Building More Python Design Patterns

by Gerald Britton

Start Course

Bookmark Add to Channel Download Course

Course Overview

Course Overview

Hi everyone, my name is Gerald Britton, I'm a senior solutions designer at TD Bank in Toronto, Canada, welcome to my course, Building More Python Design Patterns. This course builds on the design patterns with Python course by introducing you to eight more design patterns and showing you how to write them and use them in Python. This material will take an intermediate Python programmer to the next level and lay a foundation of removing beyond simple scripting to complex projects developing full-fledged, large-scale applications. Some of the major topics we will cover include a review of object-oriented design principles and the convenient acronym to remember them by, SOLID, eight more design patterns used in many programming languages today, common programming challenges made easier using these patterns, and typical business problems and how to solve them using the design patterns we look at. By the end of this course you'll have learned how to apply these new design patterns to break down tricky problems into simple components and create Python programs that are easy to write, easy to read and easy to maintain. Before beginning the course, you should be familiar with basic Python programming including how to write classes, functions and methods and how to create and use modules and packages. I hope you'll join me on this journey, to learn how to use classic design patterns in Python with the Building More Python Design Patterns course at Pluralsight. com

Introduction

Course Introduction and Overview

Hi, my name is Gerald Britton. I'm a Senior IT Solutions Designer and Pluralsight author. In my earlier Pluralsight course, Design Patterns with Python, I introduced the ideas underlying programming design patterns along with eight essential patterns that every object-oriented programmer needs regardless of the language used. If you haven't already, this is a great time to view that Pluralsight course. Since there are more than 50 recognized design patterns, eight is simply not enough. Consequently, in this course, I will introduce you to another set of eight frequently used patterns. In this introductory module, I'll give you a quick look at the patterns to be covered in this course and the problems they address. I'll also briefly review the main principles of object-oriented design, known by their acronym, SOLID. So that you can get hands on experience with the code and the patterns you'll be learning, I'll point you to the required tools you will need to complete the course, along with some suggestions for IDEs that can make your work and life easier. Since the target language is Python, you will learn the basics of meta class programming, which will be used throughout the course. For more detail on these topics, you can view any of the excellent courses available on Pluralsight that cover object-oriented design and programming. In fact, I just did a quick search in the course catalog and found close to 60 courses on the subject, including the foundational course in this series, Design Patterns with Python. Finally, I have to give credit to the authors of the book Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson, and Vlissides, also known as the Gang of Four or simply G of F. This is one hard copy book that you should have on your programming reference shelf. This course would not be possible without it.

Design Patterns Covered

I'll be showing you how to implement eight essential design patterns in Python. I'll start with the facade pattern, implementing a common interface to access a variety of APIs that have similar goals but differing representations. Programs using facade need not be aware of particulars of how to talk to the underlying APIs. For example, you might use facade to build a common API to access different databases from different manufacturers for persistent storage for your application. Next up will be the adapter pattern, which can be used to convert a new, related interface into one that clients expect. Using adapter, you won't have to change client code to add new functionality. Perhaps your program can use a customer class, but now it also needs to use a

vendor class with similar, but not definitely not the same, attributes. To add functionality without breaking the open-closed principle, you'll see how the decorator pattern can help. Suppose you have an application that builds car objects and now you want to add the ability to change the engine or wheel size. Decorator can help with that and favors composition over inheritance. The template pattern defines the basic outline of an algorithm and lets subclasses implement one or more steps while preserving the order of the steps. The iterator pattern encapsulates a collection of objects to make iteration easy and hide the collection's implementation. Iterators are a fundamental part of almost every Python program, so this is an important pattern to learn. The composite pattern makes it easy to compose objects in tree structures so that clients can handle individual objects and sub-trees through a common interface. A corporate hierarchy or a family tree is a natural example. The state pattern follows the mathematical concept of a state machine. Behavior changes according to the state of the object. For example, a web site that implements a shopping cart can change its behavior if the cart is empty or not or items have changed. The last pattern you will see is the proxy pattern. It can be used to control access to another object, perhaps for filtering or security reasons. Maybe you have a weather application and for one client, you only want to return sunny days. Adding these eight patterns to your toolkit will enable you to solve many perplexing programming problems while producing solutions that other programmers will find easier to read and understand. Remember, that other programmer could just be your future self who's forgotten ever writing that program in the first place.

Object Oriented Programming Design Principles

SOLID is an acronym to help us remember five basic principles we should try to follow. The first is the single responsibility principle. This means that a class should have only one basic thing to do or one type of object to handle. Adding too many features to a class can lead to breaking the second principle, the open-closed principle. This states that a class should be open for extension usually by inheritance but closed for modification. Imagine you started to use a class I designed only to find out later that I had changed it. You'd be frustrated, and I would've violated the open-closed principle. Next up is the Liskov substitution principle, named after Barbara Liskov. It tells us that subclasses should be able to stand in for their parents in a program without breaking anything. The interface segregation principle tells us that many specific interfaces are better than one do it all interface. The name reflects its origin in languages like Java that lack multiple inheritance. It is equally applicable in Python though, where we can exploit multiple inheritance to preserve interface segregation by using mix ins. The dependency inversion principle says that we

should program towards abstractions, not concrete implementations. In Python, abstract-based classes can be used for just this purpose, and you will see that throughout this course.

Tools You Will Need

You will need some basic tools to complete the exercises. Naturally, you'll need Python. There are a few flavors of the language implemented with different goals in mind. However, recall the dependency inversion principle? We should not program to a specific implementation. For this course, we recommend either the most recent version in the Python 2. x or the 3. x series. Both are easily downloadable from the Python website for all common platforms. Though not strictly needed, a good integrated development environment or IDE can make your life simpler. There are many available. Some are free, others are commercial. One simple IDE comes with Python, IDLE. Some popular third party IDEs are PyCharm, Wing IDE, PyDev, and Visual Studio, along with many others. Online, you can visit the Python Wiki at the link shown to get a good oversight. In this course, we will be using Visual Studio code with suitable Python plugins. You can find it at the link shown.

Abstract Base Classes

In Python, interfaces are defined using abstract base classes. Here's a brief overview of how that works. For details, consult the official documentation. First, the abstract base class module must be imported. Then, to define an abstract base class in Python 3. x, use the meta class property in the class definition, as shown. Here's an example of an abstract property definition and an abstract method definition. Note that there is no implementation. For Python in the 2. x series, abstract classes are defined a little differently as shown. Instead of specifying the meta class parameter in the class definition, we set the dunder meta class attribute. By the way, if this is the first time you've heard the word dunder, it is a contraction of the phrase double underscore. The names of special attributes and methods in Python are usually surrounded with dunders. The implementation of an abstract base class begins by defining a class that inherits from the ABC. Then, the property is implemented, as is the method. Finally, the class is instantiated and that lone property printed to demonstrate that it all works together. Try this on your own using the code provided in the IDE of your choice or in the Python REPL.

Summary

This module introduced the design patterns to be covered in this course. We reviewed the principles of object-oriented design known by their acronym, SOLID. We looked at some required and optional tools you will need to complete the course. Be sure to get them installed and running before you continue. We also reviewed how to build interfaces in Python, which are abstract base classes. Now, it's time to learn some new design patterns.

Facade Pattern

Introduction and Motivation

In this module, we'll take a look at the Facade Pattern. As the name implies, it puts a new face on something, an API in this case. It's really handy when you have a complex API or a collection of APIs, and you want to simplify them for use in your application. It's also useful when you have several APIs that do similar things, but in different ways and you want to have just one API to use in client code. For a motivating example, consider what you typically need to do to access a database, maybe to get some employee records. Well, you have to ask the DBA what database to use, and then import the right Python modules to support it, then you instantiate a control object to interact with that database, that means you build a connection string and connect to the database. Once connected, you will issue a query against it and get some results to process. When you're done, you disconnect from the database and release any resources used.

Demo 1: Retrieving Employees from a Database

For this demo, I've written a short script to retrieve and print employee records from a database. I'll be using a free SQL server sample database called AdventureWorks that has been created by Microsoft and is provided for us by Redgate and hosted in Microsoft Azure. See this link for more information and how to use it. Here, in visual studio code, let's walk through the script. I start by importing a module to interact with the database, pyodbc is a good one and uses the odbc protocol. I set up a constant for the connection stream, in a real world application, this would likely be read from a configuration file or passed in as a parameter. Next, I define a function to do the work and then down at the end of the script I call that function. The function connects to the database using the connection string provided. The query simply gets the first five employees and sorts them by last name and first name and, by the way, if you're new to SQL, there are quite a number of excellent SQL courses on Pluralsight. Next, I get a cursor object, which is then used

to execute the query. Once the query is complete, I iterate over the rows returned printing out the names retrieved. Finally, I commit the transaction and close the connection. Well, let's run this to see if it works. To do that, I start by clicking the bug icon, then the green start arrow, which will compile the program. Once compilation is complete, I will start the program up and results will appear in the debug window. And, as you can see, I have successfully retrieved and sorted the first five employees from that database. So what's wrong with this? Well, just look at all the code I had to write to get the results and most of it is what I call boiler-plate, required by the pyodbc module to set up and run a query. I wouldn't want to have every one of my developers writing this code each in their own unique way, I can just see the line-ups at my cubicle asking for help. Also, there are lots of other methods in pyodbc that this simple program just doesn't need, I'd rather hide them away. Now, imagine that one day the DBA comes and tells me that the database is changing, worse than that, suppose that I can't use pyodbc anymore, now I have to use a proprietary module from some third party vendor, well, not only I, but every developer using this code will then have to change their code and debug it all over again. The Facade Pattern can help ease the pain, let's see how it works.

Structure of the Facade Pattern

Facade is classified as a structural pattern, that means it provides a new way to put a program together. It is used to present a unified interface to a set of interfaces, those interfaces need not be related to each other in any way. Now that means that Facade is a higher-level interface, it makes the set of underlying interfaces easier to use thus reducing complexity. In the example we just looked at there are five interfaces, connect, cursor, execute, commit and close. Implementing the Facade Pattern will hide all those details behind one, easy to use API. Facade has a simple structure, it provides a way to talk to many interfaces through one higher-level interface. Those interfaces may be related, as in the example we just looked at, or they may be interfaces from completely unrelated subsystems. Facade brings them all together, so the client programs can use one simple interface instead of knowing about and interfacing with all the underlying subsystems. Facade knows which subsystem classes are responsible for a request and delegates client requests accordingly. The subsystem interfaces then handle the requests made by the Facade object, to achieve the result requested by the client program. The subsystems themselves have no knowledge of the Facade. In order to make the Facade a little more flexible, here is a structure I'll implement. First, I have an abstract facade base class to find, this allows for multiple concrete facades, perhaps one will use SQL server to access the employees, perhaps another will use a web service or a flat file or some other source. All will implement the abstract facade and its

one method, called simple API in the diagram, in our working example, that's the get employees method. Client programs will call a facade factory to get an instantiated facade to use, as suggested in the diagram, the factory will import the desired concrete facade, this will be configurable and I'll show you one way to do that. The factory then returns an instantiated facade to the client, then the client program will use the simple API method to do the desired work. Let's see what this looks like in code.

Demo 2: Applying the Facade Pattern

To start off this demo, take a look at the folder structure in the explorer view. Note that I've created a subdirectory, get_employees, that contains a Python package to hold the Facade members. Opening the package, you can see the required dunder init module, oh, and note that the pycache directory, is created by Python when the package is compiled. Init holds three constant definitions, provider, which defines which module will provide the concrete implementation of the facade, constr, which is a connection string used by most databases. Note that if the implementation were completely different, there may be no need for a constr constant or other constants may be required. The third one is query, the query we want to execute. Setting it up this way gives us flexibility, need a different SQL server database? Just change the constr constant. Or, change the query if the tables change. Need a different provider? Implement the abstract facade class in a new module and put that module in the package. Add any other constants you need. Now, let's take a look at the abstract facade class. AbsFacade declares just one method, the get employees method, each concrete facade that implements AbsFacade must implement this method. The SQL server module implements AbsFacade, the code is quite similar to that in the first demo with a few notable exceptions. For one thing, we now have to import the AbsFacade class we just looked at, also, the SQL server implementation requires the const string and query constants, which are also imported here. In the get employees method, we now use that constr to find in the packages init module, similarly, we use the query to find there. Separating these to the init module, means that we don't have to change this code if either the const string or the query changes. Now, to truly separate concerns, I should move the loop that prints the results out of this method and into the main program. If you'd like to do that, you'll probably need to modify AbsFacade, the concrete facade and the main program. The facade factory does the job of instantiating a facade object. This code borrows from the discussion of the factory pattern in the Design Patterns with Python course, it uses the functions from the import, lib and inspect modules to enable dynamic run time importing of the required concrete facade implementation. Basically, it imports the module from the current package by name, then

looks through its contents to find the first class that is not abstract but is a subclass of AbsFacade. For more details, see the official documentation of import, lib and inspect. The sharp-eyed will have already noticed that I've included no error checking in this code, I left it out intentionally to make the important bits easier to see but in a production system, this would not be acceptable. Okay, now we're ready to look at the main module where the client code resides. It is quite simple, it imports the constant provider and the facade factory, then, in the main function, it creates a facade for the provider using the factory, then executes the get employees method. Does it work? Let's see. Click the bug arrow and the start icon, now our program is ready to run so let's run it. Opening the debug window we can see we have the same result as before, our Facade implementation is good.

Summary and Considerations

In summary then, Facade has a number of compelling advantages, it shields clients from the details of the subsystems the Facade uses, giving client programs a simple API to use instead. This reduces the number of objects clients need to interface with, which promotes weak coupling, a desirable goal. Facade also lets you vary the subsystem, or even add new ones, without changing any client code, also, nothing is lost. Clients can still use the subsystems directly if the need arises. Consider using Facade in your next project or perhaps the very one you're working on now.

Adapter Pattern

Introducing the Adapter Pattern

In this module, we'll take a look at the adapter pattern. This pattern is used to make a one API look like another one that has different methods and signatures. The purpose is to present client code with an abstract API to program against rather than a number or different APIs that have similar goals. One of the guiding principles in object oriented programming is that we should favor coding to abstractions instead of concrete objects. This is the D in solid, the dependency in version principle. Adapter can also help us achieve the O in solid, the opened closed principle. In every day life, we use adapters all over the place. Many modern tech gadgets have AC adapters, those little black boxes that you plug into a wall socket with a cable to plug into your device. Then, plumbers use adapters to change between different sorts or sizes of pipes. And have you

ever traveled between North America and Europe? Then no doubt you've used a power adapter so that you can use your laptop or mobile phone while on the road. Now let's take a look at a programming problem that can be solved using an adapter. For a motivating example, suppose you have a program that prints customer names and addresses from a customer object. The users like your program but come to you with a request, make it work with vendor objects. After all, both customers and vendors have names and addresses. Unfortunately, the vendor API is different. For example, customer objects have a combined address property containing the street number and name, but vendor objects separate these into number and street properties. So, what are your options? Make a new version of your program? Copy your program to a new one that handles vendor objects? Well, that violates the don't repeat yourself principle. Add some conditional logic to handle both objects in one program? Well, let's try that and see where that gets us.

Demo 1: Modifying Client Code

For the first demo, we'll start with the original program which prints customer names and addresses. Then we'll modify it to support vendors as well. Here in visual studio code, you're looking at the customer class. It's really pretty simple. The constructor takes a name and address and saves them in private variables. Then I have name and address properties to return those variables to the client. Now to provide some test data, I've got a mock customers module, which looks like this. It imports the customer class we just looked at and uses it to build a simple tuple of three customer objects. In the main program, I import the mock customers. And then in the main function, I iterate over them and print their names and addresses. At the bottom of the module, standard start up code gets things running. Let's see if it works. I'll hit F5 to start the debugger, and hit F5 again to run the program. And in the debug console, you can see that the program works just fine. Now, I need to add support for vendors. So, let's look at the vendor class. You can see that it is pretty similar to the customer class, except now the address is separated into number and street. I've also got a mock vendors module with three vendors in it. The next thing to do is to modify the main program to handle this change. For starters, I import the mock vendors. Next I simulate a run time parameter to indicate whether the program should process customers or vendors. Note that in a production system, this would probably be passed in as a parameter to the main program. Then, in the body of the main function, I need some conditional logic. One section for customers, and a second section for vendors. And I need to check for an invalid type. Okay, time to test again. So, F5 to start the debugger, and F5 to run the program. Open the debug console, and we can see it still works fine. But let's think about it. The main

program now knows about two classes and how to use them. What if I get a new request to add government customers, or perhaps, foreign vendors? Well, I'd have to break the program apart and add new logic. This is starting to look messy. Actually, whenever you start to code along an If LFL statement, ask yourself if there isn't a better way. Other patterns, including the strategy and command patterns, can help with this sort of thing, though their intents differ from the adapter pattern. Now, you may be thinking why not just change the vendor class? Well maybe I can't, maybe it comes from a third party package that is delivered as binary only. The adapter pattern can be used to sort out this type of problem.

Adapter Pattern Details and Object Adapter Structure

Adapter is classified as a structural pattern. That means it provides a new way to put a program together. It is used to convert the interface of a class into another one that clients expect. Adapter lets classes work together, even though their interfaces are incompatible. Often, the adapter provides functionality that the adapted class doesn't have. Interestingly, there are two kinds of adapters, object adapters, which use composition and can easily work with subclasses as well as parent classes, and class adapters, which use inheritance, and let the adapter override some of the classes methods. Each type has its pros and cons, but generally, you should favor composition over inheritance whenever you can. Composition leads to a flatter class structure, which can be easier to understand, maintain, and debug. The adapter pattern is also known as the wrapper pattern. The structure of the adapter pattern looks like this. First, an abstract adapter is defined. This satisfied the dependency inversion principle, which says that you should program towards abstractions rather than implementations. The abstract adapter will have one or more concrete implementations, all of which are hidden from a client program. The concrete adapter uses composition to access an object of the adaptee class. As shown in the diagram, the adapted operation, which often has a signature that is unknown to the client program, is used by the concrete adapter to present an operation that is known to the client program. Finally, the client program obtains an instance of the object implementing the abstract adapter, perhaps using a factory method, and uses its operation method to do its work. What happens behind the scenes in a concrete adapter and the adapted class is unknown to the client.

Demo 2: Using the Object Adapter

For the second demo, I'll implement the object adapter pattern for our working example. First, here's the abstract adapter base class. It implements a standard Python ABC. Note that a

constructor, the (mumbles) init module is not abstract. This is permitted with Python abstract base classes and handy here. The constructor expects an adaptee, which will be used by the concrete adapter classes, thus composing the adapted class with the adapter. Since I saved the adaptee in a private variable, I also have a property defined to retrieve it. Two abstract properties are defined. These echo the same properties as in the customer class in the first demo. I want to adapt the vendor class to make it look like the customer class to the client. The concrete object adapter for the vendor class looks like this. It imports the abstract adapter we just looked at, and implements the two required properties. The name property is returned as is from the adapted class, using the adaptee property from the abstract base class. The address property is more interesting, it converts the two vendor class attributes, number and street, to the address property the customer expects. In the mock vendors module, I now wrap the vendor instances with the vendor adapter. This is where the composition takes place at run time. The main program then imports the marked vendors and processes them as if they were customers. Let's run it to check this version. F5 to run the program, and in the debug console, you can see the same results. So, this works as expected.

Class Adapter Structure

The class adapter has a different structure, and uses multiple inheritance instead of composition. In the diagram, there is a client program that uses some class called original class here, with some operation. The adapter subclasses this class, exposing the same operation. Here though, it also subclasses the adapted class so that it can use its adapted operation to prepare the results the client is expecting. The client program then instantiates the subclass adapter instead of the original class, and through it, transparently interacts with the adapted class. Let's see how that works out in code.

Demo 3: Using the Class Adapter

For the third and last demo, let's look at how to implement a class adapter in Python. The customer and vendor classes are the same as before, but we've changed the vendor adapter, it now looks like this. Now the adapter imports both customer and vendor classes. The new vendor adapter class uses multiple inheritance to access both sets of attributes and methods. The constructor must pass the arguments received to the super class. According to the rules of the Python's resolution order, this will be the vendor constructor, so a vendor object is instantiated. The adapter then overrides the address property that is found in the customer class, and the

override constructs an address string using the number and street properties of the vendor class, and returns the result to the caller. The mock vendors module now uses the vendor adapter class instead of the vendor class to build the test samples. The main program is the same as for the object adapter example, so let's run it. Start the debugger and run the program. The results are the same as before, so the class version of the vendor adapter works too.

Comparison and Summary

So, which is better, the object adapter or the class adapter? The object adapter favors composition over inheritance, which can be more flexible and leads to a flatter class hierarchy. On the other hand, the class adapter uses subclassing, which allows it to adapt to a very specific class or subclass. The object adapter delegates calls to the adaptee, which gives it great flexibility. But the class adapter just overrides adaptee methods, which keeps it simple. The object adapter works with all adaptee subclasses, provided they adhere to the L in solid, and can stand in for their parent. However, if some subclass adds a new behavior, the adapter would also need to change to support it, breaking the opened closed principle, the O in solid. The class adapter is committed to one adaptee subclass, but if new behavior is added to the subclass, the adapter needs no work to support it, thereby obeying the opened closed principle. To sum it all up, use the adapter pattern when you need to use some class but the interface does not match what you need. You can use it to create reusable code that works with new unrelated or unforeseen interfaces. The object adapter works with several subclasses, and is useful when it is impractical to subclass each one. The class adapter works with a specific subclass and can be simpler to implement. So, which one should you use? I can only give the standard, noncommittal answer, it depends. It depends upon the application, existing and new libraries, and business requirements. However, all things being equal, consider using the object adapter as it can be the more flexible approach.

Decorator Pattern

Introduction to the Decorator Pattern

In this module we'll take a look at the Decorator Pattern. This pattern can help reduce a profusion of subclasses by adding additional responsibilities at run time instead of compile time. Imagine that for every house class you had a subclass to indicate the color of the paint in the kitchen, or

for every custom computer class, you had to subclass it for every possible combination of components, you'd get an explosion of subclasses. The Decorator Pattern can help us with that. At the same time, it helps us obey the open/closed principle, the O in SOLID. If you need a refresher on these principles, you can review the introductory module in this course. For a motivating example, imagine you run a car dealership, you sell various models of cars, perhaps economy, luxury, and sport, with many kinds of options but let's pretend there are only three, the engine size, which could be an inline-4 or a V6, the paint color, which could be white, red or black and the upholstery fabric you like, leather or vinyl. The cost of the vehicle of course depends on these options. Let's try a class-based approach.

Demo 1: Using Subclasses to Encapsulate Options

For the first demo we'll start with a simple abstract based class for cars, then we'll define a concrete class for each model. To handle the various options, we'll subclass the models to handle all those combinations. In visual studio code, you can see that I have created a cars package, this package contains an abs_car class, an abstract based class that will be used for all models. The abs_car class defines two required properties, description and cost. The first concrete model is the economy model, which implements the abs_car ABC. It defines the description and cost properties as required, however a customer will always order a car with some combination of options so I've created the first two of them. Here is the model with a four cylinder engine, white paint and vinyl upholstery, I've overridden the description and cost to match these options. Here's the same model, with a six cylinder engine, this model costs more. Now I have a little program to test these classes, it imports the models we've just looked at and prints out the description and cost so let's run this program. If I expand the debug console, we can see that the classes all work as expected but let's stop a moment and think about this approach. The first demo used subclasses to account for the various models with their options, one subclass per combination but in that example, I only created two subclasses for one car model. Doing the math on a complete solution, we would have three models, two engines, three paint colors and two upholstery types for, well, that's 36 subclasses. If you've ever shopped for a new car, you know that there are more than 36 combinations, your local dealer can probably offer thousands of them. Taking the approach in the demo will give us a subclass explosion and a maintenance nightmare. For example, imagine that you now need to add three interior color schemes, that would be our original 36 subclasses times three, which is 108 subclasses, if my math is correct. Let's try again.

Demo 2: Using Properties Instead of Subclasses

I'll begin the second attempt the same way as in the first demo, I'll start with an abstract car class then I'll define a concrete class for each model of car. This time though, I'll use properties for the options. Here's what the solution looks like now. The abstract based class has grown, we've added properties for the engine, paint color and type of upholstery. Also, the cost method is no longer abstract. Now it computes the cost of the options, this will be used in the concrete classes, which know the base cost, to compute the total cost. As the abstract based class has grown so has the economy concrete class, I need to enhance the constructor to add the new properties and I have to implement those new properties themselves. Then I have to implement the cost method differently. I call the cost method in the super class, the abs_car class we just looked at, then add the base class for the model. In the main program, I only have the one economy model to import, then, when I instantiate the cars, I pass the options into the constructor, let's run this one. Running this you can see that it produces correct results, but are we better off than before? Let's review. In the second demo I used properties instead of subclassing to get the desired results, on the positive side, now there is only one concrete class per car model, a definite improvement, however, this comes at the cost of more properties to implement per concrete class, the engine, the paint, and upholstery properties and a more complicated constructor to accommodate the new properties and what about maintenance? What if the options' prices change? Well, we'd have to open up the abstract based class and change the cost method. What if we want to add an interior color property? Then we'd have to open up the abstract based class and the concrete classes to accommodate the changes. There are a number of object-oriented programming principles I've violated in the first two demos, the single responsibility principle was violated in the second example, what business does the abstract car based class have calculating the aggregate option cost? None at all. Both examples violated the open/closed principle, since adding or changing options would potentially mean opening up both the abstract based class and the concrete classes. The interface segregation principle is violated in the second example as well, the cost method would be better off with its own abstraction. The dependency inversion principle is violated also since concrete classes now depend on the implementation of the cost method in the abstract based class. Finally, the don't repeat yourself principle is violated practically everywhere, we need a better approach. We need the Decorator Pattern.

Description and Structure of the Decorator Pattern

The Decorator Pattern is classified as a structural pattern, since it offers a new way to put a program together. This pattern adds new abilities to an object, dynamically, at run time. In the

first two demos, I tried to add new abilities by subclassing and then adding properties but both attempts led to object-oriented programming principle violations. Decorators, on the other hand, provide a flexible alternative to these approaches. The Decorator Pattern is also known as the Wrapper Pattern, you may recall that the Adaptor Pattern is also sometimes called the Wrapper Pattern, both Decorator and Adaptor wrap their underlying objects through composition but where Adaptor is used to help incompatible APIs work together, Decorator is used to add additional responsibilities. Let's look at the structure of this pattern. The Decorator Pattern is structured like this, a component is the object we want to decorate, so we start with an abstract component. One or more concrete components will implement the abstract component, then an abstract decorator is defined, which also inherits from the abstract component. Note it is composed with a reference to the component it decorates, and has access to the components' operation. One or more concrete decorators implement the abstract decorator, and each concrete decorator maintains a reference to the component it decorates so that it can use that components' operation. Also, each concrete decorator may, optionally, add some new state, or new operations, as well as its own implementation of the abstract operation. Note, that neither new state or new operations are required, though in practice, you'll likely see one or the other or both.

Demo 3: The Decorator Pattern in Python

In this final demo, you'll see how to build the Decorator Pattern in Python. We'll follow the Decorator Pattern structure we just looked at and we'll be using decoration instead of subclassing, which will lead to a cleaner solution. In visual studio code, the abstract car based class is the same as in the first example, nice and clean. Concrete classes only have to implement the description and cost properties. The economy car concrete class is also simple, it implements the description and cost properties as required. The luxury and sport classes are similar, differing only in name, description and cost. Now I've put the decorators in their own package to keep things neat. The abstract decorator inherits from the abstract car. If you're wondering about the squiggly, green underline, this is coming from pylint, which warns me that the two abstract properties have not been overridden but we'll do that in the concrete decorators. The abstract decorator also implements the constructor, which will be the same for all concrete derived classes and is used here to save a reference to the concrete car we'll be decorating. I've also added a new property called car, which will be used to return that reference, this is where the composition takes place at run time. The V6 decorator implements the abstract decorator, the description property starts with the description found in the car to be decorated and adds the string V6 on

the end, likewise, the cost property starts with the cost of the car and adds $1, 200 to it. The other concrete decorators look similar. You can see in the package, that I've implemented all of the decorators for engines, upholstery and paint colors and you can find them all in the exercise files. The main program now uses some of the decorators to show how it works. First, I import the base car, economy in this case and the three decorators I want to use. In the main function, I instantiate an economy car, then decorate it successively with an engine, some upholstery and a paint color, showing the results at each step. Well, let's run this. The output shows the car at each stage. As it gets decorated, the description expands and the price grows.

Consequences, Python Decorators, and Summary

So what are some of the consequences of the Decorator Pattern? Well, compared to static inheritance, Decorator is much more flexible and can respond at run time rather than being compiled in. Compared to adding lots of properties, Decorator keeps things simple, each decorator need only be concerned with its specific attributes and the object it is decorating. There's no practical limit, other than storage to the number of decorations you can apply to a given component and decorated objects are transparent to clients. But there are some downsides, a decorated object has a different type, since it lives in its own subtree in the class hierarchy. This could be problematic if client code is sensitive to specific types, however, the principle of coding to abstractions rather than implementations, is a great way around this, that's the D in SOLID. A finished system may have many little objects, one for each decorator, which can increase the learning curve for new developers. On the other hand, the Factory and Builder Patterns, covered in the Design Patterns With Python course on Pluralsight, can really help out here. You may be wondering how the Decorator Pattern differs from Python Decorators? First, they differ syntactically, the Decorator Pattern is implemented with concrete classes that derive from an abstract base class, Python Decorators are implemented with function definitions or classes with cullable instances and the @ sign syntax. The Decorator Patterns wraps objects, which are class instances whereas Python Decorators wrap function, method, and class definitions, not instances. The Decorator Pattern operates on class instances at run time, Python Decorators, using the @ sign syntax, are expanded into function calls at compile time. The actual function call happens at run time but can only wrap function, method, and class definitions. The Decorator Pattern adds functionality to class instances, Python Decorators add functionality to functions, methods, and classes. The Decorator Pattern has a very specific purpose and narrow focus, Python Decorators are much more general purpose and can be used for many things. The Decorator Pattern you're learning about is a Python implementation of a classic gang of four

pattern, see the introductory module of this course for more information. Python Decorators were introduced in PEP 318, which you can find at the link shown. The PEP author notes that there have been a number of complaints about the choice of the name Decorator for this feature, the major one is that the name is not consistent with its use in the gang of four. The author states that the name Decorator, as implemented in the Python language, probably owes more to its use in the compiler area. So, when should you consider using the Decorator Pattern? Well, use the Decorator Pattern when you want to add new functionality to existing objects. It's a better approach than adding many subclasses with little variations, it's also better than adding many properties to a higher level class but because you may wind up with many decorators, consider using the Factory and/or Builder Patterns to return decorated objects and be sure to watch the other modules in this course to build your personal design pattern library and know how to implement them in Python.

Template Pattern

Introduction and Motivation

Hello, welcome back to the course Building More Design Patterns with Python. My name is Gerald Britton. In this module, we'll take a look at the template method pattern. Sometimes you find yourself creating or working on classes that comprise some algorithm or recipe. For example, you might have one for baking a cake and another for making bread. Most of the steps are similar, though their names and actions might differ somewhat. The template method pattern provides a way to encapsulate the algorithm so that the basic high level process is consistent, yet allows for different implementations of some steps at a lower detail level. In this way, the pattern encourages code reuse while ensuring that all required steps are implemented. For a motivating example, imagine you are a passenger bound for New York or perhaps Amsterdam. If you live close to NYC, you might just take a bus to New York City. If you live in Canada though, you'd probably fly to New York. If you live anywhere in North America, you'd probably fly to Amsterdam, though the adventurous might go by boat. Now both buses and planes serve a common purpose, moving you from point A to point B. How they get you to your destination is of course quite different. Let's look at one way to model them in Python.

Demo 1: Modes of Travel

For this demo, I'm going to model the ride to New York or Amsterdam using distinct classes, one for a bus trip, and another for a plane trip. Then, we'll test them out. In Visual Studio Code I have the two classes in two separate files. The bus class has a bus trip method to encapsulate the trip, and uses four other methods, one to start the diesel engine, then leave the terminal, drive to the destination, and arrive at the destination where the passengers disembark. The airplane class looks similar, but it has different engines, a different method to leave the terminal, and it flies to the destination, and lands there. The main program tests these classes. I've created functions for taking the bus and taking the plane respectively. Note that this would work if I chose to take the bus to Amsterdam, though I suppose that would require a very special bus. Note that the two operative functions are quite similar, as are the classes they use. Mainly it's the names and contents of the steps that differ. The overall structure is pretty much the same. So let's run it. If I expand the debug console, you can see that the classes work as expected. But is this the best way to do this? There's one glaring principle of object oriented programming that is violated here. Can you guess which one? It's the don't repeat yourself principle. The bus and airplane classes are almost the same, but not quite. The template method pattern will get them in line.

Template Method Pattern Description and Structure

The template method pattern is a behavioral pattern. So it changes how classes interface with each other. It defines the skeleton of an algorithm, deferring some of the steps to sub classes. The overall structure of the algorithm does not change, however. It starts with an abstract base class with three types of methods. Abstract methods, and these must be implemented by any sub classes. Concrete methods, these methods are common enough that many sub classes will want to use a default implementation, although they can be overridden in derived classes if necessary, and hooks. These methods do nothing, but may be overridden by sub classes if they need to do something specific at some point in the algorithm being implemented. The order of the methods is fixed, like a recipe. You can't put the pan in the oven until you've mixed the flour for the cake or the bread. This fixed processing order is encapsulated in a separate method called the template method, from which the pattern gets its name. The first demo had this method too, called bus trip in the bus class, and plane trip in the airplane class. If you can't quite remember what they look like, go ahead and replay that demo, or download the exercise files and examine them at your leisure. Now, let's look at the structure of the template method pattern. The template method pattern is structured like this. To start off, there is an abstract base class. That class contains the template method, which determines the order of execution of the primitive operations. The primitive operations in the abstract base class may be abstract, concrete, or empty. Empty

operations are called hooks in the template method pattern. Concrete classes derive from the abstract base class. These must implement the abstract methods. They may override the concrete or hook methods if they want to, but that is not required. Note that the template method in the abstract base class calls the methods defined there in a predetermined order, and that the concrete classes cannot override that order, although they can hook in if desired. Now, let's see how that looks with our bus and plane trips.

Demo 2: Travel Using the Template Method Pattern

In this demo, you'll see how to use the template method pattern using the structure we just looked at, to simplify the bus and airplane travel example. Here in Visual Studio Code you can see the new AbsTransport class, an abstract base class that follows the template structure. Let's look at it closely. After the constructor, which just takes the saves to destination, there is a take trip method. This is the template method, and you can see that it contains five calls to various other methods. The start engine method is abstract, so it must be implemented at any sub class. The leave terminal method is concrete but overrideable. You'll see an example of just such an override. The travel to destination method is also abstract. It must be implemented by any derived class. The entertainment method is concrete, but has no implementation. This then is a hook method. The arrive at destination method is also concrete, and can also be overridden if necessary. The airplane class now implements the abstract base class. It implements the start engine and travel to destination methods as required. It overrides leave terminal, adding its own particulars. It also overrides the arrive at destination method since an airplane must make a landing. Finally, it overrides the entertainment method, that hook method, since flights can be long, and passengers can get bored and need some kind of distraction. The bus class on the other hand is much simpler. It only implements the required methods. The concrete methods in the base class are just fine, and bus riders need no entertainment, or at least that's what this bus company thinks. The main program now has a travel function that can take any destination and mode of transport. It just instantiates the transport class and takes the trip. So, let's run it. Looking at the output, you can see that both transport methods work, even though they produce different output. Bus and airplane both implemented AbsTransport but did it differently, and yet they both follow the same basic steps in the template method.

Consequences and Summary

Thinking about the consequences of the template method pattern you can see that it's a great platform for code reuse. Some standard methods can be fully implemented in the abstract base class, while ensuring that all required steps are indeed implemented, yet allowing some steps to be overridden. Using hooks enables some concrete classes to inject special requirements at predefined points of the template method. Also, the overall order or structure of the algorithm is enforced in the template method. Note however that template is not useful if the algorithm itself must vary. The template method pattern is comparable in some respects to the builder pattern, where builder's director corresponds to template's template method. However, builder is all about constructing objects, while template is action oriented. So when should you use the template method pattern? Well, use it when you recognize similarities in two or more classes that implement equivalent algorithms. Take advantage of the three types of methods in the pattern, abstract, which must be implemented by all sub classes, concrete, which are implemented but overridable, and hooks, where additional functionality can be injected. Template will help you follow the key design principle don't repeat yourself.

Iterator Pattern

Introduction and Motivation

Hello again, and welcome back to this course on Python design patterns. My name is Gerald Britton. In this module, you will learn about the Iterator Pattern. Pretty much all software handles collections of one sort or another. Whether a list of ingredients for a menu, a table of employees in a company, or a database for order processing, collections are everywhere. And whatever the collection, at some point someone will want to retrieve items from it one at a time. That operation is called iteration. Iterating over a collection means returning objects to the caller one by one. It's hard to imagine any meaningful software that does not use iteration at some point. For many systems, iteration is central to the operation. Now, for better or worse, programmers can be quite creative when it comes to building objects. That creativity can extend to the methods exposed for iterating over collections. Naturally, good objected-oriented programs hide their implementations and only expose the absolute minimum necessary. But that could lead to inconsistencies between collections. However, if collections conform to some pattern, you no longer have to guess how to iterate over them. That's what the iterator pattern is all about. For a motivating example, consider a collection of employees. The collection holds zero or more employee objects. And client

programs need to iterate over that collection, perhaps to give all employees generous year-end bonuses. In turn, the collection needs to expose a method for performing that iteration, while hiding the implementation of the collection. Now, there are certainly many ways to do that. Without some pattern to follow, creative programmers will devise all sorts of novel approaches, which means that every collection may do things in a unique way. There's no conformity. The solutions are not conformant to any recognized pattern. Check out the following demo to see the kinds of problems this can get you into.

Demo 1: Building Iterators for Employees and Departments

For the first demo, we're going to start by looking a collection of employees. That could be a list, a set, a dictionary, a tree, or anything at all really, though that is hidden from the client. Note that hiding implementation like this is a good principle of object-oriented programming. You'll see one possibility for a method to iterate over that collection. In Visual Studio code, I've got four modules to begin with. The employee module defines the employee class, which is pretty basic. After all, we're here to talk about iterators, not employees. I've also created an employee collection module, which defines the employees class, a collection used to hold employee objects. Under the covers, the collection is maintained as a Python dictionary, where the key is a number representing the order in which an employee was hired, starting at one. The value of the dictionary item is an employee object. The collection defines two methods, an add_employee method and a get_employee method. Add_employee takes a new employee object and adds it to the collection. Apparently we hire for life and even beyond, since there's no delete_employee method. Get_employee returns the employee by looking up the number in the dictionary. There's also a property, headcount, to return the current maximum employee number. To test this, I've got a little main program, which just prints out the information from the employee collection. It starts off by importing some test data, so let's look at that. Test data just builds an employee collection using a set of test employees. Some of these names you might even recognize. There are also, down at the bottom, some test departments, which I'll use shortly. In the main program, I have a for-loop to retrieve and print the items in the collection. I'll run it to show that it works as is. I hit F5 to start her up, hit F5 again to run it, then open the debug output window. And you can see all those employees that we talked about listed successfully. So I guess it works just fine. But is this the best way to do this? Since there's no other method exposed, I'm using the get_employee method to iterate over the collection. Not exactly Pythonic, is it? Now, I also have a second version of the main program. This version uses the departments collection in the test data I mentioned a minute ago. Let's look at the department class and its collection. The department

class is similar to the employees class, but the attribute names are a little different. The departments collection now uses a Python list to manage the collection. Also, it returns a tuple to use in a call to the range function. There's also a corresponding get_department method, which looks a lot like the get_employee method we had before. In the new main program, notice that I use the departments_range method as input to the range function, using star syntax to unpack the tuple returned. Now, I've started to write a generic print_summary function that is intended to print either employees or departments. But I have a problem or two. Since the two collections have different methods and properties, I cannot write the for-loop without knowing the type of collection, which would mean a runtime test for type. Then, I have different methods for retrieving employees and departments, even though they function in a similar way. In fact, it's so messy, I just gave up. You know, I really want to write this polymorphically. There must be a better way. That would be the iterator pattern. Let's see what it looks like.

The Iterator Pattern Exposed

The iterator pattern is classified as a behavioral pattern. In this case, it adds new capabilities to a collection or aggregate that enables iterating over the elements of the collection without exposing the underlying representation. This preserves encapsulation and promotes information-hiding, two important and related principles. Using iterator properly can solve the problems that we encountered in the first demo. Oh, and the iterator pattern is also known as the cursor pattern, since the way it operates is similar to cursors in other contexts, database processes for example. The classic iterator pattern has a structure that looks like this. First, we have an abstract iterable. This is just one method: CreateIterator. Note that in the demo, the employees and departments collections are iterables. A concrete collection or aggregation that follows the pattern implements this method. On the other side, there's an abstract iterator with standard methods to navigate the collection. A concrete iterator implements the abstract iterator, which then can be used in a polymorphic way. In the demo, the departments and employees collections have two different ways of iterating over their collections, which is the very problem we want to solve. The client program uses the concrete methods that have been implemented to iterate over the collection. Note that the classic structure, as pictured here, has three methods not usually used in Python programs. You will not often see implementations of the CurrentItem, First, and IsDone methods in Python. CurrentItem is not needed in Python, since iterator items are returned using the Next function. You simply use the reference returned. First is not used, since instead of resetting the iterator to the first item, the Pythonic way is simply to get a new iterator if you want to begin again. The IsDone method you will likely never see. Python iterators raise a StopIteration

exception when an iterator has been exhausted. Before diving into the next demo, I want to review the special support Python offers for iteration. There are actually two different types of iterators available in Python. A sequence iterator, which implements the dunder _getitem_ method. This works with arbitrary sequences. And note that this method has no direct equivalent in the classic structure we just looked at. A callable object implements the dunder _iter_ method, which corresponds to the CreateIterator method in the structure, and the dunder _next_ method, which is the next method in the diagram. Note that in the Python 2. x series, the method to be implemented is actually called "next", without the double underscores. Both types of iterators can be used for loops and generators. You can choose the one that fits your application the best. All of these things are built into the compiler and are ready to use. Python also provides abstract base classes in the collections module that are ready to use for iteration, Iterable and Iterator for general iterable collections, and Sequence for read-only sequences. We'll use both of these in the demo.

Demo 2: Simple Iterators

For this demo, you'll see how to build iterators for both the employees and department collections. I'll actually use both types of iterators so that you can see them both in action. I'm also going to use a simplified version recommended in the Python documentation, where the Iterable and Iterator are defined together in one class. Once built, we'll use the new iterators in the main program. And then we'll complete the print_summary function. Back in VS code, you can see the revised main program. That's all I really want to do. Really simple. I just want the print_summary function to handle employees and departments in a polymorphic way. No type testing, no special casing, just a simple for-loop. To do that, I have to change a few things. The employee collection now inherits from iterator, an abstract base class in the collections module. The Python documentation states that an iterator is also an iterable, since both require the dunder _iter_ method. I also added the dunder _iter_ method and the dunder _next_ methods as required. Now, the dunder _iter_ method sets or resets the employee ID to zero, then returns itself. So it becomes an iterator. The next method does the heavy lifting. As long as the employee ID is less than the headcount, _next_ increases the ID by one and returns the matching employee object from the dictionary. Once that employee ID hits the headcount, there are no more employees to return, so StopIteration is raised. The department collection now uses the second type of iterator, so it inherits from the Sequence abstract base class and implements the dunder _getitem_ method, which just returns the item by the supplied item number. Since the underlying data structure is a list, that's easy. Oh, I also have to implement the dunder _len_ method, since

that's required by the abstract base class. One last thing. I renamed the attributes in the employee and department classes so that they are the same. With all these changes in place, let's see if the main program works. Start her up, run that program. Open the output window. And we can see that all the objects are returned correctly, as we had hoped. So it works. Our iterators, both of them, do the job required. The Python compiler does the heavy lifting behind the scenes, and our code is nice and simple. There's a subtle bug here, though. What happens if we have two simultaneous iterators over the same collection? First, we'll see the bug, and then we'll see how to fix it.

Demo 3: Multiple Active Iterators

Here I have a little program to show what happens when you have two simultaneous iterators over the same collection and you're using the simplified Python implementation, as in the preceding demo. You can see I'm using the built-in iter function to get two iterators for the same employee collection. Then, a little loop I have here tries to iterate over them at the same time. Note that I'm exclusively using the Next function to get each item. Now, before I run this, see if you can guess what will happen. Got a hunch? Okay, here we go. Let's run this thing. F5. And kaboom. One, two, three, four, then a traceback. Is that what you were expecting? Oh, also, did you notice that the assert statement passed? That means we really only have one iterator. The counter-variable in the employees collection is incremented with each call to the next function. As a result, StopIteration is raised early. In the middle of the print function, as a matter of fact. Clearly we can't do it this way. So what if we actually do want multiple iteration? Here's a new version of the employees collection module that supports multiple iteration. That is, multiple simultaneous iterators. It imports both Iterable and Iterator from the collections module. The employees class is now an iterable and is simpler than it was before. The dunder _iter_ method now just returns a new instance of an employee's iterator. And the dunder _next_ method has been moved to the iterator itself. Moving down, the EmployeesIterator class implements the actual iterator. The logic is pretty much the same as we had before, except there is no need to reset the employee ID in the dunder _iter_ method. Now, whenever we ask for an iterator over the employees collection, we get a new one. Going back to that buggy program, let's see what happens. I'll hit F5 to start her up again. And away we go. And boom, we've got another traceback. Why? Because the assertion failed. But that's exactly what we want. That tells us that the two iterators are actually different. So I'll comment that out and do it again. Hit F5 once more. And run that. And now it runs exactly the way we hoped it would in the first place. The two iterators run simultaneously and

independently. Now, are there other options? Well, there are indeed. Let's see how to do the same sort of thing, but even easier, using a generator.

Demo 4: Iterators Using Generators

This time we'll implement our iterators using generator expressions. Recall the general syntax: x for x in iterable or perhaps f of x for x in iterable for some function f. You can even add a predicate with the optional if <condition> syntax. Now, generators are evaluated lazily. That means that the next item is not evaluated until it's called for. That can be really important if function f is expensive in terms of computation or memory. Or when the iterable is infinite but you only need some of the results. If you haven't seen these before, I recommend you check out the Python Fundamentals course on Pluralsight. com. Now let's see what our iterables look like using generators. Here's the generator-based version of the employee collection. The dunder _iter_ method now returns a generator expression, which does the work of iterating over the employees dictionary items. Note also that the collection no longer implements the dunder _next_ method. That's done by the generator expression instead. Well, what about the department collection? It too is using a generator expression. Even simpler in this case, since the collection uses a simple Python list. And the main program, well, it's the same one as before. And if I run it, well, let's do that, it still works and produces the same output that we had before. Exactly what we want. Just listings of all of those items. Now, what about that multiple iteration bug we fixed? Remember this program? If we run it now, what do we get? Well, it fails at the assertion again. Why? Because the two iterators are now unique. And that's good. If I comment out this assert statement and try again, it should work. And indeed it produces the correct output. Note that if a generator expression is too limiting, you can use a full generator function to do whatever complex logic is required. Generator expressions and functions are covered in the Python Fundamentals course on Pluralsight. com.

Consequences and Summary

The iterator pattern has three important consequences. First, it provides a simple, standard way to process the members of a collection. Then, any program can use a simple for-loop, list comprehension, or generator to process the items in that collection. The collection can implement the iterator in any way it needs to. The implementations can vary widely, even for the same collection. For example, consider an n-way tree that can be processed depth first or breadth first. You'd need quite a different algorithm depending on the direction you want to traverse your tree.

You can have more than one iteration active at once. And separating the iterator from the iterable allows for multiple active, independent operators. In the classic model, this is done with separate iterable and iterator classes. But you also saw that you can do this with generators and considerably less muss and fuss. In summary then, when should you consider using the iterator pattern? Well, use it when you want to provide a way to iterate over a collection while preserving the encapsulation of that collection and not exposing its internal implementation. Also, use it if you want to support multiple active iterators. Using a generator can help you do this easily. Use iterator when you want to provide a uniform interface for client programs, which allows polymorphic iteration. And be sure to watch the other modules in this course to build up your personal design library and how to implement them in Python.

Composite Pattern

Introduction to the Composite Pattern

Hello, and welcome back to this course on Python design patterns. My name is Gerald Britton. In this module, you'll learn about the composite pattern. This pattern is all about trees. Employee hierarchies, family trees, nested groups, and evolutionary taxonomy are all examples of tree structures. People tend to naturally organize things into trees. Oh yes, we should also mention actual trees. One thing interesting about trees is that any part of the tree can be thought of as a mini tree. Lop off a limb with branches and leaves and stick it in the ground and you've got something that looks like a smaller version of the original tree, at least from a distance. Formally, we call these things part-whole hierarchies. Whether looking at the whole thing or just a piece of it, it looks similar. The composite pattern is a way to handle these kinds of hierarchies in code. The goal is to be able to handle either a part of it or the whole thing with simple, understandable code.

Motivating Example

For a motivating example, consider a family tree or set of family trees. Now, a nuclear family consists of parents and some children. Now, let's say we want to find the oldest person. But some of the people we want to look at are not in our families, still, we want to include them. After all, one of them may very well be the oldest person. And what if some of the children are married and have their own families? And we don't want to leave out the grandparents, who are enjoying

their grandchildren and may very well be the oldest. Well, let's make a first attempt at finding that oldest person. For the first demo, I'll start with a simple family: A couple of parents and a bunch of kids. To make it more challenging, I'll add in some unattached single people. Then I'll show you one possible solution to finding the oldest person among them all. Then see if I get into any kind of trouble with my solution.

Demo 1: Working with Families and People Separately

To start off with, I've created two classes. The person class represents one individual and hold just a name and birthdate, which should be a date object, as per the Python standard library date/time module. Now the sharp-eyed will complain that I've done no type checking here. All I can say is guilty as charged. However, we're here to talk about the composite pattern, not data verification, so I'll leave it as it is. The family class is used to hold the members of a family, which should all be person objects. Note that I've implemented an iterator for the family, which should come in handy later. And in the main program, after the required imports, I first build up a family from a few people, and then build a list of singles. And then I've added some simple logic to find the oldest person. Here I have to use two loops, one for the family and another for the singles. And when I found the oldest person, I just print out their name and age in years. Well, let's see if it works. So to run this I'll just hit F5, it'll compile and then we'll run it, and open the debug window and it reports that Arthur is the oldest, a little over 51 1/2 years old. But am I happy with this? Nope, I'm not. I don't like having to use two loops in this way. Also, suppose I wanted to add a second generation to the family, that is, I want a family object to be able to hold families as members to any depth. But to accommodate that, I probably have to change the first loop to a recursive function, which would look even more different than the loop for the singles. Something here is not right. This is a perfect situation for the composite pattern.

Composite Pattern Structure

The composite pattern is a structural pattern. It's a way of putting objects together in tree structures. These structures represent part-whole hierarchies. That's a mouthful. In the first demo, a person is a part, so is the family, and so is that list of singles. Take them all together, though, you've got the whole. We're going to organize these objects into a tree structure. That will mean the clients can handle individual objects, like the person objects and collections of objects, like the family object or the list of singles, using the same code. No need to change it to increase the tree depth or add new collections or objects. The composite pattern has a simple structure.

Clients use the abstract component interface to handle the objects in a tree. The abstract component must be implemented by both the leaf and composite classes. Instance of leaves and composites are called nodes. The abstract component may implement some default behavior for some operations, especially for the add/get child, and remove methods. You'll see why in a moment. A leaf node is at the bottom of a tree, or the top if you're looking at a real tree. They have no members, yet they inherit the add/get child and remove methods. That's exactly why the abstract component may implement some default operations for these methods, so the leaf doesn't have to do that. The operation method of the leaf node will just return some information about that node. A composite node, on the other hand, is a subtree. It will have a set of children, some of which may be empty. And those children can be leaves or other composites. Note that its operation is actually an iterator and follows the iterator pattern discussed earlier in this course. It can also be that a leaf has some operations or attributes that make no sense for a composite. For example, the birthdate attribute has no meaning for a family tree by itself. Nevertheless, all operations are declared in the abstract component for both leaves and composites, some with default implementations.

Demo 2: Restructuring Using the Composite Pattern

For this demo, I'll implement the composite pattern following the structure you just saw. That means I need to create a tree to hold both families and singles. The family and person objects now have to implement the abstract composite interface and client code should be much simpler. Here in VS code, let's start with the abstract base class, abscomposite. I've only defined one required method here, get oldest, which must be implemented by the concrete composite and leaf objects. The person class now inherits from abscomposite and implements the get oldest method, which just returns itself, since for its node, it is the oldest. The tree class now replaces the family class we had before. This corresponds to the composite concrete class in the diagram. Tree has a few things going on, so let's dig in. It inherits from both iterable and abscomposite. I'll define and use an iterator in a moment. The constructor is the same one we had for the family constructor. It just expects and saves a list of members. Since this is a composite node, though, those members can now be person objects or other trees to any depth. Here I've defined an iter method, which implements the iterator interface required by the iterable abstract base class. We'll use that later. The get oldest method is an interesting one. To save coding and reduce errors, I'm using the reduce function from the functools module in the Python standard library. I'm giving it three arguments. Starting from the right, null person is a pseudo person object, which has the maximum possible birthdate. Null person is defined here like this. It's an example of the null

pattern at work. To find out more about the null pattern and how to use it in Python, see the design patterns with Python course on Pluralsight. com. The middle argument is the iterator over the members of the tree, which may be persons or other trees. The first argument is a function reference, defined here. Now the way reduce operates, it passes pairs of items from the iterator and expects one item back. Here, I want to return the oldest item from the two items. If either one of those happens to be a tree, well, the function will then look into that subtree to get the oldest member. This is then, a recursive, depth-first traversal of the tree. The main program starts off as before, except that the family object is now called tree. I've also added a new person and assigned it to the loner variable. Now I can construct trees in any combination I like. I've got a few set up. Tree one is just the hitchhikers. Three two is the singles and that new loner. Three three is composed of trees one and two. I have a little loop here, which just gets and prints the oldest member from each of the trees. Let's run it. So I run it, then I'll open up the output window, and we can see that it works. The tree returns the oldest person, even when composed in a complicated way. I can change the shape of the trees by moving subtrees around in any way I like. Yet, they'll still return the correct results. This is what the composite pattern is all about. Whether looking at the whole tree or just a part of it, the client code works without any modification. In a large application, the tree would likely be built in another place and passed in to the client. No matter, all the client cares about is that it can talk to an object that fully implements the abstract composite interface.

Consequences and Summary

The composite pattern organizes data into tree structures. This permits unified access to subtrees and leaf nodes, which makes for simplified client code. Clients no longer have to have separate paths or run time checks for different components, since they look the same. This also makes it easy to add new kinds of components that implement the same interface without ever having to open up client code or that of the other components, thus following the open/closed principle. However, putting code for all component types in the abstract base class can be seen as a violation of the single responsibility principle. This is essentially a trade off. Simpler code at the expense of the single responsibility principle. In summary, then, when should you consider using the composite pattern? Well, use it when your data naturally fits into a tree-like structure. And client code can treat leaves and subtrees uniformly, without needing to know which is which. You can also enhance composite in a few ways. For example, child nodes can maintain references to their parents. This can improve performance when navigating up the tree, but requires additional care when updating to preserve referential integrity. It's also possible to share components. For

example, in a family tree, a child could have two consecutive sets of parents if they've been adopted at some point. Sharing components can reduce storage requirements. Though if parent references are also maintained, this can be difficult. Also, composite can make your design too general, since it may violate the single responsibility principle. Often, though, the advantages of the simple design outweigh a more granular approach. Consider using composite pattern in your next project.

State Pattern

Introduction and Motivation

Hello. Welcome back to this course on Python design patterns. My name is Gerald Britton. Now I don't know about you, but sometimes my kitchen looks awful. Dirty plates and pots and pans in the sink, food not put away, spills and splashes on the countertop. You might say it's in quite a state, if you have a gift for understatement. That kitchen of mine could be in a tidy state, a messy state, an active state, when someone is cooking, and a cleanup state, somewhere between messy and tidy. If you still remember your college computer science classes, you probably can still see in your head the state diagrams used to model all sorts of real and theoretical problems. Maybe you thought you'd never use it on the job. Nevertheless, in this module, you'll meet the State pattern, learn how to apply it, and learn how to implement it in Python. It turns out that there are plenty of applications for it in all sorts of places. Let's look at one of them. For a motivating example, think about a shopping cart. It makes no difference if you see yourself in the supermarket or on some company eStore. That shopping cart can be in various states, at least empty, containing some items, at the checkout, and paid for. Now, getting from one state to another involves transitions: adding and removing items, checking out, paying for your purchases. But not all transitions are possible in each state. This diagram shows what's going on. You start with an empty cart. If you add an item, you transition to the Not Empty state. If you remove the last item, you get back to the Empty state. At the Not Empty state, you can add or remove items as long as the cart does not become empty. When you're done shopping, you go to the Check Out state. Even then, it's possible to remove items and return to the Not Empty state, or even remove all the items and return to the Empty state. Finally, you pay for the items, and move to the Paid For state.

Demo 1: The Shopping Cart Problem

For this demo, I'll model the shopping cart we've been talking about. So I know what state I'm in, I'll use a simple variable to track it. Then I'll create methods for all the state transitions. I'll run that model to test out my solution, and then we'll step back and think about it to see if we like the result. In Visual Studio Code, I've modeled the shopping cart with its states and transitions. It's a little long and repetitive, but here goes. First, I defined the four states as integer constants. In the constructor, I initialized the state to EMPTY, and set the item count to zero. I've got a method for each transition, and that means I always need to check my state to ensure that the transition is legal, and then I set the following state if it changes. For example, let me scroll up the add_item method. If the state is EMPTY, I report that I added the first item, and that the state is now NOT_EMPTY. Also, I increment the item count. If, on the other hand, the state is in NOT_EMPTY, I add the new item, and report the new item count. If the shopping cart is AT_CHECKOUT, it's not legal to add new items, so I print an error message. Otherwise, the items have already been paid for, so I shouldn't be trying to add any more. I have to do the same thing with the remove_item method, if I scroll it up so you can see it. I have to check the state. If it's legal, I have to print a message and then decrement the item count. If it's illegal, I have to print an appropriate error message. Note the if the cart is AT_CHECKOUT, it is okay to remove items, but not to add them. However, I now have to check the item count to set the new state correctly. At checkout, more state checks and more errors. The only legal input state is NOT_EMPTY. The new state is then AT_CHECKOUT. The paid_for transition checks the states again. The only valid state coming in is AT_CHECKOUT; all others generate error messages. Okay, now I have a little main program to test this. Let's try it out. It simply instantiates the shopping cart, and tries a bunch of operations on it. So let's try it. Well, it looks like it works. But do I like it? Well, do you? Let's go back to the shopping_cart class. For each transition, I have to check each of the four states. Now suppose for a moment I had to add a fifth state, called SAVED, so shoppers can pause their shopping and resume later. To do that, I'm going to have to modify each transition method, to add checks for the new state and take appropriate action. If you haven't guessed it already, this violates the open/closed principle. Other than that, the code is just plain ugly. Long sections of if, elif, else statements are a sure sign of trouble waiting to happen. Other patterns, such as the Strategy pattern and the Command pattern, can help with this sort of thing. But their intents differ. As you're about to see, the State pattern can get us out of this mess.

State Pattern Description and Structure

The State pattern is a behavioral pattern. That means it controls how your program operates. Formally, the pattern operates in some context, and that would be the shopping cart in the demo.

Instead of putting transition methods in the context object, however, we put them in state classes, using one class for each state. The context then delegates transitions to the state objects, while the clients only ever interface with the context. The State pattern looks like this. If you think you've seen it before, you're not wrong. It resembles the Strategy pattern, thought the Strategy pattern has a different intent. The context is the object clients want to interface with. That's the shopping cart in the demo. It keeps a reference to the current state, and that reference is used for every request, which are the state transitions. All states derive from an abstract base class, which defines an abstract method for each type of request, and each concrete state has a method to handle each type of request. Now, let's see how that works out in code with the shopping cart.

Demo 1: Using the State Patter for the Shopping Cart Problem

For this demo, I'll implement the State pattern for the shopping cart application. That means I'll need to create a shopping cart context object, and then I have to create state objects. And those state objects have to handle the various transition requests. And finally, I'd better make sure it still works. To start off with, here's the abstract base class, AbsState. It begins with an implemented constructor. This one requires the context, the shopping cart that is, to be passed in as an argument. The constructor just saves this in an instance variable to use later. Then there are five abstract methods, one for each of the five transitions: add_item, remove_item, checkout, pay, and empty_cart. Each concrete state must, of course, implement all five methods. Now we have one class for each of the states the shopping cart can be in. We have an Empty state, a NotEmpty state, a Checkout state, and a PaidFor state. Each of these must implement the methods in the abstract base class. But before we look at them, you need to see the new ShoppingCart class. ShoppingCart starts off by importing the various state classes. Then, the constructor instantiates those four states, and assigns those to instance attributes. You'll see why later. After that, it sets the item count to zero, and then sets the initial state, which is Empty of course. Now, here's the Empty state. You can add an item to an empty shopping cart, and you'll get a message that that's just what you did. You can't remove anything, though, since the cart is empty. You can't go to checkout, since you haven't selected anything to purchase. In the same way, you can't pay for it. And trying to empty and empty cart is, well, futile. Let's look more closely at the add_item method. It starts off by incrementing the item count. It also sets the cart state to the NotEmpty state, using that instance variable we mentioned. You know, the one that saves the reference to the state in the context, that is, the shopping cart. The other state classes are similar. They implement the transition methods in a way that makes sense for that particular state. Later, when an operation is performed on the shopping cart, add_item for example, the shopping cart's

add_item method just calls the method of the same name for the current state the shopping cart is in. And note that at the moment it makes the call, the shopping cart neither knows nor cares what that state actually is. The other four action methods in the shopping cart do the same kind of thing. Remove, checkout, pay, and empty_cart just call the method of the same name for the current state the shopping cart is in. Now, go back to the Empty state for a second. See how after adding an item, it sets the state of the context, which is the shopping cart, to NotEmpty? That means that when the next operation is performed against an instance of the shopping cart, the corresponding method in the NotEmpty class will be called. Keeping that in mind, check out the NotEmpty class. It has one state transition, which can happen when you remove an item and there's nothing left in the cart. The state of the cart is then reset to Empty. It also has a second state transition when you call the check_out method. Then the new state is set to CheckOut. The CheckOut state has three transitions. One for remove_item, in case that leaves your cart empty, in which case the cart will be set to the Empty state, one for pay, which moves your cart to the PaidFor state, and a third for empty_cart, which removes all the items and sets the cart back to the Empty state. Finally, the PaidFor state only issues messages, all of which are errors. Also, there's no transition to another state. This is actually the final state. If you want to buy something else, you'll need to get a new shopping cart. With all that in place, we arrive at the main program. It has two rounds of tests to see if our state machine works. In the first round, I get a shopping cart, add and remove some items, then checkout and pay. In the second round, I start off the same way, but I empty that cart at checkout. That should send me back to the Empty state, where I can begin again. And lastly, I try to add an item after paying, and that should trigger an error message. Okay, let's try it out. I hit F5 to compile it, and we'll run it, and then look at all the messages we've received. Looking at the messages for the first cart, you can see that the messages echo the operations that were performed, and I wind up by paying for my two items. The second cart looks like it worked fine too; even after I emptied the cart at checkout, I could still add a new item again. And finally, what about my error? I was in the PaidFor state. Even so, I tried to add another item. I was told, correctly, to go and get a new shopping cart. So it all works together quite nicely. If it seems a little complicated, I encourage you to watch this demo again before continuing. It's also a good idea to download the exercise files, which include all these we just went though, so that you can try it out on your own.

Consequences and Summary

The State pattern has some important consequences. To start off with, it encapsulates state-specific behavior. You saw that in the demo. The behavior of the shopping cart in the Empty state

was quite different from its behavior in the CheckOut state, yet the shopping cart was unaware of the current state, or when the state changed. This makes it easy to add new states. For example, you might be asked to add a Saved or Suspended state to the shopping cart. The bulk of the work would be in the new states. You can also add default behavior to the abstract base class, so that you only need to implement methods that cause state transitions in the state subclasses. The pattern distributes behavior across state classes. This makes the solution less compact, but eliminates long conditional statements. And the pattern makes states' transitions explicit, without complicating code in the context, that is, the shopping cart. If there are no instance variables, it is possible to share the state objects, saving memory. And this would actually be an example of the flyweight pattern. The State pattern does not dictate where transitions are defined. The demo happens to define them in the state subclasses, but you could also do it in the context, especially if those transitions are fixed. In the demo, I created all the states up front, but they could be created at transition time instead. It's a trade-off between memory consumption and CPU time. So in summary, when is the State pattern applicable? Well, whenever an object's behavior depends upon its state, and that behavior changes as the state changes. You should use it to replace long if/elif/else code segments when you are checking an object's state. In this sense, it's similar to the Strategy pattern, though there we are not concerned with objects' state, per se. If you're interested in learning more, check out the Strategy and Command patterns from the Design Patterns With Python course on Pluralsight. com.

Proxy Pattern

Introduction

Hello again, and welcome back to this course on Python design patterns. My name is Gerald Britton. Have you ever held a few shares in some publicly traded company? I have. Not enough to make me rich on dividends or growth, sad to say. If you've done the same, chances are you may have received a ballet in the mail before the shareholders annual meeting. Asking if you'd allow your shares to be voted by proxy. A proxy vote is a ballet cast by one person on behalf of a shareholder of a corporation who would rather cast a proxy vote than attend a shareholder meeting. Of course, you have to trust the person casting the proxy vote and feel comfortable that they have a similar position as you do regarding the current and future plans of a company. Basically you're giving them control over your vote. The proxy pattern is a lot like that, hence the

name. When implemented correctly, a proxy stands between your client code and the actual object you need to access. In the shareholder analogy, you are the client, the company is the object, and the proxy voter is, well, the proxy. In this module, we'll explore some of the common uses of the proxy pattern, and see how to implement them in Python. There are four principle types of implementations of a proxy pattern. A remote proxy provides a local representative for an object in a different address space. A great example is a web client that uses a database on a separate machine. This allows the database to be kept behind a corporate firewall while the web server is in a DMZ facing the internet. This enhances security by reducing and controlling access to the database. A virtual proxy creates expensive objects on demand. Imagine that your client needs to get information from a database on another server. That query may run long or return thousands of rows to fully populate the object. A virtual proxy can provide a way to return the data in a lazy fashion, only when actually needed. A protection proxy controls access to the original object. This is useful when the access rights of clients depend on other factors. Such as a user ID or time of day. Again, the database is a natural example here. Not everyone could see your personal information stored in a database, nor should they. A smart reference proxy can perform additional actions when an object is accessed. One use is reference counting so that an object may be released when it is no longer accessed. In managed languages like Python however, that is usually handled by the garbage collection subsystem. Another use may be to add locking to an object to ensure that some access requests are single threaded. If you dig a bit, you'll find out that database management systems use all of these types of proxies. Remote proxies are used to access databases on other servers. Virtual proxies are used with buffering to minimize IO. Protection proxies are used extensively. Limiting access to objects and even parts of objects. For example, a database might limit access to certain rows or columns in a given table. Smart reference proxies are used for copy on writing, auditing, logging, locking, and maintaining statistics and all kinds of other things. Also, web servers oven use proxies when loading images. A virtual proxy allows the server to display a message like image loading, when retrieving and rendering a large image, for example.

Demo 1: The Employees Problem

To demonstrate the need for the proxy pattern, consider an employee object. You need to control access to it since it contains sensitive information, such as an employee's birthdate and salary. Say you also have an AccessControl object. With employee IDs and a flag that indicates if the accessor can see personal information or not. Now you need to write a client program that accesses the employee object while restricting access to the personal information, according to

the entries in the access control object. Well let's take a crack at that problem. Here in Visual Studio Code, I've created a little project to see if I can solve the request to provide access to the employees object, but restrict who can see what. The employee object is pretty simple. Just a few attributes. The birthdate and salary attributes are personal and sensitive information though. That's what I want to restrict access to. I'm going to use the access control object to indicate who can see what information. It just has an employee ID and a bullion can see personal. If true, the system should allow a request from that employee ID to see the birthdate and salary of employees. I've got some test data set up that looks like this. I just build two dictionaries. Now in a real application, that data would no doubt be loaded from persistent storage. In the main program, I import the test data then set up a function to retrieve and print the results. The function takes in two arguments. A list of employee IDs to print and the ID of the requester. Then it loops through the employee IDs and gets the matching object from the test data if it exists. Then it constructs a details line to be printed. First it adds the employee ID and name, since these are not sensitive. Next the function needs to add the birthdate and salary, but only if the requester is authorized. So it looks up the requesters ID in the access control dictionary, and checks that the can see personal attribute is set to true. If so, it adds the birthdate and salary information to the details to be printed. Finally, it just prints the details accumulated so far. At the bottom of the program, I call this function twice. Once with a non-authorized requester and again with an authorized requester. Let's see what happens when we run it. So if I open up the debug console, we can see the results. The first two lines are from the first call to the function. The confidential information is hidden. The last two lines are from the second call to the function, using a requester ID of an authorized user. Now you can see the birthdates and salaries have been added in. This is far from ideal however. If the employees object adds new kinds of access controls. The main program will need modification. The same thing if more attributes are added. You know we really should separate the concerns of the main program. Which is retrieving and printing employee data from the data access itself. Which requires some security processing. This is an ideal scenario to use the proxy pattern. We'll use the protection proxy in this case.

Description of the Proxy Pattern

The proxy pattern is classified as a structural pattern. It acts on a real subject. That would be the employees in the demo. The proxy keeps a reference to the subject, and this is a great example of composition. It exposes an identical interface to the client. Which is unaware that it is even using a proxy. And the proxy controls access to the real subject, and may even be responsible for creating and deleting it. The classic proxy pattern has this structure. We begin with an abstract

subject. The subject represents the object that needs to be controlled in some way. The request operation is whatever operation the client needs to gain access. There can be multiple operations. The abstract subject is implemented by a concrete subject. Which is the actual thing we want to use. It is also implemented by the proxy. Which will control access to the subject. Note that the proxy is composed with a reference to the concrete subject. That reference is used when the client program makes a request. The proxy controls the request, even to the point of denying the request all together, and returns the results, possibly transformed in some way by the proxy, to the client. The client however is unaware that a proxy is in use. It only knows that it is talking to an abstract interface that supports the methods in the abstract subject. Now there is a second simpler way of implementing the pattern using inheritance, shown here. The client program then interfaces directly with a proxy. However, one of the principles we want to follow is to favor composition over inheritance. Use the second pattern only in limited and highly contained situations.

Demo 2: Using the Proxy Pattern

For this demo I'll implement a protection proxy for the employees example. That means I'll need to create three new classes. An abstract base class for the subject. Which is the employee's collection, and note that the request method will be to get employee info method we had before. I'll need a concrete subject. Which is the employee's collection itself that implements the abstract base class and actually retrieves the data. Then I'll need a proxy subject for the employees collection. Which will be composed with a concrete subject to add the protection part. In the main program, I'll use a factory to get a reference to the object to use, and then I'll test the solution to make sure it still works. Back in VS Code, here is the abstract subject. Since the subject for this example is employees, I've called it AbsEmployees. There's just one method that must be implemented. The get employee info method. This corresponds with the request method in the UML diagram we looked at. The employees class implements the abs employees abstract base class, and the required get employee info method, but all it does is return a generator that yields the employees and the list of IDs passed in. It uses the same test data as in the first demo. Of course in a real application, this would probably be loaded from a database. The other class we need to implement is the proxy itself. Which looks like this. First the constructor takes in a reference to an employee's object. Which is where the composition happens. It also accepts a requester ID to be used later. In this implementation of get employee info, it loops through the employees using the method of the same name in the reference it saved at instantiation. Then it implements the protection logic and returns different employee objects depending on the

requesters authorization level. If the employee ID matches the requester ID, or the requester ID is in the list of those who can see restricted information, the returned employee object contains all information. This actually enhances the original logic. Since now I allow an employee to see their own personal information. Otherwise, the birthdate and salary are masked. Note that since this method uses the yield statement, this is also a Python generator. Now before we get to the modified main program, I want to show you a factory function I've created. The factory imports the employees collection and the proxy class. Then the get employees collection function, which is the factory function, returns a new proxy object. Passing in the employees collection and the requester ID. When the main program calls this function, it will be unaware that it is getting a proxy instead of the full employees collection. Now here is the modified main program. Its only external reference now is to the factory function. The print employee details function uses the factory to get a reference to an employees collection. There's no way to tell at this point what the factory will return. Except that it will implement the abs employees ABC. Once it gets the collection, it simply iterates over it printing out the details. To test the new program, I'll call the function twice. Once with an authorized requester that can see everything, and once with a requester that can only see their own confidential details. Let's see if it works. In the output we see that the authorized requester could see all the details about every employee requested. On the other hand, the ordinary employee, who is Ted, could only see his own personal information. The details for the other employees have been masked, and this is the desired effect of the protection proxy.

Consequences and Summary

The proxy pattern has some important consequences. First of all, it introduces a level of indirection when accessing the real subject. You saw this in the protection proxy demo. When the client program asks for employee information, it was unaware that the returned objects pass through the proxy. Which controlled the access. A virtual proxy can be used to perform lazy instantiation, or to cache results. In Python, the func tools module has an LRU cache decorator which is, in effect, a virtual proxy. A remote proxy can hide communication details. For example, the popular Python package pyodbc can be used to access a database which may be on your laptop or half way around the world. That makes it a type of remote proxy. A smart proxy can add housekeeping duties. For example, you might want to use a proxy for locking to access the data structure for multi-threaded application. Since the proxy is hidden away from the client code, the proxy actions take place apart from the real subject. So the solution obeys the open/closed principal. The classes are open for extension but closed for modification, and by implementing the

classic design, we show a preference for composition over inheritance. Which helps keep class hierarchy shallow, making maintenance easier. In summary then, when is the proxy pattern applicable? Well use it whenever you want to add controls to an object, but you want to stay true to the open/closed principle. Keep in mind that it is possible, even common, to use proxies in combination with each other, while remaining completely unaware of each other. Thanks for watching this module on the proxy pattern. Look for opportunities to use it in your own projects.

Summary

Course Summary

Congratulations! You have made it all the way through this course on building more design patterns with Python. Adding the eight patterns presented here to the eight discussed in the Design Patterns with Python course, you now have 16 patterns you can use. However, in the world of design patterns, you're only about one third of the way through. Let's take a few minutes to review the patterns learned in this course. The facade pattern is a way to implement a common interface to access a variety of APIs that have similar goals but differing representations. Programs using Facade need not be aware of particulars of how to talk to the underlying APIs. You saw how to do this, to talk to SQL Server, to get employee information, and learned that you could easily modify the Facade, without ever having to touch the client code. The Adapter pattern converts an API into an interface that clients expect. You saw two different ways to implement this pattern. The first were object adapters, which use composition and can easily work with subclasses as well as parent classes, and class adapters, which use inheritance instead, and let the adapter override some of the adapted class's methods. Using the Adapter pattern, let the client program retrieve information from vendors or customers, without being aware which one it was using. The Decorator pattern provides a way to add responsibilities to an object, without violating the open-closed principle, or introducing a profusion of subclasses. You saw an example of this, involving a car dealership, where options, like the engine size or the interior upholstery, were added using decorations, without modifying the base class. The Template pattern defines the basic outline of an algorithm, and then lets subclasses implement one or more steps, while enforcing the order of the steps. Some steps may be required, some optional, and some may have default implementations in the abstract base class for the template. Optional steps have no implementation, but can be used to hook into the algorithm at pre-defined points. We used the

Template pattern to set up different kinds of transportation. The Iterator pattern is a way to add iteration to any collection. That makes it easy to write for loops against the collection. We looked at a few options, including simple iterators, which only allow one instance at a time, generators, which provide lazy iterators, and multiple iterators, which allow multiple active iterations. You saw how the Composite pattern can be used to provide a single interface to objects in tree structures, both nodes and leaves. Although in a way, this pattern violates the single responsibility principle, the benefits often outweigh this transgression. We use the Composite pattern to provide access to a tree of families. We used the State pattern to simplify the coding for an E-store shopping cart. You saw how to build states and transitions, in a way that is transparent to the client code, and simple to maintain. The Proxy pattern gives us a way to control access to objects. There are at least four kinds of controls that are common. Protection proxies control access to objects or parts of objects. We used this in the demo to restrict confidential employee information to authorized users only. Virtual proxies can create expensive objects on demand. Python comes with one of these, in the form of the LRU Cache decorator. Remote proxies hide communication details. You'd use this sort of thing to communicate with a remote machine. Smart proxies can add housekeeping. This can be great for adding locking, auditing, or performance tracking to an object. I hope you've enjoyed this Pluralsight course on building more design patterns with Python. If you haven't already, be sure to watch the Design Patterns with Python course, which also covers some of the history and philosophy behind design patterns, and introduces eight essential patterns. I'm Gerald Britton, and I hope to see you again on Pluralsight.

Course author

Gerald Britton

Gerald Britton is a Pluralsight author and expert on Python programming practices and Microsoft SQL Server development and administration. A multiple-year of the Microsoft MVP award, Gerald has...

Course info

Level	Intermediate
Rating	★★★★★ (27)

My rating ★★★★★

Duration 2h 7m

Released 10 Apr 2017

Share course