Introduction and Prerequisites

Introduction

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 coworker and they said it sounds you were reinventing the wheel. They might have even said that it sounds like you were describing the decorator or a proxy pattern. If this sounds like a situation that you've been in, then this course is for you. Our focus in this course in on presenting design patterns described in the Gang of Four in an example-driven way while using Java to do so. Hi, I'm Bryan Hansen, and welcome to Structural Design Patterns using Java.

Why Learn Patterns?

You might ask yourself why design patterns are important. I first learned patterns as a means of communicating a problem to another developer. You may very well already known how to solve a particular problem, and it might follow the structure of a pattern, but is 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 are more easily applied to a particular problem set than another. Revisiting pattern material 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. A lot of people have used singletons or maybe a decorator, and the thing they have used with that design pattern is they'll find out that that's all they know and there's much more than just singletons or decorators or adapters out there.

Pattern Classifications

The Gang of Four has patterns broken out into three groups. The groups are creational, 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 are focused on how you use or utilize objects. It could be for something like performance or refactoring, or memory utilization just to name a few concepts. Let's look at what we're going to cover as far as patterns are concerned in this structural group.

Which Patterns?

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?

Each pattern in 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 Java API, a demo in which we code our own pattern, the pitfalls associated with this pattern, and then we'll contrast it with another pattern that may be kind of similar to what we're looking at, or something that really abstracts or contrasts what we're doing with this particular pattern, and then we'll follow up with a brief summary.

Prerequisites

The prerequisites for this course are quite simple. I am doing all of my coding using Java 7 or later, and I'm going to be running this from Spring STS, which is just a flavor of Eclipse Honestly though, this should work in any IDE 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

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

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

Plug Adapter

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

Concepts

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

Design Considerations

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

Example: Arrays.asList()

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

Demo: Arrays.asList()

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

Exercise - Create Adapter

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

Demo: Adapter

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

Demo: Second Adapter

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

Pitfalls

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

Contrast to Other Patterns

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

Summary

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

Bridge Design Pattern

Introduction

Hi, this is Bryan Hansen, and in this module we are 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 adapter works with legacy code.

Concepts

The concepts surrounding why you would choose the bridge pattern are 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 decouples 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 bridge is in a lot of ways just a driver. A good example of this in the Java API are JDBC drivers. We have an interface that we work with and a driver that works with the underlying database.

Design Considerations

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, implementer, refined abstraction, and a concrete implementer. Let's look at that diagram now.

UML Diagram

If you aren't very familiar with UML, don't let the names of the objects in this diagram confuse you with the types of classes that they are. The abstraction in this case is an interface. This interface can be refined into our refined abstraction, and this is just a more specific implementation of that interface. From here we have our implementor, which is the hierarchy of our abstract classes. Our abstract classes will then be defined into a more concrete implementation, which will utilize composition of our various pieces of our bridge pattern to define those concrete implementations. There are a few more moving parts in this pattern we will look at in 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

JDBC is an API for executing SQL statements. Classes that implement the interface are JDBC drivers, and applications that rely on these drivers are abstractions that can work with any database for which a JDBC driver exists. The JDBC architecture decouples an abstraction from its implementation so that the two can vary independently, thus being an excellent example of a bridge pattern. Let's see this code in a live demo.

Demo: JDBC

In this demo we're going to go through and use the JDBC API to go ahead and create a database table in a Derby database. So this example is pretty straightforward. In fact, there's a more detailed example of this in the Creational Design Patterns course under the singleton that shows how to wrap all of this, so we only get one instance of that connection each time we want to utilize it. But in this example we're going to go ahead and register our driver, and then we're going to build our URL, 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 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 API that we had. So from there we can create our statement and then execute our statement, which will tell us that we've created a table. So let's go ahead and run this, and if I go ahead and execute this you can see that our table was created. It's a pretty straightforward example, but it's a good example of the JDBC API and how it is a bridge to an underlying database. We could swap out our database with MySQL or SQL Server, Oracle, any of those, and our client could stay the same, or our client can change without our underling database having to worry about the details or whatever other configurations we need to, we can just utilize our driver to make that bridge between the two.

Exercise - Create Adapter

Now that we've seen a live demo of the bridge in use, let's go ahead and create our own. We're going to 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 going to 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 going to 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 on to that.

Demo: Shape Without a Bridge

Here's an example that illustrates the problem that we are trying to solve with the bridge pattern. So it's a simple application. We have some shapes, a circle that we're going to 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 applyColor to this. Now the problem with this is that it can't grow with us, 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, oh, I'm going to use inheritance to solve this problem, well inheritance doesn't really solve this problem. I have a shape, I 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 and a blue square. The problem is, and this code will run just fine when we go to run our application it'll run and it'll apply our colors to it, so you can see we have our blue color and our red color, but what happens when we want to say, add a green square. Oh, well now we've got to go through and create a New, Class, and say oh, well we want a GreenSquare, and its superclass is going to be the Square, and we'll go ahead and click Finish and it will go through and generate a method for us, and we'll say oh, well let's apply our color, so we'll do System.out.println and say that we're going to be Applying green color. This isn't that hard because this is a very simple example, and that's what I really don't like about this example, it's such a simple solution, but you see that for every color I want to add I have to go add a GreenSquare, I'd have to do that same thing, I'd have to add a GreenCircle, and now we have yellow and we're going to add a YellowSquare and a YellowCircle, 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've got to go through and add a rectangle that's red, blue, green, yellow, and it just keeps expanding and expanding from there. So just to show that this works fine, the code will run just fine with us, so we'll do a GreenSquare, it'll run just fine for us, = new GreenSquare, and then we can apply that, so we can say greenSquare.applyColor, and I know it'll run just fine, you know it'll run just fine, so we can see all of our colors, our blue color, red color, 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

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 are 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 an abstract class, it still has an abstract void method for applying color, but now we take a colon in the constructor to utilize through composition. Color is its own interface now, and we have the applyColor method defined in that interface. So Shape has applyColor 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 and we can 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 and our red color. But now let's show by adding another color or a shape to this how it's not the issue of an orthogonal problem like we had before. So let's go ahead and add another color here, and we'll 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.out.println, just for our basic example, and we're going to apply the green color, so we're applying our green color. And to save this, now everything works here, we go back to our sample, and now if I want a green circle, we already have our shape and our square here, let's go ahead and create a new circle, let's say Shape, and we'll do a greenCircle = new Circle, and I can 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, so we'll say Color green = new Green, and save that. Now we can come down here, and just to execute it and make sure that it runs right, we'll say greenCircle.applyColor, and run it. 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 an instance of a GreenSquare now, I can also say Shape greenSquare = new Square, I can now pass in that same green color instance to it and apply that color down here, greenSquare.applyColor, and save it. Now if we run this it'll also work as well. So we were able to now add changes, add colors, we could add another shape too to really illustrate it, but you get the idea. The bridge is the way we are 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 are more of an attribute, 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 add a color without having to go through and change 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

This bridge pattern demo is a little bit larger than some of the other patterns that we have gone through, so I'm going to go ahead and pre-code some things to show you what we need to create the pattern, but not bore you with the details of watching me type. So you can see inside of here I have a Movie object that's already created and it's populated, and it's a simple POJO, there's nothing about it, it doesn't inherit from anything or implement anything, it just has a classification, runtime, title, and the year in 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 use to pass information between the bridge pieces that we're 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 that we're going to be building. So let's start off by creating our interface, which is New, Interface, Formatter. Now this Formatter represents the same thing as we were doing with the color in 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 go to details, and then let's save this and import our java.util.List, and now we have our Formatter interface. This interface, like I said, is very similar to what we had with the Color interface. Now, we're going to 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 are going to extend this or implement various aspects of this that are going to be our concrete implementations. So now we have a couple methods we need in here, and I'm going to type these ones out because this really is the heart of the bridge pattern, the rest of the stuff I'm just going to copy and paste in, but we're going to have a method inside of here that is our print method. So we're going to say String print, and this is going to take the Formatter instance in. Now remember this is our composition. So we have our instance of our formatter, and this returns formatter, formatter.format, and it's going to pass in our getHeader, and this method isn't defined yet, and our getDetails. You can quickly see how you could add stuff for getting a footer, or those various aspects inside of here, formatter. And now let's go ahead and create those two abstract methods inside of here. So we want to do abstract protected List Detail getDetails, and this is what we're going to call in 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 getHeader, so we're going to do abstract protected String getHeader. Now we have all the pieces in place for our bridge, we have our formatter, which is representative of our color and shape as the one-half of our bridge that we're going to 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 we had with our colors where we had a red square or a blue circle, now we can have printers and formatters stand independently of one another. So we could have a web printer or an HTML printer, as well as a print printer, more concrete strings, whatever we want to do, we can build upon those to do whatever we want and have these two change independently of one another. Now that we have those pieces put in place, let's build our print formatter and our movie printer. So we're going to come over here and create a New, Class, and we want to start off with the MoviePrinter, and it is going to extend Printer. Go ahead and click Finish, and you can see it's put our two instances, our two methods in here already from our abstract class that we inherited from printer. So let's go ahead and store an instance of the movie in here, so we're going to say private Movie movie, and we're going to pass this in, since it is through composition, we're going to pass this in and say public MoviePrinter, 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 play. So this.movie = movie. Now the details of getting the details and getting the header, I'm going to go ahead and paste those in from some sample code that I had earlier. Let's grab the details there and go into our MoviePrinter 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 creating those name value pairs as we go, and then I'm going to do the same thing with the header, and the header is really straightforward, I could have just typed that, but it'll work just fine. We'll say MoviePrinter and replace our header and save that. And now we have our MoviePrinter built. Now we need to do the same thing with our formatter on the other side. So we have one-half of our bridge done, let's do the other half of the bridge now, which is our print formatter. So we're going to come over here and right-click and say New, Class, and do PrintFormatter, and it's going to implement the Formatter, not extend the superclass, it's going to implement the interface, the Formatter interface. We'll click 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.println was applying before to our screen, this is going to build the string for us that we're going to return to our client. So I already have that done in here as well, which is just a basic String builder, and you can see what we've got going in here, let's paste this out. I go through, create an instance of a string builder, and start appending the header and the body details to it, I cycle through each one of the details, build that to the string, and then return that builder.toString to the calling code. We're actually done, that is the bridge pattern. Now let's go back to our demo and start uncommenting these. You can see our PrintFormatter and our MoviePrinter. Notice that MoviePrinter takes an instance of the movie in to store it, and then we can take and call moviePrinter.print and pass in our printFormatter. So let's look at our MoviePrinter and we can see it has our, it extends our printer method, or our printer class, and if we look in our printer class there is our print method that takes an instance of the formatter. Notice it doesn't know anything about the formatter that's being passed into it, just the contract through the interface. Now if we come back over here we can see that moviePrinter.print is going to return our printed material, and we will be able to 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:2:15. You'll notice that this knows nothing about the MoviePrinter object itself and the MoviePrinter object knows nothing of the PrintFormatter, all of that stuff can change independently. Now this is 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 to, in a business sense, I want to print a catalog, 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 an HTML print formatter instead of just the print formatter 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

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 formatter to do an HtmlFormatter. You can 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 HtmlFormatter, and it's going to implement the Formatter interface, click Finish, and you'll see that it just goes ahead and like it did with our PrintFormatter, create a basic method signature for us. So now we can 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 PrintFormatter. I've gone ahead and created an instance of a String builder and then done a builder.append, and you can see that I've added on here the headers and the different columns, and then got into the body of it, and the sky's the limit with what you want to do here, but what I want to point out is all I do to add another formatting type is create one instance of this class, and now I can utilize it anywhere. So I'm going to go head and save this. If I go back to my BridgeDemo, I now have my HtmlFormatter, and notice that I didn't have to create a new instance of MoviePrinter. I'm using the same MoviePrinter that we had before and just passing in the different formatter and getting that material back. So now I can save this and run this, and you'll see down below here that we have our 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 HtmlFormatter and pass that in to my MoviePrinter, 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, a book, so you would just create an instance of the movie and change that to utilize it as a book, and then create a BookPrinter and go through and build those details that you want you. Maybe you'd have page count, maybe you would have the Dewey decimal number, or whatever other features you want, the ISBN number, those types of things that you would build into that details object. You don't have to use that detail object the way that I did here, this little class, it was just a convenient way for me to pass things through generically and share that information from those objects from one to another, you can utilize that however you want. But all you'd have to do to create the other half of the bridge, like I said, was create a different type of object from Movie, maybe Book, and then a BookPrinter and our PrintFormatter and our HtmlFormatter would already work, you could just grab those values and pass them in and you're good to go.

Pitfalls

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 code-as-you-go scenario. It is definitely more than just good OO principles. One comment that is 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 can just be a little confusing as to what goes where in your code.

Contrast to Other Patterns

To contrast the bridge pattern with another one, let's go ahead and look at the adapter pattern. The bridge pattern is designed up front, it's something we consciously go into our application with a mindset that we want to do. The abstraction and the implementation can vary. As we saw with the demo that we did, we have our printer object and our formatter 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 more 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 try to tie in legacy code with a new application. And in doing so, it's typically retrofitted in, it's something that we're 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 a get a different interface for existing or legacy code, we are 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 changes independently of one another.

Summary

Two wrap up the bridge, let's just quickly recap what we've covered. The bridge is designed for uncertainty. It can be complex, and oftentimes 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 complexity 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 lastly just point out, it is much more than composition. We do utilize composition, we also utilize inheritance through abstractions and interfaces inside of our code, so we're really taking a lot of the principles of good OO design and wrapping them into this one pattern.

Composite Pattern

Introduction

Hi, this is Bryan Hansen, and in this module we are going to look at the composite design pattern. The composite pattern is a hierarchical type pattern that deals with tree structures of information.

Concepts

The concepts 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. In 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 Java API are java.awt.Component, almost everything from the java.awt library is built this way, as well as JSF widgets. Now JSF isn't part of the core API, but almost everything in JavaServer Faces is built around this same structure. Another useful example is RESTful 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 RESTful service, don't worry about it, but there is a couple of courses out here on Pluralsight, specifically one on RESTful services using Jersey that talks about how we navigate and recurse down through the directory structure using this composite model.

Design Considerations

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

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 component's 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 component's methods as well, but typically delegates the functionality to its children. So a leaf and a composite have the same functions inside of them, a composite just knows about which children it has and what it can do with those children.

Example: Map

The map and other collections are an interesting implementation of the composite pattern. Much of the collections API have the option to addAll, or in this case staying with the naming convention of a map, putAll 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 coding example now.

Demo: Map

Here is the map example that we were just previously describing. In this example we're going through and building a list of personAttributes and a list of groupAttributes, and then combining them all in one to represent our secAttributes using the putAll method, which treats each map as a composite of the previous one, so it treats the individual object the same way. And if we run this application, you can see that it has eventually combined all of our roles into one security map. Now I've done this, this is actual code that I've 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, although 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 menuing systems, which is typically where you see the composite pattern being used.

Exercise - Create Composite

Now that we've seen an everyday example of the composite pattern, let's go ahead and create our own. We're going to start off by creating a Menu, MenuItem, and MenuComponent to represent our composite, leaf, and component of the UML diagram. Then we're 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

To show this example, I have created a couple of stubbed out classes just to help us get things rolling. First let's start with the Main method. Inside this Main method I just have a basic menu structure that we're building that has a Main menu, then there's a Safety MenuItem that is its own menu underneath that, and a Claims menu, and then a Personal Claim menu, and all of that builds into a much larger composite menu. So we have a menu with submenus, and those have individual items underneath them. So to facilitate this I've started off with a MenuComponent class, and the MenuComponent class is pretty simple, it has a name and a url and an array list of menu components, and you can just see the basic features we have in here, a getter for the name, a getter for the url, and I've put an abstract toString method in here. We're going to 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 we can add and remove things to it, as well as it has a name and a url itself. 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 we'll do to this is we're going to also add a toString method to print out this menu structure, just for display purposes, but to also see about which pieces should know about the children and which shouldn't in there. The last piece that we're going to implement is the MenuItem. The MenuItem is not going to change a whole bunch from this, it just has a name and a url, but we're going to also add some toString method stuff in here to just help round out the object and show all the pieces as they work together. So let's get started with the MenuComponent, and as we go through this you'll see these red squigglies underneath our objects here go away, our red Xs. So let's start with the MenuComponent. Now the MenuComponent 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. Let's go through and I have some sample code already put in here, I'm going to add this print method that I've already premade and just paste this in the bottom of this object. Now the print method here goes through and creates an instance of a StringBuilder, which by the way is another design pattern that you can look at in the creational section of my other course, it talks about what the builder pattern is and how it works, but the builder pattern here we're going to go through and get the name of this string, of this MenuItem and append the colon to it, and then the url and return that object out so we can just see the menu hierarchy get built as we go here. So we have our menu here, now let's go ahead and move onto the Menu class, we have our MenuComponent, 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 an add and remove method, as well as we want to build out the toString 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. Now 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 as you go down the road if you want to know which object was removed or which object was added, you just return the reference to it. And so I've got an 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 @Override commented out, and I'll explain that later, it's because you don't have to implement these features for this to be a useable class, but we'll talk about why I've got those commented out here later. Now I still have a compilation error in here, and that's because I don't have the toString implemented yet that is required from our MenuComponent class. So we have this toString method here, our Menu object is looking for it. So let's go ahead and put a toString in there as well. And this one is fairly complex. We're going to walk through what happens here. So let's open up the Menu object here and paste it in. Now the toString in here has a couple of things going on. Just like in the MenuComponent, we have a StringBuilder, we also inside of our toString here have a StringBuilder, but we call the print method for MenuComponent, so this print method here is referring to the one inside of our MenuComponent, and then we iterate 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 object structure can be handled the same way as its own leaf, and so we just hand off it to do what it needs to and append that back, and you'll notice we're doing exactly that. When we get our next menu component we go ahead and just append the menuComponent.toString to us and return it back. So we recurse 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 MenuItem 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 toString. So we're going to 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 and added our add and remove for our child or our composite structure here, so we know about our children, and navigated down recursively through our toString method by calling the menuComponent.toString and having it walk down through anything and its children. And then our MenuItem we added a toString inside of this, and this is just calling the print function up in the tree structure of our MenuComponent. 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 built, our Safety menu, Claims submenu, all of that stuff, if we run this, we'll see that it prints out our Main menu, Safety menu, Claims, and Personal Claim menu as we go through and build our structure. We just called the System.out.println on our MainMenu, and it navigated through all the structures we went through here. Now if you wanted to take this example a step further, you could change that toString method to go ahead and print out the entire path down to it, so /main/safety/claims, or /main/claims/personalClaims 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 problem solved quite simplistically, and as far as our client is concerned, they just add them and treat them just the same, and it handles it pretty elegantly.

Demo: Unsupported Operations Exception

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 you remember in our Menu class we had these @Overrides commented out, and I said that we would talk about them later. Well that is because we can add in features to our MenuComponent and paste those throughout our hierarchical structure, but we don't have to use this pattern. Let me explain what I mean by this. So inside of our MenuComponent, we can go ahead and add these two methods in here, and these two methods are what establishes 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 MenuItem. What if we had a complex MenuItem, or a graphical MenuItem. You can see we can start adding different types of MenuItem 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 it abstract because we are overriding as we go through our child objects, we can now implement these features where we want, but we can throw an UnsupportedOperationException at other points in our hierarchical 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 MenuItem 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 UnsupportedOperationException Feature not implemented at this level, so you can establish that contract a little better. Let's just run this again to make sure that everything works alright, which it does, and it works as we expect it to.

Pitfalls

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 can make it difficult to restrict what we want to add to it. Everything eventually is treated the same, and that is the intent of 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 if dealing with a very large composite, or if implemented incorrectly. This doesn't have to always be the case, but if child objects are held in a collection and each object itself contains a collection, its size can grow fairly quickly. Typically composites aren't that big though, and I don't find this being a real practical issue.

Contrast to Other Patterns

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 is really just composition. Composition is just an object containing another one. It differs in that the decorator modifies the behavior 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

Let's do a quick recap of the composite pattern. The composite pattern generalizes a hierarchical structure. As we saw in our examples we built a menu structure that had menu items such as leaves 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 and it didn't care if we had a menu, a submenu, a MenuItem, it just handled it all gracefully the same way. Likewise with 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

Hi, this is Bryan Hansen, and in this module we are 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

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 effecting other parts of the hierarchy if you don't want to. It is more than just simple inheritance though. You are controlling which pieces compliment 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 subclasses that decorate your object. This candidly makes it feel a little bit like a creational pattern, but it is adding behavior through creations, so it is in fact still a structural design pattern. Examples of this in the API are the java.io.InputStream 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 java.util.Collections API also has a checkedList method, but it's not really clear as to why that is a decorator pattern, so we're not going to use that for our example. And also I should note that almost all UI components in the AWT and Swing APIs are implemented following a decorator pattern.

Design Considerations

The design of the decorator is that it is inheritance-based. Often the 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 subcomponents. It is also an alternative to subclassing because it adheres to the single responsibility principle that we talked about in the concepts section where this class will just do one thing. Typically when you 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 UML diagram are a Component, ConcreteComponent, Decorator, and a ConcreteDecorator. Let's go ahead and look at that example now to see what the UML diagram looks like.

UML Diagram

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 ConcreteComponent of this diagram. The ConcreteComponent is what we are going to eventually decorate, and the Decorator is the base decorator or wrapper that we will extend and create other decorators from. Now it should be of note that both the ConcreteComponent and the Decorator extend the component so that they can be treated the same. From here we can then create multiple ConcreteDecorators to decorate our object and provide functionality as we develop.

Example: File

Here is an everyday example using the InputStream class from java.io that utilizes decorators to build out its functionality. In this example we're going to go ahead and create an instance of a file and write that file out to the OutputStream. Using an OutputStream, which is a base decorator, we can add on to its functionality using a FileOutputStream and a DataOutputStream 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 appending its functionality on. Let's go ahead and look at a live example of this code.

Demo: File

As we mentioned earlier in this example, we're going to create an instance of a file and write that out to the OutputStream. Here we've created an instance of a FileOutputStream and assigned that an OutputStream. In this example our OutputStream would be our base decorator and our FileOutputStream would be one of the instances of a concrete decorator. We then turn around and pass that FileOutputStream into a DataOutputStream so that we can write physical data out to the system. Pretty straightforward, but if you haven't used the java.io classes, or even if you have, they're oftentimes confusing because people don't understand why the OutputStreams are built the way that they are. Well it's so that we can daisy chain that functionality on to 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, Java Application, and if we go over to our file structure here, we'll see an output.txt 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 example is pretty straightforward, but we get all of our daisy chaining of our decorators built out here that ends up writing it into our file system, and it's a good example of a decorator because the OutputStream alone can't write files the FileOutputStream can write to it, well the FileOutputStream doesn't know about writing out data, so the DataOutputStream goes through the FileOutputStream, which goes through the OutputStream. So you can see that kind of reverse hierarchy that's being built there using that structural pattern. It's a great example of a decorator pattern.

Exercise - Create Composite

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 creational 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'd 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 ConcreteComponent, and then a Decorator and a ConcreteDecorator. With these parts we will have everything that we've needed to create our decorator pattern and can demo it. Then to cement those concepts, we will implement another decorator, and lastly we'll discuss that this is not a creational pattern and kind of show the differences between this and the builder pattern.

Demo: Composite Menu

Alright, to start off with our decorator pattern, I've gone ahead and created some classes just to fill in some of the blanks and not make 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. So I'm going to just drop this in here. Now this is just for us to make a sandwich, we're going to build out our sandwich. Now you'll see already that the placeholders I have automatically throw some red Xs because they haven't implemented those yet. So now we're starting at the root of our hierarchy, if you remember our UML diagram, now we're going to go down and make a concrete instance of our sandwich. So here in our SimpleSandwich, we want to go ahead and just implement a basic implementation of that make method. Now I'm going to go ahead and grab this, which is just some basic code that says we're going to return an instance of our sandwich, which basically will be our ingredients. We could say Basic Sandwich, or we could return Bread here, it doesn't really matter. This is going to be the basic implementation of what we want for a sandwich. If we ran it right now all we would get back when we made our sandwich is bread, so there's nothing on it yet. You can see how this functionality will build upon that. Now let's go ahead and look at the right half of our UML tree and look at our SandwichDecorator. Now our SandwichDecorator is our abstract class and is 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 decorator 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 SimpleSandwich with meat, a SimpleSandwich with cheese, a SimpleSandwich with dressing, a SimpleSandwich with whatever, these decorators help built this out. So now let's go ahead and add some basic pieces to this. I'm going to go ahead and grab my notes over here and then talk through what we're adding. So inside our decorator we're going to add a protected instance of our Sandwich, which is what all the sub-decorators are 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 can 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 a Sandwich and then add a decorator and another decorator and 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 method. So right now in this basic class, this base implementation, we're just saying customSandwich.make. Now let's move on to the MeatDecorator. The MeatDecorator is our first ConcreteDecorator of all of our classes. So we started off with our base component, which is Sandwich, and that's just our base interface, the top of of our UML tree. Then we had the left side of our UML tree, which is our SimpleSandwich, 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 MeatDecorator comes into play. So inside this MeatDecorator we're going to go ahead and implement many of the same features that we did in the SandwichDecorator, but add some stuff to it. And I'm going to grab my snippet of code from over here and paste this in, and we can walk through it. It'll be very similar to the SandwichDecorator, except you're going to see that we add a method down here called addMeat, as we build the sandwich, we can add functionality to it and say oh, now this is essentially a meat sandwich. You could do another thing if you wanted, tofu or vegetarian or whatever you wanted to do, 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 SimpleSandwich every time we want to change up what type of class we have. This is the same thing that the I/O packages are doing using InputStream inside of the java.io API. Now as we go in here into our decorator, let's save this, and 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 that we have our custom sandwich that we call make on and go ahead and add something to it, let's add one more just to cement this concept in. So I'm going to go ahead and create a New, Class, and it is going to be the DressingDecorator, and it has a superclass of SandwichDecorator, and let's click Finish. Now inside of here it's going to be very similar to the MeatDecorator, in fact I'm going to go ahead and just grab some of this stuff already because I have it 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, we're 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 appending 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 where you can see what's going on. Inside of our Sandwich we have a DressingDecorator that takes a MeatDecorator that takes a SimpleSandwich. The SimpleSandwich is our actual base sandwich, but the two decorators can take either the sandwich or the decorator in. Now if we go ahead and run this, Run As, Java Application, you can see that we have Bread + turkey + mustard, just like we would have expected it to do. Now one thing that's confusing about this is this starts to feel a lot like a creational 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 SimpleSandwich 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 creational pattern, it's actually a structural pattern because we're modifying the structure of that SimpleSandwich by utilizing a decorator, and we get a lot of power in that hierarchy of that decorator.

Pitfalls

What are some of the pitfalls of a decorator? You end up building a new class for every feature that you want to decorate. Realize 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'd rather create a decorator and don't mess up that hierarchy of our concrete objects.

Contrast to Other Patterns

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 for the client. It provides 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 live examples that we did, we utilized both inheritance and composition. Composition just contains another entity in it, it differs in that it modifies this behavior of that container entity, so we want to use decorators to modify that behavior of the contained entity. This is usually adding functionality to an entity that it didn't originally have, and we just want to decorate the underlying object, but not necessarily change that concrete class.

Summary

Let's recap what we learned with the decorator pattern. The original object stays the same, so we don't have to keep 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 oftentimes 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. A 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. It's still a great pattern though if you don't want to modify that base object.

Facade Pattern

Introduction

Hi, this is Bryan Hansen, and in this module we are 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 API.

Concepts

First off, it should be noted that the facade pattern is pronounced "fe sod" and not "fe cade". 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 are when you want to make an API easier to use, oftentimes you will encounter a poorly-designed API and can wrap a facade around it to the 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 refactoring pattern. You would usually want to implement a facade to wrap a poorly or complex written API. Examples of this in the Java API are java.net.URL. There is a lot of functionality built behind the URL 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 Java API, and that is probably a good thing because it's usually the result of a poorly designed or complex API, but another good example of the facade pattern is the JavaServer Faces API, specifically the FacesContext. Now Faces is part of the J2EE pattern, so we're not going to talk much more about that, but suffice it to say that Faces has a fairly complex API, and interacting with the context can be quite difficult, so this is a great example of the facade pattern as well.

Design Considerations

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 lifecycle 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 associated 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 the UML diagram now.

UML Diagram

There really isn't a standard UML diagram for the facade pattern because the facade typically wraps whatever other API 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 APIs that it's providing a simpler interface for the end user.

Example: URL

Let's look at an everyday example of the facade pattern in action. The URL class is an interesting example of this pattern. At 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 URL and opening a stream to its content. You can see here that all we have to do to open up a stream is call an openStream method, and it handles all of the connections and connection information behind the scenes. It should also be of note that wrapping the InputStream with a BufferedReader is an example of a decorator. Let's go ahead and look at this code in a live sample now.

Demo: URL

Here's the URL code that we were just talking about, and you can see in this example we go ahead and grab an instance of the URL, and it's pointing to Pluralsight /author/bryan-hansen, and this is my Author page on Pluralsight.com, and I'm going to go ahead and open up a stream and wrap that with an InputStream and a BufferedReader. This is where I was talking about there's a decorator inside of here. So we're showing a facade, the nice clean usage of the URL class, and then also you get a great example of a decorator in action. I'm going to go ahead and grab this content and put it into an inputLine and just dump out what this page returns. Let's go ahead and run this code now, we can just say right-click, Run As, a Java Application, and it will dump out all this text of what the page is behind the scenes, 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 URL, just using a basic get and dumping this content out. Now great example, the URL class is doing a lot of stuff here, opening and closing connections, making sure we've got things formatted the right way, handling streams on the back-end, 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, and a great example of this pattern.

Exercise - JDBC

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 JDBC API. We're going to start with showing the complex client without using a facade that utilizes the JDBC API. Then we will implement the facade that will just have a client, the facade in between, and then our access to our JDBC API. And lastly we'll show the simplified client code in a demo that wraps up all the pieces as we put them together.

Demo: JDBC Without a Facade

Here is an example of some code using the JDBC API without a facade wrapping it, and you get an idea really quick of how busy this code is, or how much stuff is going on that 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 seeing that I used an instance of the singleton that we created in the Singleton module of the Creational Patterns course right here, and then we go down in our code and grab a Connection, and then a Statement, and 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 our result set to see if we grabbed a record from this table. You can just see there's a lot of stuff going on here and a lot of little loose ends. If you've dealt with JDBC very long, at some point you've had connections that were left open or various memory leaks because you forgot to close off resources when you were done using them. Let's go ahead and run this code now. And we can see there are tables created, we have 1 record that we created, and then we retrieved that record from the database. But let's see what a facade pattern can do for us for cleaning up this client API.

Demo: JDBC Facade

Looking at our JdbcDemo we can see that there are three distinct areas we're working with inside the JDBC classes. We have it where we're creating 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. So we're going to say New, Class, and we're going to call this the JdbcFacade, click Finish. Now inside of here we want to outline our three instances that we're dealing with, or our three areas that we're working with. So we're going to have a public int createTable, and I'm going to type this out because this really is the crux of this pattern, and then we're going to have a public int insertIntoTable, and then we're also going to have a public, and I'm going to return a List of Address objects, and we haven't created the Address object yet, I'm going to getAddresses. Now this is really the main pieces of the facade that we're going to be dealing with. So let's go ahead and fill this in now, and I've got this already done for us, so I'm just going to copy and paste some things in here, but first let's get an instance of our DbSingleton, and we'll just say instance equal to null, and this is the same singleton pattern that we used in the singleton demo, and let's create a constructor here, just a no-args constructor, so we'll say public JdbcFacade. And then inside of here we're just going to say instance is equal to DbSingleton.getInstance. Now for our createTable, basically what we're going to do is we're going to go ahead and copy the same code that we had at this JdbcDemo over here, but I've cleaned some things up a little bit. So I'm going to 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 createTable I'm going to get an instance of, grab my imports really quick, I'm going to take our database instance, grab a connection to it, create a statement, execute that statement, and then close my resources 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, it doesn't need to know about what the SQL was that was executed. Now if we wanted to change this and make it to where we could make it dynamic for any table, we could definitely pass a string in to our arguments here, something like this that would say sql, and pass that in for our client to use to create tables in a generic fashion, but you get the idea of what this method is going to do. And we're going to do the same thing with the insertIntoTable. 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 printing 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 JDBC, it has nothing to do with this design pattern, but it does have a concept of object-oriented programming is why I bring it up. The retrieval on 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, so a little inner class. We'll say class Address, and put an open and close on here, and we'll go ahead and throw some records in here, I've already got the class created. I'm going to just do an id, a streetName, and a city, and we'll just right-click and say Generate Getters and Setters, so we'll go Source, Generate Getters and Setters, yep, we want to select all of those, hit OK. Now make sure as you're doing this that you are putting everything in the right order here, so we have our objects all in the right brackets. And what I mean by that is that we now have our inner class down here below our main class declaration. If you haven't dealt with inner classes much, and it can be an external class, it's not a big issue, but you get the idea. For this type of retrieval I like to use inner classes often to pass back information. Now that we've got this object in place, we can do our select statement. So I'm going to go ahead and grab this code that I've already 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 going to import the result set. You can see very much like the insert record and the create table, we are going to go ahead and grab a connection and a statement, but then we also have our result set. We're going to execute this query, loop through it, and dump out what we have going on here. Now I need to do one other thing to the client here, I want to append 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 = new Address, and then I can say address.setId rs.getString 1, and we can do the same thing for the other fields. So we can say setStreetName and change that to getString 2, (Typing) save that, and then the same thing for the city. So we'll do setCity, say rs.getString 3. Now when we're all done with that we want to go ahead and add that to our addresses object. So we'll addresses.add, 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 JdbcDemo that we already used, and then our FacadeJdbcDemo, and let me put these two side by side and we can see here that our facade cleans up our client code a lot. So our facade demo is very simple, is 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 it, so our client doesn't need to know about any of the connection information, or our singleton or any of the other stuff that we've got going on. Let's run this and see how it works. We can say Run As, Java Application, and we can see that we had our table created and our record inserted, and I did go ahead and print this out twice, that was simply because we had our for loop here that was printing it out, and we also had it inside of our facade that was printing it out in our select statement. So you 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 facade is hiding all of this complex code, all these connection statements and all that 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 insertIntoTable statements and change that to take some string so it would work for any object and insert table we could make it take the address object in there, but this shows the simplicity that our client now experiences using the facade pattern.

Pitfalls

What are some of the pitfalls of the facade? Well it's typically used to clean up code that was potentially designed incorrectly or poorly to begin with. If you're using it in a new API or interface, you really should look at the design of your API and see if another pattern will help solve the problems that you are trying to solve with the facade. It typically shouldn't utilize inheritance, so the facade deals with a flat problem or structure. And it is the singleton of structural patterns, and what I mean by that is that it often is misused or overused because it is such an easy pattern to implement. You saw through our example we just put a basic class in and masked some of those things that were going on behind that 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 overuse 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

To contrast the facade pattern, let's compare it to the adapter pattern. The facade pattern simplifies an interface, and it works with typically just composites. It provides a cleaner API to something that was designed incorrectly or complexly to begin with. The adapter on the other hand is also a refactoring pattern very similar to the facade, but it modifies behavior and typically is used to 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 API or an interface up to something more current in your application, where the facade is just trying to make it easier to work with.

Summary

So let's briefly recap what we learned with the facade pattern. It simplifies the client interface, and that's really the only and main goal of this pattern is to make things easier and hide some ugly nuances of an API. 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 refactor things. And it's definitely a refactoring pattern. This isn't something you should design in up front, it's something that you want to use down the road after you realize that 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. As you'll notice in our JdbcDemo, our client had to know about connections, statements, result sets, and all that type of stuff, and after we put our facade in, the only thing it needed to 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

Hi, this is Bryan Hansen, and in this module we are going to look at the flyweight pattern. The flyweight pattern is a pattern that minimizes memory use by sharing data with similarly typed objects.

Concepts

The concepts surrounding why you would choose the flyweight pattern are when you need to make a more efficient use of memory. A flyweight 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 object state can be extrinsic. Extrinsic properties are ones that are not essential or inherent. Examples of this in the Java API are java.lang.String. Strings are immutable objects and are loaded from a string literal pool that is basically the flyweight factory. Caching is generally a hint also that you are or could be using a flyweight as well. There are quite a few other examples in the Java API, specifically java.lang.Integer, and the valueOf method, which is another great example of a flyweight. In fact, all of the primitive objects with wrappers such as Boolean, byte, character, short, and long have a valueOf method that is similar to the integers one that is a flyweight in action.

Design Considerations

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 flyweight objects after they've been created. The flyweight also often encompasses both the creation and structure of the object as far as the pattern is concerned. So it has a creational pattern inside of this structural pattern. It can and often does manage the entire lifecycle of the object. If you are unfamiliar with the factory pattern, you can watch the Factory Pattern module in the Creational Patterns course. The pieces of the UML diagram are a Client, Factory, Flyweight, and ConcreteFlyweight. Let's look at that UML diagram now.

UML Diagram

The UML for the flyweight starts with a Client. The Client is what is requesting the Flyweight object, although oftentimes it doesn't even know that it's a flyweight. It requests it from a FlyweightFactory. 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 ConcreteFlyweight is in the end what gets returned to the client, although it thinks it's just getting that object back. Oftentimes a client 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

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 cache that all strings in the JVM 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 wrapper object for primitives such as Integer also make a great use of the flyweight pattern as well. Here in this example when we call the valueOf 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

Here's the integer code that is utilizing the valueOf method as we can see here to grab an Integer object from a primitive value. You can see on the first two lines that we call valueOf with the valueOf 5 passed in, and get 2 different objects from that. Well as we run this, you'll notice that down here in the System.out.println we have a System.identityHashCode, and what this is a little helper class that's in the system package to go out and retrieve the identity hash code of this object. When we run it, you'll notice that the hash code for the firstInt and secondInt is the same because they're both for the value of 5, meaning that it's grabbing that flyweight object. And the third object, Integer thirdInt 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 first two objects print out the same object address, and the third one is a different address, and that's because the first one is creating it and storing it in the literal pool, the second one is just pulling it out of that literal pool, or that flyweight cache, and then the third one is creating a new object and retrieving that from the cache as well. So great example, and notice how your client really doesn't known anything unique or what's going on about the valueOf method, but under the hood it's a perfect implementation of a flyweight pattern.

Exercise - Flyweight

Now that we've seen a live demo, let's go ahead and create our own flyweight 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 flyweights, an Order, and an Item. The items will be the actual implementation of the flyweights, or our ConcreteFlyweight. We're going to implement the flyweights in a factory and we'll run a report to see how many objects were actually created for the orders out of our inventory management system.

Demo: Flyweight

There's a few little moving pieces and parts to this pattern, and rather than trying to code through it I've coded out the pattern, and I'm just going to walk through the individual pieces with you. To start with, we have an object named 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 flyweight instance, and notice that everything in here is final and immutable, and there's no setters or any of that type of information, we can just set the name through the constructor and then call a toString on it. We could put a getter in here if we wanted to, and you could add other attributes to this class if you wanted, you could have a UPC code or description or a link to an image, it 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 neat, it really doesn't have anything particular to do with the flyweight pattern other than Order is going to consume Items. So you'll notice that the Order object has a constructor for an orderNumber and an item to be passed in to it, and then we process our order, and this is just to show that we're doing something with it. Now the Catalog is where things get a little bit interesting. The Catalog, which is how we view, or you could also think of it as your search, how we go through and look at our items, it contains the HashMap of our items and the factory method to see if we have these, and if we don't to create them and what we want to do with them. So it has some very interesting attributes about how this gets handled and whether or not the client is getting back a cache event 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 I want to talk about before we go to the client is the InventorySystem. And the InventoryStystem, and this is just mocking out a much more robust one on the back-end, has an instance of the Catalog, 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.totalItemsMade, so it will go through and look at how many items we made from our Catalog object, which just contains the size of it. So for our client, we're going to go through and create an instance of our inventory management system, or basically bootstrap our inventory management system, and start taking orders. And now I've just got some basic strings but in there, we could something more sophisticated, but we can order Roombas, 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 a Samsung TV. When we take all those orders we're eventually going to close out that system or close out the report, so we're going to do a process, and then we'll print out a report of that. Let's go ahead and see how many items are created. 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 lookup from the catalog for that itemName. So the itemName that gets passed in is a Roomba, a Bose Headphones or a Samsung TV. Our inventory management system takes the order and looks it up out of the catalog and says oh, I'm going to 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 HashMap. If it contains it, it just returns that instance and you're good to go. Now when we go back to our inventory system, it just adds those, it creates a new order each time, and it 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, you 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 orderNumber associated with it. 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 to 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 million orders if we only had three of those objects, it would only create three of them because that's all we had to choose from our flyweight objects. So you can see how this can be a very big performance boost for your system, especially if you start looking at some of your bigger online retailers, such as 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

What are some of the pitfalls of a flyweight? It is a bit of a complex pattern. As you noticed as we went through our example, there's a lot of little moving pieces to utilize it the right way and you have to deal with ah 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. And 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 up front, 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 flyweight is part of the factory of what the factory is returning, and you get the example, it just becomes a little bit more complex. If you're looking at other examples out there, a lot of them are graphical examples. Its value is well beyond graphical implementations, but a lot of the research would lead you to believe that this is where its only value is at. You can see through our example that we would utilize this in any sort of management system, 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 utilizing, 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'll look at out there definitely are graphical centric.

Contrast to Other Patterns

To contrast the flyweight pattern, let's compare it with the facade. And honestly there aren't really a lot of other 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 an optimization pattern and deals with immutable objects. The facade on the other hand is a refactoring pattern, and is usually implemented after the fact. So the flyweight is definitely implemented up front and early on, if not, you're going to be refactoring a lot of code, where the facade is definitely something that's thought of after the fact. The facade is also centered around making a simplified client, or making it easier for the client once we realize that the API we're dealing with is 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 knot 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 the facade to provide a more simplified interface where the flyweight was designed up front to already deliver a simple interface to the client.

Summary

Let's do a quick recap of what we learned with the flyweight 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 we're 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 be a little bit of a complex pattern to deal with, not necessarily 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 it 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 API, 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.lang.Integer valueOf or Byte valueOf, or Character valueOf or strings are all great examples of flyweights, 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 in this series, 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

Hi, this is Bryan Hansen, and in this module we are going to look at the proxy pattern. The proxy pattern is a pattern that acts as an interface to something else.

Concepts

The concepts surrounding why you would choose the proxy pattern are when we want to wrap a real 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. A proxy can be used to solve multiple problems such as security or simplifying an interface to something, a remote service call, or just an expensive object to create, we'll oftentimes wrap it with a proxy and display Loading Message or something like that for an expensive object that we're trying to display to our UI or just load into memory. The proxy itself is called to access the real object. So we'll have an interface, then a proxy that's wrapping the real object, and then the underlying real object. Examples of a proxy in the Java API are actually kind of interesting and different from some of the other patterns that we've looked at. The java.lang.reflect.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 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.lang.reflect.Proxy. Also the whole java.rmi package is focused around proxy and remote method invocation, so as we looked at the concepts up above, one of those being remote, the java.rmi package is all about accessing remote objects and retrieving that data across the wire.

Design Considerations

The design of the proxy is simple in concept, but it can 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 Java API recognizes the need for the proxy pattern, and incorporated an interface, the InvocationHandler, and a class, the java.lang.reflect.Proxy class to facilitate this. The pieces of the UML diagram are a Client, an Interface, InvocationHandler, Proxy, and its Implementation. Let's look at that UML diagram now.

UML Diagram

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 class is that we want to retrieve. The Proxy, using an InvocationHandler and the Proxy class in Java, intercepts that call and makes the call to the RealSubject, or if it's a case like security would deny it or do something different, and turns around and sees if it needs to load that, if it's going to pull it from a cache, or whatever it's going to do, so it then decides that, yes I am going to load this RealSubject or this real object back, and returns that back to the Client. And 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

This example is going to be a little bit different than some of the other examples we've done for other patterns. We're going to actually use the Proxy class in this example and create a small sample that we're going to tie into a real application to pull a message back from Twitter. So to start with, we'll walk through the InvocationHandler and the Proxy class to build out a small piece, and then we'll continue that on for a larger example.

Demo: Proxy

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 we'll have the TwitterService interface, and go ahead and click Finish, and inside of here we're going to add two 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 two methods, a getTimeline and a postToTimeline. And 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, we're going to break up, we're going to start off by creating a stub, which is just a method for us to try something out that isn't going to call the real service itself, we're 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 going to call this the TwitterServiceStub, and it's going to implement our TwitterService interface, and click Finish. Now it has two basic methods inside of here that it's overridden to automatically generate for us, let's just return a string, and we'll just put something in here, some fake text, My neato timeline. And we can get rid of this TODO for now and just leave this blank. So now we have our service and our stub, we're going to swap this out with an implementation 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 the SecurityProxy, and we're going to make this implement the InvocationHandler 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 InvocationHandler interface. So let's go ahead and get rid of this TODO out of here, and I'm going to copy in some code that I already have written for the SecurityProxy just so you're not sitting here watching me type. So I'm going to go to my demo-notes and pull in a newInstance 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, the object that we're wrapping with the proxy inside of here. So we're going to go ahead and say alright, let's call this constructor with an object, we'll say private Object obj, and we'll make a private constructor, SecurityProxy. We're basically making a little factory pattern here. We'll say Object obj, this.obj = obj, 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 to do is go ahead and handle any exceptions or other things we have going on in here, so let's go ahead and hit Ctrl+Shift+O 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 went ahead and copied and pasted in here. So we implemented the InvocationHandler, and that forces us to have this @Invoke over here. In fact I could put the @Override annotation on here for the compiler and save that, but another thing we did is we turned this into a small little factory that we're going to go ahead and grab an instance of this object back. And this is where the proxy pattern comes into play. So you'll that we return a java.lang.reflect.Proxy.newProxyInstance where we pass in the obj.getClass getClassLoader of this and the interfaces. So it builds this proxy around the class and any interfaces it's implementing, and then returns a new SecurityProxy object that it's wrapping. And then down below the InvocationHandler is where the magic happens of us invoking these calls. So this is where the proxy really come into play. So we have our proxy up here that we're going to create, but this is what tells it what methods we will or won't invoke on when we call this. This is where we'll eventually add some security around this. As it stands right now, we're saying anything you do I'm just going to 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 pass 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 going to go ahead and now right-click and say New, Class, and I'll check the public static void main create checkbox, and we'll call this the TwitterDemo, and Finish. 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 our stub that we created. So first we want to start off by creating an instance of the TwitterService to assign our object to, so we'll say service =, 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'll say TwitterService, and we want to do SecurityProxy.newInstance, and we're going to pass in to that a new TwitterServiceStub, and we want to go ahead and put a semicolon on the end of that line. And we can now just say System.out.println, and we'll pass in the service.getTimeline, 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. So let's go ahead and run this now. I'm just going to right-click and say Run As, Java Application, and you'll see it reported back the My neato timeline that came from our Twitter stub, 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 we'll take and replace this TwitterServiceStub with an actual call to Twitter, and I've got to pull in a third-party library for that, but it's a real world example of something you can do and see how that SecurityProxy comes into play.

Exercise - Twitter Proxy

To continue on with our proxy example, we're going to go ahead and download an implementation of the Twitter API 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 class called the TwitterServiceImpl, it could honestly be named whatever we want, but typically you have the service and then 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 API, but you'll see how you can hook insecurity in that InvocationHandler in the Proxy class that we created.

Demo: Twitter Implementation

To get started to actually make our implementation call the Twitter service, we're going to need to download a third-party JAR, and it's a small JAR, it's real simple, it just makes this example a lot easier to use. So I'm going to go ahead and come out here to twitter4j.org and click the Download link, and it'll take us down here to the latest stable version. Go ahead and click Download. I already have it downloaded. Once you have yours, you're going to unzip it into a folder like this and go into the lib directory, and all we care about is the twitter4j-core example. So I want to go ahead and drag that over into my lib folder, and then 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 as Maven, a Maven project inside my IDE, but to keep it as simple as 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 an 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. So I'm going to switch back to my browser and go to dev.twitter.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 to the very bottom and you'll see a link for Manage Your Apps. Once we go in here, it'll say You don't currently have any Twitter Apps, go ahead and create one. So let's create a new app, and I'm going to do Pluralsight Proxy, and This is an app to demo calling a proxy that calls twitter. And for the website you want to just put http://, yep, that'll work, www.Pluralsight.com/author/bryan-hansen, you can put your own personal website in there. And we don't have a callback URL, so that's fine. Yes, I agree, Create your Twitter application. Now this is going to 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 going to 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 your on GitHub, sorry, gist, jist, you can call it whatever you want, to where you don't have to type all this code in. It doesn't add anything to the example, but I wanted this to be a real world example. So I'm going to go ahead and copy this code in, and I'm going to go over to my IDE and create a New, Class in our proxy package that is the TwitterServiceImpl, 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 ConfigurationBuilder and takes our keys, and our TwitterFactory and StringBuilder and all that stuff, and paste it in here and finally it returns that out to our client, but you'll see right now this is where we're going to put those keys. So let's grab that Twitter key, there's my consumer key, copy that, and make sure you copy it, and paste that over your put values here, your secret key, and then we'll have to generate two other tokens, that's why I didn't just copy these in here. So you're going to come down here and say you have no access token. So I'm going to 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 token secret, copy that, and now we have everything put in place to do our call. So the steps there we had to download from twitter4j.org, unzip it and grab the core library, we had to go to Twitter, log in, and eventually go to manage our apps, go down there, create our Pluralsight proxy app, and then generate our access key. So we had our consumer key, our consumer secret, and our access token and access secret, and then I had the TwitterServiceImpl gist here, and this will always exist here. In fact I'm going to go ahead and copy this URL for you, and it's a public gist 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 URL, that'll always exist for you out there. Now let's go ahead and go back to our TwitterDemo public static void main method, and instead of doing the stub, we're going to replace that with the Impl, and save it. Now when we call this, we should get what's on my timeline back. I'm going to right-click and say Run As, Java Application, and it went out and pulled the most recent things off of my timeline, or anywhere that they could search and have @bh5k 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, and this is why your dependency injection frameworks do this, is now I can control whether or not this object gets created, it's lazily loaded, if it's a remote resource and there's a big expense to it we don't pull it in until we need to, we can wrap security around it, so you can see there's a lot of benefits around having this proxy in place, and we did it with a real world example. The real catch is just making sure you get those keys in the right way and creating that application. Now if you try these keys that I put here, they're not going to work, you have to replace these with your own value because I'm deleting these off my account right now.

Demo: Security Proxy

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, because currently if we call it, it will let that invocation go through. I'm going to show you by modifying the TwitterDemo class that it will do just that. So if I call service.postToTimeline and give it a screenname, bh5k, 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 Impl 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.out.println in here just to show that it is in fact happening, and let's go ahead and say, oh, let's just paste the message in there and save that. Now if we call our TwitterDemo 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 and now doing that. Now to implement the security, and we're just going to do a poor man's security here, you could do a real full-fledged security interface like Spring security or something else, look at some 509 certificates or OAuth or whatever, I'm going to open up our SecurityProxy 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 the things I can do is I can come down here and I can say m.getName, and this will return the name of the method I want to call. So we're going to just say if m.getName.contains, and we'll just pass in post, then let's go ahead and we'll close that if block off and say else we'll allow it to go through. So if it does post, then we'll throw an exception. So let's go up in our if block here and say throw new Illegal, and let's do IllegalAccessException, and Posts are currently not allowed, and close that off. So all we really did is grab 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 CRUD functions, or 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 call for us. So now if we go back and run our TwitterDemo, right-click, Source, right-click, Run As, Java 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 such. So it's a real nice feature and it shows the real power of that proxy where we're intercepting those calls in the middle.

Pitfalls

What are some of the pitfalls of a proxy? Well you can only 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 chain or wrap, 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 if 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 add a little confusion if you're trying to determine which pattern you should be using.

Contrast to Other Patterns

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 functionality is set at compile time, it's not really the focus of what the decorator is. We are going to determine up front what class we're trying to call a remote call to or whatever other type of interface we're 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 too. I 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 in its hierarchical chain. And the other contrast is that its functionality is usually determined at runtime instead of compile time like the proxy. We're going to go ahead and chain these things together at runtime, and that object will figure out then what it can do.

Summary

Let's go ahead and recap what we've learned about the proxy pattern. There's great utilities built into the Java API to implement the proxy pattern. You can see there's the Proxy class and the InvocationHandler interface that 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 to, 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 proxy as you're adding different features to it. I should also point out that this is used a lot by various dependency injection and inversion of control frameworks, so if you're using Spring or Guice, or tools like that, we briefly touched on it, but 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 at a 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?

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 and a great asset, and I hope you'll refer to this back in time and look at it as you get more experience with patterns and refresh your 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 Creational 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 for patterns as I've already had requests from other people to do so. I've got many courses out there as well on Maven Fundamentals, Spring Fundamentals, the Spring MVC and Spring MVC 4 Intro courses, as well as Spring with JPA 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 me through the author aliases on the Pluralsight website or my Twitter handle, which is on all the slides, or connect with me on LinkedIn as well.