure. We just called thes system out print Lynn on our main menu, and it navigated through all the structures we went through here. Now, if you want to take this example of step further, you could change that to string method to go ahead and print out the entire path down to it, so slash main slash safety slash claims or slash main slash claims slash personal claims to go through that entire tree structure of our objects. But very simple. You can see that we are treating the Children the same as the object itself, and we get a very complex problems solved quite simplistically. And as far as our clients concerned, they just Adam and treat him just the same and it handles it pretty elegantly.

Demo: Unsupported Operations Exception

[Autogenerated] now we're not done here just yet. This pattern actually works exactly how it's intended. And as I mentioned earlier in our design section, you can implement this this way and it's perfectly legitimate. It will work just fine. But remember in our menu class, we had these at Overrides commented out, and I said that we would talk about them later. Well, that is because we can add in features to our menu component and paste those throughout our hierarchical structure. But we don't have to to use this pattern. Let me explain what I mean by this. So inside of our menu component, we can go ahead and add these two methods in here, and these two methods are what establish our contract for any other type of menu options that we have. So right now we only have two menu options. We have a menu and a menu item. What if we had a complex menu item or a graphical menu item? You see, as we start adding different types of menu item here, there are some features that we may want to implement by establishing this contract in our parent class and notice I did not make abstract because we're overriding it as we go through our child objects. We can now implement these features where we want, but we can throw a nun supported operation exception other points in our hierarchal structure if we want to, we don't have to, but it is a feature, and it does round out this pattern nicely. And this example, it really isn't going to gain us a whole lot because we aren't doing multiple menu item types. But if you wanted to implement another

one and not necessarily implement the remove feature, you could and then that feature would throw an unsupported feature exception. And I'm supported Operation Exception feature not implemented this level so you can establish that contract a little better. Let's just run this again to make sure that everything works all right, which it does, and it works as we expect it to

Pitfalls

[Autogenerated] what are some of the pitfalls of a composite pattern? It can overly simplify a system. This may seem like a strange drawback at first, but in building the hierarchy the way that we want to do it, it could make it difficult to restrict what we want to add to it. Everything eventually has treated the same, and that is the intent of the pattern. But you end up relying on runtime checks to see if objects being added can in fact be added instead of compile time. Safety implementation can also be costly of dealing with a very large composite or, if implemented, correct, incorrectly. This doesn't have to always be the case. But if child objects air held in a collection and each object itself contains a collection, it's size. Congrats fairly quickly. Typically, composites aren't that big, though, and I don't find this being a real practical issue.

Contrast to Other Patterns

[Autogenerated] to contrast the composite pattern. Let's compare it with the decorator. The composite is a tree structure. Its intent to make a leaf and a composite have the same interface to the client. It provides a unity between objects. The decorator, on the other hand, contains another entity. Now this may sound like a composite, but it really just composition composition is just an object containing another one. It differs in that the decorator modifies the behaviour of the contained entity. This is usually adding functionality to an entity that it didn't originally have. It decorates the underlying object but doesn't necessarily change it.

Summary

[Autogenerated] let's do a quick recap of the composite pattern. The composite pattern generalizes a hierarchical structure. As we saw on our examples. We built a menu structure that had menu items such as leafs and other composites, and it helped us navigate that structure in an easier fashion. It can simplify things too much. As we learned in the Pitfalls section, you can make things too overly simplistic, which can later make it harder to restrict what's getting added to a menu. And you can sometimes have to rely on runtime checks. It definitely makes things easier for the client. In our example, we looked at the client, didn't it? Didn't care if we had a menu, a sub menu, a menu item. It just handled it all gracefully the same way. Likewise, with our our collections example, we went through and could add individual items or an entire collection, and it gracefully handled both as well. The last thing that we should talk about is the composite does not equal composition difference. The composite pattern is dealing with that hierarchical structure where composition is just one object containing another, and I bring this up because the next pattern that we're going to look at is the decorator pattern, and the decorator pattern utilizes composition, but it is not a composite structure.

Decorator Pattern

Introduction

[Autogenerated] Hi, This is Brian Hansen, and in this module, we're going to look at the decorator pattern. The decorator pattern is a hierarchical type pattern that builds functionality at each level while using composition from similar data types.

Concepts

[Autogenerated] the concepts surrounding why you would choose the decorator pattern are when you want to wrap another object. To add functionality to it, you can add behavior to an object without affecting other parts of the hair key if you don't want to. It is more than just simple inheritance, though you are controlling which pieces complement your object, not necessarily trying to override it. Like with inheritance. The decorator also follows the single responsibility principle, which states that every class should have responsibility over a single part of the functionality provided by the software. And that responsibility should be entirely encapsulated by the class. Basically, this means that it should do one thing and do it well. You can compose behavior dynamically by using one of the sub classes that decorate your object. This candidly makes it feel a little bit like a creation all pattern, but it is adding behavior through creations, so it is in fact still a structural design pattern. Examples of this in the a p i. R. The Java Io input stream classes, and these are a great example, often confusing to people as to why it does things the way that it does But hopefully this tutorial and walking through an example of it will clear up why they built it the way that they did the job. You till collections AP. I also has a checklist method, but it's not really clear is to why that is a decorator. Patterns were not going to use that for our example. And also, I should note that almost all you I components in the A, B, T and swing AP eyes are implemented following a decorator pattern.

Design Considerations

[Autogenerated] the design of the decorator is that it is inheritance based. Often the confusing, confusing part of a decorator is that it is more than just inheritance. It utilizes composition and inheritance. To achieve this, there is a common component, but functionality is added in the sub components. It is also an alternative to sub classing because it adheres to the single responsibility principle that we talked about in the concept section where this class will just do one thing. Typically, when you're subclass, it is to completely rewrite or extend the parent class. The last piece is that the constructor requires an instance of the component from the hierarchy, which enables it to build upon that and use composition rather than inheritance, to override which individual fields that it wants to. The pieces of the you are of the UML diagram are a component concrete component decorator and a concrete decorator. Let's go ahead and look at that example now to see what the UML diagram looks like

UML Diagram

[Autogenerated] here is the UML diagram for this decorator pattern. The component class is typically an interface, although it can be an abstract class that has a concrete instance represented here in the concrete component. In the of this diagram, the concrete component is what we're going to eventually decorate, and the decorator is Thebe, base decorator or rapper that we will extend and create other decorators from now. It should be of note that both the concrete component and the decorator extend the components so that they could be treated the same from here. We can then create multiple concrete decorators to decorate our object and provide functionality as we develop.

Example: File

[Autogenerated] here is an everyday example, using the input stream class from java dot io that utilizes decorators to build out its functionality. In this example, we're gonna go ahead and create an instance of a file and write that file out to the output stream using an output stream, which is a base decorator, we can add on to its functionality using a file output stream and a data output stream. As you can see here, these three are all chained together, building or necessarily, in this case, decorating functionality from one stream to the next and a pending its functionality on Let's go her head and look at a live example of this code.

Demo: File

[Autogenerated] as we mentioned earlier in this example, we're going to create an instance of a file and write that out to the output stream. Here we've created a instance of a file output stream and assign that to an output stream. In this example, our output stream would be our base decorator, and our file output stream would be one of the instances of a concrete decorator. We then turn around and passed that file output stream into a data output stream so that we can write physical data out to the system. Pretty straightforward. But you haven't used the job I o classes or even if you have there, oftentimes confusing because people don't understand why the output streams air built the way that they are. Well, it's so that we can daisy chain that functionality on two different streams without having to have a specific instance of each class to build that out. Let's go ahead and run this and see what it does. I want to say run as job application, and if we go over to our file structure here, we'll see an output text, and once it reloads from being written, you can see that it has implemented text in there, and that may be a little bit small for you, but that's just the text output from it. The examples. Pretty straightforward. But we get all of our daisy chaining of our decorators built out here that ends up writing into our file system. And it's a good example of a decorator. Because the output stream alone can't write files, the file output stream can write to it. Well, the file output stream doesn't know about writing out data, so the data output stream goes through the file output stream, which goes through the output stream. You see that kind of reverse hierarchy that's being built there using that structural pattern a great example of a decorator pattern.

Exercise - Create Decorator

[Autogenerated] Now that we've seen an example of a decorator pattern, let's go ahead and build our own for our decorator. We're going to build a hierarchy that builds a sandwich. Now I'm taking this example from our builder example that we used in the creation all section of this course that implemented the different build options for building out a lunch order. And this is a real world example that I had used with that builder. Well, I like the functionality of that application that I went ahead and used it for this decorator. We're going to implement a few pieces, specifically a component, a concrete component, then a decorator and a concrete decorator. With these parts, we will have everything that we've needed to create our decorator pattern and condemn a wood. Then, to cement those concepts, we will implement another decorator, and Leslie will discuss that this is not a creation pattern and kind of show. The differences between this and the builder pattern

Demo: Decorator Implementation

[Autogenerated] all right to start off with our decorator pattern, I've gone ahead and created some classes just a fill in some of the blanks and not make so such mundane coding throughout this example, I've started off by creating an interface which is just our sandwich interface. We want to build a sandwich, so we're going to go ahead and start with that now, the interface doesn't have anything currently in it. We're going to go ahead and add a method signature called Public String make. I'm gonna just drop this in here now, This is just for us to make a sandwich. We're gonna build out our sandwich. Now, you see already that the placeholders I have automatically throw some red Xs because it they haven't implemented those yet. So now we're starting at the at the root of our hierarchy. If you remember rul UML diagram. Now we're going to go down and make a concrete instance of our sandwich. So here, in our simple sandwich, we want to go ahead and just implement a basic implementation of that make method. Now, I'm gonna go ahead and grab this, which is just some basic code that says we're going to return and instance of our sandwich, which basically will be our ingredients. We could say Basic sandwich or we could return bread here doesn't really matter. Um, this is going to be the basic implementation of what we want for a sandwich. If he ran it right now, all we would get back when we made our sandwiches bread. So there's nothing on it yet. You see how this functionality will build upon that. Now let's go ahead and look at the right half of Ruml tree and look at our sandwich decorator. Now our sandwich decorator is our abstract class, and this the base of our decorators. Currently, there's nothing in here. But one thing I want to point out is that it does implement the sandwich interface, and that is because decorators should be treated as objects because we're creating these decorators so that we don't have to keep creating various implementations of that sandwich so we wouldn't have to have a simple sandwich with meat. A simple sandwich with cheese, a simple sandwich with dressing, a simple sandwich with whatever thes decorators helped build this help. So now let's go ahead and add some basic pieces to this. I'm gonna go ahead and grab my notes over here and then talk through what we're adding. So, insider decorator, we're going to add a protected instance of our sandwich, which is what all the sub decorators air going to extend and build upon. Then we add a constructor that, through composition, takes in an instance of that sandwich. Now this is a key point that implementation of a sandwich can be the concrete class. Or it could be an instance of another decorator, and that's what we're going to see. As we build upon this

example, we're going to start off with sandwich and then at a decorator in another decorator in another decorator to it as we build our sandwich. And lastly, in this snippet of code, we are implementing the required interface of sandwich with our make methods. So right now in this basic class, this space implementation were just saying custom sandwich dot Make. Now let's move on to the meat decorator, the meat decorators, air first concrete decorator of all of our classes. So we started off with our base component, which is sandwich, and that's just our our base interface are top of Ruml tree. Then we have the left side of Ruml tree, which is our simple sandwich, and then the right side, which is our abstract class. But we currently don't have any decorators that are building anything on it. Now the meat decorator comes into play. So inside this meat decorator, we're gonna go ahead and implement many of the same features that we did in the sandwich decorator. But add some stuff to it, and I'm gonna grab my snippet of code from over here and paste this in and we can walk through. It would be very similar to the sandwich decorator, except you're going to see that we add a method down here called Add meat. As we build the sandwich, we can add functionality to it and say, Oh, no, this is essentially a meat sandwich. You could do another thing if you wanted tofu or or vegetarian or whatever you wanted to dio. You could go through and make different decorators to build this out. You can see in our sandwich example here that we don't have to create new instances of simple sandwich every time we want to change up what type of class we have. This is the same thing that the Iot packages air doing in using input stream inside of the job. I o a p i. Now, as we go in here into our decorator, let's save this. We can see that as we build these out. We're calling upon whatever implementation came into the sandwich, not necessarily a decorator versus a concrete object. So now we have our com custom sandwich that we call Make On and go ahead and add something to it. Let's add one more just to submit this concept in, go ahead and create a new class and it is going to be the dressing decorator. And it has a super class of have of sandwich decorator. And let's click finish now inside of here. It's gonna be very similar to the meat decorator. In fact, I'm gonna go ahead and just grab some of this stuff already, cause I haven't done and paste it in here Now. Inside of this, you can see that we have the constructor that calls super passing in the instance that we have utilizing composition, but we're also utilizing inheritance were using inheritance to extend the decorator but we're using composition to pass the instance in that we're working with. The nice thing about this is we're actually adding our functionality through composition. Then the next thing that we're doing is as we're making our sandwich. We are a pending on that functionality and overriding the features that we want to through that composition. So now that we have our decorator sandwich done here, let's go ahead and look at the demo, and this is where you see what's going on inside of our sandwich. We have a dressing decorator that takes a meat decorator that takes a simple sandwich. The simple sandwich isn't is our actual based sandwich, but the two decorators can take either the sandwich or the decorator in. Now. If we go and run this run as Java application, you can see that we have bread plus turkey and mustard, just like we have expected it dio. Now, one thing that's confusing about this is this starts to feel a lot like a creation all pattern, because we are passing all of these features in in constructors and it looks like we're building these out on constructors. But note that what we're doing is we're adding functionality to this simple sandwich that didn't exist there before. We're just doing that through constructors and using composition in those decorators. So although it feels like a creation all pattern, it's actually a structural pattern because we're modifying the structure of that simple sandwich by utilizing a decorator, and we get a lot of power in that hierarchy of that decorator.

Pitfalls

[Autogenerated] What are some of the pitfalls of a decorator? You end up building a new class for every feature that you want to decorate, realized that the decorator enables us to not need to extend the concrete object but rather implement a new decorator itself. The side effect of this, though, is that you end up with a lot of little specialized objects. Decorators can also be confused with simple inheritance. In Java, you only have single inheritance, and some features shouldn't be part of the hierarchy. Decorators give us a unique way to add functionality without creating concrete objects. For every feature that we want to implement, we rather create a decorator and don't mess up that hierarchy of our concrete objects.

Contrast to Other Patterns

[Autogenerated] to contrast the decorator. Let's go ahead and look at the composite. And if you just finished the composite module, I'll be honest with you. This is the exact same comparison that we made earlier. The composite is a tree structure, and its intent is to make a leaf and composite have the same interface. The client. It provides a unity between objects. The decorator, on the other hand, contains another entity. Now this may sound a lot like a composite, but it is really composition. As you saw in some of the life examples that we did, we utilize both inheritance and composition. Composition just contains another entity in It differs in that it modifies this behavior of that contained entity. So we want to use decorators to modify that behavior of the contain entity. This is usually adding functionality to nitty that it didn't originally have, and we just want to decorate the underlying object, but not necessarily change that concrete class

Summary

[Autogenerated] Let's recap what we learned with the decorator pattern. The original objects stays the same, so we don't have to creep creating concrete objects to add functionality to them. We can utilize the decorator to do that. It's an interesting and unique way to add functionality to those concrete objects. It's often time confused with inheritance, and you can see that because you use inheritance in the decorator. But we don't have to change our base object. So that's where the confusion of inheritance sets in. It can honestly be a little bit more complex for clients. Ah, lot of the other classes that we've looked at and patterns and methods and approaches to solving these things. You'll see that it gets simpler for the clients. I feel that the decorator exposes a lot of that functionality to the client and sometimes makes it a little more confusing for them. Still a great pattern, though. If you don't want to modify that base object

Facade Pattern

Introduction

[Autogenerated] Hi, This is Brian Hansen, and in this module, we're going to look at the facade pattern. The facade pattern provides a simplified interface to a complex or difficult to use system that is often the result of a poorly designed A P I.

Concepts

[Autogenerated] First off, it should be noted that the facade pattern is pronounced facade and not FIC aid. This is a common mispronunciation. When discussing this pattern when going to a job interview, make sure you call it the right name, the concepts surrounding why you would choose the facade pattern or one thing you want to make an A P I easier to use. Oftentimes you encounter a poorly designed A P I and can wrap a facade around it to hide the details from the client. It also helps to reduce dependencies on outside code. The main point that I usually look for is that it will simplify the interface or client usage. We typically want to wrap complex code with an interface using this facade to make it simpler for the end user. It should also be thought of as a re factoring pattern. You would usually want to implement a facade to wrap a poorly or complex written a p I. Examples of this in the job a p i. R java dot net dot u R l. There is a lot of functionality built behind the Ural class, and it provides a simple interface to the end user. There actually aren't a lot of good examples of the facade in the J via Java, a P I. And that is probably a good thing because is usually the result of a poorly designed or complex AP I. But another good example of the facade pattern is the Java server faces a P I specifically the faces context Now faces is part of the J two ee pattern, so we're not gonna talk much more about that. But suffice it to say that Faces has a fairly complex A p I, and interacting with the context can be quite difficult. So this is a great example of the facade pattern as well.

Design Considerations

[Autogenerated] the design of the facade is actually quite simple. It is a class that utilizes, typically just composition in its design. You shouldn't have a need for inheritance, and if you feel like you need to, then you probably ought to be looking at a different design pattern. The facade also encompasses the entire life cycle of whatever object you're dealing with, but it doesn't necessarily have to in order to be considered a correct usage of this pattern. The UML pieces associate with this UML diagram are simply a class and the packages or classes that the facade is making easier to use. Let's look at that UML diagram now.

UML Diagram

[Autogenerated] There really isn't a standard UML diagram for the facade pattern because the facade typically wraps whatever other AP I you're working with. Simply stated, though, the facade contains other classes, and this is the basis for this diagram. We have a facade class

that does something or some operation, and it contains, usually through composition, multiple other packages or AP eyes that is providing a simpler interface for the end user.

Example: URL

[Autogenerated] Let's look at an everyday example of the facade pattern in action. The euro class is an interesting example of this pattern. It face value. It seems like a simple class that doesn't do much when in fact it handles all of the connections for opening a UL and opening a stream to its content. You can see here that all we have to do to open up a stream is calling open stream method, and it handles all of the connections and connection information behind the scenes. It should also be of note that wrapping the input stream with a buffered reader is an example of a decorator. Let's go ahead and look at this code in a live sample now.

Demo: URL

[Autogenerated] Here's the U R L code that we were just talking about. You could see in this example. We go ahead and grab an instance of the u. R L. And it's pointing to plural site slash author slash Brian Desh Hanson. This is my author page on plural site dot com, and I'm gonna go ahead and open up a stream and wrap that with an input stream in a buffered reader. This is where we're talking about. There's a decorator inside of here, so we're showing a facade the nice clean usage of the U. R L class, and then also you get a great example of a decorator inaction. Gonna go ahead and grab this content and put into an input line and just dump out what this page returns. Let's go ahead and run this code. Now we just say, right click run as a job application and it will dump out all this text of what? The pages behind the scene, and it's really not too interesting. It's just a bunch of HTML. But it was a good example of it going out and grabbing that you are l just using a basic git and dumping this content out. Now a great example. The U. R L classes doing a lot of stuff here, opening and closing connection to making sure we've got things formatted the right way, handling streams on the back and all sorts of different things that are going on here. But it just has a nice little facade that wraps that all up for us, A great example of this pattern.

Exercise - JDBC

[Autogenerated] Now that we've seen an example of the facade in action, we're going to go ahead and build our own. That simplifies Working with the JBC, a p I. We're in a start was showing the complex client without using. If Assad that utilizes G A B C, a p I, then we will implement the facade that will just have a client, the facade in between and then our access to R J B C A P I, and lastly will show the simplified client code in a demo that wraps up all the pieces as we put them together.

Demo: JDBC Without a Facade

[Autogenerated] Here's an example of some code using thes e b c a p I without a facade wrapping it, you get an idea really quick of how busy this coat is or how much stuff is going on your client has to know about. There's just a lot of loose ends that you have to deal with. You

can start off by saying that I used an instance of the singleton that we created in the Singleton module of the creation all patterns course right here. And then we go down in our code and grab a connection that a statement. We execute that statement, close it. Then we insert a record into the table that we just created. Go ahead and close that. Then we create another statement, come down here and execute a result set to see if we grabbed a record from this table. You just see there's a lot of stuff going on here in a lot of little loose ends. If you've dealt with JBC very long, at some point you've had connections that were left open or various memory leaks because you forgot to close off. Resource is when you were done using them Let's go ahead and run this code now and we could see their tables created. We have one record that we created, and then we retrieve that record from the database. But let's see what a facade pattern conduce for us for cleaning up this client A P I.

Demo: JDBC Facade

[Autogenerated] looking at our JBC demo, we can see that there are three distinct areas were working with inside the JTB CSI classes. We have it where we're creating a table where we're inserting records and then we're selecting records back from the database. So let's go ahead and create a facade for this. I'm gonna say new class and we're gonna call this the jdb. See facade click Finish Now inside of here we want toe outline are three instances that we're dealing with. There are three areas that we're working with, so we're gonna have a public and create table. I'm gonna type this out because this really is the crux of this pattern, and then we're going to have a public and insert into table, and then we're also going to have a public, and I'm gonna return a list of address objects and we haven't created the address object yet. I want to get addresses now. This is really the main pieces of the facade that we're gonna be dealing with. So let's go ahead and fill this in now. And I've got this already done for, so I'm gonna just copy and paste some things in here. But first, let's get an instance of our DB single 10. I would just say instance equal to know. And this is the same singleton pattern that we used in the Singleton demo. And let's create a constructor here, just a no or constructor. So we'll say public JBC facade and then inside of here, which is to say instance, is equal to Devi singleton dot get instance now for our create table. Basically, what we're going to do is we're going to go ahead and copy the same code that we had of this JBC demo over here, but I have clean some things up a little bit, So I'm gonna go through and open up my demo notes here and paste this in, and I'll walk through what I'm doing here. So what I'm going to do is inside of our create table. I'm going to get an instance of and grab my imports really quick. I'm going, Teoh, take our database instance, grab a connection to it, create a statement, execute that statement and then close my resource is out and return a count of how many tables or records or statements we executed. Notice that I'm hiding all of this stuff from my client. My client doesn't have to know about getting the database connection. It doesn't have to know about our Singleton even doesn't know. I need to know about what the secret was. It was executed. Now, if we wanted to change this and make it where we could make a dynamic for any table, we could definitely pass a string into our arguments here. Something like this that would say sequel and past that in for our client to use to create tables in a generic fashion. But you get the idea of what this methods gonna dio. We're gonna do the same thing with the insert into table. Let me grab this code that I had written here copy and paste it in. And the same thing It's hiding the connection, the statement opening and closing. I am just putting out stack traces. If you wanted, you could throw an exception back to the client or return something different. But we get a good idea of

what we're doing here now for our retrieval for our address. And this is one thing that I don't like about J. D B C. has nothing to do with this design pattern. But it does have a concept of object during a programming is why I bring it up. The retrieval of an object You want to actually deal with objects. So let's go ahead and actually get create an address object. And for sake of this demo, I'm just going to create a class within this class. A little inner class will say class address and put an open and close on here. And Wolf, go ahead and throw some records. And here I've already got the class created going to just do an idea street name and a city, and we'll just right click and sage generate getters and centers source generate getters and centers, yet we want to select all of those hit. Okay, Now make sure you're doing this, that you are putting everything in the right order here. So we have are objects all in the right brackets. What I mean by that is that we now have our in her class down here below our main class declaration. If you haven't dealt with in a class as much and it could be an extra class, it's not a big issue, but you get the idea for this type of retrieval I like to use in her classes often to pass back information. Now that we've got this object in place, we can do our select statement. So I'm gonna go ahead and grab this code that I've already get got written for it and copy and paste that in here and now we can get an idea of what we've got going on. I'm gonna import the results sets you can see very much like the insert record and the create table. We are going to go ahead and grab a connection in a statement. But then we also have our results set. We're gonna execute this query, loop through it and dump out what we have going on here. And I need to do one of the thing to the client here. I want to upend on the record for the client to utilize. So I purposely left this out for us to work together on If I want to go ahead and say address address equals new address and then I can say address set i d rs dot Get string one and we could do the same thing for the other fields. So he can say Set said ST, and change that to get string to save that and then the same thing for the city. So the set City se rs don't get String three Now when we're all done with that, we want to go ahead and add that to her address is Objects will say addresses dot ad and add in our address and return that to the client. Now let's take a look at how much simpler our client is, and I've gone ahead and wrote the client code for us already. So we've got our JBC demo that we already use and then our facade jbc demo and let me put these two side by side and we can see here that are facade cleans up our client code a lot. So our facade demo is very simple, simply an instance of the facade. And then I've got some debug code in here, but we're creating the table, inserting the table and producing a result set from our get addresses that loops through its Our client doesn't need to know about any of the connection information are singleton or any of the other stuff that's got that we've got going on. Let's run this and see how it works. You see run as Java application and we can see that we had our table creator in our record inserted, and I did go ahead and print this out twice. That was simply because we had our four loop here that was printing it out, and we also had inside of our facade that was putting it out. In our select statement, she saw a couple of concepts there. We utilized our singleton, which was nice to take advantage of that. But we also got to see how this is. This facade is hiding all of this complex code, all these connections, statements and although type of information from our client, which results in our client being much simpler and only focusing on what it needs to now. As I mentioned, we could take the table and the answered in the table statements and change that takes mystery and so would work for any object and our insert table. We could make it take the address object in there, but this shows thesis implicit E that our client now experiences using the facade pattern

Pitfalls

[Autogenerated] what are some of the pitfalls of the facade? It's typically used to clean up code that was potentially designed incorrectly or poorly to begin with. If you're using it, a new A P I or interface, you really should look at the design of your A P I and see if another pattern will help solve the problems that you're trying to solve with the facade. It typically shouldn't utilise inheritance, so the facade deals with a flat problem or structure, and it is a It is the single 10 of structural patterns. And what A. What I mean by that is that it often is misused or overused because it's such an easy pattern implement You saw through our example. We just put a basic class in and masks some of those things that were going on behind the object behind it. It's an interesting pattern in the sense of object oriented programming, because it's just dealing with making a simpler interface to that client. But you can see how people often over use it or misuse it to just hide some of those ugly things that they've designed into their code. And if you're doing that up front, you probably need to think about the design of what you're trying to do inside your application

Contrast to Other Patterns

[Autogenerated] to contrast the facade pattern. Let's compare it to the adapter pattern. The facade patterns simplifies an interface, and it works with, typically just composite. It provides a cleaner a P I, to something that was designed incorrectly or complex Lee to begin with. The adapter, on the other hand, is also a re factoring pattern very similar to the facade, but it modifies behavior and typically is used toe add behavior to an object. So with the facade, we're just trying to clean up its usage. Where the adapter is adding behavior to the object that we're working with. It also provides a different interface to code. So the adapter is trying to maybe bring an A P I or an interface up to something more current in your application, where the facades just trying to make it easier to work with

Summary

[Autogenerated] So let's briefly recap what we learned with the facade pattern. It simplifies the client interface, and that's really the Onley. And main goal of this pattern is to make things easier and hide some ugly nuances of an A P I. It's a very easy pattern to implement. As I mentioned in the pitfalls, It kind of is the singleton of the structural patterns, because people can see how it will help and see how to utilize it so they get a little carried away with it sometimes. But it is a simple one to add to your code and something that you should consider in trying to re factor things. And it's definitely a re factoring pattern. This isn't something you should design in upfront. It's something that you want to use down the road after you realize that. You know this code is ugly and we're passing all this stuff down to our client to utilize. Let's go ahead and put a layer here so that we can reduce those dependencies that the client is necessarily looking at is you'll notice in our JBC demo, our client had to know about connections, statements, results, sets and all that type of stuff, and after we put our facade in filling and he didn't know about was our facade, it didn't need to know about any of the exception or exception handling or connections that were going on behind the scenes. So it's a nice way to clean up that stuff and reduce those dependencies inside of our application.

Flyweight Pattern

Introduction

[Autogenerated] Hi, This is Brian Hansen. And in this module, we're going to look at the flyway pattern. The flyway pattern is a pattern that minimizes memory used by sharing data with similarly type objects.

Concepts

[Autogenerated] the concepts surrounding why you would choose the flyweight pattern or when you need to make a more efficient use of memory. A flyway is definitely an optimization pattern. This is typically an issue when you have a large number of similar objects, and this is especially the case for objects that are stateless or immutable in nature. Immutable objects are objects that their state can't be changed after creation. This is possible when most of the objects state can be in tricks. Extrinsic, extrinsic properties are ones that are not essential or inherent. Examples of this in the job. A p i. R java dot Laing dot string strings are immutable objects and are loaded from a string literal pool that is basically the flyweight factory. Cashing is generally a hint also that you are, or could he be using a flyweight as well? There are quite a few other examples in the job, a P I, specifically java dot linda in Urge ER and the value of method, which is another great example of a flyweight. In fact, all of the primitive objects with rappers such as Boolean bite, character, short and long have a value of method that is similar to the inner jurors, one that is a flyweight in action

Design Considerations

[Autogenerated] the design of the flyweight is a little more complicated than some of the other patterns we've looked at. It is a pattern of patterns. It utilizes other patterns inside of it, so to speak. It uses a factory pattern to retrieve flyway objects after they've been created. The flyweight also often encompasses both the creation and structure of the object as far as the patterns concerned. So it has a creation all pattern inside of this structural pattern. It can and often does, manage the entire life cycle of the object. If you are unfamiliar with the factory pattern, you can watch the factory pattern module in the creation patterns course. The pieces of the UML diagram are a client factory flyweight and concrete flyweight. Let's look at that UML diagram now.

UML Diagram

[Autogenerated] the UML for the flyway starts with a client decline is what is requesting the flyweight object, although often times it doesn't even know that it's a flyweight. It requests it from a flyweight factory. The factory returns the cached object, or it creates a new instance of the flyweight eventually at the end of the process, if one doesn't already exist in our factory. The concrete flyway is, in the end what gets returned to the client, although it thinks it's just getting that object back often, Time declined, doesn't know the underlying structure and just has a simple interface. But regardless, this is what happens underneath it all. Now

that we've seen the UML, let's look at some demo code that is an implementation of the flyweight.

Example: Integer

[Autogenerated] If you've been developing with Java very long, you probably know that strings are immutable and come out of the string literal pool. The literal pool is a sort of cash that all strings in the jbm are stored in and retrieved from. This is a great example of the flyweight pattern in action, but most people don't realize the other rapper object for primitives such as imager also make a great use of the flyweight pattern as well. Here in this example, when we call the value of method, it retrieves the object representing the number that we want the value of. If that object doesn't exist, it will create it, insert it and then return that object from there on out. Other calls will get the same object. Let's look at this code in action.

Demo: Integer

[Autogenerated] Here's Theisinger code that is utilizing the value of method, as we can see here, to grab an integer object from a primitive value U C in the 1st 2 lines that we call value of with the value of five past in and get two different objects from that. Well, as we run this, you'll notice that down here in this system, without print Lynn, we have a system dot identity hash code. What this is is a little helper class that's in the system package that go out and and retrieve the identity hash code of this object. When we run it, you'll notice that the hash code for first and second end is the same because they're both for the value of five, meaning that it's grabbing that flyway object. And the third object into Juror third end, where we have a value of 10 will be a unique address. So once we run this run as Java application, you'll notice that those 1st 2 objects print out the same object address, and the 3rd 1 is a different address, and that's because the first ones creating it and storing it in the literal pool, the second one's just pulling it out of that literal polar. That flyweight cash and then 3rd 1 is creating a new object and retrieving that from the cash as well. So great example. And notice how your client really doesn't know anything unique or what's going on about the value of method, but under the hood, it's a perfect implementation of a flyway pattern.

Exercise - Flyweight

[Autogenerated] Now we've seen a live demo. Let's go ahead and create our own flyway pattern for our exercise. We're going to build a flyweight that is a simplified version of an inventory management system that I worked on for a client a little while back. It will be comprised of a client, a catalog. The catalog will contain a factory of fly weights and order and an item. The items will be the actual implementation of the flyweight or are concrete flyweight. We're going to implement the fly weights in the factory and will run a report to see how many objects were actually created for the orders out of our inventory management system.

Demo: Flyweight

[Autogenerated] there's a few little moving pieces and parts to this pattern, and rather than trying to code through it, I've coated out the pattern. And I'm just gonna walk through the individual pieces with you to start with. We have an object name item item is our implementation of our flyweight. It's the object that we're going to create a lot of. Think of this in terms of like Amazon. If you went out to every object on their site and pulled it up, you would chew up a lot of memory. And if everybody that was viewing it pulled back new instances of that object, you could see how it could be very memory intensive. So our item class is an example of our flyweight. It is our flyway instance and notice that everything in here is final and immutable. There's no centers or any of that type of information. We can just set the name through the constructor and then call a two string on it. We could put a getter and hair if we wanted to, and you could add other attributes to this class. If you wanted, you could have a U. P C. Code or description or a link to an image might be the actual path to the image. The next piece I want to talk about is the order object. The order object is just part of our application. There's nothing need. It really doesn't have anything particular to do with the flyway pattern. Other than order is going to consume items. CIA noticed that the order object has a constructor for an order number and an item to be passed into it. And then we process our order and this is just a show that we're doing something with it now. The catalog is where things get a little bit interesting. The catalogue, which is how review. Or you could also think of it as your search, how we go through and look at our items. It contains the hash map of our items and the factory method to see if we have these, and if we don't to create him and what we want to do with him. So it it has some very interesting attributes about how this gets handled and whether or not the client is getting back a cache of it or creating a new one. But you can see. It's a pretty simple implementation of how we look up these objects. Now, the last piece that won't talk about before we go to the client is the inventory system, the inventory system, and this is just mocking out a much more both robust one on the back end has an instance of the catalogue and then an instance of our orders. And it just basically creates some state in our system so we can take orders and we can process orders and then run a report on them and you'll see in here that this has a catalog dot total items made so it will go through and look at how many items we made from our catalogue object, which just contains the size of it. So for our client, we're gonna go through and create an instance of our inventory management system, are basically bootstrap our inventory management system and start taking orders. And now I've just got some basic strings put in there. We could do something more sophisticated, but we can order room buzz. We can order some Bose headphones or a Samsung TV, and I'm just passing in some random order numbers on the side just to simulate our order system. If you look at this at first glance, it would look like we're creating oh, about a dozen or so items in our system when really there's only three things that we're allowing people to order a Roomba, Bose headphones or Samsung TV. When we take out those orders were eventually going to close out that system or close out the reports, we're gonna do a process and then we'll print out a report of that. Let's go ahead and see how many items were created. We do run as Java application, you're going to see that we had all of our orders. But at the end, we only made three objects. The reason for that is when we take an order in our inventory management system, we come through here and we do a look up from the catalogue for that item name. So the item name that gets passed in is a Roomba, a Bose headphones or Samsung TV. Our inventory management system takes the order and looks it up out of the catalogue and says, Oh, I'm gonna hand you back an instance of this object. If it doesn't contain that object. It creates a new one and puts

it in that hash map. If it contains it, it just returns. That instance and you're good to go. Now we go back to our inventory system. It just adds those creates a new order each time and adds that order to our orders collection. And then we go ahead eventually and process our orders. Now, item nothing significant there. But notice that everything inside of item is immutable. Can't change it. There's no getters, nothing on that. It will just allow you to pass. That instance of that object around order is unique because each order has its own item and order number associated with this. So order has extrinsic things in there, but item has intrinsic values. You don't need to get hung up on those details, but just realize that the pieces of order can be changed where pieces of item cannot be changed, and then catalog is what's doing our factory toe. Look up those objects. So when we run it, it looks like we're creating all these item objects when really we're only creating three for this entire system. And it wouldn't matter how many times we took orders for those. We could take a 1,000,000 orders. If he only had three of those objects, it would only create three of them because that's all we had to choose from our flyway objects. So you see how this could be a very big performance boost for your system, especially if you start looking at some of your bigger online retailers such like Amazon or Walmart, or things like that that have a lot of objects or items that people could take advantage of and take orders for.

Pitfalls

[Autogenerated] what are some of the pitfalls of a flyweight? It is a bit of a complex pattern, as you notice is, we went through our example. There's Ah lot of little moving pieces to utilize it the right way, and you have to deal with a factory and some of the other pieces that you want to utilize just to take advantage of this pattern. I am personally not one that likes to prematurely optimize my code, and this is a pattern that you do that with. You are already planning up front for optimization. That's not necessarily a bad thing. If you've done a spike or you understand your product or architecture well enough that you know you're going to need to optimize upfront, then that's not necessarily a bad thing. But I often find that people figure that out later in their product lifecycle. Another pitfall with this is that you must understand the factory pattern now. That's not necessarily a bad thing, but if you have a pattern inside of a pattern, sometimes it can be a little bit confusing about which is the factory and which is the flyweight, and that the fly weights part of the factory of what? The factories returning and you get the example. It just becomes a little bit more complex. If you're looking at other examples out there. Ah, lot of them are graphical examples. Its value is well beyond graphical implementations. But a lot of research would lead you to believe that this is where it's on Lee Value is at. You can see through our example that we would utilize this in any sort of management system, you know, inventory management or a shopping cart system. Something to that nature where we have a lot of small objects that a lot of people will be utilising, and we need to take advantage of that memory management. So not just a graphical pattern, but a lot of the examples and samples you look at out. There definitely are graphical centric

Contrast to Other Patterns

[Autogenerated] to contrast the flyweight pattern. Let's compare it with the facade. And honestly, there aren't really a lot of other Strack structural patterns that are very similar to

the flyweight, but this is a good one to compare it to. For various reasons, the flyweight is focused on memory optimization. It is by nature a optimization pattern and deals with immutable objects. The facade, on the other hand, is a re factoring pattern and is usually implemented after the fact. So the flyweight is definitely implemented upfront and early on. If not, you're going to be re factoring a lot of code where the facade is definitely something that's thought of after the fact. The facade is also centred around making a simplified client or making it easier for the client. Once we realize that the A. P I were dealing with his very complex in nature, it also provides a different interface. Its entire goal is a different interface for the client, where the flyweight was designed up front for the client to not know that that pattern was being utilized under the hood. All it knows is that it has an instance of whatever object it was trying to get so we would use a facade to provide a more simplified interface where the flyweight was designed up front to already deliver a simple interface to the client.

Summary

[Autogenerated] Let's do a quick recap of what we learned with the flyway pattern. It is great for memory management. It's a pattern that we want to utilize if we know we're going to have a lot of objects that were going to pass around in our system and don't want to create those for every client or end user that's going to be accessing them from our system. It can't be a little bit of a complex pattern to deal with unnecessarily bad once you know the details of it. But you have a factory pattern in there. At a minimum, you have to make your objects immutable, and you're dealing with all these small little objects and how you pass them around where a lot of times people haven't thought of in terms of composition, you're going to pass an object into an object, and that's going to be an instance of your flyweight. It is used a lot by the core AP I. So there are some great examples out there, and now you know what you're looking for. You'll see all the time that oh, I'm using the java dot langdon integer value of our byte value of our character value of her strings are all great examples of fly weights, and they're not just used for graphical implementations. The next pattern that we're going to look at is the proxy pattern if you're following along this Siri's and that's the last one for the structural patterns. So let's go ahead and take a look at that now.

Proxy Pattern

Introduction

[Autogenerated] Hi, This is Brian Hansen. And in this module, we're going to look at the proxy pattern. The proxy pattern is a pattern that acts as an interface to something else.

Concepts

[Autogenerated] the concept surrounding why you would choose the proxy pattern or what we want to wrap it really object with a proxy. For various reasons, you create an interface to an object by wrapping it with a class to create that proxy. It can also add more functionality to that wrapped object. Proxy could be used to solve multiple problems such as security or

simplifying and interface to something a remote service call or just a expensive object to create. Well, oftentimes, rabbit with the project proxy and display loading message or something like that for a expensive object that we're trying to display to Are you? I or just loaded a memory. The proxy itself is called to access the rial objects will have an interface than a proxy that's wrapping the real object and then the underlying real object. Examples of a proxy in the Java empire actually kind of interesting. In different from some of the other patterns that we've looked at. The java dot lang dot reflect dot proxy object is a mechanism to facilitate creating proxy patterns using Java, so Java deep down, felt like this was an important enough concept that they create created a proxy class and object invocation handlers to facilitate creating proxies. And many of the frameworks you're used to seeing are built with java dot ling not reflect dot proxy. Also, the whole java dot r M I package is focused around proxy and remote method invocation. So as we looked at the concept above one of those being remote, the Java Dot are in. My package is all about accessing remote objects and retrieving that data across the wire.

Design Considerations

[Autogenerated] The design of the proxy is simple in concept, but it could be utilized in many different ways. The basis for it is an intermediary object that intercepts calls. That being said, it is typically interface based. Many frameworks like spring and some uses of hibernate and other various dependency injection frameworks use it and in doing so typically have an interface and an implementation class that the proxy resides in between the job, the A P. I recognize the need for the proxy pattern and incorporated an interface. The invocation handler and a class the java dot Laing don't reflect up proxy class to facilitate this. The pieces of the UML diagram are a client, an interface invocation, handler proxy and its implementation. Let's look at that UML diagram now.

UML Diagram

[Autogenerated] Here's the UML for the proxy pattern. As you can see, we start off with a client class that's going to make a reference call to some object, some subject and rather than it retrieving the real subject that we want, it's going to be intercepted with this proxy. So that subject would be an interface to whatever the implementation classes that we want to retreat. The proxy, using an invocation handler in the proxy class in Java intercepts that call and makes the call to the real subject. Or if it's a case like security, would deny it or do something different and turned around and sees if it needs to load that if it's gonna pull it from a cash or whatever, it's gonna dio. No, it didn't decides that. Yes, I am goingto low this really subject or this real object back and returns that back to the client. You can see how, really we've just got an implementation and a class, and that proxy handles the call in between those two

Example: Proxy

[Autogenerated] this example is gonna be a little bit different than some of the other examples we've done for. Other patterns were going to actually use the proxy class in this example and create a small sample that we're gonna tie into a real application to pull a

message back from Twitter. So to start with will walk through the invocation handler and the proxy class to build out a small piece, and then we'll continue that on for a large, larger example.

Demo: Proxy

[Autogenerated] So for this example, we're going to start off and create an implementation of a proxy that's going to be used to call Twitter to do so. We're going to start off by first building an interface because we typically have an interface and an underlying implementation class. So to start with will have the Twitter service interface and go ahead and click finish and said here, we're gonna add to method calls and I've already got these typed out. Just go ahead and grab these and copy and paste them in. We'll have to methods a get timeline in a post a timeline. What we're going to do is we're going to put a security proxy in between this and the implementation class now, because this demo would be too long to just do it. Once when you break it up, we're going to start off by creating a stub, which is just a method for us to try something out that doesn't isn't going to call the rial service itself going to do that in the second part of this example. So now let's right click and create a new class, and we're gonna call this the Twitter service stub, and it's going to implement our Twitter service interface and quick finish. Now it has two basic methods inside of here that it's overridden automatically generated for us. Let's just return a string and we'll just put something in Harris some fake texts, my need o timeline, and we can get rid of this to do for now on. Just leave this blank. So now we have our service and our stub. We're going to swap this out with an implementation letter later. Let's put a proxy in between here now. What we want to do is we want to create a new class and we'll call this thesis a cure ity proxy. And we're gonna make this implement the in vocation handler and click finish on this and you'll see that it drops in this invoke method for us that we have to implement as part of the invocation handler interface. So let's go ahead and get rid of this to do out of here, and I'm going to copy and some code that I already have written for the security proxy. Just you're not sitting here watching me type, so I'm gonna go to my demo notes and pull in a new instance and the invoke method that I have already written and paste that in here. So now there's a couple things I need to add. The way I have this written. I'm going to store a private instance of the security object that the object that we're rapping with the proxy inside of here. So we're going to go ahead and say, All right, let's call this constructor with an object. Say private object will be J and will make a private constructor security proxy were basically making a little factory pattern here. I will say Object object. This stuff object equals O B. J. And we're done with that section of it. So that facilitates us passing in the object down here. Now the next thing we need do is go ahead and handle any exceptions or at the things that the other things we have going on in here. So let's go ahead and hit control shift. Oh, and save this so that organized our imports. Now we have the basis for this class in place. Let's step through the code that I that I went ahead and copied and pasted in here. So we implemented the implication handler and that forces us to have this at invoke over here. In fact, I could put the at override annotation on here for the compiler and save that. But another thing we did is we turn this into a small little factory that we're gonna go ahead and grab an instance of this object back. And this is where the proxy pattern comes into place. You'll see that we return a java dot Laing dot Reflect up proxy, new proxy instance where we pass in the object that get class, get class _____ of this and the interfaces. So it builds this proxy around the class and any interfaces its implementing and then returns a new security proxy object that is wrapping

and then down below the invocation handler is where the magic happens of us invoking these calls. So this is where the proxy really comes into play. So we have our proxy it here they're gonna create. But this is what tells it what methods we will or won't invoke on when we call this. This is where we will eventually add some security around this as it stands right now, we're saying anything you dio I'm just gonna go ahead and pass through. So any method you try to call, let's invoke whatever action you wanted to on that and return the result from that so you can see that we go ahead and pass things in and past results out, and now we have the basis of our proxy in place. This is all you have to do to create that proxy. Let's go ahead and create a demo class for this so you can see it in action. So I'm gonna go ahead and now, right click and say New class and I'll check the public static void Main create check box and we'll call this the Twitter demo finished. Now let's go ahead and go in the main method that was created for us and create an instance of our proxy that wraps are stub that we created. So first we want to start off by creating an instance of the Twitter service to assign our object twos will say service equals and this is where it gets a little more interesting. I'll tell you right now we have to cast it, so I know I'm going to end up doing this. I say Twitter service and we want to do security. Proxy dot knew instance. And we're gonna pass into that a new Twitter service stub, and we want to go ahead and put a semicolon on the end of that line. And we can now just say, system dot out of print Lynn and will pass in the service. Get a timeline, and we'll just It's a stub right now. So whatever value we pass in, it's going to just return our default test data for us. Let's go ahead and run this now Gonna click, run right, click and save Run as a Java application and you'll see it reported back the Mayan Ito timeline that came from our Twitter stubs. So this is what we're returning out of our stub right now. In the next example, we're going to dive into a little bit more of what we did with this invoke method, and we're not going to allow people to post back because we don't have any security there and will take and replace this Twitter service stub with an actual call to Twitter. Gotta pull on 1/3 party library for that. But it's a real world example of something you can do and see how that security proxy comes into play.

Exercise - Twitter Proxy

[Autogenerated] to continue on with our proxy example. We're going to go ahead and download and implementation of the Twitter AP I with a little jar that's out there for calling it. We're going to modify our stub to be an actual implementation, so we'll switch that over to a classical. The Twitter service simple, could honestly be named whatever we want. But typically you have the service and in the implementation of it in that naming convention. And then we will finally restrict the post call to requiring you to have some security. Now we're going to stop that exercise a little bit short and just not allow post calls right now with their A p I. But you'll see how you can hook in security in that invocation handler in the proxy class that we created.

Demo: Twitter Implementation

[Autogenerated] to get started to actually make our implementation. Call the Twitter service. We're going to need to download 1/3 party jar, and it's a small jar. It's real simple. Just makes this example lot easier to use. So I'm gonna go ahead and come out here to Twitter for j dot org's and click the download link and will take us down here to the latest stable

version. Go ahead and click download already have it downloaded. Once you have yours, you're going to unzip it into a folder like this and go into live directory. And all we care about is thedc Twitter for core Twitter for J. Desk or example. So I want to go ahead and drag that over into my live folder and we'll say, copy files and then we can right click on it and say, Build path, add to build path. Now the other way you could do this is, of course, pull it in through maven, and this project is actually set up. Is maven maven project inside my I D. But to keep it a simplest possible, I just downloaded that link and copied it in there. Now the next thing we need to do is to go out to Twitter and create a APP account to pull this information in. Now I will tell you you have to do this because I am going to show you on the screen and I'm going to delete my keys afterwards. So if you try to put the keys that I'm using in, it will not work. I'm gonna switch back to my browser and go to dev dot twitter dot com and you'll need to log in with your account and verify that you're a developer and everything else, and you go down the very bottom and you'll see a link for manager APs. Once we go on here, say you currently have any Twitter outscore hadn't create one. So let's create a new app and I'm going to dio plural site proxy. And this is an app to demo calling a proxy that calls Twitter and for the website. When I just put http colon ford slash ford slash Yep, that'll work. W w dot portal site dot com Author Brian Hansen You can put your own personal website in there and we don't have a callback, your l so that's fine. Yes, I agree. Queer Creator. Twitter application. Now, this is gonna come in here and do two things for us. You'll see that it shows some various consumer keys and that type of stuff. What we want to do is we want to go to our keys and access tokens and you'll see that there is a consumer key and a consumer secret. I shouldn't be showing you this, And that's why I'm going to delete mine when this is all said and done. So I'm gonna copy that consumer key, and we're going to eventually paste this into our application. Let's go create the class that we're going to utilize this in. I have created a gist for you on get hubs or a guest, Jesse, and call it whatever you want toe where you don't have the type of this code in. It doesn't add anything to the example. But I wanted this to be a real world, real world example. So I'm gonna head and copy this code in, and I'm gonna go over to my I d e and create a new class in our proxy package. That is the Twitter service Simple and click finish, and I'm just going to replace all of this in here with this class here and save it. Now you can go through and look at what it does. It does a configuration builder and takes our keys and our Twitter factory and string builder and all that stuff and pace it in here and finally turn returns at at to our client. But you'll see right now, this is where we're gonna put those keys. So let's grab that Twitter key. There's my consumer key. Copy that. And make sure you copy it. Pace that over your put values here, your secret key. And then we'll have to generate to other tokens. It's white. And just copy these in here. So you're gonna come down here and say you have no access tokens. I'm gonna create my access token for this and you'll see there's my first access token. Be careful. If it's got a hyphen in there, it won't just select all for you. We'll put our access token in, and then our tokens secret. Copy that. And now we have everything put in place to do our call. So the steps there we had to download from Twitter for j dot org's unzip It grabbed the core library. We had to go to Twitter, log in and eventually goto manage our APS, go down there, create our plural site proxy app, and then generate our access key. So we had our consumer Keilar consumer secret and our access token and access secret. And then I had the Twitter service Empoli just here, and this will always exist here. In fact, I'm gonna go ahead and copy this u R L for you. It's a public just so you can always download it and just put it in a comment inside the application here for you to see that you can go download it from this u R l That will always exist for you out there. Now let's go ahead and go back to our Twitter demo, Public static void, main method. And instead of doing this stub, we're going to replace that with the simple and

save it. Now. When we call this, we should get what's on my timeline back, right? Click and say, run as job application and it went out and pulled the most recent things off of my timeline or anywhere that they could search and have at B H five k inside of that string. So you can see here we have a real world example where this is calling a remote object, and we could control access to it or other information that we wanted to retrieve from that implementation. One nice thing about this, this is why your dependency injection frameworks do. This is now I can control whether or not this object gets created, its lazily loaded. If it's a remote resource and there's Nick big expense to it, we don't pull it in until we need Teoh. We can wrap security around its. You see, There's a lot of benefits around having this proxy in place. And we did it with a real world example. The rial catches just making sure you get those keys in the right way and creating that application. Now, if you try these keys that I put in here, they're not gonna work. You have to replace these with your own value, cause I'm deleting these off my account right now.

Demo: Security Proxy

[Autogenerated] Now that we have our example working, let's go ahead and add that security to not allow people to call the post method in there cause currently, if we call it, it will let that in vocation go through. I'm gonna show you by modifying the Twitter demo class that will do just that. So if I call service dot post two timeline, give it a screen name, the H five k and a message, some message that shouldn't go through it currently will let it go through. Now. I don't have it actually implemented in the simple to go through and make that call because we're going to disallow it. But as it stands right now, it will go through. We'll just throw a system about out print linen here just to show that it is in fact happening. And let's go ahead and say no. Let's just paste the message in there and say that now if we call our Twitter, Twitter demo and run run as Java application, you'll see that it goes through and it says some message that shouldn't go through. So it's going through a now doing that now to implement the security and We're just going to a poor man. Security here. You could do a real full fledged security interface like spring security or something else. Look at some five or nine certificates or oh, author or whatever. I'm gonna open up our security proxy and go down here to the invoke method. And this is what's getting called by our proxy to actually invoke the various methods we want to call. So one of things I can do is I can come down here and I can say m dot get name and this will return the name of the method I want to call. So I want to just say, if m dot get name dot contains and what has passed in post, then let's go ahead and we'll close that if block off and say ehlz will allow it to go through. So if it does post, then we'll throw an exception. So let's go up in our F block here and say Throw new _____ and let's dio a legal access exception and posts are currently not allowed and close that off. So all we really did is grabbed the name and look at the method that's being called here and redirect or reroute it to whatever we want to do and you'll notice really quickly. We can do lots of stuff based on retrievals or posting or any of the crowd functions, or what just based off of our naming convention. So it's a little you get the power of having something like aspect oriented programming without a lot of the headaches of it, just because the proxy is here to intercept these calls for us. So now if we go back and run our Twitter demo right click source, right click, run out a job application, excuse me and you'll see that we get an exception thrown saying unexpected invocation. Exception. Posts are currently not allowed, so it did not allow that call to go through and stopped our application there. Clearly, we could clean that up a little bit and catch that runtime exception and do some various things. But you see this nice feature

now that we can go ahead and implement the security notice. Our business case didn't have anything to do with that. So when we're using this proxy pattern, we can put auditing or security or things like that in there and just allow the code to run how it was originally written. If we want to let that user do his such so, it's a real nice feature and shows the real power of that proxy where we're intercepting those calls in the middle.

Pitfalls

[Autogenerated] What are some of the pitfalls of a proxy? Well, you can Onley have one proxy. So if we want to implement security and auditing, we have to do it in that one proxy. We can't separate those out. Some other patterns allow you to change or rap. When you use the proxy, though you can only have that one instance, it also adds another abstraction layer. Now some might argue that this isn't a bad thing, but it can lead to other issues in case of like a remote proxy. If you believe you're accessing something local and it is in fact remote, you might get errors that you wouldn't maybe be expecting. It's also really similar to other patterns now. This may not seem like much of a pitfall, but it can be a little bit hard to identify. If you aren't familiar with the alternatives that you need a proxy instead of, say, a decorator or an adapter of sorts, it is easier if the proxy is for a remote object, but you can see how this can had a little confusion. If you're trying to determine which pattern you should be using,

Contrast to Other Patterns

[Autogenerated] to contrast the proxy. Let's compare it with a decorator. The proxy can add functionality, but it's really not its main purpose. We can really only have one proxy for that class instance, and it is. Functionality is set at compile time. It's not really the focus of what the decorator is. We are going to determine upfront what class were trying to call a remote call. Two or whatever other type of interface were trying to provide virtual or remote. The decorator, on the other hand, dynamically adds functionality. Its purpose is to add functionality and chain them as we go. They are a chained pattern, as I mentioned to. It also always points to its own type, where a proxy is intercepting a call to some different subclass or subtype. A decorator is usually looking at something else, and it's High Article chain, and the other contrast is that its functionality is usually determined at runtime instead of compile time like the proxy, we're gonna go ahead and change these things together at run time, and that object will figure out then what it can dio

Summary

[Autogenerated] Let's go ahead and recap what we've learned about the proxy pattern. There's great utilities built into the Java A P I. To implement the proxy pattern. You can see there's the proxy class, and the invocation handler interface would make it really easy for us to implement this pattern inside of Java. One drawback to it is that you only get to use one proxy per object. You're trying to proxy, too, so you can't chain them or build upon them. So if you have to add things to it, you might get some bloat inside that proxies or adding different features to it. I should also point out that this is used a lot by various dependency, injection and inversion of control framework. So a few years in spring or juice or

tools like that, we briefly touched on it. But this is this technique, or this concept is used a lot in there. One thing that people often think of that this pattern is that it's a great way to just implement lazy loading, which that really doesn't have a lot to do with the proxy pattern. But you can see how out of minimum that's what it could be used for. I like to use this pattern a lot when you're intercepting a call to maybe go out to a remote server and return that instance in there and not have all that code cluttered inside your object or how that gets handled.

Next

What Next?

[Autogenerated] Well, if you've made it this far, you've completed the structural design patterns course, and I thank you for sticking with it through this. Patterns are a great tool in a great asset, and I hope you refer to this back in time and look at as you get more experience with patterns and refresh her memory of what you're using and what might be a good fit for this pattern. If you're wondering what to look at next, there are various courses that I'm working on or have worked on. You may have already seen the creation of design patterns course that's out there. I'm also working on the behavioral patterns course that will finish this three part series and plan on doing more courses from patterns. As I've already had requests from other people to do so, I've got many courses out there is well on maven fundamentals, Spring fundamentals, The Spring, NBC and springing Me see four intro courses as well. A spring data J. P. A hibernate and spring security fundamentals, just to name a few out of the library. Thank you for watching this course, and if you have any other suggestions, feel free to reach out to be through the author. Aliases on the plural site website or my Twitter handle, which is on all the slides or connect with me on LinkedIn is well.