Course Overview
Course Overview

[Autogenerated] Hi, everyone. My name is Brian Hansen and welcome to my course designed patterns in Java. Behavioral. I am the director of development as software technology group as well as a plural site author. I learned about design patterns almost 20 years ago and have enjoyed teaching and sharing them with people ever since. In this course, we're going to cover the behavioral patterns as identified by the Gang of four, the Gang of four considered to be the fathers of design patterns as we know them today. Some of the patterns we will cover include ones that you may have already heard off, like the command pattern, which is used to isolate individual commands in your application. The observer pattern, which you use in a published subscribe event model. And the Temple of Method, which you may already used more than you actually realize. These are just a few of the patterns that we will cover in this course, and by the end of the course you'll have seen an example of each pattern from the job. A P I written each pattern from scratch and contrasted each pattern with another one to showcase its strengths before beginning this course, you should be somewhat familiar with Java and comfortable using an I. D. You can continue your learning by exploring other patterns. Courses focused on creation, a LL and structural design patterns in the plural site library. I'll hope you will join me on this journey to learn design patterns with the design patterns in Java behavioral course at plural site.

Behavioral Design Patterns Introduction

Version Check

Introduction

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

Why Learn Patterns?

[Autogenerated] 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 know how to solve a particular problem, and it might follow the structure of a pattern. But it is better to have a common vocabulary that you can explain to someone what that 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 particular problems than another. You should revisit pattern material. Whether it be this course or a book or something else is amazing how your perception of a pattern will change. After gaining more experience with it, you might come away with a different understanding of it after

applying it. Ah, lot of people have used the single 10 and they think that they have used design patterns. There's much more than just a singleton pattern out there

Pattern Classifications

[Autogenerated] the gang of four has Pattern's broken out into three groups. The groups are creation A ll, structural and behavioral. This course is going to focus on the patterns classified under the behavioral group. Behavioral patterns are focused on how objects interact with one another. Often times were trying to be more loosely coupled, and that leads to is just looking at behavioral patterns.

Which Patterns?

[Autogenerated] the behavioral group contains the most patterns out of the gang of four patterns. We're going to cover the chain of responsibility. The command interpreter, it aerator mediator, Memento observer, state strategy, template method and finally, the visitor pattern.

How Do We Learn Them?

[Autogenerated] to help facilitate how you learn. The patterns were going to go over an overview of what the pattern is, the concepts that you would look for when choosing the pattern. The design of that pattern. Look at a live example from the job a p I. They were to go through and do a demo which will ultimately end up coding your own pattern. And look at the pitfalls of what this pattern has. Contrast that with another pattern that you may be more familiar with and then finally summarize what we've learned about that pattern.

Prerequisites

[Autogenerated] The prerequisites for this course are quite simple. I am using Java seven, but it should work with any of the more recent versions of Java, and I am doing all my examples in Spring STS. But the functionality that I'm utilizing is actually the same as Eclipse.

Next

[Autogenerated] Although the patterns are independent of one another, we're gonna go through them sequentially. How we described in the what we're going to cover section, which is the same order that's defined by the gang of four itself. The first pattern we're gonna look at is the chain of responsibility pattern.

Chain of Responsibility Pattern

Introduction

[Autogenerated] Hi, This is Brian Hansen. And in this module where you're going to look at the chain of responsibility design pattern, the chain of responsibility pattern is a behavioral pattern that d couples a request from a handling object in a chain of handlers until it has finally recognized.

Concepts

[Autogenerated] the concept surrounding why you would choose the chain of responsibility pattern or they want to decouple the sender and receiver objects often times in an application. We want to pass a request to a receiving object without knowing who the center was and vice versa. The center shouldn't have to know who the receiver was in order for to process that request. When using the chain, the receiver should also contain a reference to the next receiver or the successor. This is an important part when choosing and implementing this pattern. It doesn't know the whole hierarchy, but it does know who's next in line. One of the main reasons for choosing this pattern is to promote loose coupling. We can modify the chain and add links to the chain without rewriting large portions of logic in the application. It should also be okay that there may not be a handler for a given request, and the application will just continue on examples of this in the job, a p i. R, the Java Util, logging Logar, a p I and although not part of the core job, a p i thes serval ap I specifically filters is another great example of chaining one of the best examples I've seen This pattern in use, though, is how spring implements their security chain filter in spring security. It's a great example of chaining.

Design Considerations

[Autogenerated] the design of the chain Responsibility has a chain of receiver objects. This can be implemented in a number of ways, but some basic form of a list is typically the most common. Each handler is based off of a main interface that defines the contract between each chain link. There is a concrete handler for each receiver or implementation that will interpret a request in building the chain. Each handler has a reference to its successor or the next link in the chain. The pieces of the UML diagram are a handler and a concrete handler and then its successor. Let's look at that UML diagram now.

UML Diagram

[Autogenerated] This is the UML diagram for the chain of responsibility pattern. The client makes calls to a handler class, and that handler is a common interface or contract for all of the handlers and is the abstraction for all links in our chain. There are multiple concrete handlers for each link in our chain and, although not really clear in the UML diagram, but the handler will pass it to the next handler in a chain if it doesn't know how to process that request. This is known as the successor and is one of the key parts of implementing this chain

correctly. To help gain familiarity with this pattern, let's go ahead, look at an everyday example of it in action.

Example: Logging

[Autogenerated] If you look very far for an example of the chain of responsibility pattern, you'll quickly find an example of logging. I didn't really want to use this in the example that we're going to develop together. But do you still think it is a great example of this pattern and decided to show it for an everyday example? The various logging methods are passed through a chain, and if there is a handler, it will log. Otherwise it will pass over it. Let's look at this in some life code now.

Demo: Logging

[Autogenerated] Here's the code for that Every day. Example, we were talking about using the logger, and you can see it's pretty simple. We create an instance of the logger by just using the get logger based off of a string name. And then instead of our code, we begin setting up the chain to handle the logging levels. So we set our logar to a level of finer. And then we create a handler, one of our concrete handlers, to also have a level of finer. And then we add that handler to our original logger. So we start building that chain now. If he had multiple handlers, we've got one for the console. We could have one to a file or some other cysts, log or other logging handler. Doesn't matter. There's a lot of various implementations we could build that chain to go through and build those handlers up. The pieces we have down below here are our actual logging statements that we're going to see what gets printed out. We have finest finer and fine three of the seven or eight various levels that are available to us as a logger, and you'll notice that we set our two finer. So our chain ends with finer finest ones. They're going to fall off the end of that request cycle, so nobody set up to handle that request. Go ahead and run this and see what prints out you can see that are finer statement and fine statement both printed out, but the finest did not. And that's just because we don't have a handler that's set up to go to that level of logging, so their link of the chain doesn't exist. Let's go ahead and create our own example of this now that's a little bit more complex than just this basic logger example.

Exercise - Chain of Responsibility

[Autogenerated] Now that we've seen some code in action, let's build our own chain of responsibility. We're going to implement in the handler show what the successor is and process a request throughout our chain. We're gonna start off by creating the handler and then from there, build the rest of our chain out.

Demo: Approver

[Autogenerated] So this exercise is actually based off a real world example that I have in my office that I work with. I have a director and I have a VP and a CEO. And for purchase approvals, it has to go through that hierarchy. There's certain things that as a director, I can

approve. But when it gets to a certain dollar amount or a certain type of purchase, it needs to go to a V P. And then if it's above that, it needs to go to the CEO. So there's a couple things you can see inside of our demo here that we're building. I've got a director of VP in a CEO, those air, all types of handlers, and then you can see that I have to set the successor. So as a director, my successor is the VP, Crystal and Crystal. As a VP, her successor is the CEO and that is Jeff. So let's go ahead and look at the handler class first. That all of those director, VP and CEO are going to inherit from the handler has two things in here that are key. It has a successor and a center for that successor, and then it has a method for handling the request. So any handler, any concrete handler we have We want to implement this handle request method. Now, requests, as you see we have in here, is just a basic class that I put together that has a request. And it's a request type in an amount associated with it. And I really like venom. So I put the request type and there is an n um, we could have conferences or just a regular purchase that we want to go ahead and approve. Now the director, I've gone ahead and implemented Director extends handler, and in doing so has to implement the method handle request. And you can see where this behavior comes in. Now a director, I can approve conferences so immediately I look to see if Okay, what type of request is it? Well, it is a request type of conference. Great. Let's go ahead and say that I have approved this conference now. Now, obviously, we would probably do something more than what a system now print Lin is, but you'll get that. That's where I can put the business logic in here for talking to the database or what it really means to a prove that conference. But if I don't have the ability to do that, I can go in here and change it to pass to my successor. Seeing see the else of that If Block is passing that off to my successor and in our demo, my successor, Waas Crystal. So we set that up on that line right there. Now, the VP, I have purposely left empty because I wanted it to implement this together. So to do that, we would go ahead and override that method. I'm going to actually pace that code in here. So I'm gonna override that method and I'm going to say that well, the VP can She can approve purchases and they had to be in a purchase of $1500 or less, you know, under $1500. Other than that, we want a hand it up to the CEO. So now we can go through and say, Well, it's over 1500 is gonna go to the next successor. And if you'll remember that next successor was Jeff was the CEO. Let's go ahead and save that. The final one in our chain is the CEO and he can handle anything he wants. He can approve anything he wants, but you'll notice that we have that business logic now across those three classes. Now that's where we got that loose coupling in there. I could add another. Approve her in the in the middle. So let's say we have somebody who approved stuff up to $750 orders, conferences and small purchase orders, or vice versa. I can add links anywhere I want in the chain, and the's classes individually don't have to be reprogrammed, but you get that. It can be a little bit questionable as to where that logic gets handled inside your application. So you need to be diligent when building your chain that you're setting it up right. This is a good example, though, because this is a real world example that I have used this pattern in. Now let's go ahead and run this. You can see that we're going to build a request of type conference. It's $500 say, yet Brian can handle that request. But you'll notice that I can also say I'm gonna let Brian handle this other request But it's up to me as a concrete handler to say, Oh, I don't have permission to do that. I'm going to go ahead and send that up to my successor. So the client that's calling it doesn't know. They just know that their requests got approved. Let's run this now so you can see the first thing that gets printed out is that directors can approve conferences, which I did go through an approved that the second thing is is that V P s gonna prove purchases below 1500 that CEOs can prove anything they want. But it wasn't necessarily me that handled that. If I swap this out with Krystal, the same thing will happen in the same code, will get printed out. But how it handled

that in the chain. It's transparent to the client. The client doesn't need to know about that. So it saves the client from running around in this example of running around to all the different people in the office to see who they should talk to, to get approval, and instead they just talked to one person and that one person knows who to escalate it. Thio, in order to get their request approved, do you see how that hierarchy ties in with that chain of responsibility?

Pitfalls

[Autogenerated] Now we've seen how great that chain of responsibility pattern can be. We did start to touch on some of the pitfalls or problems with this pattern. Let's go into that a little bit deeper. One of the pitfalls that you run up against is tthe e handling handler guarantee. We aren't guaranteed that someone along the chain will in fact handle our request. This is a direct reflection of our runtime configuration. We have great flexibility with this configuration, but it could mean that there's some configurations that haven't been tested and something might not get processed. There is also a concern about chain length as well. If you just keep tacking on handlers, it can get quite large, and the performance could start to degrade. This usually isn't a problem in my experience, but it is something to keep in mind. Let's go and compare this with another pattern. Know to see what it might be a good idea to choose it or something else

Contrast to Other Patterns

[Autogenerated] to contrast the chain of responsibility pattern. Let's compare it with the command pattern. The chain of responsibility is a pattern that deals with handlers, and each one of those should be unique. They also know about their successor. This is key because the pattern doesn't know about other commands surrounding it. The chain of responsibility will pass it on to its other successor, for it doesn't know what to do with it. The chain can oftentimes also utilize the command pattern in its implementation of those individual concrete handlers. The command pattern is actually quite similar to the chain responsibility and that its commands are also unique, just like the handlers. It's different, though, and then it it should encapsulate all of its functionality. It doesn't attempt to hand it off to somebody else if it doesn't know what to do with it commands air, also reversible or trackable in nature. We will often store a history of commands, and we don't do this with the chain of responsibility. In fact, that is a risk, as we mentioned in the pitfalls that we could have a chain not handle a request. It all and we don't know about it Although the pattern doesn't enforce this, we usually call a command because we know it will handle it. But the chain, we just send it down the chain, assuming that somewhere somebody along the way will handle it.

Summary

[Autogenerated] Let's just recap what we've learned with the chain of responsibility pattern It d couples, thesent er and receiver from requests. So the center doesn't have to know who's going to handle its request, and the receiver doesn't have to know who the sender was necessarily. You can configure this at runtime, and that can be helpful, or it could be dangerous as far as your applications concern. Usually that flexibility is looked upon as a positive thing, though it is high article in nature. In other words, it's going to build a zit goes

down the chain As to who may be able to handle that request. You do have to be careful with large chains because it can become a performance bottleneck. Or it can be confusing where that business logic might be configured inside that chain. The next pattern that we want to look at is the command pattern in the command pattern shares some of the similarities, as we mentioned in the contrast section of this course

Command Pattern

Introduction

[Autogenerated] Hi, This is Brian Hansen. And in this module where you're going to look at the command design pattern, the command pattern is a behavioral pattern that lets you encapsulate. Each request is an object. There are a lot of reasons as to why you'd want to do this. Let's look at some of those concepts.

Concepts

[Autogenerated] the concept surrounding why you would choose the command pattern or that it encapsulates. Each request is an object. If you have dealt with or are going to work with a large system, you'll quickly find that the business logic and functions inside that system can be very complex to maintain a debug if they're all just added in one file. Another key reason for choosing this pattern is that each call back a request is now object oriented instead of just another method added in sight of that growing class. Maintain ability is also increased because the sender is decoupled from the processor. This will enable the system to be more flexible and grow over time, often times, but not the only reason. You will use a command to add undo functionality to your application. The entire request should be contained within the command and then could be rolled back. Examples of this in the job a P I. R implementations of java dot laying dot rentable each implementation. It adheres to the principles of the command pattern. Another example is any of the Java ex dot swing, the action implementations Years ago, swing actions were probably the first place that I personally used this pattern and have since used it in many Web frameworks and applications that I've written on my own.

Design Considerations

[Autogenerated] the design of the command is a little different than some of the other patterns and is sometimes argued that it breaks the principles of, oh design because there is an object per command, A command is a verb, and objects usually aren't verbs, but rather the methods inside them. But people have seemed to relax their view on this. The main contract of the command is the command interface, all implementations of actions or commands inside the framework will implement this interface and in its simplest form, just contains an execute method. This method is where all of the action is performed. In the case of an undue feature, though, the interface will also contain an UN execute or undue method. But this isn't required to adhere to the principles of this pattern. Advanced implementations of this pattern

make use of reflection to completely decoupled the client from the receiver or processor using a callback. Most examples you see, though, are simpler than this version, and we're going to look at various examples is to see howto best exercise this action and implement it in your day to day use. The pieces of the UML diagram are a command and invoke er and a concrete command. Let's go ahead. Look at that UML diagram now

UML Diagram

[Autogenerated] This is the UML diagram for the command pattern. The command interface is the heart of this pattern. It provides the structure for the invoke er too abstractly execute on concrete commands. The receiver in this has an action method that the execute command is performed on the abstraction is that the client just doesn't call the action on the receiver directly, but rather works with Ian Voelker to issue the command. The action is decoupled from the client. Let's look at an everyday example of this using the java dot laying dot rentable a p I.

Example: Runnable

[Autogenerated] the rentable interface is a great example of the command pattern in the job a p I. We create a thread and pass an implementation of the rentable interface into it. You can see here that the request is completely decoupled from the processor. Let's look at this in life code in an example now of how the thread and rentable interface decouple the actual execution or action inside of our code.

Demo: Runnable

[Autogenerated] for our threat example. I've gone ahead and created a task class, and I've put these two in the same file just so you can see him side by side. The task class implements run herbal, which in our case, would be an example of the command pattern or the command interface. The request is completely encapsulated by this task object. You can see we pass our parameters in, and that is our entire request. And then we have an in Voelker, the thread that calls start, which in turn will invoke the execute method on our task. Now, we don't have much of a receiver in this example, because all I'm doing is multiplying a couple numbers together. What we're trying to do isn't really important as illustrating what the pattern does. The receiver in this case would be the consul, though, because I'm going to dump out the result to the system got out print Lynn. So as we start this, let's go ahead and run this code. You can see that it goes ahead and multiplies those numbers and dumps them out to the command line. Now, I could swap out this receiver with something different and have it act on that receiver object, but you get the principles of the code here. I've got the rentable interface, which is the same thing as our command interface. I've encapsulated our request. We haven't invoke er we execute the method and we have a receiver which are really all the principles we need in this command pattern. Now that we've seen an example of it, though, let's go ahead and create our own to further cement these concepts.

Exercise - Command

[Autogenerated] in creating our own command pattern, we're going to create a command and invoke er, a concrete command and a receiver to help us illustrate the principles of the pattern we're going to then extend that example toe look at how he managed state, and then we'll show what a macro command would look like inside of our application.

Demo: Switch

[Autogenerated] to shorten the length of this exercise, I've gone ahead and created a couple of classes to start with and one interface. The interface is the command interface, and it has one method in here the execute command. The next piece that I've done is the actual demo that we're going to run where we create a light, which is theory receiver a switch, which is the invoke er and then a command to flip a light on and off. So for this example, we're just going to control a light with a switch in an on command. Now, if you look very deep fourth command pattern, you're gonna find an example similar to this all over the place. But I'm gonna point out a few places where this fall short and how we're going to expand on it to make it a very useful command pattern. But its roots, it's great. And it's an example that I have personally used as I built a new house. I put some automation stuff in there, and this is exactly what I did for it. So to start with, let's go ahead and create the light class, so I'm just gonna go ahead and say, Create class light and we'll click. Finish on it. And just first sake of documentation, I'm gonna note that this is the receiver so you guys can follow along, gonna add two methods and hair public void on, and this is what the action is going to be performed on. So we're going to say system dot out print Lynn. Now, if you were depending on what type of automation package you had, this is where would be where you toggle something with the switch to go on or off. So we're gonna say lights switched on and we'll save that and just why we're here. Let's go ahead and put it off in there, too. We'll see off and we'll do the same thing Will say system dot out print Lynn Lights switched off. Now that we have that in place, let's go ahead and refer back to our main method where we need a switch or the invoke her in this. So the switch is pretty straightforward. Let's go ahead and say that we want to create a class switch and finish. And here there's two ways that we can do this. You can put Justin execute method. Or you can put separate storage mechanisms. Sometimes you'll see implementations where they manage the command that weaken. D'oh! For our simple example, we're going to store the command and execute the command in the same call so we can say public void, store and execute, pass in an instance of command and then actually execute on it. Command dot Execute. Now, I am not doing anything with storing it right now. We'll expand on that later. For this example. We just want to toggle that switch and this works. So for documentation purposes, go ahead and throw at the top of this that we've This is the in Volker and our UML diagram. Now we're almost done. We need to do one other piece here. We haven't created our command are on command yet. Now the command interface, if you remember from are you are you are you Mel Diagram doesn't do anything with state, but the concrete object itself does. And it has a receiver or a callback that is passed into it. So to achieve this in our concrete command, go ahead and create it Now, quick finish and full just documented the top here, Concrete command. We're gonna go ahead and pass in an instance of light to its constructors Will say private light, light and we'll say public on command light light. Now all this is going to do is just tie that receiver to us so that we can do the callback on it. So we're going to say that this on command is tied to this light. So this light equals

light. And now inside of our execute method, all we need to do is execute what we want have happened since this is the the on command I'm going to say light dot on. And we've now wired up all the pieces. Now, notice the client issues a command that turned the light on instead of the client directly calling the light dot on and that so that the command can manage state it can handle on do functionality. We haven't added that to this class yet, but you can see how this ties into all of those pieces. Now, if you go back to our demo, all of our pieces will work. We create our light, we have our switch, we creator on command and then we execute that command. Let's go ahead and run this now. Say, run as Java application and you'll see that our light is now switched on. So really quickly. Let's walk through those pieces again. We created our command in her face. We created our light, which is our receiver. We have our switch, which is our invoke er and then are on command where the meat of this really takes place and you can see inside of here. This is what's actually doing the callback on the light object. And then for our demo, we just went ahead and did the store and execute and ran.

Demo: State

[Autogenerated] one of the things I don't like about our current implementation of the command pattern is that state is nowhere stored in any of the objects and if you remember from our uml diagram we can store state in our command or in the receiving object. Let's go ahead and modify our code to do that. Now to do this, we're gonna change the light object and we're going to create a new command. So let's start off by going into the light object and adding a private bullion is on flag and start off by setting that equal to fault. Now, I'm going to go ahead and just say public void. Toggle instead of having the honor off command that we that we're going to call from our commands. So let's go ahead and say yeah is on then will call the off method and set is on equal defaults else we will call on and set is on equal to true. Now this uh this toggle is going to be used inside of our command object now. And instead of having an on command, I'm going to create a toggle command and Then delete this one. So let's go ahead and say this and say new class. And we'll call it the toggle command and click finish here toggle and save this. Now I copy and pasted that over rather than just recreating the code and it could have re factored it one of a couple of ways, but rather than call light dot on. Now, I'm going to call light dot toggle and save this and I wanted to do this to show you the two side by side that they're very similar in nature from one another. So what we had before, in fact, let's make it full screen so that you can see them easier. You can see the two side by side. All it's doing is tripping the toggle rather than the on. So I'm going to go ahead and delete this eventually. But I wanted you to see him for the two examples and I would make these both private as well. But that will break the current code that's there. So now let's go into our light again. It's going to just toggle whether it's on or off and go to our demo and we'll say toggle command. Now this code is set up to toggle whether the light is on or off. We can call it a couple of times. Let's go out and run it and see what it does now for us. Yeah. You see that we switch the light on, then off, then back on again. And I don't have to create multiple commands for that. And the state of the object is actually contained inside of the receiver and in conjunction with the command. Now I don't have a stateless application. I've maintained some state there. The next thing we're gonna look at is issuing multiple commands to turn off multiple lights inside of our house. So you can see building upon this command pattern that we want to go through and say, All right, I've

got three lights. Now. I want to turn those all honor or all off at a certain time, and we can keep building using the building blocks of the command pattern that we have.

Demo: Macro

[Autogenerated] so now that things are working real nicely with our command pattern, let's go ahead and implement a macro command to turn off all of our lights so that when we're in bed at night, we don't have to go around and switch off all the lights, which is in our house, or grab our remote control and individually click them. We can do them all with one command. So to create a little more in depth of an example, we want to go ahead and add a few more lights in here. The first thing that we're going to dio is change this to where we have a bedroom light and we'll go ahead and create a kitchen light as well. So we'll say kitchen light. Everything works fine there now notice. As it stands right now, we don't have multiple light switches that we're doing things with, but let's just run this. Make sure example still works fine. Yet our life switch was turned on. Then it was off than it was on again. Now let's change this up a little bit to create that mackerel for all of those lights. Just start off by commenting this code out and I'm coming it out because I'm going to illustrate a problem that we're gonna run into later. Gonna save this. Now, the first thing we want to dio is we're gonna create a list of lights. Well, say list lights, lights equals new array list. Now, the purpose for this is that I want to be able to shut off all of the lights in my house at once. So I'm gonna add all my lights, this command, and say it lights dot ad when add the kitchen light and then lights dot ad the bedroom light. Now we have all of the lights added. Let's go ahead and create a new command. We'll call this the all lights command. So the sequel to a new all lights command and this doesn't exist yet, so it's gonna give us a compilation air, and we'll go through and created ourselves here in just a second. And this is going to be called borrow light switch. We're gonna see a light switch dot store and execute air, eh? Lights command. Okay, let's fix this to where we create our all lights command. Now I'm gonna say, create class all lights command, and it implements the command interface. So when it brings that up just how I did it off the Red X on the site, it says, Oh, you want to create this is gonna implement the command interface? Yes. Let's go ahead and click. Finish now, are all lights. Command Looks pretty much just like our toggle command are on command Did. There's a few slight variations here. First, we're gonna say that private list light lights and store this. We're gonna create a constructor that passes all of these in. So with our toggle command, we just took one light and we're gonna take a list of lights and here's will say all lights, command list, light lights, and we're gonna sign those to our object. We have our reference we have inside of our class. So say lights equals lights, organized all of our imports. And we pretty much have the basis of what our other objects look like. Four commands with one slight variation down here, we're gonna say for each light, light, light of all of our lights, we want to say light, dark, toggle. Great. That's all we had to do to turn off all of our lights at once. Turn them all on or turn them all off. Let's go ahead and run our demo and see how it works. And, as expected, it turned on our bedroom light and it turned on our kitchen light at the same time as you see from our our output down below. Both objects were switched on. That's fine. And that works fine. Except for if we throw a little wrench in the gears of saying, let's go ahead and create a toggle for the bedroom light, so I'm gonna go ahead and turn the bedroom light on, and then I'm gonna issue all of our lights to turn them all honor all off, and you'll see that we quickly run into a problem with state. So we have one switched on, then the other switched on, and then another switched off. Well, that's gonna

pose a little bit of a problem, because if we're trying to use this to turn them all on, are all off. We now have them in a weird state. This is where our command needs toe work with our receiver to assess the state of where our objects read. So you can see that it was really easy for us to create our mackerel command. All we did was passing a bunch of lights and we can call multiple commands on it. In fact, we could pass in individual commands containing those objects, and that would be one way to fix this. We can also go ahead and just modify our code here slightly to assess state. So if we wanted to say all lights on or all lights off, we can go ahead and change this to an all lights on command. Go inside of her object here and say public Julian is on and return the flags because a return is on and save this now inside of our all lights command, we can say light dot is on and wrap that with a basic conditional statement. If it is on, then we want to turn it off. Now, if we go through and rerun that code one was switched on, we switched the other off. We didn't try and switch both of them off because it wasn't on to begin with. So you see from our code that we've now fixed that race condition of where if I have one or two on the 3rd 1 off. When I issue the all command that we can toggle it off. Very simple example. You It's very clear to see what we're doing here. As far as just working with the state of that object in assessing from our commands, you can think of a dozen different ways to work and control your objects from these command patterns. But to step through the key points that were trying to illustrate. Here we have our client, which is our command demo that we're running here that works with the receiver. Our lights notice our client doesn't call those lights directly, but rather issues a command to work with those lights. And that's where that state can come into a bigger picture of what we want to do with those objects. Imagine if you had all of that light code, all the functionality of the switch in the various commands. Now we haven't all lights command and a toggle command. We have just a basic on command as well that were going through and modifying this code to go through and do that if we did that all inside of our client. This thing would become huge and we only have three basic commands on one simple light switch. I think if we added fans in here, we added more lights. We add a dimming capabilities. You can quickly see how the code inside this command object would get huge really fast. So this is a great way for us to work with commands and decoupling that client and see how you can group those together.

Pitfalls

[Autogenerated] what are some of the pitfalls of a command. It's typically used with other patterns to be more mature. The dependence on other patterns isn't necessarily a bad thing. It just requires more knowledge on the developers part. I also often see people struggle with the use of multiple commands. Frequently, I see people make the mistake of duplicating logic in another command. A better way to do this is either the use of a composite pattern as we demonstrated or commands. Combined with the chain of responsibility pattern for undue functionality, you may want to look it using the memento pattern to handle state. If you're tracking of objects needs to store a history, you may need to also look at the prototype pattern for creating copies of commands to store on a list to get a better idea. When we shouldn't or shouldn't use this pattern, let's contrast it with another one

Contrast to Other Patterns

[Autogenerated] to contrast the command pattern. Let's compare it with thes strategy. The command is structured around an object per command or per request. The class contains the what essentially, what we're trying to do. It also encapsulate CE the entire action. The command object on Lee deals with this one exact scenario. Thes strategy, on the other hand, is similar to the command in that it is an object per request, with the focus on this pattern being per strategy different than the command. Though thes strategy focuses on the how rather than the what. Instead of encapsulating the action, it encapsulates the algorithm. This structure of these patterns are very similar, though with just some slight variations.

Summary

[Autogenerated] to briefly summarize what we've learned. The command pattern is a great pattern. And possibly after the singleton, the second most used pattern. The command pattern encapsulate. Each request is an object. It's also very good at decoupling the sender from the processor. This is one of the key reasons for using this pattern. Overall, with the command pattern there very few drawbacks and, as mentioned, the pitfalls. Maybe just a reliance on other patterns. The command pattern is also often used when we need undue functionality in our application. The next pendant that we're going to look at is the interpreter, and we'll see how that helps us process things.

Interpreter Pattern

Introduction

[Autogenerated] Hi, this is Brian Hansen, and in this module, we're going to look at the interpreter design pattern. The interpreter pattern is a behavior pattern that you used to represent the grammar of a language. A lot of tools use this pattern when parsing various aspects of a grammar. Let's look at some of the concepts considered when choosing this pattern.

Concepts

[Autogenerated] the concept surrounding why you would choose the interpreter pattern or that it represents a grammar. This could be music notation or mathematical equations, or even another language. Compilers will use the interpreter pattern too often. Times par source code. This goes hand in hand with representing the grammar, but we can then use it to interpret a sentence. This enables us to map out a domain specific language. If you've ever used Sequel or an XML par sir, this is the exact thing that is. This pattern was designed to dio define a language that can be interpreted to do things you'll often see an interpreter used when defining an abstract syntax tree or an S T. As it's often abbreviated. To examples of this in the Java A p I. R. The Java Util pattern, the pattern classes used to represent a compiled regular expression, it is an incredibly powerful way to search through strings. Another

representation is tthe e java dot text dot for mat class format is an abstract based class that is used to represent locale sensitive content such as dates, numbers and strings. Let's look at some of the design considerations when choosing this pattern

Design Considerations

[Autogenerated] the design of the interpreter is quite a bit different than most of the other patterns that we've looked at. There is an abstract based class or an abstract expression that declares an interface for executing an operation. That operation is an interpret method. Expressions are then broken into terminal expressions, which represent a leaf of a tree or an expression that does not contain other expressions. If it does contain other expressions than it's a non terminal expression. Non terminal expressions represent compound expressions and continue. Clea called itself recursive lee until it finally represents a terminal expression or multiple sub expressions. The pieces of the UML diagram are the context, abstract expression, terminal expression, non terminal expression and a client. Let's look at that UML diagram now.

UML Diagram

[Autogenerated] This is the UML diagram for the interpreter design pattern. The client builds the sentence of terminal and non terminal expression instances. The context contains information that's global toothy interpreter. All expressions will extend from the abstract expression based class or implemented. If it's an interface, it provides the contract for how we interact with one another. Terminal expressions are expressions that could be evaluated in their current form, where non terminal expressions will be interpreted again. You can see the loop back up to the abstract expression where we can call ourselves recursive Lee until it's broken down into terminal expressions. Now that we've seen this yo Mel diagram, let's look at an everyday example of this pattern in the job. A P I.

Example: Pattern

[Autogenerated] the pattern class used in conjunction with regular expressions are a good example of the interpreter pattern being used in the job. A P I. We created sentence and establish a grammar for that sentence. From there, we're going to interpret the sentence and display what we parsed using the pattern. Let's look at this in life code now.

Demo: Pattern

[Autogenerated] as we discussed in the slide for this example, we're going to go ahead and build a sentence, establish a grammar and then run that through our regular expressions to see what comes out of it. And I'm guessing that the first time we run this, it's not going to produce the output that you're maybe thinking it will. Let's go ahead and right click and run this code well, say, run as Java application and the output says that it found a tiger and it found a bear. The reason for that is because of case case does matter in this now we could change our regular expression to look for upper and lower case. Or we can just go

ahead and add another case on the end of here. Four lion, which is what I'm going to d'oh! And save this and run it again and you'll see that it found a lion, found a tiger and found a bear. And that's because the grammar is very specific. We are looking for exactly this, not sort of something similar to this or close to it, although we can loosen that regular expression. But this is all about this grammar that we've established and interpreting this sentence that we that we've passed in now. One thing that I should note about the interpreter pattern is that it has next and nothing to do about parsing but rather establishing and interpreting a grammar. And so it doesn't care if this was parsed, using four loops or some other pattern or method of of breaking this sentence down and breaking it up. It's just about establishing a grammar. In this case, we are looking for a lion, a tiger and a bear to exist somewhere in this sentence that we've passed in.

Exercise - Interpreter

[Autogenerated] for the exercise that we're going to create on our own. We're going to go ahead and build a rules validator. I won't say a full blown rules engine, but using thean interpreter pattern to interpret things that we pass in and validate whether it's part of an Andorran or a clause in our expression. So in doing so, we're going to create an expression, a terminal expression, and from that terminal expression, we're going to extend that for an end and an or expression, and then we're going to go through and parse that example to see how that applies to all the rules of what we're providing in our grammar.

Demo: Interpreter

[Autogenerated] to shorten the length of this demo, I've gone ahead and created a couple of classes just to help get things rolling. Long, quicker to start off, I created a static avoid main method that I just go ahead and build the context in which we're going to use this and build the interpreter tree. So define what our grammar is and run that context through our grammar. And right now there's nothing in our grammar. I've got it commented out and just returning null so that the application will physically compile. Now for the heart of the application. I've gone ahead and created just in interface. You can use an abstract class or an interface. I didn't have anything that I was trying to pass down to sub classes. So I went ahead and created the expression where the interpreter as an interface and you can see it has just the one method public Boolean interpret and I passed that string into it. Now I've started stubbing out a terminal expression, and this is where we're going to pick up and start coating out this application. I have a terminal expression that just goes through and says ROK, I implement the expression interface and you're gonna pass some data into me, and then I'm going to interpret that data. Let's pick up from here. I'm gonna go into my interpreter data and first of all, uncommon this line that I had in here too, So it would compile, and we can see that we have a terminal expression that's going to take a string lions. And this is starting to define how our grammar is going to interact with our interpreter. We're looking for lions with this terminal expression. So I'm gonna save this and now move over to my terminal expression. And right now, it doesn't really do anything very complex. In fact, right now it's it's hard coded to return. Fault's gonna go ahead and swap out that implementation. And I've already got it coated just to save time. So I'm going to grab this chunk of code here and paste this in. Now What this does is it's going to take the data that gets passed in and token ISAT

and loop through that just to see based off of space, off of a space, loop through it and see if it just contains that string So we're basically doing the same thing that we were doing with regular expressions, but doing it kind of in a poor man's method. But you can see how I can go through and start to grab or look for specific pieces of the grammar and work with it. Saved this terminal expression and now are very rudimentary. Implementation should work. We're going to extend it from here. But I'll build this string token Isar cycle through the tokens and see if any of them equals. If it does, I return true. Otherwise, I returned false and exit out. Let's go over to our demo here and right click on the main method and say, Run as a job application and we'll see that it finds Lyons Lyons is equal to true. So just to kind of go through those steps, we go ahead and build our context, build our interpreter tree and then define or run that context through our grammar to see what matches. Let's start doing some more complex things. With this, though First off, I have created and or expression, and I'm gonna open this one up. We're gonna create the an expression together cause it's a little bit more complex. The or expression has two expressions that get passed in. Now you already see that we're starting to work with non Terminal. So we had a terminal expression over here that very simply went through all of the pieces. And then we're going to work with a non terminal or a compound ing expression to start comparing things and interpret one or the other. And that's exactly what we do. You can see down here we take and look at the first expression and interpret it. And if it matches, great or we're going to go ahead and look at the second expression and see if it matches in return. True from one or the other. Well, as you congest imagine, we're going to do something very similar with the and expression. I'm going to go ahead and get out of full screen here and right click on our package and say, new class and expression. Now you could go ahead and say that the interface was expression here. I've actually got the code typed out, so I'm just going to click, finish and go into my demo notes and grab all of this code here. Copy this and paste it in here and we'll walk through what it does. So this code goes through just like the or grabs the two expressions that are passed in, and instead it looks at the first expression, interprets it and then ends it with the second expression, too. Look to see if both of those values returned, false or true or whatever combination of those to finally return it back to the calling mechanism that asked for the interpreter. Now we're at a little bit of a crossroads, though We want to do something Maur interesting with our demo because our demo will still work exactly how it waas. We only had a basic lions that we're passing in here. And just for sake of testing, we can comment out lions and uncommon tigers and go ahead and run it and it returns false because there's nothing tigers isn't found in our grammar are grammars only looking four lions. Let's go ahead and add a few more expressions there, so I'm gonna grab a couple over here. I've got tigers and bears created the exact same way. So I've got terminal one, just eternal expression. One of lions Terminal Expression number two, which is tigers and terminal expression number three, which is bears. The problem is, is now we don't have any of those combined using our end or are or expression yet to compound those into a much bigger grammar. Let's go ahead and create some of those. This is where it gets a little more interesting as faras a rules in general validators concerned with what we want to do inside of our application. I can go ahead and compound these and grab this and paste it into our demo. And now I have some complex expressions built for determining our grammar so you can see I have an alternation or a different use case of this where I'm looking for Terminal two and Terminal three sewing, say lions or tigers and bears. And I throw that Oren down below here, where I say, Okay, I got alternation Number two, which takes in Terminal one and alternation once you see how I'm starting to build that syntax tree out with the different variations that I want to accept or approve inside of our application, and I returned that as an and expression. Let's

go ahead and see if this works. Now, If I come down here and rerun this, I will get that Tigers is still fault. But if I look for one of my combinations down below, such as lions and tigers, save that run ahead. I'll see Lions and Tigers fault. But now if I go to tigers and bears or lions and bears, I'll see lions and bears is true and that is because of the combination that we have here. So we have lions or bears. So if we look in our combination here, we're looking for tigers and bears or our claws, tigers and bears. Now, if I do tigers and bears that one should also return true as well. Save that run it. You see tigers and bears is true as well. So you start to see the power of using the this grammar to go through and parse all the subsets and how for doing something with an abstract syntax tree to make a more complex interpreter of what we're trying to d'oh! Just like we look to see it for the and clause of the or Klaus, we could do the same thing for mathematical equations to look for parentheses or multiplication symbol or a division symbol and go through and build out that syntax tree through our code and interpret what we have inside of her application. Really powerful way to do this. And we're not just using a bunch of, if then else statements or switch statements to build out our application, which eventually could become something quite unmaintained herbal, whereas with this, I can keep building those expressions upon one another.

Pitfalls

[Autogenerated] Now that we've created our own interpreter, let's look at some of the pitfalls of it. If the grammar becomes very complex, it can be difficult to maintain, as you noticed in our rule example that we had you could see very quickly that if I started adding these different answer or combinations, it could become a little bit interesting to try and debug and walk through. So complexity can be an issue there. There is at least one class per rule, so every time we create one of our new expressions were creating another class. Complex rules will require multiple classes to define them. That's where the difficulty of maintenance can come into play. Use of other patterns might help with your specific implementation of a complex interpreter, and adding a new variant requires us to change every variant of that class. The interpreter is a little unique compared to some of the other patterns that we have looked at because it is fairly specific to the problem that we're trying to solve to better understand when we should use this or a different pattern. Let's contrast it with the visitor pattern

Contrast to Other Patterns

[Autogenerated] to contrast the interpreter pattern. Let's compare it with the visitor. The interpreter and the visitor are very similar in structure, but a different focus on implementation. The interpreter has access to properties because it contains the object. Functions are defined as methods, and since we extender implement the base interface, each interpret function is contained within a method. One drawback is that adding new functionality changes every variant recall the demo that we coated together when we were building the expression tree and the complexity that we could get by calm pounding those expressions together, the visitor is actually very similar to the interpreter, with some slight variations. Instead of having access to the properties we need, we have to implement the observer observable functionality to gain access to those properties similar to the interpreter. Functionality is found in one place, but it is in the visitors and not in the

expression objects that were building and just like the interpreter. Adding a new variant requires changing every visitor. The focus is more about whether you're adding more expressions or grammar rules or adding new visitors to interact with, and that is the main focus on choosing one over the other

Summary

[Autogenerated] to summarize the interpreter pattern. You want to use this pattern when you're defining a grammar. It is a great pattern if you're defining rules or validation criteria for objects. Our example use strings, but you could just as easily substitute objects and take advantage of the power of generics inside your interpreter pattern. This pattern is more of a special case or limited use pattern. No, it solves a very specific problem. It's great for that purpose, but limited use outside of that watch when you're implementing it, to see where you think change is likely to occur, and you may want to consider using the visitor pattern depending on where that change is, The next pattern that we're going to look at is the generator pattern, which is a great one for cycling through objects.

Iterator Pattern

Introduction

[Autogenerated] Hi. This is Brian Hansen. And in this module, we're going to look at the generator design pattern. The generator pattern is a great pattern for providing navigation without exposing the structure of an object.

Concepts

[Autogenerated] the concept surrounding why you would choose the Liberator pattern are that you need to traverse a container in Java and most current programming languages. There's the notion of a collection list maps sets are all examples of a collection that we would want to traverse historically have used a loop of some sort and an index into your collection to traverse it. In the case of the generator, we don't expose Thea underlying structure of the object that we want to navigate. Navigating various structures may have different algorithms or approaches to cycle through the data the IT aerator de Couples. The data from the algorithm used to traverse it generators air also sequential in nature. And this is an interesting concept because not all objects have a sequential set of data. The generator handles the navigation in an order that best resent represents its sequence. Examples of this in the job, a p i. R. The built in it aerator interface. This is such a pivotal part of the language that Java has decided to build in an interview later interface for us to use across all of our containers and collection that needs some sort of navigation There is also an enumeration in new Marais. Shin predated it aerator and is now simply just a copy of the functionality of the operator interface. Let's look at some of the design considerations when choosing and generator.

Design Considerations

[Autogenerated] the design of the Liberator is interface based. Whichever object you want to reiterate over will provide a method to return an instance oven iterated from it. It follows a sort of factory method pattern in the way that you get an instance of the IT aerator each generators developed in such a way that it is independent of another. It aerators don't know about other it aerators, but they are fell fast. Fell fast means that to Liberators. Can't modify the underlying object without an air being thrown coincidentally and numerator are fell safe, which means it is built in a way that they can't be used to fell. They're pros and cons to both, but they decided in the job a p I to utilize a fell fast approach rather than a fail safe approach for the it aerator pattern. The peace of the UML diagram are simply an operator interface and an instance of a concrete generator from the container. Let's now look at that UML diagram

UML Diagram

[Autogenerated] This is the UML diagram for the Javi you Till list using an ID aerator. The generator itself is quite simple, so to show how it works, I included the list classed and its implementation. The collection interface is extended by the list interface. The list interface has, among others, and it aerator factory method. The Generator Factory Method returns an instance of the Liberator interface in the case of a list, and it's implementations Thea underlying instances a list it aerator. The listed aerator is an implementation of the IT aerator interface that understands how to jittery over the various list objects in the collections a p i. It declares the interface for objects in the composition. Let's look at this everyday example in some code now.

Example: List

[Autogenerated] the list interface makes great use of the generator for looping purposes. In this example, you can see that we create an instance of Honore List and after initializing some values in her iterated over it, notice in this example that were also modifying the original list by using the remove method on the IT aerator. Let's look at a couple of examples of this in life code now.

Demo: List

[Autogenerated] Here's the actual code for the samples that we were just looking at. Where we go ahead and create an instance of Honore List. Add some people to it and then get an instance of an incinerator to victory over that object in a simple wild loop. And if we run it, it's really straightforward code. You'll see that it runs just fine. It was a job application. You'll notice that we cycle through it. It prints out Brian, Aaron and Jason, and then we call the remove each time. So when we finish, the actual loop is of or the object itself, that the loop on over is off size zero. Now great. It works. How on it aerator should work. But let's look at some other variations of this. One thing that people don't often realize is that the four H Loop is using an instance of the generator. Sophie go for each and hit control space bar. Notice what the text says here for each iterated over an array or an literal object. So if I hit enter here, it's

using an operator behind the scenes. Now we don't need this one anymore, cause it's grabbed its own instance of it. But the object it has here is now available for use as an it aerator. So it has an honorable object behind the scenes that it called on that interface. So it knows using the it aerator pattern that we can now construct this for each loop and the same thing. If I just do a system dot out dot print land on the name here, we'll change our string up here to name and save it. We can now enter a over this object the same exact way, and we can utilize the remove if we want to, or how we want to go through some of that functionality. But you can see we're not exposing that index anywhere. Traditionally, we would have written this as a four loop and gone through and said, for anti equal to zero I less than the size of names. So we do names dot size and then do I plus plus. And now we have an instance of that. Now we need to go through and actually grab that based off of its location so he could say names dot don't get at an index, and we could use I, and this would return our names so we'd say string name equal to this. And now I can go through and print this up a lot more work, a lot more things to get wrong. It still works, though, So a lot of people go Well, why would I ever want to use the generator? Because I have this functionality built in front of me that will work this way and I'll show you exactly why. Because if I come up here and create a new instance of a set and say, it's a set of strings and we'll call this one names new hash set and save that we have to import all of our objects. What's happening on this line right here? There isn't an index to get an object by an index location. I can't do this anymore. But if I look at some of my other examples, my four loop that's using the terrible under the hood, it works just fine. We can go through, we can run it. It'll do the same thing, and that is because a set doesn't have a get based off of an index. And just for the sake of showing the entire example will go back to what we had originally with Wild Loop and you'll see that it runs as well. We can go through save that, run this example, and our example will work, and it will also allow us to remove that object at that location. So this is really a great way to generate over a set or some of the other collections and have an index to a location and be able to modify that collection. You can't do that with a four loop and a simple dot get based off of an index. You see some of the power of using this generator and why they did that is based off of how that object gets structured underneath that collection.

Exercise - Iterator

[Autogenerated] Now that we've seen some examples of this being used, let's go ahead and create our own, and it's gonna be a pretty simple example. We're gonna go through and create a repositories yaar basic example of a collection, and then we're gonna create an it aerator to cycle over that repositories. There's a few things I'm gonna show you inside of that repositories, that air indicative of what goes on behind the scenes in a collection, things such as re sizing and how we remove and those types of actions.

Demo: Iterator

[Autogenerated] to help shorten the length of this exercise, I've gone ahead and created some sample code to show you the pieces that we want to set up. These don't really have anything to do with the actual implementation of the IT aerator. It's just to help make this run a little bit quicker. So I've gone ahead and created a demo method where we set up an instance of the bike repositories. We're going to create and add a couple of bikes to it, and I

have taken the simple route this time and just created some strings. I probably would really use some bike objects if I was going to implement this in a true application, but you can see what we're doing by example. From here, I'm going to call an instance of the generator, and right now that method isn't defined, so you can see that I've got a compilation air on it from here. I just jittery over that iterated that we return back from calling the method from our repositories. Let's go ahead and switch overto our bike repositories and start adding those features in here Now, inside here. Very simple. I've backed it by an array. I initialize it to a default size and to make it a little more realistic, I've added an ad bike method in here that you can see, goes ahead and looks at the index size and goes ahead and creates an instance Ah, larger instance of the ray. If we need Thio and copies, everything is into into it. And then it assigns the actual bike to that position in the array and returns from here. So kind of a simple collection that we've created on the back end on her own. Now, I'm gonna do two things here. Gonna make this object, implement Ita ra ble and you'll see from doing this implements literal. If I choose a terrible, you'll notice that it says implementing this interface allows an object to be the target of the four each statement, and we want that. So we're gonna go ahead and it's asking for generic here. I'm in a hard Cody in string now. Obviously, if this was the bike repositories, I'd probably want to create that bike object that I mentioned earlier and substitute bike in here. It's a trivial example and it doesn't really have anything to do with this pattern so you can do that exercise on your own Now, in doing so and implementing that interface, you'll get it in those. I get another competition. They're telling me that I need to add on implemented methods. And as you might imagine, it tells me you need an ID aerator. Great. That's exactly what we want to D'oh! We're going to create an instance. Oven iterated er, And to do so, I'm going to just do this as an inter class and anonymous in her class. Inside of this object. Well, say it aerator and a string se I t equals new it aerator, and it will go ahead and create our basic structure for us. And it's gonna tell us that it wants the unimplemented methods as well. So we'll add those three unimplemented methods inside of here. The last thing we want to do and just get it out of the way is I want to return the it aerator that we've created. So there's our entire anonymous in her class. Now we're gonna go ahead and start filling in these values to use for our generator to start with. I want to go ahead and add an index for us to track our progress through iterating. I'm say private and current index is equal to zero and save that now I'm gonna start off with Has next has next is a little bit more complex. We're going to go ahead and say current index less than cause notice. This is creating a Boolean bike stop length and bikes at the current index does not equal no. Now, what this is just doing is saying that make sure that our index is less than the length and that the object that we're looking at Cousin Array can be larger than what our current our index sizes and have no values in there. So look, that are indexes below our length and that the object that we're looking at that location is not No. The next thing we're gonna dio is for our next method. We're going to return bikes at the current index and just as a shorthand, I'm gonna do index plus Plus, what this is going to do is it's going to grab bikes, the current index And since we're calling the next one, once it's returned that it's going to increment the counter. So it's doing two things at once, a little bit of shorthand. You maybe a little confused by it. Don't worry about it. You could break that out into two calls. If you wanted to grab the bike object, return it on its own and then in credit now for the remove. The remove just is going to grab something from that location and nullify that spots what's not available in the future. Going forward. And we should resize the array just like we've done up in our ad method up above here. I'm going to let you do that on your own. It's not go. It doesn't add anything to the actual liberate herself. So I'm going to throw a new unsupportive operation exception in here and supported Operation Exception, and you can implement it if

you want to. But this shows wth e the intent in the implementation of that it aerator pattern. Let's go ahead and go over to our demo now, and you can see our compilation error is gone. Right? Click and say, run as a job application and we'll return. Survey Loa Scott and a Fuji Three really great bikes now because we made that an irritable object. We had our cetera ble at the top and it forced us to create an instance of the Liberator method. We can also go ahead and take that same chunk of code and safe for each. And it will just automatically choose our repo for us. You say bike 20 system that out dot print Lynn on the bike and save that well, coming up the wild just so it doesn't display them all twice and run that you'll see it does that if we try and call the remove on our wild loop. Obviously, at this time it will throw an object unsupported operation exception and you can go ahead and implement that on your own if you like. It doesn't really do anything to show what the actual it aerator does. You see how we made an aerator? We didn't have to expose an index into our object, but we also made it usable by the for each that was implemented in Java 15

Pitfalls

[Autogenerated] Now that we've seen a couple of working examples and created her own, what are some of the pitfalls of innit? Aerator? Well, if you haven't noticed and it would be pretty hard not to because we've discussed it a couple of times. You don't have access to an index of any sort. If you want to get an element of a certain position, there is no way to do that without just iterating through and stopping on that object in the case of sets and maps and some of the other collection. So there isn't a method for you to grab an element at a certain position, So this is somewhat of a moot point. The Basij aerator interface in Java is also on Lee Yoon a directional. It can only go forward. Some of my limitations, though, offer bidirectional access. So if you've noticed the list, it aerator not just the generator that we're using to the list it earlier. It has a forwards and backwards capability built into it. I don't give you the wrong impression here either. And in most cases, Thean Aerator is the most efficient solution to looping through an object bid in a couple of select scenarios, the generator may be slower and just slightly slower than using an index and looping through it. I should note that larger, looping instances the generator is almost always more efficient. I don't wanna say it always is because someone will prove me wrong. But in every scenario I've seen in every test ran, the generator is the most efficient way to go through a large collection of objects.

Contrast to Other Patterns

[Autogenerated] with all of the other patterns we've covered. We have contrasted the pattern with another to show its strengths. The generator is a little different than others, though, and stands on its own feet to have something to compare it to. Let's look at the traditional four loop instead, as we've already talked about some of its strengths and weaknesses compared to the generator, the generator is interface based, and by doing so, helped us remove the Traverse A LL algorithm from the client. There is not an index available. Yeah, it doesn't seem to be much of an issue, especially when we're iterating over some of the collections that don't have a dot get at a certain index. It also helps us with concurrent modifications. So two things trying to modify that collection at the same time the four loop pushes all of it's true, traverse a ll code to the client. So this is, ah, huge change. You'll notice that we had an algorithm that was contained in the repositories as we generated that anonymous in her class inside that

object, and it kept track of indexes and how we may or may not navigate over that object It also exposes an index which could be beneficial or not, depending on your usage of it. It doesn't change the underlying object cause you're dealing with that object. It also doesn't allow you to work with the four each syntax. If you're going to be using the four each syntax, you have to be utilizing an object that is implementing the literal interface. And that shorthand is really nice. And it typically is slower in most situations the way that we're navigating, using the four loop so the generators a little faster, and it handles things like concurrent modification as well, a cz keeping that algorithm contained from the client. So keeping your clients simpler for both to use. Both work well. There's pros and cons to each. But many people often overlook the it aerator early on because they knew how to deal with a four loop. There's a lot of scenarios where that traitor is a much better for iterating than that for Luke

Summary

[Autogenerated] Let's recap the IT aerator. It's an efficient way to traverse an object. We've shown that both in speed as well as keeping your client simpler than what may be done with just a simple four loop. It also hides the algorithm from the client, so that simplicity for your client is contained within the algorithm that you have coated in the container. You're creating an illiterate or four, so it helps us simplify that client. It also lets us take advantage of the new four H syntax that was introduced in Java 15 which definitely simplifies iterating over that object in a four loop or some of the various methods using a wild loop. The next batter we're going to look at is the mediator pattern, which is used to handle how objects interact with one another.

Mediator Pattern

Introduction

[Autogenerated] Hi, this is Brian Hansen, and in this module, we're going to look at the mediator design pattern. The mediator pattern is used to define how objects interact with one another without having them refer to each other explicitly. Let's look at the concepts when choosing this pattern.

Concepts

[Autogenerated] the concept surrounding why you would choose the mediator pattern or that you need to achieve loose coupling between objects. This is present when we're dealing with a set of well defined objects that communicate in complex ways. Oftentimes their spaghetti code between objects that make loose coupling seem impossible and makes it tough to create a reusable components. Because of inter object communication, it's simply acts as a hub or a router in your application and which all communication will be routed through examples of this in the job, a p i. R the Java Util timer class, the timer class mediates through its schedule

methods. Also, the Java laying reflect method class has an invoke method that allows us to mediate objects that were reflecting upon. Let's now look at the design of the mediator

Design Considerations

[Autogenerated] the design of the mediator is interface based with a concrete class, although it could be implemented as just a class. Utilizing the interface gives us the option of cycling out. The various mediators based on the situation typically would want to do this for variances across different platforms that were deployed on. The mediator often minimizes inheritance and our application because it is handling the communication rather than classes inheriting functionality. The mediator also knows about all the colleagues rather than colleagues knowing about each other. Pieces of the UML diagram are the mediator and a concrete meteor. Let's look at that UML diagram now.

UML Diagram

[Autogenerated] This is the UML diagram for the mediator pattern. The mediator classes an interface or abstract class but doesn't necessarily need to be. If it is abstract, it is extended to a concrete mediator or the functionality will just extend from itself. This is also that the pier objects or colleagues don't communicate directly with one another, but rather through the mediator. Let's look at an everyday example of this pattern in action.

Example: Timer

[Autogenerated] the timer object and more specifically, the schedule methods are a good example of a mediator. We can schedule tasks that do complex things without having them communicate with one another. Let's look at this in life code now.

Demo: Timer

[Autogenerated] So let's walk through the live code that we were just talking about in the slide on how to implement a timer object to schedule some tasks, go ahead and create an instance of this class. It's a little bit larger to see it than we can fit on one screen inside of here in the constructor. I just pass in the number of seconds that I want this to run for a very simple time or demo. I go ahead and create an instance of a remind task that I've created that extend this timer task that requires us to have a run method. And because sound doesn't carry through very well from running applications and record a demo, I went ahead and commented out the tool kit where I am grabbing an instance of the tool kit and making a beep noise. Then I have created another task that does remind task with out of Beep, And the two of these just assure you that I can schedule these two tasks and they don't need to know about one another. That's all handled through the mediator, which is our timer and our scheduling method. Let's go ahead and run this and see what it does now, right? Click on the main method and say, Run as Java application and you'll see that it's schedules. The task on the task is scheduled, and then it fires off the first reminder task. Time's up. And in the 2nd 1 this is times now really up. These didn't need to communicate with one another to know

how they were scheduled when they were gonna run. They all ran independently of one another without having to be hard coded with references to each other. So this is a good example of basic example of how the mediator pattern works and not requiring objects to communicate directly with one another.

Exercise - Mediator

[Autogenerated] Now that we've seen an example of the mediator, let's go ahead and implement our own to implement our own mediator. We're going to create one that will implement a mediator, object and a colleague object. And to do this, though, we're going to fix a problem that we experience with the command pattern that we discussed in that module. To demonstrate the solution of the mediator. We're going to reconfigure our light and light switch object to exercise the mediator.

Demo: Mediator

[Autogenerated] to start off with this demo. I actually went ahead and grabbed the command interface from our command pattern object, and you'll notice that I changed the comment in here to say that it's colleague because inside this example, all of our commands are in fact our colleagues. I also grab the light object from that command just because I'm going to utilize this to demonstrate the complexity that we had inside the command pattern from having a a command object. Call multiple other objects to aggregate or create mackerel commands. So to start off, let's go to our mediator object and you'll notice I've created this mediator class, and as I mentioned before, a mediator could be a interface or an abstract class. Or it could just be a concrete instance. I don't plan in this example on having multiple mediators, which you would usually implement to swap out for different platforms. So I just went ahead and created a mediator class and inside of here. I've got a ray list of lights I can register a light, which just adds it to that collection, and then when I say I want to turn on all lights, I just cycle through all of them, checked to see if it's on and toggle it on or off. So the object itself controls its state, but the mediator will go through and determine whether or not it should or should not. Call it now to demo this. I've gone ahead and created a basic mediator demo, which we can go through and see that I have an instance of my mediator. Then I created two lights, a bedroom light and a kitchen light. Then I register those lights with my mediator. So I call the mediator dot register light and I go through and taken instance of our command pattern now. And this is straight out of the command pattern module that we did. I grab an instance of this turn on all lights command, and then I executed. Let's look at that turn on all lights command. It has an instance of the mediator where before it had an instance of objects that was trying to call, let's go ahead and open up our previous command to see the difference between using this with the mediator pattern. So if I go up back up in your command demo and grabber on command. You'll see that it had an instance of a light and we tried to start doing things with, like are all lights command that had a list of lights and then I would pass those in and I would cycle through there. But our command started getting smarter and smarter with what we were trying to dio as faras aggregate commands were concerned. So this all lights command we're now implementing as a mediator. Let me switch back to my turn on all lights and you'll see that there's no reference to lights in here. There's only a reference to the mediator in here, and this is our concrete command. Now, to run

this, I'm just gonna go ahead and go back to my main method and right click on it and say, Run as job application and you'll see that the bedroom light is switched on in the kitchen. Lights switched on. Well, I don't know about your household, but mine. The kids seem to have a problem with shutting lights off, not turning them on. And so I want to implement a command that will go through and turn off all the lights is we're going to bed or we're leaving the house. So what? I want to go ahead and do is now do that same thing with turning it all off. So I have a turn off all lights command that we're gonna walk through an implement the functionality ourselves to d'oh! I'm gonna uncommon this line here that just gets an instance of that. Turn off all lights, command, and we're gonna go ahead and execute it. So do that now so that it doesn't have the instance. Not used air message. And inside the turn off all lights command, there is a method that were overriding is part of the command pattern for the execute that I'm gonna go through. And now uncommon this. Now this is gonna give me an heir because that method currently doesn't exist inside of her mediator. Few things I wanted to point out here. Notice that our commands still do what commands should d'oh! And our mediator handles that aggregate or that calm pounding of command. So I really like to use these two together. In fact, I went out of my way to make this example. Include the concrete command patter that command pattern in here with this concrete command so that you could see how these two are used in concert or in conjunction with one another rather than standing alone by themselves. So you don't use one or the other. I like to use them together. Let's go ahead and create this method now so I can cover here and say, Create method, Turn off all lights inside of here and we're gonna do something very similar to what we did up above here. We're gonna do it for each and we'll liberate over and Ray were terrible. And we're gonna get an instance of light light, and it's all of our lights and we'll say, If the light is on, then let's Tuggle it might dot toggle So we'll save this now. This is all I had to do to go through an implement another command that was a mackerel command. And you could do all sorts of things here. If you want to extend this and I purposely try to trim this example down, I was going to do the same thing with fans, so we'd go through and create a fan object and turn all the fans on or off all the fans off for a toggle individual rooms. You could tie it to an alarm clock that it's 6 a.m. It turns on the bedroom light and the fan, but it doesn't turn on any of the other ones in the house. So you can start doing these very cool compound commands through this mediator object that controls that object. But the objects don't know about them. Their own state. Let's go back to our demo now and see how this runs a right click and say, Run as job application and you'll see that I switched both lights on and then I switch both lights off. So much better example, in my opinion of these compound commands and not having to pass around this collection of lights or register all the lights associate ID with that object we just passed the mediator around. If we look at this main method versus the command pattern demo that we had, this client code, in my opinion, is much simpler and easier to understand If we go ahead and look at the command demo. You see, we had a bunch of different things going on here. We had a list that we were creating, adding all those objects to the list where we just register it with the mediator and pass an instance of that mediator around. Much simpler, in my opinion, and you get the benefit of both patterns out of it.

Pitfalls

[Autogenerated] Now that we've created our own instance of a mediator, what are some of the pitfalls of it? Well, you have to be careful not to create a deity object. You can easily build

a mediator that is everything to everybody and becomes unruly. This usually isn't a problem, but you can see how, by adding different scenarios, it can become large rather quickly. The mediator structure can limit subclass ing, too, although, as we discussed in the design section, you convey eloped mediators to extend and provide various mediators for various situations. But those are usually used for platform differences. They can also be a little confusing whether or not you should use this instead of the command pattern. This is why I went out of my way in the demo to show the command pattern in conjunction with the mediator, as they feel, is a really good use of this pattern. Let's compare this with some other patterns, though, to solidify some of these concepts

Contrast to Other Patterns

[Autogenerated] to contrast the mediator pattern. Let's compare it with the Observer. The mediator defines how objects interact with one another, and its intent is to decouple objects by eliminating references to each other and instead just have a reference to the mediator. It is also more specific, and I'll explain what I mean by specific here in a second when I compare it with the Observer. The Observer, on the other hand, is a one too many broadcast. Instead of defining interaction like the mediator it broadcast to all listeners, they choose whether or not they want to do something with it. It also focuses on object to couple ing just in a different way. By using that broadcast using this approach, it makes things more generic, adding, this feature requires just becoming another listener rather than modifying the actual mediator.

Summary

[Autogenerated] to recap what we've learned about the mediator. It helps us achieve loose coupling between objects, and it does this by simplifying the communication between those complex objects. You do have to be careful of mediator complexity. You can see how, if we add too much to it at once, that it could become this unruly object for us to maintain, and I like to use it with the command pattern rather than in lieu of the command pattern. But they do stand on their own two feet, and you don't have to use them together. The next pattern we will look at is the memento pattern, which is a great pattern for providing rollback functionality in your application.

Memento Pattern

Introduction

[Autogenerated] Hi, This is Brian Hansen. And in this module, we're going to look at the memento design pattern. The memento pattern is used to externalize an object state, usually to provide rollback functionality.

Concepts

[Autogenerated] the concept surrounding why you would choose the momentum pattern or that you need to restore an object to a previous state. We achieved this by Externalizing and objects internal state so that that object could be returned to it later. We do need to be careful to not violate encapsulation while implementing this pattern. Typically, we would use this pattern to implement, undo or roll back functionality inside of an application. Another benefit of the memento is that it shields complex eternal internals from other objects. Examples of this in the job a p i. R Javi you till date. The date object is internally represented by a long value, so we can easily return the date object to its previous state by setting it back to that long value. A better example, though, is probably java dot io dot serialize Herbal serialize herbal allows you to implement any object to have its state recreate herbal. Let's look at this design associated with this pattern

Design Considerations

[Autogenerated] the design of the memento is class based. There are three roles in this pattern. The originator, which is the object that we were wanting to create a copy or save point of the originator creates an actual memento. The next role is that of the caretaker. The caretaker is what manages the copies or Mementos that we have created. Think of it is the list of undoes available in your menu and then, finally, the memento itself. The momentum represents the copy of the originator that we want to store. The memento consists of what's often termed as a magic cookie. The magic cookie is the combination of fields that it takes to recreate or copy the state of the object, and it is stored inside the memento. It isn't an actual object, but you will often see people refer to it when discussing the memento, the pieces of the UML diagram or what we already discussed the originator caretaker, a memento. Let's go ahead and look at that UML diagram now

UML Diagram

[Autogenerated] This is the UML diagram for the mental pattern. The originator is the object that we want to create a copy of or have the ability to return to its state. It has state and then the ability to set the memento and create a memento of itself. The state captured from the originator is contained in the memento object. The mental has stayed as well and methods to access that state. The copies of those Mementos that we make our then stored in the caretaker. The caretaker knows why and when the originator needs to be saved and how to restore itself. These three pieces make up the memento pattern. Let's look at an example of this code in action.

Example: Serializable

[Autogenerated] the built in serialize herbal interface in Java is a great implementation of the momentum pattern. We can create an object as we have done here, and then implement a serialized method that will write it out to the file system. Let's look at this in a live example now.

Demo: Serializable

[Autogenerated] Here's the code for the live demo that we were just discussing about using the serialize herbal interface. And you can see here I've got a main method that goes ahead and creates an instance of an employee object. And we'll look at that object here in a second. We said the name, address phone number, and then we were serialized that object out the employees object. All it has to do is create an interface and implementation of the interface of serialize herbal. That's it. There's nothing else in here that needs to be done to implement serialize Herbal. Now you will notice that I have set the suppress warnings for serial up here just to be full disclosure that is set so that we can implement future versions of serialize herbal. I just put suppress warnings on here because we don't need it for our example, you can read more up on what it does in adding a default cereal. You I d. Inside your code. So inside of her object, we have this serialize herbal interface and now I can call any method that's looking for a serial Izabal object and inside of our code. I've gone ahead and created this serialized method that gets an instance of a file output stream and then passes that to an object output stream. Just typical file Io and Java and then writes that object out to the file system. And it's gonna ride it out to our temp employees dot S E R file and then closes all those up and it's available for de serialization you see down below in our code that I then d c realize that object and get a new reference to it and that it will return an instance of that object for me. So let's go ahead and run this code and see what it does. Now it's gonna go ahead and get me back that instance of Brian Hansen, So I will go ahead and run it. It's serialize is this object out just by writing it out, and then it will go ahead and call D serialized, which will read that object in off of the file system. Pretty plain simple example. But you can see how I can then create instances that I can pass around and have these safe points. Now, one thing that this example is lacking is the caretaker, so we have our employees object, which is our originator. The memento is really the serialized object that's implement using serialize a ble, but we don't have a caretaker. Let's go ahead and create our own example that will then use the caretaker and all the pieces of the memento pattern, the originator of the memento and the caretaker and see how that's implemented.

Exercise - Memento

[Autogenerated] so to create our own memento pattern, we're going to go ahead and implement all the pieces wth e originator, Thekla air taker and the memento object itself. And we're gonna do that to just create a full fledged memento pattern instance. Let's look at all those pieces now, and we're going to change our employees object that we had that was implementing Serialize Herbal to create our own memento just to see that internal workings of what we need to do there.

Demo: Memento

[Autogenerated] So to start with, let's open up. Our employees object from thesis realizable demo, and I'm gonna get rid of the serialize herbal interface and just clean that up. So now we are a basic object, just a po Jo. There's nothing special inside of here that is our originator. It's the object that we want to create a copy of her an instance off now, to do so, we need to create our memento object. Go ahead and right click and say new class, and we're gonna call this the employees momento. Now, these air pretty simple objects. I could have

pre created them, but I want you to get the full demo of walking through all the pieces of what we need to do. So this is our actual memento, as the name may suggest, and we're gonna put two fields in hair were to say that to create an instance of an employee. We care about their name and we care about their phone number. We don't care about their address. This is one of the reasons I like creating our own memento rather than just using the serialize herbal instances because we have control over the fields that we want to put in sight of this code. So let's just create a constructor here, save public employees Memento, and we're gonna pass in string, name and string phone number. Name equals name, and this dot phone equals phone. Now, the only thing we care about on a mental object is whether we can get the state. We set the state through the constructor, so that guarantees our contract. So we're gonna provide getters for the name and phone number. So the source generate getters and centers. Let's select get name and get phone number and click. OK, go ahead and sort our code, and we are now get This is our momentum objects. Everything looks pretty good inside of here. We've got a basic object that's going to allow us to get that has state is going to allow us to set it through the constructor and get it when we want to return to this object state. Now, inside of our employees object, we want to provide two methods. I'm gonna scroll the bottom here, and I'm gonna say public employees memento and we're going to say save, and this is going to return a new instance of employees Memento with name and phone number. So we have the safe portion done. And now we're gonna provide the revert or undo people oftentimes put really long names in sight of here. I like to just use wth e the Revert name. It's kind of synonymous with what we do inside of some of the source code control tools that we use, like subversion or get or things like that. There's a revert inside of there were going to say Revert, and this is going to take an instance of the employee. And I got this method already done at the employee Memento, and we're just going to take and past that in and revert the state. So you see what this does Employee Memento passage, and we're gonna set the state of this object to this dot name equal to employees memento dot Get name and this dot for phone equal to employees memento dot Get phone. Now we have one more piece that we need to implement here, and then we can run our demo and I've got a pre code a demo set up. We need to create our instance of our caretaker I'm gonna go ahead and right click and say New class, and we have the caretaker object that we're going to create. Now there's a bunch of different ways to create this caretaker object. One of the ways is to implement an array list, and I like a ray list. They're great for a lot of things, but they don't. It's not really the implementation of this pattern. Usually we build these with a stack, so to speak, that we pop each one ofus. We revert through the history. I am going to go ahead and implement this caretaker with a stack so we'll say Private stack. And it's a stack of the employees memento objects and that employee mental object. We'll just be our employees. History equals new stack, and that will go ahead and create that for us. Now we need to implement to methods inside of here when you do a public void save very similar to what we did with in our employees object. We're gonna pass in an instance of employees in and say employees, and we'll just push this onto our stacks will simply history dot push and will stew in plead save now this is going to call the same method which is going to return an instance of the employees Memento and push it onto our employees history. And then we'll go through and do the revert will say Public void revert. And this is going to also take the employees object in now. One thing I'll, I'll point out here is that this is going to do this object by reference association. So I'm gonna pass this employee object in and keep a hold of the reference and change the values in sight of it. And you'll see what I'm saying here in just a second. So say employees don't revert and we're going to do employees history dot pop now the This is why I like the stack is it's going to just grab the previous state off of that stack

and push that on top of our object. So reassign those values for us Now. We've got our caretaker built. We have our employees, object are originator. Are memento created and our caretaker. Let's go ahead. Look at the demo. Now I've pre coded a demo for us here and inside of here. You can see we create an instance of the caretaker object and an instance of the employees. And then we go through and sets and values on the employees. I said a fake name set name said Address, phone number. And then we call save. Then we change the phone number and call Save again. Then we set the phone number and we don't call Save. This is a key point. I've got the comment. They're showing that we don't call, save and call Revert, and it will just set those values back to where we were at. And then I'll call Revert again, and I'll take us back to the original. Let's run this and see that it works. I'm gonna right click and say, Run as Java application and you'll see that our employees, before save our employees before save was that value. Our employees after changed phone number save was that value and then we changed it. But we didn't call save and just reverted back to it and then reverted to the original. So you see that we have our full history implemented and the emphasis that it puts on those values. Do you see that we have our our object that We changed here but didn't call. Save on it and we went ahead and reverted back. Sometimes people get a little hung up on that. I haven't called the save, and that hasn't popped anything back on that stack. You see that we have a full history of our object available, and all we had to d'oh was change. Our employees object. Add that save method and that river method and then our memento itself. That just creates that state. Now you can change this to have whatever pieces you care about for reverting notice. We left out the address. We purposely did that. If we were using the serialize herbal interface, we could mark something as transient. But we just went ahead and said, We don't care about that for a moment, our memento and left it out and then inside of our caretaker, we have our history here that's implemented with Stack. You can choose another implementation of the array list or linked list or whatever other tools you want to. But the stack really. Wilke works well for this demo and then finally, our final result of implementing with the caretaker and the caretaker tracks are history and rules that out for us

Pitfalls

[Autogenerated] now that we've created our own memento, let's look at some of the pitfalls of using this pattern. It can be expensive. If the memento is a large copy of the originators data, we might incur a lot of overhead with each copy that we store. The caretaker, although maybe simple and lightweight, needs to consider deleting that history or how much of the history it should keep around. You also need to be careful not to expose originator information. State needs to be transitioned to the memento, but not outside of their. In our example, we had methods for the access er's, but we didn't convey what our state was. Two other objects, just the memento.

Contrast to Other Patterns

[Autogenerated] to contrast the memento pattern. Let's compare it with the command pattern. The memento is used to capture ST. Each state that we capture is independent so that we can roll back or recreate it in the future. Memento is also focused on building a history with the caretaker object. The command, on the other hand, is very similar, almost identical

in fact, to the memento pattern. But it is focused on requests rather than the state of an object instead of independent state. It's focused on independent requests. The main difference with command is that a lot of people implement the command without thinking of history. It is available, but typically not the focus of the request of this object. History is just a side benefit benefit. Typically, when using the command all the pieces of their though the option is available, most people don't utilize it.

Summary

[Autogenerated] Let's recap what we've learned with the memento it's used to capture ST. We have to be careful because it can get heavy with history. If we have a lot of Mementos that we've stored, or it's a large amount of data that we're copying for that state, they can get quite large. We use it to recreate state of an object. So once we've captured it, we can then return an object to that state. And it is very similar, if not almost identical, to the command, but with a different emphasis on history and state rather than requests. The next pattern that we're going to look at is the observer pattern, which is one of my favorites, and one of the very first patterns that I learned when trying to figure out the power of design patterns.

Observer Pattern

Introduction

[Autogenerated] Hi, this is Brian Hansen, and in this module, we're going to look at the observer design pattern. The observer pattern is a decoupling pattern. When we have a subject that needs to be observed by one or more observers. Let's look at the considerations when choosing this pattern.

Concepts

[Autogenerated] the concept surrounding why you would choose the observer pattern are that when you have a situation where a subject has one too many observers, this is generally when we're trying to decouple objects. Usually we will use this pattern but are not limited to situations where we need event handling capabilities. Some people use the term publisher subscriber interchangeably with the observer pattern. But there is some debate here. The observers typically used in a synchronous fashion where the pub sub model is typically used a synchronously. This pattern is most often used in NBC situations, where the view is event driven. Examples of this in the job, a p i. R. The observer pattern just like the generator pattern that we covered earlier in the course, the creators of Java felt like the observer pattern was useful enough in an abstract way that they baked this functionality into the core. A p I. The job you till event listener is the base interface for virtually every event, an event listener and ate a beauty and swing. This same pattern is also used in J. M s when implementing topics for message driven applications. Let's look at the design of this pattern

Design Considerations

[Autogenerated] the design of the Observer is that of a subject that needs to be observed. The subject is typically an interface or abstract class that we make concrete implementations off. The subject is a class that observers will then register themselves with. The Observer Self is interface based with various concrete implementations and in the case of the Java Util observer implementation, the subject implements an observable interface. Concrete observers are typically views in an event driven application. The pieces of the UML diagram are a subject, a concrete subject observer and concrete observer. Let's look that UML diagram now.

UML Diagram

[Autogenerated] This is the UML diagram for the observer pattern. The subject class can be an abstract class or a concrete implementation. The attach and detach methods are to register observers for notifications. The notify methods cycles through each registered observer, calling it's update method. The concrete subject is an implementation of the subject if they have been separated out, although often times it's just one object itself. It contains the state that the observers are wanting to be notified about. The Observer interface contains one method. The update method that gets called when something changes in the subject that issues and notify the observer interface also makes it so that we can decouple observers from the subject. Since we handle everything uniformly and consistently through the interface, the concrete observers are simply implementations of the observer interface and are the objects that act upon the changes in the subject. These air typically views in your application. Let's look an example of the pattern using the job you till observer interface

Example: Util

[Autogenerated] the job you till observer interface. And Javi, you till observable abstract class provide much of the functionality that we want to use to implement our own observable pattern. In this code, we're going to simulate a Twitter news feed with messages and notifications that someone has tweeted something that we're following. Let's look at what it takes to wire up this code using the observer and observable in a live example now.

Demo: Util

[Autogenerated] So in this example, we have a couple of basic pieces we have are observable or our subject and our observer. And I've created two classes inside of here to simulate Those two are Twitter. Stream is the object has some state that we want to observe or our subject and our observer, which is basically our client or display that's gonna be updated from our from the changes of state on our observable subject, you'll notice inside of the stream that there is a method called someone tweeted where we set changed and notify observers with the implementation of observable and observer, you can see that we have to call set changed first because it signifies to the observer that we have changed our state so that notify observers actually gets fired. They notify observers, met that goes true and calls the update method and just basically prints out this system out print line to our

screen, saying that we have updated so on so stream that they had tweeted something. So we can you get the idea that of the messenger of in nature of this application and set this up that we went through and just created an instance of our concrete subject, our Twitter stream we created to clients and then we added them is as observers and then fired off a tweeted message. I'm gonna run this, but I'm gonna point out a couple of shortcomings inside this example. That's not a problem of their examples, actually, a problem with the way that Java uses the job, you till observer rule and observer interface. And after class, let's run this. I will see that it did in fact, fire off that message that Mark Stream was updated and Brian Stream was updated. You see down here that we didn't have a direct call that it notified it through the notify observers method that we call up here. So it's loose. Couple. We've achieved that decoupling. All we had to do was create these two clients, registered them, is observers and then perform something on that subject. Well, if you remember Ruml, there's a couple of problems here. Are observers really know nothing of the concrete subject that we're trying to observe? And so there's they're a little too disconnected in that sense. We can't actually grab an instance to that object at that time to see what's changed, and there's a little bit of code or or wiring up that has to take place here. In the main method you see, that seems like it's a little bit of overkill for what this pattern is. We're gonna go through and rewrite this code in our own observer observable, and you'll see how we eliminate a lot of the shortcomings from just using these built in interfaces.

Exercise - Observer

[Autogenerated] two better suit the observer pattern for what we're trying to do. We're gonna go ahead and create our own implementations and show some of the shortcomings of the Java Util observer an observable implementation. To do this example, we're gonna create our own subject concrete subject observer and concrete observer. And then we'll compare this pattern with the Java Util examples to see the shortcomings of what they've done and how ours is better suited to what we're trying to do within this pattern.

Demo: Observer

[Autogenerated] to shorten this demo, I've gone ahead and created a couple of interfaces and abstract classes based off the UML diagram that we looked at previously. You can see here and the observer demo class that I've created thes message stream hasn't been created yet. It's a concrete implementation of the subject. The subject is a very basic implementation, but complete implementation of the observable or subject class you can see here we have a list of observers. We have a set state a get state that are both abstract because they're gonna be implemented by the concrete implementation and then attach observer detached observer and a notify observers method. Now, if we go back to our observer demo, the next thing you can see that we create our various clients, a phone client and I've got commented out a tablet client. The phone client is a conclave concrete implementation of an observer. The observer implementation or interface is quite light. It's just a nabs tracked class and could honestly be a interface itself. But we do have a reference to the subject inside of here. Now, this is where it differs from the Java Util observer and observable implementations from the job You till eight p I. We have a reference to the subject in sight of our observer that we can get state from. We go look at our phone client. You can

see that in our constructor we pass in the subject that we're dealing with and we actually do the attachment there. This is where it was a little bit weird with our job, You till observable example. We were doing all this set up ourselves in the client where the actual clients or view modules are now taking care of this themselves. We also have access to get state from this object as you'll notice we do it down here in the update method down below. Let's go ahead and create our message stream so that we can finish out this example of our concrete subject. So here's our basic subject. We're going to go ahead and create a concrete Strub subject of this class. So let's go ahead and right click on our observer and say, knew class and we'll call this the message stream and it is going to extend subject quick, finish inside of here. We have a couple of methods that we have to implement as part of our contract with it being an abstract class from subject subjects, abstract requires us to implement set state and get Steve ST go back to our message stream. Now, this is where the realism of this example comes into play. We want to provide what a riel case would be to use observable in sight of our code and this message stream. We're gonna keep a message, history or other things. So think about if you go into Twitter vital. I get it on my phone and I look at it on my laptop or my iPad or tablet or whatever, They aren't necessarily the same history. I'm not looking at the same point in history, but that history is stored for my entire stream. So we want to do that same things. We're gonna create a little implementation. We're going to see private deck, and we're gonna do this of string, and we're gonna say message history equals new double ended Q type string. I actually wanted a rake. You Honore ____, You perfect. That's what we want our example Look like now this is going to be used to store our message history. So any time somebody post something to it, obviously, in a more concrete example, we would want to limit what that history could be. We would do that in our set state methods and her get state methods where we could trim that history of Let's go ahead and get rid of this auto generated implementation here and we'll say Message history dot ad and will pass in the string message. Let's change this just for clarity's sake to message. So it's clear what we're adding in here, and we'll do the same thing for our get state. We're gonna say Return message. History got kid last. So get our last message and save that Now we have the basic implementation of our observer. Done. Know how this works is we have our observer and are Concrete observer, which is a phone client. Implements that or extends that abstract class would implement that interface. If we didn't care about the subject being passed in where we go ahead and register ourselves as a observer of the subject that we've passed in, then we have the opportunity to add messages that actually go ahead and change that subject. State that we're watching and we update from our notify observers that gets called in our message Street. Now our message dreams are concrete implementation of our subject where we can add in our message. We need to do one more thing to notify the rest of the listeners on this After we have said our state. When you say this dot notify observers and click save now be careful there. You'll notice there's two other methods in there that are part of the job. You till object. There's notify and notify all these air First the Reds. That's why we had to color method notify observers. So don't get confused there and put the wrong implementation in ever notify observers here? We're actually ready to run our demo. Now we come in here and say, Run at US Java application. It will now print out our message. Here is a new message. It was sent from her phone and you can tell how he did that. And he said, phone client ad message. We never called that update method ad message inside of our phone client ad message sets the state the set state in our subject. Our message stream then adds that to our message history and calls this Stott notify observers. This in turn goes up through our subject and cycles through all of our observers and calls notify on them notice that we don't directly call the update method. Anyone? Our code, our observer demo are our original

client never calls the update mint that it's called through our observer and our subject contract. Now what happens if we want to add another observer? So we did all of this work Seems like a lot to just have a call out. A basic method. Well, this is where the power comes in. We want to go ahead and add another client. So let's create a tablet client. We're gonna create this one together. I'm gonna go ahead, right, click and say, New class tablet client, and we're going to say that it is. We'll just finish. This extends observer, and there's a handful of things that it's gonna want us to do. It's gonna have us add these unimplemented methods. The update I've actually got the implementation done. I'm gonna copy and paste it in just so you don't need to watch me type, we'll grab these methods here and paste it in. So now we have a basic constructor that takes the subject in. We register ourselves as an observer on that subject, and then we have the option to add a message here and notice that just to differentiate the two, I've put a slash sent from tablet in the Tablet client and a cent from phone in the phone client. Now I am just dealing with strings here. You could make a much more complex object and pass an object around in store objects, a string as an object, and you could do something more complex, just a short in the demo. I made it a little smaller by saying that we were just going to use the strings here instead of the concrete object. Now we have our tablet client, and as soon as it's done compiling, we get rid of that red line. We have our basic subject that we passed in. We've registered ourselves and our update message that's different from our phone. Let's go ahead and uncommon that tablet client line that just goes ahead and adds another message on there and run this again, say, run as Java application and you can see that we have a message sent from our phone that's updated the stream on our tablet and a message that sent from our tablet that updates the stream on her phone and her tablet so thes two objects know nothing of one another. The only communication has handled through that subject. Now, this feels a lot like the mediator, and we're gonna talk about the differences between those two because they're very similar patterns in nature. But before we move on, let's look at the job. You two example that we had where we had to set up our client there, notice we had to create an instance of our subject and then are two clients and then our actual client code. Our main method had to know about registering them and communicating through the subject, which feels a little weird. Why should we communicate with our subject directly? If I'm on my phone, I don't have to go to my desktop to update Twitter. I can update it from my phone or I can update it from my tablet or I can update it from my desktop. I shouldn't have to goto one place to update them, which is where our examples a little bit better. In fact, if I put those two side by side and show you our demo, you can see that in the example that we did before using the job you till observer. There's a lot more code in here that fills like it shouldn't belong here. The registration and how we communicate with the subject is a little bit different. We're doing Maurin ours, and there's fewer lines of code, and it just seemed a lot client cleaner from the client perspective from our actual main method.

Pitfalls

[Autogenerated] What are some of the pitfalls of the observer? Well, since the subject doesn't know about its observers, there could be unexpected updates. An object can notify that it has changed without knowing what has changed. This leads to the next point, and that is of large update consequences. If an object is really large and we traded an update that forces that change, there can be a performance hit, even though we may not necessarily want update. I think of it in terms of something like files and notifying a change in a file for copying 100

make file every time something's been updated, we might not always want to do that. Although not using this pattern isn't going to make that risk go away, either. It's just something to consider when you're implementing it. Overall, just not knowing what has changed With this disconnected system, it can be confusing as to what has changed in that subject that signified such event to be updated. This disconnected nature can makes debugging often difficult and a little bit troublesome to work through

Contrast to Other Patterns

[Autogenerated] to contrast The Observer. Let's compare it with the mediator. The Observer is a one too many pattern. We have one subject and many observers of its state. It is primarily used to decouple an object from those that want to watch it. One of the key difference is is that it's uses a pub sub array broadcast communication mechanism. The mediator, though, is more of a 1 to 1 too many model and an object talks to a mediator, and then the mediator is in charge of communication with other objects. It's also used for decoupling but usually used for handling complex communication in a more direct model than that of a published subscriber at pub sub model. It should be noted, though, that the mediator can be implemented with an observer in the notification mechanism. The mediator would simply be part of our subject and to take our Twitter example a little bit further. That's actually how we would probably implement something like that in Twitter is our client would communicate with a mediator, which then in turn would broadcast through an observer

Summary

[Autogenerated] Let's briefly recap what we've learned about the observer pattern. It's one of the first patterns that I learned when first learning about patterns, and I really like it. It really strikes home. And I, like the publisher, describe broadcast model of the observer pattern. It's often used for decoupled communication where we have these objects that we want a broadcast stuff, too, and they don't necessarily know about one another. Our phone doesn't necessarily have to know that our tablet updated our Twitter feed. It's got some built in functionality in the job. A P I, the creators of Java, thought that this pattern was important enough that they built that into the job a P I and it typically could be used with and should be used with the mediator. If your case warrants it, I wouldn't shy away from using those two patterns together. If you're in a very complex situation that might benefit from both models, the next pattern we're going to look at is tthe East eight pattern, which we use to encapsulate the state of our application inside of an object

State Pattern

Introduction

[Autogenerated] Hi, This is Brian Hansen. And in this module, we're going to look at this state design pattern. The state pattern is as it sounds used when we need to represent state in an application.

Concepts

[Autogenerated] the concept surrounding why you would choose the state pattern. There. You want to localize state specific behavior. More specifically, the state pattern makes it so that your current application's state is stored in an object rather than a mix of variables throughout your application. The state pattern also helps us separate what state we're in from where we're at in our application. It effectively makes your objects follow the open, close principle of software design. The class is closed for changes, but the states are open to extensions. Examples of this in the job a P I. R. Actually, none of all the patterns There really isn't a good example of the state pattern in the core, a p I. There is an instance of this pattern in JSF, with its life cycles and phases, and some people argue that the Liberator implementations are implementations of state pattern. But I don't feel that they really are The reason, they argue this is that because it can change objects behavior runtime based on its state. Let's look at some of the design considerations when choosing this pattern

Design Considerations

[Autogenerated] the design of the state is abstract class or interface based. More often than not, this pattern is implemented with an abstract class. Because of the various states, an object can be in don't necessary apply to every instance of the state class and a rather just a default implementation. Each state is then class base and represented by an individual concrete class per state. The context or subject that has multiple states is unaware of various states using this pattern where before it knew all of those pieces. If this sounds a little bit complex, look at it this way. Before the state pattern. One object had a very large, if then else diagram, innit? That went through all the different cases that we could have inside of our application. Whereas once we implement this pattern that's externalized into each condition being its own state class, the pieces of the UML diagram are a context, a state and then the individual concrete states. Let's look at that diagram now

UML Diagram

[Autogenerated] Here's the UML diagram for the state pattern. The Context class has specific requests that air passed through our state interface or abstract classes. I choose to call it our contract that we're guaranteeing to our context for each state that our application can be in. There is a concrete states such as thes illustrated here. Overall, it is a pretty simply UML diagram. The complex T of this pattern comes into play when we were working at removing those if else blocks from our code. This helps us reduce what is called cyclamen attic complexity. Inside of our application, those many paths that we can go through each method, let's look at what this pattern fixes in a live example of our code now.

Example: if/else

[Autogenerated] Since there really isn't a great example of the state pattern in the core Java a p I we're gonna go ahead and walk through how a lot of developers will mimic state in their application. And that is through the use of managers. You can see in this example we have two states in our application and on state and an off ST. And this is all fine if this is the only

two things that were going to represent in our application. But in the case of this fan example that I'm gonna walk us through, we have a lot more state that we want to add to it. So let's see how this transform and how that if else statement continues to grow and where how we can utilize the state pattern to minimize that are actually get rid of that cycle medic complexity out of our applications altogether.

Demo: if/else

[Autogenerated] Here's a basic implementation of an application that would track state. I've got a simple class here that's gonna go through and simulate us, pulling the chain on a fan to turn it off and then to the next speed and vice versa. And as I run this demo, you can see run as job application that it's going to just go through and say the fans off. Now we're turning the fan on the low, the fans on low and we're turning the FAM back off, and now it's off. So it shows you what the state is, and it shows you what the action is that we're performing. This code is really quite simple. There's nothing that complex about it looking at it and you'll see we have an often a low and then for each time we interact with it. When we pull the chain, we look at what the state is, and when we have to print out what are two string as we have to do the same thing as well. Now let's look what happens if we want to add another state, so we want to go through an ad medium on hair souls Go ahead, say medium and set that to an imager of two. And now we're gonna come in and add another, if else block. Now, one of your first clues with any pattern that you're maybe doing something wrong is if your first instinct is to copy and paste to the next state. Now notice. I also have to come in here and say, Turn state, we're not gonna go to turning it off. Now we're turning fan onto medium, and the next day is medium and mediums gonna go to off now and set that state off. So we have our three states represented here. Notice how it gets pretty complex, pretty quick about the pieces that were doing. We're also gonna go down here to our two string and say if the state's off for the fans off its low turned low, if it's medium, we're going to print out that it's also to medium and his own medium. Now let's run this and see what it does here. Everything looks like it should run just fine. And it does print out. Fan is off and is now low. Fan is medium and so forth. Now you can see that as we go through here in ad states, it gets more and more complex. And look at the conditional statement that we have honestly, with our fan example. It's a pretty small example in the sense that we're probably only gonna have four states, maybe five. If you throw lights in, there may be some more states, but you see how this gets a little bit unwieldy and think of a little bit larger application of all the pieces that we have in there. We want to keep this example small. So that's why I chose this fan demo. But notice how the copying, pasting and how this class is just growing and growing. When I asked this question at local user groups or things that I'm giving this presentation for this state pattern, the response is always we've we've implemented a huge state machine before. That really wasn't a state machine. It was just this big. If then else statement. That's a couple of 100 lines long. So this is what we're working on, reducing the cyclo Matic complexity of these conditional statements. Let's see how we're gonna transform that now into use, utilizing thes state pattern

Exercise - State

[Autogenerated] Now that we've seen what appears to be a fully working example of state being represented in an application, let's go ahead and make some modifications to it. We're going to add a state abstract based class, and this could be an interface. But we're going to add some default implementations to it and dust. We use the abstract class instead. Next, we'll create a concrete state object for each of the states that could be represented inside of our application. And lastly, we're going to remove any conditional logic that was surrounding the old manager approach and show how the state objects contain and eliminate that now.

Demo: State

[Autogenerated] to get started with converting our fan to a implementation of the state pattern. She wanna point out that our fan object is exactly where we left it off at. We have our off low and medium states. And what we're working to do is to get rid of these. If l statements that Aaron are to string and our pull chain and all that different cycle Matic complexity that's contained within our application to start off, I'm gonna go ahead and create a state object If you notice in my example, I have to read X is down there. I've copied a couple of my states in for just keeping this demo going faster. You'll quickly see that they're just the same implementations for the high end medium state. So let's go ahead and create a new class and hair and we're gonna call this state and it is abstract and we'll click, finish, and then what we're gonna do is gonna come in here and we're gonna put one method inside of here. It's a public void handle request, and then we're going to say system dot out of print. Lynn and I'm just gonna put a default string in here that says that we shouldn't be able to get here now. If you wanted to, you could throw an exception here. You could do a couple of things. It really depends on how you use this pattern. This particular case, I only have one request that we're trying to handle cause our fan is very simple. We pull the chain and if I'm in the lowest state, I pulled the chain. I'm gonna go to medium. If I pull the chain again, I'm gonna go to high. If I pulled 1/4 time, I'm going to go to the off ST. Now think of it in different terms off different things that I could be working with, such as opening or closing a garage door or dealing with lights on a fan or some different things. So I might not always just have the one handle request method in here. And they have different things that I'm working with. So now that we have our state are abstract based state, let's go ahead and come back to our fan and look at what we're gonna do here. We're gonna get rid of these final static ends at the top of our fan class, and I'm gonna start with just getting rid of one will say state, and we want to go ahead and start off. We don't have the implementations for these. Yeah, so we'll say fan off, and then we're gonna do another one. That is state, and we're gonna do Fan Lo State. And just for sake of consistency, the fan off state fan Low stayed state fan, medium stayed and while we're at it and I'll wait to add this implementation till later But we'll do a fan High state. So let's get rid of those two static final ends there. And now we're gonna change this because we're no longer tracking state by an imager. Going to say state equals nothing yet because we're going to sign that in our constructor now. So we have a state variable that we're tracking what state were currently in, and now we can go through her application. So from here, what we want to do is start initializing what the's various fans states are. We'll start off by saying fan off ST equals new fan off state, and we haven't created this yet. Passing the variable instance, this. Let's go ahead and create this object. I'm gonna just

right click on our package and say, Knew, Class and it is the fan off ST and it's super Class is ST Click Finish now inside of here. We need to do a couple of things. Bear with me on this 1st 1 It's not that complex, but there are a few moving pieces first. Want to create an instance variable to hold a reference to our fan in, and then we're going to create a constructor. Wants a public fan off ST Fan fan. We're gonna sign that to the variable that we've created. So we'll say this dot Fan is equal to fan. Now we're almost done. We're gonna say at Override, and we're going to say Public void handle request. And now we have our basic signature that we need here going to do two things inside here. We're gonna say system dot out print land, and this is just so you can see what's going on. We're going to print out the verb or the action that we're doing. So if we're in the off state and somebody pulls the chain, we're gonna turn the fan on too low to say, turning fan on too low. And then this is where it gets a little bit interesting for people. We need to now say fan dot set state. We haven't created this method yet, and we're going to get an instance of fan dot get fan Low State. So this is the state that we're transitioning to. So as we are, if we're in our fan off state and we pull the chain, we're going to now set the state of the fan too low. I had somebody asked me a question while I was giving a demo on this at a user group meeting as to why we wouldn't just create an instance of the fan Low State here, rather than getting it from that fan, was part of that state machine. The state machine knows about the different states that it can be in, but the individual states on Lee know aboutthe one that they can transition to. And so instead of having the fan off state create a new instance of the fan Low State, we get another level of indirection and that I can return it back if it doesn't make sense, not a big deal. I just have had some people ask questions about while we're using a getter in a center in this method right here. Now, before we go create those two methods, I'm going to finish out this class and create a very basic to string in here to represent the state of this Objects will say public string to string We're gonna return The string fan is off Now this object is done. We just have to go put these two implementations in our fan for setting the state and getting the fan Low State. Let's go ahead and use the quick fix method here. We're gonna create a method. Get fan, Low State and Tight fan. It's gonna return an instance of state and we're going to return fan off ST Now that's done. The next thing we need, D'oh! Before this class will compile over here is we also need to do that fan set state. We're going to create a method fanned upset state. That's also in fan. So now we have our fans state. We're going to say this stops State is equal to ST Now that we have that all done in our fan off state is compiling. Let's start cleaning up our fan object again. The goal of this again is to remove all of this if else stuff that's going on in our poll chain and to string method. So I want to go back up here to my pull chain and all of this stuff is going to just get deleted. And now it's gonna be ST dot handle request. Now we're not completely done, but we are going to be able to do the same thing Hair will say. Return ST dot to string. So look at our fan object Now. There is next to nothing in here. We haven't represented all of our states yet, but we were able to now get rid of that conditional code that's going on everywhere. Let's finish implementing out the rest of our states. I'm gonna go back in our constructor and say fan and we're gonna go to our low state and say equals new fan Low State. If you remember, this is what we were just implementing down below with the get Fan Lo ST method that was required for our fan off state because the off state will go to the low state in our low state. Is it going to go to our medium state? And so on now inside of here, we can go ahead and create our fan Low State, and it's going to look very similar to what we did with our off ST. So when we go in here to our low state, we will right click and create a new class, and we'll have fan Low State and it's based classes state, same thing. And for the sake of time, I am just going to copy and paste the innards out of the off one to the low one. Be careful doing this because you will make some copy pay

stairs and we're going to say that it's to string is low and the constructor is obviously the family estate. Now we just need to change the action that we're doing inside of the handle request. So if we're in the low State, we're gonna now be turning the fan on two medium and we already have the set state there. But the next day we're transitioning to his medium, so we need a get fan, medium, state and of course, this is gonna break because we don't have that created yet. So now we go back to our fan object and we have, ah, fan medium state Instance variable here. We just haven't created anything for it yet, So let's go ahead and do that same thing now, mind you, I had already copied a medium and high state in here just to save time. And I'll show you what the code is. It's exactly what we've done with the off in the low state. So say fan medium stayed equals new fan Mead State. Pass in this and say this instance why we're in here? I'm also to go ahead and say that state starts off as fan off ST. So that's just what it's initialized too. Now, we don't have our fan Lo ST just yet, so I'm gonna come down here, right, click and say source generate getters and centers and all I want to do is to get her. We don't want to have that be able to set the state will come up here to our get fan medium state click. Okay. And it will drop that in there for us. Now. We are basically all done. There is one catch. We haven't done that same thing for our high state just yet. So as an example, you can do this on your own. I'm gonna go ahead and implemented as well. Let's just create the fan High State in here and now you're starting to see how easy it is. First, add additional states. So say fan High State, all I have to do is come through here and create a high state just like the other states we have been creating. So now we have high go through and say Fan, hi. And the only thing I have to provide is that get her down here for that high state come right click source, generate getters and centers. And again, I only want to do the get her for it. So we'll say, Get fan High State and save that. And now we have all of those states represented inside of our application. So notice there's no if then else nothing. We've created all those states, and now the fan off state just knows about the low state. The low state knows about the medium state. And if I open up the medium. You'll see. It's the same thing. I'm gonna go through here and uncommon that so that it was about the high State and same as the high State. It's going to cycle back. Toothy off ST. Pretty straightforward. Now we haven't created our method to get our fan off state yet, so we can go ahead and add that into our fan object. Come up to the top here and right click and say Source. Generate getters and centers go to our off state just at a getter for the fan Allstate, click. Okay, save it. And now all of our states are represented here. I have my off low, medium, high state, the set states and our two strings are all implemented. Are our state objects all contain what they dio None of it's contained in the actual state machine itself. And now we can run our example. So let's go ahead and go into our state demo now. One very interesting thing to note about this our state demo. Our client didn't change it all. This is all about fixing complexity of the back end. So if I go ahead and hit run on this, you'll see that we now go through all of our states from off the low to medium. We can add a couple more in there so that we go to see the entire round trip. Let's just grab a couple of those, and that should get us toe high. And that should get us off. And we'll save this and run it again and you can see that we do the full round trip. So we go from off the low to medium to high and then backed off. So that's how we've changed our application to not be this conglomerate of, if then else statements. And at first it seems like this is a lot more work. Think if you're doing a lot more complex things in your state rather than just printing a system dot out print land. In fact, for a live demo, I like to hook this up to a raspberry pi or some Arduino and have it interact with light, and there's always some code wrapped around doing that. So thes states usually aren't just this simple. One liner with a system out print Lynn and a little transition in there were usually tying that to some other interface that we work with on the back end, and that's all just contained and signed this

handle request method. Now, if I pull up a older example of what we had there, I have one say that our demo notes. Here you start seeing that we have things where the state is open and closed and all that various into jurors. This is a particular garage door instance, but you see that it gets a lot more complex for those individual cases. And if we add one more to it, it's just a lot more copy paste code that goes into our application. So getting these all combined in their own states and contained is sure a lot cleaner way for our actual object we're trying to contain state on.

Pitfalls

[Autogenerated] Now that we've seen, how do you convert an application to use the state pattern? Let's look at what some of the pitfalls are. You must know all the states of your application. This may seem obvious, but often times people don't go through the exercise of clearly identifying each state, and what is needed to revisit represent that state in an object. When implementing the state pattern, you will invariably end up with more classes than you would have if you had implemented with the more traditional, method based approach. This was even evident in our small example. Another thing that you need to be careful of is keeping logic in the state objects and not letting conditional logic creep back into the context. Another common misstep is not identifying what triggers a state change. In smaller examples, it's easy, but clearly identify what triggers that change in the state of your application

Contrast to Other Patterns

[Autogenerated] to contrast the state pattern. Let's compare it with the strategy pattern. The state pattern is interface based, with a collection of concrete states estate on. Lee knows about the next state that it can transition to each state is also contained in its own class. The strategy pattern is actually almost identical to the state pattern, but its focus is different in that it is on algorithms, or strategies instead of state representation. It is also interface based, just like the state pattern. But one major difference is that strategies don't know about the next state. Their algorithms are independent from one another. They also like the state, have a class per algorithm. As the state pattern has a class per state. The UML diagram is almost identical for the strategy pattern, as it is for the state pattern.

Summary

[Autogenerated] Let's quickly recap what we learned with the state pattern. It simplifies Psychlo Matic complexity, thus removing those, if else spaghetti code blocks that we have inside of our application. It makes adding additional states much easier instead of having to go through each. If else blocking our application weaken, just add a class and change the state before it to point to this one. One of the drawbacks is that there are more classes involved with the state pattern, but that's not necessarily a bad thing. They're typically lightweight, and they are definitely isolated and contained, representing that open, close principle that we talked about earlier and software design and it's very similar and implementation to the strategy pattern. The two UML diagrams are honestly identical, and their focus is very similar. The only difference being is one's focused on algorithms

where one's focused on ST with that comparison, let's look at the next pattern, which is thes strategy pattern

Strategy Pattern

Introduction

[Autogenerated] Hi, This is Brian Hansen and in this module where you're going to look at the strategy design pattern, the strategy pattern is a behavior pattern that is used when you want to enable the strategy or algorithm to be selected at runtime. Let's look at the concepts when choosing this pattern.

Concepts

[Autogenerated] the concept surrounding why you would choose the strategy pattern or that you want to, or see the need to eliminate conditional statements in your application. The strategy pattern encapsulates the algorithm options in the individual classes that eliminate the conditional statements in your application. Another key point of when you would choose the strategy is if it's difficult to add new strategies or cases inside your application. When using this pattern, the client is typically aware of the strategies that are available and usually chooses the strategy that we're ultimately going to use in our application. A great example of this pattern in the job a p I is Java Util comparator. The comparator is used when we want to implement various comparison strategies in our application for passing into a sort method. Let's look at the design of the strategy pattern now

Design Considerations

[Autogenerated] the design of the strategy could be implemented with an interface, but typically it is usual izing, a abstract based class. All of the concrete classes are then implemented per strategy, where each concrete class will then implement the algorithm that is unique to that strategy. A key point with the strategy pattern is that we want to minimize, if not eliminate, if else condition ALS from her application. A key point with the design of this pattern is that the client knows about the strategies. But other strategies don't necessarily know about each other. The pieces of the UML diagram are a context, strategy and concrete strategy implementations. Let's look at that UML diagram now.

UML Diagram

[Autogenerated] This is the UML diagram for the strategy pattern. The context receives a request, and it decides which strategy to use based off of the situation strategies all adhere to the same interface. And then, finally, the concrete implementations of which there may be many are then selected based off the situation. Each strategy is contained within its own class. Let's look at this pattern in a live example now using the comparator that we talked about earlier.

Example: Comparator

[Autogenerated] the Comparator is a great example of the strategy pattern. Here's a snippet of a larger example that shows how we can swap out the sorting strategy of a collection by implementing a new Comparator and this example we're sorting list of people by their age. Let's look at this in the larger example now.

Demo: Comparator

[Autogenerated] to make this example go a little faster. I went ahead and created a person object that just has an age name and phone number and then a default constructor and one taking the name, phone number and age. There's some simple getters and centers, but besides that, this is a plain Poggio with nothing out of the ordinary in it. Inside of our main method, I've gone ahead and created three instances of the person object Brian, Mark and Chris with their associated information and then added those people to Honore list and then finally created a print contents method that just cycles through each item and prince amount. Now, if we run this just right, click and say Run as it's going to print out that Brian, Mark and Chris is not sorted, and it says that sort of by ages. Brian Market Chris, which isn't true because we have not sort of that by age yet. So let's go ahead and add that functionality in here. Oh, I'm going to do this using a anonymous in her class, so we'll say collections dot sort and we're based this off of our people list, and I'm gonna add in a new comparator object. Now, I could do this, not using an anonymous in her class, but it works for this example. So I'm gonna go ahead and put the person type in there, and then we're going to add the unimplemented features and inside of here, you'll see that we have to implement the override. And if you haven't done an override before using the Comparator, you just need to go through an implement the compare method on that method and it will ask us three things will ask us to return a one. If this object is greater than that, one less than one. If it's less than that object or a native one and zero, if they're equal, is gonna go through and say if 01 dot get age is greater than 02 dot get age then we want Thio return one, and then we have to do the same thing. If it's less than so, I'm gonna say if 01 dot get age is less than 02 dot get age. Then we want to return negative one. And if they're equal, we returned zero and you could write this a couple of different ways. This is just a quick and dirty way to get that return functionality built in there. We go ahead and run this again. We're gonna see that. Now it's sorted by Chris, Brian Mark, which is by our age, if you come up here. Chris is 38 Brian's 39 marks 41. Well, we have one other thing that we can do down here that I've actually stubbed out to sort by name. And if I ran that right now, it wouldn't be sorted. So let's go ahead and do the sort by name Comparator as well, just to help cement in these ideas and concepts of the strategy pattern. This one's a little bit different because I'm going to utilize the compare to method in sight of the string class to help us, right? This is going to say collections dot sword. It's just like we did the other one. We're going to pass in our people object and then new comparator, and we want to utilize the instance of generics with our person object and now override that compare method inside of here. I'm gonna not really cheating him to go ahead and use the strings compared to method m s a return 01 dot get age or name on this one Because we're gonna do it by name and say dot compare to, 02 dot get name and then close off our anonymous in her class And are you just utilized the strings compared to feature So it's already built in. It's already done for us. Now if I run this again, it will go ahead and show us our first instance where they're not sorted where it's Brian, Mark

Chris, which is how we add them to the array list. Show us how it's sorted by age. Chris, Brian, Mark And then finally show us how it's sort of by name. Brian. Chris Marc. Now we have thes two different strategies built into our application that are implementing the Comparator interface, which in this instance would be our strategy interface. We have our concrete implementations. You see how we can swap out how we're sorting that list very easily, and it also proves another point, which is Thekla. I in't knows about the strategy that we're dealing with

Exercise - Strategy

[Autogenerated] Now that we've seen the strategy pattern in action, let's go ahead and create our own. We're going to create a context, strategy and concrete strategy, and they're going to show how we switch strategies inside of her application to utilize the different concrete instances that we've created.

Demo: Strategy

[Autogenerated] for this exercise. The context that we're going to be working in was that of a credit card. We're going to go through and implement a credit card validation that utilizes multiple, different strategies to start with. I've gone ahead and created a main method and added the card credit card information in here. I'll tell you right now, this is not my real credit card number. So sorry to get your hopes up. If you're not aware, you can actually Google credit card numbers that are generated because a credit card is actually the number itself is very simple and generated using a check digit and a sum to verify that it is a valid credit card. Enough of that, though. Let's get rolling with this exercise, and we're going to start by creating a class, and we're going to implement the credit card to begin with. So we're going to say it's a simple class credit card, and inside it here there's a couple of things that we want to pass in. We're going to do a private string number, which is just gonna represent the credit card number, private string date, and for this example, we're gonna keep it simple and just keep them all as strings and then a private string cvv number. Now, before we finish this out, there is one other piece that we're gonna do. We're actually gonna jump ahead and start by making our strategy in her face and to do so. The reason why is we're going to implement that is part of this credit card to do so we're going to just go ahead and create an abstract base class, say, knew class, and it will call it validation strategy. And we want to check abstract on this and go ahead and click finish. Now, to start with, there's only one method we care about in here. We're gonna add another default implementation inside of this a little bit later on, I'm gonna say public abstract Julian is valid, and this is just to validate the credit card that we're passing in is, in fact, a valid credit card based off the credentials and criteria that we've set up. So we're gonna pass in the credit card instance and go ahead and save this method. Now we can go back to our credit card and finish up a couple of things here and the credit card class itself will be complete here. So we're gonna take and pass in a private validation strategy and we'll just say that that's our strategy. And then we want to go ahead and create our actual constructor for this. So we want to say, Public credit card and we're going to pass in the validation strategy to the constructor. It's a strategy, and then we want to go ahead and save thestreet egy instance in here. So I'll say this stop strategy is equal to strategy. Now we're almost done with the credit card class itself, but I'm gonna go ahead and

put a convenience method in here. We don't have to necessarily do this as part of the strategy pattern. It's just a lot easier way to work with it. In terms of the context will say, public, Boolean is valid and we're just going to use our strategy to say return strategy dot is valid and will pass in this. So the object that this is contained within now we can, for just ease of working with our object right click and say source, generate getters and centers. We won't do it for the cvv, the date and the number. We don't want to do it for the strategy and click. Okay, And now this object is complete. We have our number, date, cvv and our validation strategy passed in. And then we have the is valid method that we're going to be working with. So now our credit card is pretty well complete. A few other things we need to finish out on this and that is we We need a concrete instance of our validation strategy before we jump ahead on that, though, I want to add one thing in the side of here when checking credit cards. There is an algorithm called a loon algorithm and you can go look it up Doesn't matter as far as what we're doing with this example. But a loon algorithm is how they go through and verify that the card that we're passing in is in fact, a legitimate credit card. What it does is it goes through every other number, doubles um, and then mods it by 10 after its sum them all up. And if the response comes back a zero, it's a valid credit card number. This is used on all credit cards to my knowledge, and it's a simple way to eliminate auto generated numbers as easily. I put this in here because this shows the example of having a base class that has some functionality in there that all of the concrete strategies are going to utilize. So now if we go look at our strategy demo, we can see that we have everything but our concrete instance of our strategy in place. Let's go ahead and create that. Now we're gonna go ahead and right click and say New Class, and it is an a Mex strategy, and it's super class is validation strategy, and we'll quick finish. And inside of here, the only thing it cares about is tthe e is valid method, which is exactly what we want to do it with. And so we will say Julian is valid is equal to true, and I'm gonna go through and code this one out. You can go through on your own and do one's for MasterCard, Discover Visa, whatever you want to dio, but you'll get the idea really quickly of how this works, we will say is valid is equal to credit card dot get number, and if it starts with 37 then it is a valid American Express number. Or if credit card dot get number starts with 34. So those are the two prefix is that a valid American Express card can start with again for the example. It doesn't really matter. This is just to show you how we have different strategies in place for these types of credit cards. Now, the next thing that we're going to d'oh ISS look to see if its length is 15. All American Express cards have a length of 15. So if it is still valid, and I wrote it this way on purpose so that in the Comparator example we had multiple return types that actually increases your cyclo Matic complexity. I wrote this one so that does have the check and doesn't only has one return type out of the methods. So if is valid, is equal to, and that is credit card dot get number. The length is equal to 15 so all American Express cards have a length of 15 and you could do things with looking for spaces or dashes or any of that stuff and then finally, we're gonna utilize what's in our base class and say, if is valid and we're going to say is valid is equal to passes. Loon from our parent object will say credit card dot get number and then lastly, we're gonna return is valid. Now we have our entire American Express strategy built inside of here, and it's pretty well ready to go. So we'll go ahead and go back over to our strategy demo. And you see, I have a valid American Express number inside of here that I went and just Googled and found one. And if we go ahead and run this right click, it's a source run as Java application. We can now come in here and see that is American Express. Valid is equal to true. Now, if you don't trust things, run right the first time. We'll go ahead and give it a fake credit card number. So in come inside this application and we'll take the same one. But just change it by one digit. So 2832 and save that and this one will fell. So if you run this, you'll

see inside of our console. The 1st 1 passes true in the second pass, his fault now to create another strategy. It's really quite simple. In fact, we can go ahead and create a copy of our American Express strategy really quick. I'll just grab the contents of our is valid and right click and say, New Class, and it's a Visa strategy. It's Super Class is going to be validation strategy. We'll click, finish and sight of here. We're going to just return that same thing, but we're gonna look for a few different things. American Visa, rather than American Express, always starts with four, and they don't have two different digits. They go through and it's always 16 and then we're gonna make sure that it passes the loon. You can see, though, how if he had all of this content in sight of our main class like our credit card, we'd be looking to see if it's American Express than if it's American Express that if it has this many digits and then it passes this and that, we could have all this big contest ID context. Harry code in sight of our credit card class. We're with this. It's all contained inside the strategy, and I do have to f statements in here. But that's just so we don't have multiple return methods outside of our application. We could very easily just return out of those and not have to check each time to see if it's there. Now, if you go to our strategy demo, come through here. We have our visa strategy and grab a visa card and now paste this information in there. So we have our visa card with its new visa strategy. And if this is a valid credit card number, is gonna return is true for the is valid visa. So let's run this and see what it returned to us. And we have a valid visa. So we have an American Express that's valid. America's dress that isn't any valid visa, and all we had to do as far as our client was concerned, was swap out our strategies. But each one can have a completely different set of rules based off of what we're trying to do inside of here. As far as our credit card is concern, it just knows about the strategy that we pass in. So our credit card context stays the same. What changes is our actual validation strategy. Very powerful pattern. We eliminate having that nested, if else conditional code inside of our credit card and her client stays clean and simple. It does have to know about the different strategies we have in there, but as far as it's concerned, it's pretty simple for us to go through an implement this.

Pitfalls

[Autogenerated] Now that we've seen the strategy pattern in action, let's talk about some of the pitfalls of it. The client does have to be aware of the concrete strategies that are available. This is quite a bit different than the state where the client didn't have to choose what state it was in. Strategies have to be chosen one of the other drawbacks, or that there is an increased number of classes in the application. It really isn't a big problem because we usually don't have more than a few strategies in our application, but it does add a little overhead. The maintenance of our application is much easier, though, and definitely worth this small overhead. Speaking of the state pattern, let's contrast the strategy with state pattern to Seymour of the commonalities that they share

Contrast to Other Patterns

[Autogenerated] to contrast the strategy pattern. Let's compare it with the state. The strategy pattern is almost identical to the state pattern, but it's focuses different and that it's for algorithms or strategies instead of transitionary states. It is interface based, just like the state pattern with mud. Major difference, though, is that strategies don't know about their next

state. In fact, they are independent of one another, also similar to the state. There is a class per algorithm instead of a class per state. The state pattern, on the other hand, is interface base, just like the strategy and has a collection of concrete states that we can transition to. A state only knows aboutthe next state that it can transition to, and each state is contained in its own class. So they're very similar a class per state or a class per algorithm. They're both interface based. The few major difference is is that he state knows about the next day that is going to transition Thio and the strategy doesn't know about other alternative strategies. The major difference, though, in my opinion of the strategy, is that the client typically knows all of the strategies that are available to it where it doesn't necessarily know all of the states that it can be in. We can request it, but it doesn't assign that up front like it does with the strategy.

Summary

[Autogenerated] let's recap what we've learned with the strategy pattern. We want to utilize this when we want to externalize algorithms inside of our application. It should also be known that the client knows about the different strategies that we can implement inside of her application. It doesn't have to know about all of them, but it typically does. In the case of the strategy pattern, there is also a class per strategy. We typically utilized this pattern when we want to reduce conditional statements inside of our application based off what we're choosing to utilize to solve a problem. The next pattern we're going to look at is the template pattern, and we kind of got a glimpse of the template pattern when we looked at our solution of the Comparator and how that utilize strategies. We're going to see a great example of this with the template pattern

Template Method Pattern

Introduction

[Autogenerated] Hi, This is Brian Hansen and then this module, where you're going to look at the template method designed pattern. The template method pattern is used to define an algorithm that allows sub classes to redefine parts of the algorithm without changing its structure. Let's look at the concepts as to why you would choose this pattern.

Concepts

[Autogenerated] the concept surrounding why you would choose the template method pattern or that it is a fantastic technique for code reuse. The use of this technique is common in designing frameworks and libraries. The template method pattern has been around long before inversion of control frameworks, but IOC containers use this to allow plug double components inside their frame works all the time. IOC frameworks really helped keep this pattern mainstream. In my opinion, the real focus of this pattern is surrounding the algorithm of what we're trying to solve. The template method enables us to force an algorithm, but allows pieces to be configured by the user. Examples of this in the Java a p I r. The job you till collections sort method. If you watch the module on the strategy

pattern, you will notice that this is Theseus. Aim example. The difference here is that the strategy allowed us to swap out the entire comparator, where the template method has is focusing on the method to override with sorting. Another example is tthe e index of method in the abstract list class. Let's look at the design considerations when selecting this pattern. Now

Design Considerations

[Autogenerated] The design of the template method is based around an abstract base class. The base class is responsible for calling wth e child class and not the other way around hooks air put into place, which may be overridden. Operations, though, are also put in place and must be overridden. The pieces of the UML diagram are an abstract based class and a concrete class. Let's look at that UML diagram now.

UML Diagram

[Autogenerated] the UML diagram for the template method pattern is quite simple. Actually, the abstract based class contains the entry point or, in this diagram, the template method that is called by the client. It contains operations, and in this instance, Method one and method to that must be overridden by the concrete class. The concrete class overrides those operations and any optional hooks to fulfill the algorithm defined in the original template method Entry Point. Let's look at a live example of this code now.

Example: Collections.sort

[Autogenerated] the collection stop sort meant that provides us the operation to override. As part of our template, method collections and its base class provide the algorithm. From here we could implement a comparator, which is an example of the strategy pattern, or focus on the compare and compared to methods which are part of the actual template method pattern. Here is a snippet of the larger example that shows how the sorting is implemented. In this example, we're sorting list by their age, and specifically, the compare method is the portion of this that pertains to the temple method pattern. Let's look at this code in the editor now.

Demo: Comparable

[Autogenerated] if you've seen the strategy module. Ah, lot of this code is gonna look similar because I used a very similar example for demonstrating the Comparator, which is an implementation of the strategy pattern over comparable which we're going to use the temple method with. So this class goes through and create some person objects and stores them in array list, which we eventually want to call the collection Stop Sort on the collection Stop sort. If Ayan comment this right now, it's going to complain because the array list of people or the person objects are not a comparable instance that I can call sort on. So the sort is looking for a template method of the comparable interface to sort them on. We pull up our person object over here. We can see right now that it's just a plain object. Well, go ahead, say implements comparable, and it's of type person object. So when we say this, it's gonna give

us a compilation air asking us to add those unimplemented methods. Go ahead and add that to the bottom here, and you're gonna see that we have our compared to method implemented here. And this is the temple method. It wants us to implement this compare to method that's part of the temple method pattern of that interface. So we've implemented this interface, telling us that we have to provide this functionality to pass into that sort method. And we're gonna just really quickly throw a if this stott age is greater than oh dot age that we want to return one. And then we want to say if this stop age is less than oh dot age to return negative one. Now, if you've not used the comparable or compared to interface before, all this does is a sign it awaited value that it knows to sort this object before or after that object. So I really don't need to go into the details of how the sorting works. Just realized that the compared to is being implemented as part of a template method. This method is what's implementing our algorithm or are sorting functionality for that larger sorting method from that collections class. So now if we go back over to our template method, you can see inside of our demo our temperament, the demo collections out sort is not complaining anymore. When we print this out beforehand. It's gonna print it out in the order that we've entered these in 39 41 38 as faras the Ages concern. Then we're gonna call sort on it, and it will spit them out as 38 39 41. Let's run this and see if it does what we're expecting it to. D'oh! So say it run as job application. And when it's unsorted, you see Brian, Mark, Chris and when it is sorted, it's Chris, Brian, Mark. And that's what we were expecting, Chris, Brian and then Mark as faras. Their age is concerned, but you see how this is a really great little way of just implementing this one method that the algorithm for the sorting is contained within and Java did take advantage of doing that through the comparable interface on that object a little bit different in the strategy pattern, little different approach. It's all contained in just this method. Instead of having a strategy that we swap out at runtime

Exercise - Template Method

[Autogenerated] Now that we've seen an example of this pattern in action, let's go ahead and create our own. To do so, we're going to create an order template, a Web order and a store order class to demonstrate how the template method pattern works, a crossed implementing an algorithm, and this will finally just result in us creating our own template method.

Demo: Template Method

[Autogenerated] to get started. I've gone ahead and created a main method that's going to just demonstrate what we're doing with our order template and the individual order cases that we have. So you can see from this example we're going to create an order template and that it will have a web order associated with it. So let's right click and say, Create new class and this is gonna be an abstract class. So we'll say order, template, and go ahead and click finish. And this will give us the basic structure that wanna do. And I'm gonna throw a couple of methods in here, and I'm gonna go through and type all of these out as you kind of get the idea of what we're creating with our process order, our template method. So there's a few options we're gonna put in here that air hooks and operation's gonna put a public Boolean his gift, and this would be an example of a hook it could be ever overridden, doesn't necessarily have to be. And then the next one is going to be public abstract void do check out. And this is part of the algorithm that we want to create inside of our application. The next one we want

to put in a side of here is public abstract, void due payment. And then finally, public abstract, avoid do receipt. And I named these do just cause it signifies that there's an action associated with it. You don't have to necessarily name your methods this way. It's not really something that is particular to this pattern, but you can just get the idea of I've got an action associated with this, so I'm gonna have a do in front of the name. Finally, I want to do a method called Public Final Void Wrapped gift. And this is just what it sounds. We're creating an order entry system. We want to have the option for them to wrap their gift. I'm gonna put a system dot out dot print lin inside of here just to signify that when they we call this that the gift was wrapped and we're gonna close this off Now. We're down to the final parts of our application and let's throw one other piece inside of here. First, we're going to implement our actual template method that is public final and we want this final because we don't want it to be overridden by the sub classes. So this is part of that sequence that you have to get right. We always want our algorithm to be the same, and we'll say Process, order now This is where we get to define our algorithm. So first we want Thio force them to do the check out when this is called, and then, regardless of what they do, we want them to pay for it so we'll do a due payment. Now we get to see if it is a gift or not. So we say, If this is a gift, then let's wrap the gift. Otherwise, we want to include the receipt with this so we'll go ahead and say else do recede and then we could add one final piece inside of here. Let's add a do delivery mechanism as well, so we'll just as a final closing of our algorithm, will say do delivery and we'll need to define this algorithm up above. Still, this part of our algorithms will say public abstract void do delivery. So now we have all the pieces of our algorithm in place and I'll expand this, you can see it. We have our checkout payment receipt and we added a delivery on the end of there. And then we have the option to have it gift wrapped. So we have a Boolean flag that if it is a gift, then let's go ahead and wrap that as well. Now let's go ahead and create the Web order that's going to fill out our template demo here. So you see, it's had a template demo. We go through and grab, gather a Web order and process this order. So I want to come over and say, Knew Class and we'll call it Web order and it's super class is the order template we could finish. You see, it automatically goes through and defines these methods that we have to fill out as part of our template. So we define the site of our algorithm here that we wanted it to do. Do check out due payment wrapped gift if it was a gift and we made that one so it couldn't be overridden. It was the same for either option we had and then finally do our receipt and your delivery. So it's filled out the methods of our template method pattern for us now, rather than watch me tied these all out, I'm in a copy and paste. Just some demo code I have in here. But you can see how we have these hooks. Now that we've overwritten and some of these air operations that are required us to override as part of our template method, I'm gonna grab this code and just pasted over the top of these the same method signatures. I just went ahead and implemented all of them. And instead of our check out, we get the items from the car, we set the gift preferences, and I'm just sip, signifying what we would do in an usual checkout process that doesn't have anything to do with our actual method pattern. You see how the hooks are there and you can do what you want with it. Then we process the payment. We're gonna process it without the card being present because it's a Web order. And then we're gonna ship the item to an address and email that final receipt. Let's go ahead and look at our demo now and see if this will run for us. Right click run as Java application and you'll see we have our Web order and we get all the items from, Marquardt said. The gift preferences delivery address billing, address all of that information and then process that payment without the card present e mail them re seat and ship the item to that address. Now the beauty of this is we can add a whole nother order type, and we don't have to change our order. Template it all. The algorithm stays the same. I

have actually gone ahead and already done that. I've created a store order, and as you can imagine, it's quite similar to the Web order. Fact. Let's set these two side by side. Just we can see him here store order with its methods, do check out due payment, do delivery and do recede. And then there's the Web order next to it, with the exact same things we have stored on Web order with Do check out due payment, do delivery and do receipt. But it's different implementations for each one of those methods. The algorithm is gonna be called the same. We have our recipe. I like to think of it. This is an example of building out just example of cookies. It's the same gonna end up with the same result. We just have a different recipe for it, but we go through. The same exact steps are checkout, payment, delivery and recede. Now, if I go back over to our demo and a NCAA meant this, our order templates stays the same. And when we run it, we get all of our output for a Web order and Oliver output for a store order. Completely different set of business rules are different implementation for each order, but the same algorithm or same rules applies. It's just handled differently. Great pattern. A great way to reuse this and frameworks. You'll see it all the time with J D B C code and like spring JDC template or other tools like that, you're mapping in data layers. It's a very powerful way to guarantee things get done in a certain order of operations.

Pitfalls

[Autogenerated] what are some of the pitfalls of a template method. You need to make sure that you restrict access to certain methods correctly. Also, it is important to implement hooks and operations correctly as well. The template method can create a confusing class hierarchy, typically in an object oriented structure. We don't interact with an algorithm across multiple classes. Usually objects contain the entire a unit of work, where with the Temple of Method, it is purposefully divided across multiple classes. And overall, depending on how many templates you make, it can make for a difficult program flow. I do like this pattern, though, and the template method makes for a great use of implementing an algorithm. Let's compare it with different patterns to see more of its particular benefits and which situations you should use it in.

Contrast to Other Patterns

[Autogenerated] to contrast, the template meant that pattern. Let's compare it with this strategy. Both patterns are focused on the algorithm. The template method is focused on the same algorithm, but with different implementations. Though it is always class based, the template extends the base class and concrete implementations just implement pieces or hooks in the algorithm. The temple method is also chosen at compile time. The strategy, on the other hand, implements the entire algorithm per strategy. The strategy is also Maura about the contract or the interface. Another major difference is that this strategy has always used when you need to pick the algorithm at runtime he intended. This pattern is to allow the client to select while using the application where, with the template method, we've defined that beforehand and are aware of what we want to use at compile time.

Summary

[Autogenerated] Let's quickly recap what we learned about the template method. It guarantees algorithm adherents. It's commonly used in IOC containers and is often referred as thehunt Leawood. Principle. Don't call us, we'll call you. It's a easier implementation, then the strategy. As far as how we're going to interact with this algorithm, there's typically just a method that we hook into and just have to build out that simple piece of the algorithm. The next pattern that we're going to look at is the visitor pattern, and that is the last one of the behavioral patterns.

Visitor Pattern

Introduction

[Autogenerated] Hi, this is Brian Hansen, and in this module, we're going to look at the visitor design pattern. The visitor pattern is a great way to separate an algorithm from an object structure. Let's look at the concepts when choosing this pattern.

Concepts

[Autogenerated] the concept surrounding why you would choose the visitor pattern or that you want to separate the algorithm from an object structure. There are many reasons as to why you might want to do this, but the key reason is typically that you are expecting a lot of change and adding new features and can't modify the existing object structure that you've already created. This approach helps us maintain the open closed principle in our design. The visitor class contains the changes and specializations rather than changing the original object. Examples of this in the job a p I r a little bit obscure, but actually our great implementations of the pattern the Java laying model element element class and its corresponding visit earthy element Visitor are great examples of this pattern, but they're typically used for the internals of language itself and processing. How the language gets compiled, interpreted. Let's look at the design considerations of this pattern now

Design Considerations

[Autogenerated] the design of the visitor is interface base. There is a twist with the visitor pattern, though, and it is that we must design around the visitor from the beginning. It's hard to retrofit after the fact the application has elements, and these elements are the pieces that we expect to change and need to add functionality to overtime. The way we add to this changes by implementing a visitor in each of the elements. Each element has a visit method, and each visitor knows of every element. The pieces of the UML diagram are a visitor, a concrete visitor, an element and a concrete element. Let's look at that UML diagram now.

UML Diagram

[Autogenerated] This is the UML diagram for the visitor pattern. It is a little busier than most of the other patterns that we have looked at. The client creates elements which are all instances of the element interface, the elements air, all some form of a concrete element. And this is really where the pattern and the abstraction of the algorithm comes into view. Instead of changing the concrete element every time that we want to add functionality, we instead architect it to have an except method on a visitor. The visitor has a concrete element method for each type of element that we want to interact with. Finally, the types of visitors that we want to create are contained in concrete visitors. You can have multiple concrete visitors. In fact, it's encouraged to do so realized that this is where our changes will take place rather than in the elements themselves.

Example: API

[Autogenerated] the examples from the core Java A P I. R. Someone of ex cure and not representative of an everyday example for this pattern. Instead, we're going to look at a larger example of creating our own and walk through, creating two visitors.

Demo: Visitor

[Autogenerated] to demonstrate the visitor. We're going to first look at the same set. A code written without using thieve. Is it? Er, pattern. And then we're gonna turn around and create our own visitor using a visitor interface element interface, concrete visitor and a concrete element.

Demo: Without Visitor

[Autogenerated] to demonstrate the visitor of gun ahead and created an application without using the visitor pattern to demonstrate how you may write this code historically. And then we're going to adapt that code to use a visitor in the next example to start off. This is an actual example I have used on a client project a few years back, and we go ahead and create orders for a TV parts. So we have a parts order, and I've kept it quite simple. It just added a will offender and oil, and then we're going to eventually go through and calculate shipping on this and to keep everything the same, I've gone ahead and created an A T V part interface so that all of the parts are of this type. And you see, Fender just implements that oil implements that and our will implements, that is, Well, then we have our parts order. Our parts order has a few methods inside of here. It creates an array list apart. And then we have an ad part method that just add it to our list, and then we can get all of our parts, as you could see in the visitor demo. I had already gave a hand test. What we're gonna d'oh gonna calculate shipping now? This method currently doesn't exist. So let's go ahead and create that method inside of our parts order. And we probably want to go through and just it right through all over parts and go ahead and start calculating shipping on there. Now, we could go about this a poor man's way and just put an if then else in here of all of our parts. But since we're in a pattern, of course we're gonna try and apply some of things that we have learned here and just good old principles about how we want to navigate through this. But we're still going to

say for each and then we would want to probably go through each part and start calculating shipping. So it's a TV part dot calculate shipping. Now you're starting to recognize that this is where the visitor patterns coming into play. Currently, we don't have a calculates shipping method on the A. T. V part. We're gonna open up that interface and say, Well, sure, we want to add this public double calculate shipping method in here and save that and then you'll notice that we just broke the only three parts that we have. But I have to go through and add that code to all of these. So I'm gonna come through here right now. It's gonna return. Zero. Let's just say that for offender. It's pretty easy to ships were gonna return a three here and for oil. We're gonna do the same thing. We want to calculate shipping. It's hazardous. So it's gonna be $9 to ship that and then for Will's since their bulky, we're going to do the same here. And we're going to say that it costs them $12 to ship those. But you quickly realize I've had to go through and change every product that we've had. Now imagine if I had thousands products and thousands of different objects that we can go through and create these Now, of course, we could probably do some stuff in base classes, but that's not the point. This is just for shipping. What if we start adding other things in here? We'll go back to our method here, and we also broke this because this implements a TV part as well So we're gonna make this return a double now as well. So now we've got our double shipping cost and we're going to go through and start calculating our shipping costs across all these parts. So say shipping Cost plus equals a TV, and then we're going to return shipping cost. So you're starting to get the idea of any minor change I make. We have to go through and modify all of those objects that we have inside the system. We'll go back to our visitor demo now on, catch this double and now run it and we'll print it out just to see what it says. But we kind of have an idea of what it's going to do anyways. Well, say shipping cost. Save that. And now let's run. This is a job application and we see that it came back with $24. Okay, that's great. Let's see how to solve this same problem using the visitor pattern and as well as adapt that to other things that we may want to do with those objects and notice that we don't have to change any of our base objects that we've already created.

Demo: Shipping Visitor

[Autogenerated] So this code base is right where we left off before adding wth e calculate shipping method in the previous example. And I'm gonna show you how to use the visitor to do that instead of going through and adding those individual methods to each one of the elements that were working with. So in this example, a TV part is playing the role of the element interface. Where will fender and oil are taking the place of the concrete elements? Now, we haven't created our visitor interface or are concrete visitor. Yet besides that, this examples exactly the same. So if I look at our A TV part now and the A t. V. Part that we just created to do calculate shipping, you see that we added that calculate shipping method inside here. What? We're gonna do the same thing with accepting the visitor in this example? So let's go ahead and say public void, except we're gonna taken a TV part visitor in this Now we haven't created that yet, and we're gonna go ahead and save this. Now you'll see that it automatically broke my will fender oil and parts order objects. And that's because we don't have that method in here for except yet I have, in my demo notes an example this already copied, and that's just a safe time of us implementing this. So let's go through here and paste that in there, paste that in there and paste this in here. And all this is is just the unimplemented method, and you'll notice one thing that's a little weird with this is the method. Signature is exactly the same for all of these visitor dot visit this. That's one

complaint I see people have about the visitor pattern all the times that we add this generic method in there. It's the only way for us to get this hook in there with the way that this is built from the ground up. Now we can go ahead and create are a TV parts visitor, and this is a basic interface that the concrete interface it the concrete visitors are going to extend from sewing right click and say knew interface. And this is the A TV part visitor and we'll click finish. And now this is still gonna break because each one of these the will, the fender of the oil, the parts Order does not have a visit method in here yet we can go through and use the quick fix to create this. And it will add that to this method, automatically go through an ad, this method again being add fender, Say that been going do the same thing in oil. And this is just finishing out that hook inside of our parts visitor. So now we have all of these pieces in here, and I have gone ahead and commented out this one inside of the parts order. And I did that because I wanted to talk about this for a little bit as we went through it. This is a little bit different because this container builds the whole piece of our order. And so it references the other a TV parts or elements that we have as well as itself as part of this visitor. So let me go ahead and add the final method in sight of our parts visitor. So now we have our element interface, which was a TV part and our visitor interface, which is the A TV parts visitor. From here. We can go ahead and create our own concrete visitor because this is the interface. Now we want to go through and create a concrete implementation of this. So if we look at our demo right now, instead of saying ordered out, calculate shipping, I'm gonna say order dot except new a TV parts shipping visitor. If we go ahead and create this class, it will go through and say, Oh, we want to create this class with a TV parts visitor. Let's go ahead and click finish and it will go through and stub out all these methods for us. Now what happens as we cycle through this? Calculating the shipping is done in this class rather than in the individual elements that we had. So instead of putting that in fender oil parts order and will, it gets done in this class here. So to do this, we now could just go ahead and say, Well, let's do a double shipping amount and set that equal to zero. And then we come inside of each one of these elements and calculator shipping, and it's different for each one. We could direct them to one method if we wanted Thio, But will you say, Well, let's ah set the shipping amount equal to $15. And we could even do a little bit of reporting inside of here if we want. We say system dot out dot print Lynn, That Will's are bulky and expensive to ship and finish that off. And now we've got this done for the will. We could do the same thing for fender oil and our parts order. Now I'm gonna go through instead of you just watching me. Type these all out. I have these already done. I'm gonna copy them in. So the same code. Go ahead and grab that method for all of these. And copy that in to our shipping visitor and paste this in. Now, let me make this full screen just so you can see this for a second. I've got the will were shipping $15 the fender shipping $3 oils, $9 notice. I have the algorithm in the logic tied to this inside this class, it's not spread all over all the fender oil will and any other a TV parts that we may have. I ran into this in the real world example that I had where we started combining parts and there was no shipping because it was a combined part. So instead of charging for a will on a rim, it was just one shipping cost instead of two individual ones. You see how they're some business logic that can get tight into these, which is a good thing now. The one piece that I want to talk about down below here is this Visits, parts order. We can add some logic in here if there's a certain number of parts or rebates or things like that where we reduce the shipping amount and calculate that as well. So now our parts order is very simple. It looks just like our other example did with one slight little change instead of it being us calculating the shipping. We just have a visitor, and we visit this one method. So we have our visitor that goes through and we just visit ourselves, and now we can run this to see what it does inside of our visitor. Demo looks just the same. We've swapped out the except

this visitor with that calculate shipping. We can right click on this now run as job application and you'll see that we have a whole set of text to go with what we're doing, and it will calculate our entire shipping costs to see what we have. And none of that logic was added into those individual objects. So it's the difference of kind of dealing with a rich domain with utilizing this visitor to give us back the information that we're expecting.

Demo: Display Visitor

[Autogenerated] to really illustrate the power of the visitor. Wanted to go ahead and quickly create another visitor inside this code with you to show you that you don't need to add anything else but the visitor to it. So I'm gonna copy this line, and we're gonna create an A T V parts display visitor. And the display visitor is going to just go through and display what we already have inside of our order to create this To create this a TV parts display. It's gonna implement the A TV parts visitor. We'll click, finish, and I'm basically done. I'm gonna go through an ad, a system that out print Lynn and we're gonna say that we have a wheel and save that we can do the same thing. We can go through with fender and oil, and then we finally have our entire order. When you say that we have an order and that's all I had to dio I don't have any other code that I need to implement anywhere else. My order stays the same. My actual individual objects stay the same. Now I can go through my visitor demo and just add this one, Colin site of here. If I run this, it will go through and do the shipping visitor as well as the display visitor. And I didn't have to go through and change any other code inside my application. So you can really see the power of the visitor and how it allows us to add functionality to our application without changing other parts of the structure inside of our app.

Pitfalls

[Autogenerated] What are some of the pitfalls of a visitor? You have to plan on adaptability. This could be a little bit troublesome and lead toe over architect. In your application, you may build inflexibility that has never needed nor used. Thean direction of working with an object outside of its domain can be a little confusing as well. Each visitor may not implement all of the methods, so you may want to implement the adapter pattern. This isn't much of a pitfall, but it does add to the complexity of the visitor. We cover the adapter and an earlier module, and to solidify the usage of the visitor, let's contrast it with another pattern.

Contrast to Other Patterns

[Autogenerated] to contrast the visitor. Let's compare it with the generator. The visitor is interface based. Its focuses on adaptability through Externalizing changes, adding visitors is easy and encouraged. The interrogator, on the other hand, is also interface based but is typically implemented using an anonymous in her class. The IT aerator encapsulates navigation but doesn't necessarily externalize it. Although weaken implement multiple types of it aerator, we typically only have one inside of our application.

Summary

[Autogenerated] Let's do a brief recap of what we've learned with the visitor pattern. We want to use this pattern when we're expecting changes inside of our application, but we don't necessarily know what they might be up front. It does add a touch of minor complexity to the application, but nothing outside of standard object during innit principles. And we use this pattern when we want to externalize change. So if we can't change the base a p I once we've deployed it, the visitors a great way to go ahead and add functionality without changing what we already have inside of her application. It's also a great way for us to have tests inside of her application that we're not breaking the contract with by adding functionality down the road.

What Next?

What Next?

[Autogenerated] Well, if you're seeing this slide, you've made it through the design patterns behavioral course, and I want to thank you for staying with me through the whole thing. If you're wondering what toe look for next, if you haven't already taken the 1st 2 parts of this course the design patterns in Java creation all course and the design patterns and Java structural course, I would definitely recommend watching both of those videos to see the other counterparts of the three groups of patterns from the Gang of Four design patterns. Some of my other popular courses are Maven Fundamentals, Spring Fundamentals, Introduction to Spring, NBC and Introduction to Spring, NBC. Four. Those do build on top of one another, so I would recommend watching the spring NBC one first and bringing me see for one next and then spring with J. P and hibernate and finally, spring security fundamentals. And I'm also open and would like to hear feedback from you on other courses that you would like to see. You can reach out to me on my Twitter handle or my linked in profile that are at the beginning of each slide. Thanks again