Writing Testable Code

by Matthew Renze

Start Course

Bookmark Add to Channel Download Course

Course Overview

Course Overview

Hi, I'm Matthew Renze with Pluralsight, and welcome to Writing Testable Code. Testable code is code that makes automated testing easy. Unfortunately, most developers never learned how to write code for testability, so they struggle with software practices like unit testing and test-driven development. However, learning how to write code that is easy to test is the first step in making automated testing quick, easy, and even enjoyable. As an overview of this course, first we'll learn about testable code and how creating seams in our code makes our code testable. Then we'll learn a series of best practices that make testing code easier such as simplifying object construction, working with dependencies, and managing application state. For each of these practices, we'll walk through an open-source demo application that applies these practices so you can see first-hand how they are implemented. By the end of this course, you'll have the skills necessary to write code that is easy to test. In addition, practices like unit testing and test-driven development will become significantly easier because you'll have a deep understanding of what makes code testable versus what makes code difficult to test. As an introductory course, there are no prerequisites for this course; however, having basic experience with at least one C-like programming language in unit testing will be beneficial, but it is not required. I'll walk you through

everything you'll need to know as we progress throughout this course. So please join us today at Pluralsight and learn how to create code that is easy to test with Writing Testable Code.

Introduction

Introduction

Hello, and welcome to Writing Testable Code. I'm Matthew Renze with Pluralsight, and in this course we'll learn about how to write code that makes testing easier. The skills you'll learn in this course will make practices like unit testing and test-driven development practices much easier for you because you'll have a deep understanding of what makes code testable versus what makes code more difficult to test. But first, let's start with a quick story to explain why we want to write testable code. Imagine for me, if you will, that we've been asked to design a brand new automobile. We can build this car in one of two ways. First, we could build this car with all the parts fused together as a single, giant, complex machine. So rather than spending the extra time and effort making all the parts of the car purposely interchangeable and replaceable, we would simply fuse everything together as quickly as possible. We wouldn't spend time or effort designing a hood that opens and closes; we would just create the shell of the car as one large piece of formed metal. In addition, we wouldn't spend time or money adding unnecessary things like tire pressure valves, interchangeable spark plugs, or replaceable light bulbs. While these cost-cutting decisions might make this car less expensive to create, it would probably come at the expense of some pretty significant long-term costs. Without access to the engine and removable, interchangeable parts, we couldn't easily test or replace any of the parts that were defective. So, if the car failed to start, we would have no way to test an individual component in isolation to see which part was causing the failure. The only way to test the car would be to turn the key in the ignition and see if the car could drive or not. If the car failed to start, we couldn't identify or replace the faulty part, so we'd have to scrap the entire car and buy a new one. On the other hand, we could design our car like all modern cars are built today. We could design a car using interchangeable parts for testability and maintenance. This way each part could be individually removed from the vehicle, tested in isolation, and replaced if necessary. If the car doesn't start, we could simply open the hood, test each part in isolation, and determine which part was defective. Then we could easily replace the defective part rather than needing to scrap the entire vehicle. Clearly, the second type of car has some serious advantages over the first type of car in terms of

testability and maintenance. In fact, it probably seems absurd to us that anyone would even consider building the first type of car in this day and age. However, the reality is that many software developers are creating software just like this first type of car. They're fusing all of the components together in ways that make software easier to create, but as a result make testing and maintenance extremely difficult. From my own personal experience with over 17 years in the software industry, this is actually much more common than you might think. As software developers, we need to learn how to create software like the second type of car, software that is easy to test and maintain even if it requires a bit of extra time and effort up front. Practices like unit testing and test-driven development are great for helping us create software that is testable and maintainable; however, many developers struggle with learning, adopting, and sticking with these practices. This is often because they find these practices too difficult, too time-consuming, or too painful. Over the years, I found that the two best ways to make unit testing and TDD fast, easy, and relatively painless are first, to learn how to write testable code so that your code is easy to test (this eliminates much of the difficulty with unit testing and test-driven development), and second, learn how to write effective unit tests to make it easier to write the tests themselves. In this course, we're going to focus on the first of these two practices, learning how to write testable code.

Course Overview

As an overview of this course, first we'll introduce the topic of testable code. We'll learn what it is, why it's important, and how to write testable code. Next, we'll learn about seams and how creating seams in our code makes code testable. Then, we'll learn how to construct objects in ways that are easy to test. Next, we'll learn how to work with dependencies in ways that make testing easier. Then, we'll learn how to manage application state for testability. Next, we'll learn how maintaining the single responsibility principle can benefit testability. Finally, I'll show you where to go next to learn more about writing testable code. In addition, throughout each module of this course, we'll see sample code demonstrating each of these topics in detail. The purpose of this course is to learn how to write testable code. This is not a course on unit testing or test-driven development specifically, rather, it's intended to complement unit testing and test-driven development practices. Having a deep understanding of the physics of what makes code testable will make unit testing and test-driven development significantly easier for you. In addition, because writing testable code improves cohesion and coupling in our code, coincidentally, it also makes software easier to maintain. We'll learn about all three of these ideas throughout this course. The intended audience for this course are developers who are either writing or learning

how to write unit tests and developers who are either performing or learning how to perform test-driven development. However, for developers who have not already learned how to create unit tests or develop software using test-driven development, this course will make it easier to begin unit testing and test-driven development when you're ready. Since this is an introductory course, there are no required prerequisites; however, having basic programming experience with at least one C-like programming language will help you understand the code used in the demos. In addition, having experience with unit testing or test-driven development will be beneficial as well. However, if you don't have sufficient experience with either of these two optional prerequisites, don't worry. I'll take time to explain unit testing, test-driven development, and all the code in the demos throughout this course. So now let's learn what testable code is, why it's important, and how to write testable code.

Testable Code

Let's start by discussing the current state of testing in the software industry. Unfortunately, despite great advances in both testing practices and technology to enable software testing, many software developers still do very little testing of their code, and when they do, it's often relatively ineffective or highly inefficient. There are numerous reasons given for why developers don't create high-quality automated tests for their code. These reasons include that it's too hard to create and maintain automated tests, not having enough time to create tests, and it's not my job to test software (testing is for testers not developers), along with many other possible reasons. By learning how to write testable code, we'll see that creating automated tests can be easy, save us significant time in the long-run, and even become an enjoyable part of a software developer's job. Before we learn about why we should write testable code, let's take a step back. First, let's address the more basic question of why should we write automated tests for our code in the first place? Testing our code provides several important benefits. First, testing our code helps reduce bugs, both bugs that occur while we're creating code and regression bugs that occur as a result of changes to existing code. Second, testing helps us to reduce the total cost of developing and maintaining an application. We spend significantly more time and effort maintaining software than we do creating it. So while it may initially cost us more to create unit tests, these tests pay for themselves in the long-run by reducing the cost to maintain existing functionality. This, in turn, reduces the overall cost of the software project as a whole. Third, testing our code helps us to improve the design of our code. If we're writing unit tests before or in conjunction with our production code, we force ourselves to take time to think about the problem our code is solving. This generally leads to better designs than writing code without thinking about how to test it.

Forth, software acts as a form of self-documentation. If we write our automated unit tests in a language that clearly communicates the behavior that the test is attempting to verify, these tests act as an executable form of documentation describing the expected behavior of the software. Finally, testing our code helps us to eliminate fear, the fear that changing our code will break existing functionality. If we have a comprehensive suite of unit tests covering all execution paths of significance in our codebase, we can be confident that any change that we make in our codebase will not break existing behavior without causing a test to fail. There are a variety of types of tests that we can use to verify the correct behavior of our software. Some are based on what they are testing; for example, unit tests, integration tests, component tests, service tests, and coded UI tests. Some are based on why they are being tested; for example, functional tests, acceptance tests, smoke tests, and exploratory testing. Still, others are based on how they are being tested; for example, automated tests, semi-automated tests, and manual tests. In this course, we'll be focusing primarily on writing testable code for automated unit tests that verify the functionality of our code. However, these testable code practices apply to many of these various types of testing. Good software testing starts with a comprehensive suite of automated unit tests and works its way out from there. Essentially, if you've written code that makes creating unit tests easy, all of the other types of tests are generally easier to create as well. So what is a unit test? A unit test is a type of automated software test that verifies the correct behavior of a single unit of production code in isolation. For example, here we have a unit of production code. This could be a single class or a single method, essentially any isolated unit of functionality we wish to test. We create test code to verify the correct behavior of this isolated unit of functionality. We'll take a look at an example of unit testing in our upcoming demo. Now that we've reviewed software testing in general, let's begin discussing testable code. Just to be very clear about what we're discussing, what is testable code? In essence, testable code is code that makes testing easier, not harder. It's code that has been designed with testability in mind so that creating automated unit tests is relatively quick and easy. So how do we write testable code? First, we create seams in our code that allow us to inject tests into this code. Second, we simplify the construction of our objects to separate the testing of their construction versus the testing of their behavior. Third, we work directly with the dependencies we need rather than digging through a chain of dependencies. Forth, we decouple our code from global state in our application so that we can unit test in isolation and in parallel. Fifth, we maintain the single responsibility principle to keep our tests focused and simple. Finally, we use test-driven development to drive the design of our tests. We're going to be covering all but the last of these topics in-depth throughout this course. Unfortunately, test-driven development is a topic that requires an entire course in itself; however, we'll provide a brief introduction to the test-driven

development process and the benefits of TDD during the last module of this course. In each of the remaining modules in this course, you'll be learning a series of guidelines to help you write testable code. However, these are just guidelines, not absolute rules. Context is very important in software development, and so there are always exceptions to every guideline. However, it's important that you learn each of these guidelines, understand their pros and cons, and know within which context they're applicable or not. Then, when you're creating your own software for each situation, you need to use your best judgement and implement the solution that makes the most sense for your specific situation. So now, with this in mind, let's see a quick demo to introduce the idea of writing testable code.

Demo

Now we'll look at a quick demo to introduce how to write testable code. For all the demos in this course, we're going to be building a simple application to create invoices for automobile parts and service. Our demo code will involve various tasks such as calculating, printing, and emailing these invoices. In these demos, I'll be showing you the simplest code possible to demonstrate each of the ideas in this course. This is so that we can focus on the concepts in this course without getting lost in the details of building a complex invoicing system. However, this also means that we'll be omitting many aspects of modern software development that you would typically find in real-world software such as logging, security, caching, exception handling, and more. While these examples might be useful for teaching how to write testable code, please keep in mind that they may not be representative of other best practices in software development. All of the demos in this course will follow the same three-step process. First, we're going to show a sample of code that is hard to test. Next, we're going to show the same code refactored to be easy to test. Finally, we'll walk through the unit tests for this refactored code so that you can see how it is tested. The demos in this course will depend on a few technologies. First, we'll be using Visual Studio as our integrated development environment, or IDE. In addition, all code samples will be written in C#. NET. For our unit testing framework, we'll be using NUnit. For our mocking framework, we'll be using Moq, spelled M-o-q. We'll also be using a tool that makes mocking test doubles easier called AutoMoq. Finally, we'll be using a dependency injection framework called Ninject for all of our dependency injection demos. While we'll be using. NET technologies to demonstrate the ideas in this course, these ideas are applicable to a wide array of programming languages, frameworks, and technologies. So even if you're not specifically using. NET technologies, you'll find these testable code practices applicable to most programming languages and technology stacks. All of the source code for the demos in this course are available as free

open-source software. You can download a copy of the demo code from the Exercise Files tab on the homepage for this course, or you can view, download, and modify the source code from my GitHub repository at the following URL. I encourage you to download, explore, and play around with the code used in these demos. In addition, feel free to use, modify, or redistribute this code in accordance with the included open-source license. Now let's take a look at a very simple demo of code that is difficult to test, followed by code that has been refactored to be easy to test, and finally, the unit test for our refactored code. This demo will likely be review for anyone already familiar with unit testing or test-driven development; however, it's important for developers new to unit testing to see a very simple example. In addition, it's important that we're all on the same page with the terminology and the technologies used throughout this course. For this demo, we're going to look at a basic console application that performs just a single function. This application computes the total price for the invoice based on the price of the parts, the price of the service, and an optional discount. We start with the console application's main method that is the entry point for the application. First, we get the price of the parts from the first argument passed into the console application and convert it into a decimal value. Next, we get and convert the price of the service from the second argument. Then, we get the amount of the discount to be applied to the invoice from the third argument. Next, we compute the total price by adding the price of the parts to the price of the service and we subtract the discount. Finally, we print the total price out to the console window. The part of this application that we're going to focus on is the calculation to compute the total price of the invoice. While this example is the simplest example I could think of to communicate the essential idea here, we could easily imagine much more complex logic that we might need to test. However, the important question right now is how would you go about writing an automated unit test for this code? The way this application was coded, we can't easily write any automated test for this calculator logic. We would essentially have to manually launch the application, type in some user input, and visually inspect the results. We would need to do this for every combination of values that we wanted to test and for every feature of the calculator. Unfortunately, this is how many developers are currently testing their entire applications. They launch the application, navigate to the part of the application they want to test, and manually test the functionality from the outside. However, if we were to extract the calculator logic into its own class, we might be able to create automated unit tests for it. So let's give this a try. Let's create a class called Calculator that contains the logic for performing the total price calculation in our demo application. This class contains a single method called GetTotal. The method takes, as input, the price of the parts, the price of the service, and the amount of the discount. Then it calculates the total price in the body of this method and returns the total price's output. Now we can use our new Calculator class to replace the embedded logic in our original

console application. We construct a new Calculator class. Then we call the GetTotal method on this class, passing in the user's input and receiving the total price as our output. The application should behave exactly the same as it did before; however, the total price calculation logic is now testable in isolation via automated unit tests. Please note that I have not refactored any of the code outside of the calculator logic for this example. In practice, you would also refactor and test any logic of sufficient complexity or risk. However, to keep things simple for this demo, we're just focusing on the calculator logic and nothing else. In order to test the functionality of the Calculator class, we create a Test class, also known as a TestFixture, called CalculatorTests. This class is marked with a TestFixuture attribute, which tells NUnit that the class contains unit tests that should be run by our test runner. Next, we have a private field called Calculator that contains our subject under test. That is the invoice class that we're testing. Then we have a method called SetUp, which is marked with the SetUp attribute. This attribute tells NUnit that the method should be called before each and every test is run to ensure that the test is set up properly. Within our SetUp method, we construct our Calculator class so that it is initialized before each and every test. Next, we have our first Test method marked with a Test attribute. This attribute tells NUnit that the method contains a test that should be run each and every time the unit tests are run. The Test method is titled TestGetTotalShouldReturnTotalPrice. The name of the method should make it clear to other developers what is being tested by the test method. Within this method, we execute the action being tested, that is we call our GetTotal method passing in three numbers, that is $1 for parts, $2 for service, and $0. 50 for a discount. Then, we assign the return value to a variable called result. Finally, we assert that the result is equal to $2. 50. If we coded the logic in our GetTotal method correctly, this test should pass; if not, the test should fail. With our code refactored so that our calculator logic is easily testable, and with our unit tests in place, we can verify the correct behavior of our calculator logic using our unit test runner. A test runner executes the unit test code in order to verify the correct behavior of the subject under test. In this case, I'm using the built-in unit test runner that comes with ReSharper, which is a developer productivity plugin for Visual Studio. However, you're free to use any unit test runner you prefer. As we can see, the unit test passed, so our code is behaving as expected. What we've just seen is the most basic example of the essential idea behind this course, how to take code that is difficult to test and make it easy to test. For some of you, this example may have been too simple. For others, this might have been the first time you've seen code being tested like this. In either case, no worries. The remaining topics in this course will become more complex and will mirror many real-world problems that you'll encounter in your day-to-day lives as software developers. However, I'll do my best to keep our code demo simple and explain everything as we go.

Summary

In this module, first we saw an overview of the course. We covered the purpose of the course, its focus, and its intended audience. Next, we learned about writing testable code. We learned that testable code is code that makes testing easier, not harder. Finally, we saw a demo where we introduced the idea of how to write testable code. We saw code that was difficult to test, we refactored it to code that was easy to test, and finally we saw the unit tests that verified the correct behavior of our testable code. In the next module, we'll learn how creating seams in our code allows us to make code testable.

Creating Seams in Code

Introduction

Hello again, and welcome to our next module on writing testable code. I'm Matthew Renze with Pluralsight, and in this module we'll learn how to create seams in our code to make it testable. As an overview of this module, first we'll learn about creating seams in our code. We'll learn what seams are and why creating them makes our code testable. Next, we'll learn about problems that exist for testability when there are no seams in our code. Then, we'll learn how to identify symptoms that our code may be lacking seams for testability. Finally, we'll learn how to refactor untestable code by creating seams to make it testable. So let's get started.

Seams

Let's think back to our two hypothetical automobile designs again, that is the one with all the parts fused together and the one built with interchangeable parts. Imagine that a spark plug, which uses electricity to ignite fuel in the cylinders of an engine, stopped working in our fused vehicle. How would you test if the spark plug itself was defective or if the problem was with the electrical system or the engine cylinder? In short, you couldn't, or at least not easily. In order to test an individual spark plug in isolation, we would first need to be able to disconnect it from both the electrical system and the engine. However, in an automobile with interchangeable parts, testing a spark plug in isolation is easy. We simply disconnect the spark plug from the electrical system and remove the spark plug from the engine. Then we can connect it to an appropriate power supply and a power meter for testing. If the spark plug fails to develop sufficient voltage to generate a spark, we can be relatively confident that the problem is with the spark plug itself.

However, if the spark plug generates a spark, the problem is most likely with the vehicle's electrical system or the engine. Using a removable and interchangeable spark plug creates a seam between the automobile and the spark plug. This seam allows us to remove the spark plug from the engine and the electrical system and test the spark plug in isolation. We can create seams in our code as well. A seam is a place in our code where we can alter the behavior of the program without manually editing the code in that place. Michael Feathers first introduced the concept of a seam in his book Working Effectively with Legacy Code. Seams allow us to replace the calling class with a test fixture and replace any dependency with a test double, which is a class that stands in for the actual dependency for testing purposes. This allows us to test the behavior of the production class in isolation, that is testing our subject under test without involving the calling class or any dependencies. When there are no seams in our code, this creates a few problems for testing. First, without seams, we cannot easily pull apart the individual pieces of our code. This means that we cannot connect a test harness to the class we want to test. In addition, we cannot replace any of the class' dependencies with test doubles to remove those dependencies while testing. Ultimately, without seams we cannot test our code in isolation. There are several symptoms of this problem that we should be on the lookout for in our code. First, we should watch out for the new keyword in our application logic. Whenever we see the new keyword, it's an indication that we're creating a dependency from inside of our class. There are cases where this may be okay, for example creating a new integer or a new string inside of our method; however, any dependency that is sufficiently complex or is tied to an external resource will make testing more difficult. Second, we should keep an eye out for classes that make calls to static methods on one or more of its dependencies. Static methods create very tight coupling between the caller of the method and the class with the static method being called. Once again, there are cases where this may be okay, for example using math. min, math. max, string. join, or int. parse. Since these are leaf nodes in our call stack with no external dependencies and they have already been tested by a third party, they're usually not a problem from a testability perspective. Third, we should watch out for places in our code where we have direct coupling to third party frameworks and external resources. For example, datetime. now, file. write, and console. writeline. Once we're directly coupled with frameworks and external resources, we can no longer test our code in isolation. The solution to this problem is that we need to create seams in our code to separate components that are wired directly together. Seams allow us to decouple our code from its dependencies. We refer to this type of code as being loosely coupled rather than tightly coupled. There are several ways we can decouple our code from its dependencies and create loosely coupled code. In many of the C-like programming languages, for example C#, C++, and Java, the way to do this is to program to interfaces rather than programming to

implementations. This makes our components interchangeable just like the spark plug in our earlier example. Finally, we want to inject the dependencies of our classes into the classes themselves. There are several ways we can do this as well; however, in general, we should prefer to inject our dependencies via constructor injection. That is we want to inject each of our dependencies as an argument into the constructor of the class that needs the dependency when the class is being constructed. This allows us to swap out the dependencies when we're constructing an instance of the object from the class. If we decouple from our dependencies by programming to interfaces and injecting our dependencies, this means that we can test any class in isolation. We can swap out any of the class' dependencies with a test double, that is a mock, a stub, or a fake, when we're constructing the object so that we are only testing the class in question and not any of its dependencies. So now let's see a demo where we create a seam in our code to make that code testable.

Demo

Now let's see a demo where we create seams in our application to make our code testable. Imagine that we're creating a feature in our automotive invoicing application to print an invoice. When the user clicks the Print button in our application, it will trigger a command to print the selected invoice. The command will retrieve the specified invoice from the database and will write each line of the invoice to the printer. For this demo, we're only going to worry about printing the invoice number, the total price for the invoice, and the date the invoice was printed. We'll worry about the rest of the invoice fields later. First, let's look at the code that is difficult to test. We'll begin with the PrintInvoiceCommand. This class contains all the logic to print an invoice. It takes an invoiceId from the user interface of the application. First, it creates an instance of a class called Database, which provides access to the invoice data in the database. Next, it gets an invoice from the database using the specified invoiceId. Then, it uses a static WriteLine method on the Printer class to write lines of text to the printer. It prints two lines to the printer, that is the Invoice ID and the Total price for the invoice. Finally, it gets the current date from the static Now property of the DateTime structure and prints the date that the invoice was printed to the printer as well. This code is hard to test in isolation because of the dependencies on the Database and Printer classes and the DateTime structure. In order to test this code, we would need an actual database and printer connected, plus we'd also need access to the current operating system date and time. In addition, we'd also need an actual invoice in the database with a known invoice ID and a known total price in order to verify the results. Clearly, creating automated tests for this class would be quite difficult. This code is hard to test in isolation for three different reasons. First, there's a

dependency on the database. When we use the new keyword to construct a dependency inside of a class, we are tightly coupled to that class, so we cannot test in isolation from the dependency or any of its dependencies either. Second, we're using a static method on the Printer class rather than an instant method of a printer object. This creates tight coupling between the PrintInvoiceCommand class and the Printer class. However, for the purposes of this demo, we'll assume that the Printer class is a class that we created ourselves, so we'll be able to refactor it on our own to make it easily testable. The third problem is that we're calling the static Now property on the DateTime structure. This problem looks similar to the previous problem with the static method call in the Printer class; however, because the DateTime structure is part of the. NET Framework and thus not part of our code, we cannot change this code ourselves to make it easily testable, so we're going to have to take a slightly different approach to solve this problem using a wrapper class. Now let's see how to refactor this code to make it easily testable. In order to make our classes interchangeable, like the spark plug in our earlier example, we need to use interfaces. An interface in programming is code that describes the actions that an object can perform. It describes the necessary inputs for each action and the expected outputs. In essence, an interface acts as a contract between a class using the specified interface and every class that implements that specific interface. Because of this, we can use an interface as a placeholder for an actual class and then substitute in any class that implements that interface later. For example, we've created an interface for our Database class called IDatabase. This interface defines the contract between a caller, which calls methods of the database, like GetInvoice, and the class or classes that will implement this interface, like our Database class. Here we have our Database class. Notice that it implements the IDatabase interface. This means that in order for the class to be valid at compile time, it must implement all of the methods exactly as specified in the IDatabase interface. That is, the methods must have the same name, take the same input arguments, and return the same type of output. Please note that to keep things simple this demo code isn't actually connected to a real database. Instead, I'm just throwing a NotImplementedException, indicating that the method has yet to be implemented. In a real example, the code necessary to talk to the database would be contained in this method. We also do the same for existing Printer class. We create an interface called IPrinter that defines the methods of the printer. Then, in the Printer class, we refactor our existing WriteLine method, which was a static method of the class, into an instance method of the object that implements the WriteLine method we defined in our interface. Once again, I'm omitting the actual code to talk to the printer and just throwing a NotImplementedException to keep things simple for this demo. To solve the problem with the dependency on the. NET Framework's DateTime structure, we're going to have to take a slightly different approach. Since this code is part of a third party API, that is Microsoft's. NET

Framework, we can't just go out and refactor their code to make it easier for us to test. Rather, we're going to have to create a wrapper class which will wrap the DateTime structure and make it easier for us to use in a testable way. First, let's create an interface for our DateTimeWrapper to define the methods that it will provide us. We define a single method called GetNow, which returns a DateTime structure just like the Now property of the existing DateTime structure. Then, we implement this interface in a class called DateTimeWrapper. As we can see, this class has a single method called GetNow, which returns a DateTime structure. The method simply forwards the request on to the DateTime structure's Now property and returns the result to the caller. The point of a wrapper class is to decouple our code from third party code that makes our code difficult to test in isolation. We want to make our wrapper class as simple as possible to avoid any logic that might require unit testing. The reason for this is that if we keep our wrapper class free of logic, then we don't need to write unit tests for the wrapper class or the class that it's wrapping. This is because we make the assumption that the third party that created this code has fully tested it and that it works as specified. We can, however, test this functionality as part of our full system tests, which should be part of our comprehensive suite of tests for the application. Let's take a look at the same PrintInvoiceCommand class that we saw before, but now it has been refactored for testability. Rather than working with our Database class, our Printer class, and the DateTime structure directly, as we did in our first example, now we can define three interfaces, that is IDatabase, IPrinter, and IDateTimeWrapper, to act as placeholders for our actual Database class, Printer class, and the DateTime structure. Then, in the constructor of our PrintInvoiceCommand class, we pass in three objects that must conform to the IDatabase, IPrinter, and IDateTimeWrapper interfaces. We then assign these three objects to three fields in our class so that we can access them from any of the methods inside of this class. This is referred to as dependency injection via constructor injection. That is, we're injecting the PrintInvoiceCommand class' dependencies into the constructor of the class as it's being built. When the application is running in production, we'll inject the regular database, printer, and dateTime wrapper objects, and the application will work as expected using the real database, the real printer, and the operating system's clock. However, when we're running our unit tests, we'll instead inject a test double to stand in for our database, one for our printer, and another for our DateTimeWrapper. This way, we don't need the actual database, printer, or operating system clock for our tests; we'll just use our test doubles instead. Now, in the Execute method of this Command class, we get the invoice from the database object that implements the IDatabase interface, and we print the first two lines of text to the printer using the printer object that implements the IPrinter interface. Finally, we get the current system date and time from the dateTime wrapper object that implements the IDateTimeWrapper interface and write the date

printed to the printer. This code works exactly the same as our previous sample code; however, this code is much easier to test. Once we've refactored our code to make it testable, how do we actually go about testing it? Here we have a TestFixture called PrintInvoiceCommandTests. We have five private fields inside of our TextFixture, a field called command, of type PrintInvoiceCommand, which is our subject under test, three fields for our mock objects, that is our test doubles, a mockDatabase, which is a mock class of type IDatabase, a mockPrinter, which is a mock of type IPrinter, and a mockDateTime wrapper, which is a mock of type IDateTimeWrapper. These three mock classes, which are part of our mocking framework, will serve as test doubles for our Database, Printer, and DateTimeWrapper classes, respectively. We also have a field for our Invoice class, which we'll use as a stand-in for a real invoice during out tests. Next, we have two constants called InvoiceId, which is set to the integer 1, and Total, which is set to 1. 23. We'll use these two fields to hold test values for our invoice ID and total price, respectively. Finally, we have a static readonly DateTime structure called Date, which will hold a test value for our system date, that is February 3, 2001. Please note that in the. NET Framework the DateTime structure does not support literals, so we can't create it as a constant like we did with our InvoiceId and Total price variables. However, we can create a static readonly field to store our DateTime test value instead. In the SetUp method of our TestFixture, first we create an Invoice class and assign it the test values we created for InvoiceId and Total. Next, we construct our three mock classes to act as test doubles, that is our mockDatabase, mockPrinter, and mockDateTime wrapper. A mock object provides us with complete control over its behavior so that we can tell its methods to return certain values and verify the methods have been called with certain values. For example, our mockDatabase class will act as a test double for our real Database class. We can make it behave like the real Database class by telling it to return specific values when certain methods are called with specified parameters. In this case, we'll set up the GetInvoice method so that when a user calls this method with a specified InvoiceId, which in this case is 1, the mockDatabase class will return the test invoice that we set up a few lines earlier. We'll do the same with our mockDateTime object. We'll set up the GetNow method so that it returns the test date that we created earlier in the TestFixure. Finally, we create a new instance of the PrintInvoiceCommand class, passing in our mockDatabase object, our mockPrinter object, and our mockDateTime wrapper object. Please note that the. Object property of the mock class returns an instance of the mock object itself. This is what we pass into the constructor of the subject under test. Our first unit test is called TestExecuteShouldPrintInvoiceNumber. This test will verify that the correct invoice number is printed when the Execute method is called. First, we call the Execute method on the command, passing in our specified InvoiceId, using the test value we defined at the top of the tests, which was the integer 1. Next, we use our mockPrinter object to

verify that the WriteLine method has been called with the text Invoice ID: 1 passed in as an argument. We specify that this method should have been called exactly one time during the test. So if the WriteLine method on the mockPrinter has been called with this text string exactly once, then the test passes. However, if the WriteLine method has not been called with this text string exactly one time, then the test fails. We do the same for our Total price line in our second unit test. We execute the command; then we verify that the Total price line has been printed exactly once with the value $1. 23. Finally, we do the same with our date Printed line in our third unit test. We execute the command; then we verify that the printer's WriteLine method has been called with the date February 3, 2001. If all three tests pass, our PrintInvoiceCommand is behaving as expected. Before we wrap things up for this demo, there are a few things that we can do to reduce duplication and clean up our test class. I'm showing this so that you can see how writing efficient unit tests can help make your testing process quick, easy, and relatively painless. To do so, I've refactored our existing PrintInvoiceCommand TestFixture using an automocking framework called AutoMoqer. It eliminates much of the setup of our mock objects by automatically creating them as needed. We'll start again at the top of this refactored TestFixture and work our way down to the bottom. First, at the top of this TextFixture, we still have our subject under test, our test invoice, and our three test values. However, instead of having three mock classes, like we did before, we have a single field called mocker of type AutoMoqer. AutoMoqer is our automocking framework. In the SetUp method of our TextFixture, we still create our invoice as we did before; however, instead of creating our three mock classes, we'll create an AutoMoqer instead. Then, instead of using each mock class directly, we'll call the GetMock method of our AutoMoqer class, passing in an IDatabase interface as a generic method type parameter. This will return an automatically generated mock that implements the IDatabase interface. Then, like before, we can set up the GetInvoice method of our mockDatabase to return our test invoice. We'll do the same for our IDateTimeWrappper so that the GetNow method returns our test Date value. To end the SetUp method, we'll create our subject under test using the AutoMoqer's Create method, passing in a PrintInvoiceCommand class as a generic method type parameter. This will tell AutoMoqer to create a new PrintInvoiceCommand and automatically create and inject the three mock dependencies that need to be injected into the constructor of this class. Finally, we'll replace the three independent test methods with a single test method using three test cases instead. The TestCase attribute tells NUnit that this test method should be executed multiple times, once for each specified test case. The character string inside of the TestCase will get passed in as an argument into the line parameter of the Test method. Then, in the body of this method, we execute the PrintInvoiceCommand like we did in the previous test, and we verify that the mockPrinter's WriteLine method was called exactly once containing the

correct line of text for each TestCase. As you can see, using techniques like this significantly reduces the amount of setup code and test code duplication in our unit tests. By writing both testable code and efficient unit tests, we can significantly reduce the time, effort, and difficulty of testing our code.

Demo Summary

Now let's quickly recap what we accomplished in this demo to make sure that we understand the concepts. Our PrintInvoiceCommand class was tightly coupled to three dependencies, that is a Database class that provided access to the database, a Printer class with a static method that provided access to the printer, and a DateTime structure, which was a third party component, which provided access to the operating system's clock. First, we replaced the Database class with an interface representing the database. This created a seam in our code. The seam allowed us to continue using the Database class and the actual database at runtime; however, we can now swap out the database with a mockDatabase object while testing. The test harness created and injected the mockDatabase into the constructor of the PrintInvoiceCommand while testing. Then we did the same thing with the dependency on the static method of our Printer class. We created an IPrinter interface that defined a WriteLine method to use instead of the Printer class. Then we refactored the static method of the Printer class into an instance method that implemented the WriteLine method definition. This created a seam in our code, which allowed us to inject the Printer class into the PrintInvoiceCommand at runtime so we could print to the actual printer. However, this also allowed us to swap out the printer object with a mockPrinter while testing. The test harness created and injected the printer object into the constructor of the PrintInvoiceCommand while testing. For the dependency on the DateTime structure in the system clock, we created an interface for a wrapper class for the DateTime structure. This created a seam in our code, which allowed us to inject the DateTimeWrapper class into the PrintInvoiceCommand at runtime. This forwarded any calls to the GetNow method of the wrapper class to the actual DateTime class, which has a dependency on the operating system clock. However, we can now swap out the dateTime wrapper object with a mockDateTime wrapper while testing. The test harness created the dateTime wrapper object and injected it into the constructor of the PrintInvoiceCommand while testing. If you were able to easily follow along with this demo, the rest of the course should be relatively easy to follow as well. However, if you struggled with understanding the concepts in this demo, I would suggest spending a bit of time playing around with the demo code, and then re-watch this demo before moving on. While some of these ideas

might be a bit difficult to grasp initially, with a bit of practice, these ideas will become second nature to you.

Summary

In this module, we learned that we should create seams in our code so that we can test our classes in isolation. We learned how to decouple our tightly coupled dependencies, a practice referred to as loose coupling. Then we learned how to program to interfaces rather than implementations and inject our dependencies, which conform to these interfaces, into our classes. This allows us to test our code in isolation. In the next module, we'll learn how to construct objects in ways that make them easy to test.

Constructing Testable Objects

Introduction

Hello again, and welcome to our next module on Writing Testable Code. I'm Matthew Renze with Pluralsight, and in this module we'll learn how to construct objects in ways that are easily testable. As an overview of this module, first we'll learn about constructors and how objects are created. Next, we'll learn about problems that occur when we mix the construction of our objects with their behavior. Then, we'll learn how to identify symptoms that indicate that we're constructing objects in ways that are difficult to test. Finally, we'll learn how to refactor the construction of objects that are difficult to test into ones that make testing easier. So let's get started.

Construction

Building a car and testing a car are clearly two separate tasks. When we're building a car, we're assembling the components of a car in such a way that when we're done we'll have a fully operational vehicle. When we're driving a car, we're using the fully assembled vehicle to drive from point A to point B. It would seem very odd for someone to be attempting to drive a car as they were building it. However, as software developers, we often do exactly this when we're constructing objects in our application. Quite often, we're mixing application logic into the

construction of our objects. In general, this makes testing more difficult. Constructors are methods in our class that are used to build instances of objects for that class. Inside of constructor we execute code necessary to prepare the object for use. However, it's quite common for developers to do additional work inside of the constructor of a class. Oftentimes they construct other dependencies, talk to external resources, execute initialization logic, or even execute application logic. This creates several problems from a testability standpoint. First, if our constructors create any complex dependencies, this creates tight coupling between our class and its dependencies. This is the same problem that we saw in our previous module. This is why we injected our dependencies into our constructors instead. Second, if we have logic in our constructor, it is difficult to test this logic directly. The only way we can test the logic in a constructor is by constructing the object in different ways and verifying the state of the object state, which is often hidden from direct testing. Third, if we have logic in our constructors, it makes our setup process for testing more difficult. We have to navigate through that logic using each possible code path every time we set up our test to ensure that the class behaves correctly in each possible state. There are several symptoms to look out for in our constructors that indicate that we're making testing more difficult. First, seeing the new keyword in a constructor is often a symptom. Oftentimes, this is an indication that we're tightly coupling to a dependency that we might need to mock out for testing. However, constructing simple value objects like integers, lists, and dictionaries is usually just fine. Second, any logic in our constructor is usually a symptom of testability problems. This could be a conditional statement, that is an if-then statement or a switch statement, or it could be a loop of some kind. Essentially, any code other than the assignment of values is often an indication of a potential problem. So how do we solve these problems? In general, we want to keep our constructors very simple. First, we want to inject our dependencies rather than creating them in our constructors. Then, within the constructor, we simply assign those dependencies to private variables inside of our class. In addition, we want to avoid adding logic in our constructors. This includes conditional logic, any loops, or calculations. You also need to be careful that you don't just move this logic from the constructor into an initialization method and then call that method from the constructor. This isn't actually fixing the problem; it's just moving it. Finally, when building complex objects in object graphs, use well-established design patterns that separate the construction of the objects and object graphs from the application logic. For example, use the factory pattern, the builder pattern, or dependency injection via an IoC container. We'll look at how to use a factory and perform dependency injection via an IoC container during our upcoming demo. In general, we want to avoid mixing the construction of our objects with the behavior of our objects. It's very easy to test the results of the construction of an object graph, and it's very easy to test the behavior of objects in isolation

as well. However, it's quite difficult to test either of these things when they're mixed together. So, we should separate the code that constructs our objects from the code that implements their behaviors. Finally, we need to separate the construction of two types of objects, which Misko Hevery, author of AngularJS, refers to as injectables and newables. An injectable is an object that is composed of other injectables and performs work on newable objects. Injectables are generally services that implement interfaces. For example, our Database, Printer, and InvoiceWriter classes are all examples of injectables. A newable is an object at the end of your object graph. These are generally things like entities and value objects. For example, invoice, customer, address, and credit card numbers are all newables. As a general rule for testability, an injectable class can ask for other injectables in its constructor; however, it should not ask for any newables in its constructor. Inversely, a newable can ask for other newables in its constructor; however, it should not ask for an injectable in its constructor. Maintaining separation between injectables and newables in this way benefits testability. However, distinguishing between these two types of objects isn't always as easy as it sounds. So now, let's see a demo where we'll put these practices into action.

Demo

Now let's see a demo where we'll simplify the construction of objects to improve testability. The automotive parts and service company that we work for was very happy with our new invoice printing feature. However, they've been having issues with customers neglecting to pay overdue invoices. The billing department thinks the problem is that the overdue invoices look too much like regular invoices, so they've asked us to print all overdue invoices in bright red ink. While we're not a big fan of this decision, we agree to implement the feature as requested. So let's see how we might implement the solution first in a way that is difficult to test. Then we'll refactor this code to make it easy to test. We'll start again with our PrintInvoiceCommand class. As we can see, this class is roughly the same as it was before. The class takes two dependencies, that is an IDatabase and IPrinter via constructor injection. The Execute method gets an invoice from the database using the specified ID, like it did before; however, the code to write each and every line of an invoice to the printer is now contained inside of a new class called InvoiceWriter. This class takes a printer and an invoice as an argument. Then we call the Write method to write the invoice to the printer. Encapsulating the logic to write an invoice to the printer is generally a good thing to do; however, the way this InvoiceWriter class is constructed and used is where our testability issues will exist. You may have noticed a few symptoms already. For example, this code is mixing application logic with the construction of its dependencies. Coincidently, this mixing of

application logic and object construction is also creating tight coupling between our PrintInvoiceCommand and the InvoiceWriter class. This is the same type of problem we saw in the previous demo. Our InvoiceWriter class consists of two private fields, that is one for an object that implements IPrinter and one for an Invoice. These two dependencies are injected into the constructor of this object and assigned to the respective fields. However, in addition to the assignment of these dependencies, this constructor is also doing additional work. First, it's constructing a new PageLayout object and setting the page layout of the printer to this new object. This is going to create tight coupling between the InvoiceWriter and the PageLayout class. This makes testing difficult for the same reasons we mentioned in the previous demo. Next, it's testing to see if the invoice is overdue, and if so, it's setting the ink color to red instead of the default color, which is black. Once again, this code is mixing object construction with application logic. This behavioral logic is going to make it more difficult to set up each test and to verify the logic that uses red ink for overdue invoices inside of a test. Finally, the Write method writes out each line of the invoice to the printer. Based upon whether an invoice is on time or overdue, these invoice lines will be printed in either black ink or red ink, respectively. This code might seem completely reasonable, and I've seen a lot of code like this in my career; however, there are several problems with this code in terms of testability. Now let's refactor this code so that it's easy to test. We'll continue to inject and assign our IPrinter dependency via constructor injection like we did before. However, we'll also inject a PageLayout dependency adhering to the IPageLayout interface. This allows us to decouple our InvoiceWriter from the PageLayout class. As we can see, all the logic has been eliminated from the constructor. The only thing we do in the constructor is simple assignment of values. Now, within the Write method of this class, we'll take an invoice as an argument. Notice that we've moved the invoice from the constructor to the Write method. This is because invoice is a newable as Misko Hevery would refer to it. According to the rules we discussed earlier regarding injectables and newables, an injectable, like InvoiceWriter, can ask for other injectables in its constructor, like IPrinter and IPageLayout. However, it should not ask for a newable; for example, an invoice or an invoice ID. When we keep the construction of our object graphs separated into injectables and newables, it makes it much easier to test our injectables and newables in isolation. However, distinguishing between the two isn't always clear and can be a bit ambiguous. For example, our PageLayout class could be considered a newable or an injectable depending upon the context. If you're still struggling with the concept of injectables versus newables, we'll show an example at the end of this demo that might make things more clear. Within the body of the Write method, we set the PageLayout of the printer to the PageLayout we injected into the constructor of this class. Then, if the invoice is overdue, we set the InkColor to Red. Finally, we write each line of the invoice to the printer. Now that we've

simplified the constructor of our InvoiceWriter class by separating construction and behavior, let's see how this makes your InvoiceWriter class easier to test. The PrintInvoiceCommand class will also be easier to test as a result of the changes we made. Now we can inject an object that implements an IInvoiceWriter interface into the constructor of our PrintInvoiceCommand. Then, in the body of the Execute method, we simply call the Write method, passing in an invoice as an argument. This eliminates the mixing of application logic with the construction of a dependency in the body of the Execute method. Ultimately, this class will be significantly easier to test as well. Now that we've simplified the construction of our InvoiceWriter class and our PrintInvoiceCommand, let's see how this makes testing them easier. In order to test our InvoiceWriter class, we create an InvoiceWriter TextFixture. This TextFixture has three fields, an InvoiceWriter, which is our subject under test, an AutoMoqer, to simplify mocking of our dependencies, and an Invoice, which we'll use in place of a real invoice. Like our previous demo, in the SetUp method of this class, we set up the Invoice with an Id of 1 and set IsOverdue equal to false. Create a new AutoMoqer, and use the AutoMoqer to create a new InvoiceWriter class, which automatically creates and injects a mockPrinter and a mockPageLayout into the constructor of the InvoiceWriter class. The first test method is titled TestWriteShouldSetPageLayout. In this method, first we execute the Write method on the InvoiceWriter. Then, we get a reference to the mockPageLayout object from our AutoMoqer. Finally, we get a mockPrinter from our AutoMoqer and verify that the mockPrinter's SetPageLayout method has been set to the mockPageLayout that we injected into the constructor of our InvoiceWriter class exactly one time. Because this method has been moved from the constructor of the InvoiceWriter into its Write method, it's easier to test directly by executing the Write method rather than indirectly testing it as a result of the InvoiceWriter being constructed. The next method tests that overdue invoices should be printed in red. First, we set the IsOverdue property of the invoice to true indicating that the invoice is overdue. Then, we execute the Write method, passing in our overdue invoice as an argument. Finally, we verify that the mockPrinter's SetInkColor method has been called with the color of red exactly once. The third method tests that on-time invoices should be printed in the default color, which is black. Here, rather than setting the invoice to overdue, like we did in the previous test, we leave it set to false, which is a value we set in the SetUp method earlier. Then, we execute the Write method and verify that the printer's SetInkColor method has never been called with any color whatsoever. This means that the default printer ink color, which is black, will be used for normal invoices, that is invoices that are not overdue. Finally, our last test method will verify that each line of the invoice is written to the printer. We do this the same way we did in the last demo with one test case for each line to be printed. Before we move on from this demo, there's an important concept that

we've intentionally deferred until now. You may have been wondering how we efficiently construct and wire up all these objects at runtime. This is a great question, and I'd like to provide you with a quick introduction to factories and dependency injection to demonstrate how this is done. Here we have our simple console application that takes an invoiceId as an argument and prints the invoice to the printer using the PrintInvoiceCommmand. The program needs a PrintInvoiceCommand, which needs a Database and an InvoiceWriter, which needs a Printer and a PageLayout. There are a few ways we could construct and wire up this graph of injectable objects. First, we could do this by hand, like we see here, constructing a new object for each constructor parameter of the object graph. This is referred to as poor man's dependency injection since it's a quick and easy way to construct and wire up a simple tree of dependencies. However, this doesn't scale very well as it becomes overwhelming for applications with large and complex object graphs. As an alternative, we could encapsulate the logic to construct this object graph into a factory. A factory is a class that contains the code necessary to construct an object graph. Using a factory helps us to separate the construction of objects and our application logic. Factories offer benefits for testing in various ways in our application, especially because they can implement an interface and be injected into classes that need to construct an object. This means that we can mock out the Create method of the factory and inject anything we want in place of the runtime objects. A sibling to the factory pattern is the builder pattern, which serves the same purpose, but works better for building objects and object graphs that are composed in many different ways. While factories and builders are useful to improve testability throughout our application, at the root of our application there is a solution to this problem that is much more effective. This solution is dependency injection using an Inversion of Control framework, or IoC framework for short. For example, in our refactored program, we create what is called an IoC container, which in the language of Ninject, our IoC framework is referred to as a kernel. Next, we specify a binding for each interface to each class it implements the respective interface. For example, we bind the IDatabase interface to the Database class. We do the same for the IInvoiceWriter interface and the InvoiceWriter class, for the IPrinter interface and the Printer class, and for the IPageLayout interface and the PageLayout class. We get the invoiceId from the arguments passed into the application, like we did before; however, now, rather than assembling the object graph by hand or using a factory to construct the object graph, we use the IoC container and ask it for a PrintInvoiceCommand. The container knows that the PrintInvoiceCommand needs an IDatabase and an IInvoiceWriter in its constructor's dependencies, so it resolves the dependencies to the appropriate Database and InvoiceWriter class as specified in the binding. It does the same for the IPrinter and IPageLayout dependencies for the InvoiceWriter class' constructor as well. Essentially, the container has the knowledge to

construct the entire object graph itself. Finally, with our PrintInvoiceCommand constructed, we can execute the command in order to print our invoice. As you might imagine, having to specify bindings for each interface to each class would be pretty time-consuming and error prone. Fortunately, modern IoC frameworks support binding by naming conventions. This means that as long as the interface and the class share the same name, excluding the letter I prefix on the interface, the IoC framework can automatically bind interfaces to their implementations. Here we have our previous application refactored to bind by naming convention. As we can see, we create an IoC container, like we did before. Then, instead of manually setting up every binding between each interface and each implementing class, we specify that the container should bind by conventions. It should select all types from the current assembly, select all classes in the assembly, and bind the default interface to the matching class. Binding via naming conventions can save us a tremendous amount of time and effort constructing our object graphs. Please note that in general we only use our IoC container to build our graph of dependencies at the root of our application. Passing an IoC container around our application has several problems for testability, which we'll discuss in the next module. In addition, please note that we only use IoC containers to construct graphs of injectable objects rather than newable objects. This is one of the main distinctions between these two types of objects. Injectables are constructed at the root of our application via dependency injection using an IoC container, whereas newables are never constructed by the IoC container. Newables are always constructed using the new keyword somewhere beyond the root of our application. Now let's review what we've accomplished in this demo to make sure that we understand these concepts.

Demo Summary

Now let's review what we've accomplished in our demo to make sure that we understand the concepts. First, we had an InvoiceWriter class, which took an IPrinter and an Invoice as a dependency. In the constructor, it created a PageLayout object, which caused tight coupling. In addition, the constructor also executed logic to set the color of the printer's ink based on whether the invoice was overdue or not. To fix these problems with testability, we refactored the constructor of the InvoiceWriter so that it only took injectable objects as constructor arguments. We also converted the PageLayout class into an injectable object by defining an interface and injecting it in to the constructor of the InvoiceWriter class. This broke the tight coupling between these two classes, which allowed us to inject either a real PageLayout at runtime or a mockPageLayout for testing. We moved the invoice argument out of the constructor and into the Write method since invoice is a newable object rather than an injectable object. This allowed us to

pull the difficult-to-test logic out of the constructor and keep all of the behavioral logic in the Write method. It also allows us to construct the InvoiceWriter class using dependency injection via an IoC framework because the constructor is now composed only of injectables, which the IoC framework is able to provide. Then, we demonstrated how to test this refactored InvoiceWriter class. Finally, we saw how to compose a graph of injectable objects using four methods. First, we did this by hand using poor man's dependency injection. Next, we encapsulated this logic in a factory class. Then, we used an IoC framework with manual bindings. Finally, we used an IoC framework with automatic bindings via naming conventions. We covered a lot of information in this demo, so if you're having trouble following along, please spend some time playing around with the sample code. Then feel free to re-watch the demo to make sure you have a grasp on all of the concepts before you move on to the next module.

Summary

In this module, first we learned that we should simplify the construction of our objects to improve testability. We do this by injecting our dependencies into our constructors rather than creating dependencies inside of our constructors. In addition, we learned to avoid adding logic to our constructors because this makes setup and testing more difficult. We also learned that in general we should avoid mixing object construction with application logic just like we want to avoid building a car at the same time we're driving the car. Finally, we learned that we should strive to separate the construction of our injectable and newable objects. Injectables can be composed of other injectables, and newables can be composed of other newables; however, we should not mix the two when we're constructing our object graphs. In the next module, we'll learn how to work with dependencies in ways that are easy to test.

Working with Dependencies

Introduction

Hello again, and welcome to our next module on Writing Testable Code. I'm Matthew Renze with Pluralsight, and in this module, we'll learn how to work with dependencies in ways that make testing easier. As an overview of this module, first we'll learn about the Law of Demeter, a general practice of object-oriented software for creating loosely coupled code that has quite a few benefits for testability. Next, we'll learn about problems that may occur for testing when we

violate the Law of Demeter. Then, we'll learn about symptoms that generally indicate that we may have a testability problem. Finally, we'll look at solutions to refactor these problems to a more testable design.

Dependencies

Imagine that you're an employee at our fictitious automotive parts and service center. Once you've completed work on someone's vehicle, you can get the payment from the customer in one of two ways. First, you could grab the customer, reach into their pocket, pull out their wallet, and take the money that they owe you. Or, you could simply hand the customer an invoice and let them pay you. In the first case, we're digging through things that don't belong to us to take what we need. In the second case, we're asking the person responsible to give us what we need instead. While it might seem odd to even suggest reaching into people's pockets to take things that don't belong to us, software developers do something very similar to this in the code that we write on a regular basis. We refer to this as violating the Law of Demeter. The Law of Demeter is a general principle of object-oriented software development proposed by Ian Holland. It states that each unit of code should only have access to specific information. While it's a bit difficult to fully define the Law of Demeter, it can be easily summed up as only talk to your friends, don't talk to strangers. Essentially, our classes should only depend on the object's data and methods that were directly given to them. They shouldn't go reaching into the dependencies of the dependencies they were given. The Law of Demeter isn't a strict law, it's more of a helpful guideline for creating object-oriented software. As such, there are often exceptions to this rule. In this course, we're just going to be looking at how this guideline applies to testability. Violating the Law of Demeter creates a few problems for testability. First, it creates a particular form of tight coupling. For example, if we've been given an object, it's completely fine to call the methods on that object that we've been given. However, if calling one of those methods returns another object and we call its methods, we now have both a dependency on the first object and a dependency on the second object that was returned by the method we called on the first object. Essentially, the first class is an immediate friend of our class, the second object is a stranger to our class. This makes our tests more difficult to set up because we need to create a test double for both the first object and the second object as well. Finally, when we pass in objects that are just used as containers to get access to the other objects we need, our dependencies are no longer explicitly defined in the constructor of our class. This makes it more difficult to figure out what dependencies we need to mock out since they aren't explicitly defined in the constructor. This also prevents our AutoMoqer from automatically constructing the necessary dependencies, since it can't discover them from

the constructor of the subject under test. There are several symptoms that we should be on the lookout for that indicate that we may be violating the Law of Demeter. First, seeing a series of method calls appended to one another in a long line of code is often a symptom. For example, calling something like environment. getsecurity. getlogin. getuser. getusername is usually a good indication that we're violating the Law of Demeter. However, there are some cases where a chain of methods is perfectly acceptable. For example, using the builder pattern with fluent notation. Second, if we see a dependency that is only being used as a container to retrieve other dependencies, this might be a sign that we're violating this principle as well. We refer to this as a container dependency, since the only purpose of the container is to hold other objects that we might need. This is the same problem that we alluded to in the previous demo about passing around the IoC container throughout our application, rather than keeping it localized to the application root. Container dependencies often have suspicious sounding names like container, context, environment, or service locator. If you see objects with names like these, it's often an indication that they are being used in ways that may violate the Law of Demeter. The solution to this problem is to follow the Law of Demeter. We want to talk only to our immediate dependencies, we don't want to go digging into their dependencies. We do this by injecting the dependencies that we need, and using those dependencies directly rather than digging through their dependencies. Finally, we inject only what is specifically needed rather than injecting container objects that hold a bunch of dependencies that we may or may not need. We refer to this practice as making our dependencies explicit. That is, we can look at the constructor of our class and know exactly which dependencies it depends on. This makes it very obvious which dependencies we need to mock out while testing. Now let's see a demo where we'll refactor a few types of violations of the Law of Demeter into easily testable code.

Demo

Now let's see a demo where we'll refactor a few types of violations of the Law of Demeter into code that is easy to test. Our invoice printing functionality has been very well received by the automotive parts and service company we work for. In fact, it's been so well received that it's been causing a bit of a problem. Invoices are now so easy to print that employees are frequently printing them off just to save time having to look them up when they move back and forth between computer terminals. As a result, the office managers are concerned about generating too much paper waste. We suggest to them that it would probably make most sense to just make it easier to look up an invoice in our software or provide a feature to email invoices to customers to discourage printing unnecessary invoices. However, the office managers reject our suggestion

and have asked us to record who was the last person to print each invoice in order to identify patterns of waste so that they can selectively discipline employees for their wasteful behavior. Despite our reservations, we agree to implement this feature as requested. So let's see first a demo of what this feature would look like in a way that's difficult to test, then we'll refactor this code to make it easily testable. Let's start again with our PrintInvoiceCommand class coded in a way that's difficult to test. Rather than injecting explicit dependencies, like the IPrinter and IInvoice writer dependencies that we saw in our previous demo, another developer on our team has decided to inject the IoC container of our application to make it easier for them to create this new feature. Their reasoning was that the number constructor parameters was growing too large, so they wanted to consolidate it to a single parameter that they could use to retrieve any dependency that they might need now or any time in the future. Injecting a container that can be used to retrieve any dependency is referred to as the service locator pattern. However, because the service locator pattern has significant disadvantages relative to the advantages that it provides, many software developers refer to this as an anti-pattern instead. An anti-pattern is a design pattern of software development that should generally be avoided rather than followed like regular software design patterns. Using a service locator causes several problems with software in general, however, we're just going to focus on the problems that it creates for testability. In the Execute method of this class, first it uses the container to get the database dependency bound to the IDatabase interface, which is then used to get an invoice by the invoiceId. Next, the container is used to get the invoice writer, which is then used to write the invoice to the printer. Then the container is used to get the current session object. The session contains all of the objects and data associated with the current user's session. With the session object in hand, the code digs through the login object to get the logged-in user, which it then uses to get the current user name. The logged-in user's username is then recorded in the invoice's LastPrintedBy property. This is what a classic Law of Demeter violation looks like. There are so many method calls through various objects chained together that the code wouldn't even fit horizontally on the screen if it wasn't broken up into multiple lines. Finally, the command uses the container to get the database a second time and saves the changes to the database. Looking at the constructor of this class, the only dependency we know about is the container that's injected into the class. It's not obvious at all which classes this command depends on or needs to be set up for testing without scanning each of the dependencies inside of the class. In addition, the auto-mocking framework won't be able to automatically create and inject the classes dependencies because they are not explicitly defined in the constructor. Each mock dependency would need to be created by hand, and a manual binding would need to be specified in the container in order to be used inside of the command for testing. In addition, the long chain of

objects involved in getting the username from the session object will create addition problems for testing as well. First, it's quite likely that the session class is tightly coupled to the login class, which is likely tightly coupled to the user class. If this is the case, it will likely be very difficult, if not impossible, to test the command with these tightly coupled dependencies, as we'd need the application running in memory with a logged-in user just to populate the session objects. However, even if these classes were loosely coupled with interfaces using dependency injection via constructor injection, we would still need to create mock objects for each of these dependencies and wire them up in our test fixture set up method. Clearly, violating the Law of Demeter in these ways has created several difficulties for testing. So how do we fix these problems to make this class easier to test? First, let's refactor the constructor of this class. We'll remove the parameter to inject the IoC container and make our dependencies explicit again. We'll inject objects implementing an IDatabase, an IInvoiceWriter, and a new interface called IIdentityService, which will be implemented by a class specifically designed to get the logged-in users username from the application session data. In the body of the constructor, we'll just assign these values to private fields inside of the class. These changes that we've made so far will clean up things quite a bit in the Execute method of this class. In the Execute method, we'll get an invoice directly from the database dependency like we did before, we'll write the invoice to the printer by directly using the invoice writer dependency, then we'll use our newly created identity service to get the username of the logged-in user from the session data. Finally, we'll directly use the database dependency again and save the changes made to the invoice to the database. Not only do these changes improve the readability, understandability, and maintainability of the code, they have also made this class significantly easier to test. The test for this class should look pretty similar to what we've seen in the previous demos. We declare our subject under test in AutoMoqer, a test invoice, and some test data values, then we set up the invoice, the AutoMoqer, any dependencies, and create our subject under test using our AutoMoqer. In the setup method, we're also going to get a mock identity service from our AutoMoqer and set up its GetUserName method to return the test username that we specified above. Now in the first test method, we'll test to see that the invoice has been written to the invoice writer exactly one time. In our second test method, we'll verify that the invoices last printed by property should be set to the username of the logged-in user. To verify this, we'll execute the command, then assert that the invoice's LastPrintedBy property is equal to the test username we defined above. If the username was properly set the test will pass. Finally in the last test method, we'll verify that the changes to the invoice should be saved to the database. We'll execute the command, then assert that the save method has been called on the database exactly one time. If this is the case, the test will pass. If not, the test will fail. As we can see, our refactored code was pretty quick and easy to test

compared to the previous code that violated the Law of Demeter. Now, let's summarize what we've accomplished in this demo just to make sure that we understand all of the concepts.

Demo Summary

Now let's review what we've covered in this demo to make sure that we understand the concepts. First, our print invoice command had a dependency on our IoC container. It then used the container, combined with a set of interfaces, to resolve all of the dependencies for the class. This made the setup of our test more difficult. In addition, it prevented us from using our AutoMoqer to automatically create and resolve our mock dependencies. To fix these problems, we refactored the constructor of our print invoice command class to ask for the dependencies it needed directly. This allowed us to use our AutoMoqer to automatically create and inject our mock dependencies. In addition, it eliminated the need to create a container and set up all of the bindings for all of the dependencies by hand in the test fixture setup method. Second, the print invoice command class needed access to the username for the logged-in user. It had a dependency on the ISession interface, which was implemented by the session class, which exposed the login class, which provided access to the user class that we used to get the username of the logged-in user. This created a chain of dependencies that stretched from the session class, all the way to the user class. In addition, this forces the print command class to have to dig through these dependencies to get to the logged-in user's username. To test this command, we would've needed to create and set up a mock session, a mock login, and a mock user. This would have made both testing and test setup more difficult. So we refactored the solution by injecting a new interface that we called IIdentityService into the print invoice command class. This new interface directly exposed the logged-in user's username through the GetUsername method rather than forcing us to dig through its dependencies and find it on our own. Then at runtime, the identity service class provided us with the username from the logged-in user's session data. While testing, we used a mock identity service class to provide us with a fake username for our test cases. As a result, we simplified the setup process for our test, since we only needed to create a single mock object rather than a long chain of mock dependencies.

Summary

In this module, first we learned that we should follow the Law of Demeter to improve testability. The Law of Demeter states that our classes should only talk to our immediate neighbors, they should not talk to strangers. To do this, we should inject the dependencies that we need into the

constructor of our class, then we should use these dependencies directly rather than digging through their dependencies. In addition, we should inject only what is needed rather than injecting container dependencies. In our demo, we saw the problems that occurred when we tried to use our IoC container as a service locator. Finally, we learned that we should make our dependencies explicit. We do this by explicitly asking for the dependencies that our class needs in the constructor of the class. In the next module, we'll learn how to work with global state in ways that are easy to test.

Managing Application State

Introduction

Welcome back to Writing Testable Code. I'm Matthew Renze with Pluralsight, and in this module we'll learn how to manage application state in ways that make testing easier. As an overview of this module, first we'll learn about global state. That is, state that exists at the application level. Next, we'll learn about problems that global state creates for testing. Then, we'll learn about symptoms that may indicate that we have testability problems with global state. Finally, we'll learn how to refactor our code to solve these problems with testability. So let's get started.

Global State

Imagine that you're the passenger in a brand-new automobile that you and your friend have taken for a test drive. You've just finished letting your friend test drive the new car, and you're both ready to exit the vehicle. You press the button to unlock the passenger door, reach for the handle, and try to open your door, but it won't open. So, you press the unlock button a second time, reach for the handle, and the door opens just fine. You'd probably assume that the door lock was defective because it failed the first time. However, since it opened the second time, you're not really sure what to think. You come to find out that your friend was also reaching for the driver side automatic door-lock button the same time you did. However, he accidentally locked all the doors in the vehicle from the driver's side after you unlocked your door, but just a moment before you pulled the door handle. So while you thought you had unlocked your door, unbeknownst to you, your friend actually locked your door instead. In this case, both the driver and the passenger shared access to the same variable that is the lock for the passenger's door. Thus, the driver was able to unintentionally modify the state of the passenger's door lock without the passenger being

aware that it had happened. While this situation rarely happens with automobiles in the real world, it's a quite common occurrence when creating automated tests for software that are directly coupled to global state. Global state, also known as application state, is the set of variables that maintains the high-level state of a software application. These variables are usually implemented in one of two ways, depending upon the programming language. They may be contained in a single application-state object, for example the HTTPContext class in an ASP. NET application, or they may be implemented as global variables, which are variables that are accessible from anywhere within the application. Regardless of how global state is implemented, there's only ever one instance of each state variable in memory. So, if we have a variable called isloginenabled, there would only ever be one single copy of this variable in memory, accessible from anywhere within the application. As a result, this allows us to set a value in a method in a class A, and then retrieve that value within a second method in an entirely different class B. Storing and retrieving information from application state is often necessary in our applications. For example, we often need to access the currently logged-in user's permission in a desktop application. However, there are also reasons for using global state that are not so good. For example, developers often use global state as a quick and easy shortcut to share data between two or more classes. Either way, using global state can present significant problems in terms of testability, if it's not used properly. There are a few problems that global state creates for testing. Any time we access global state using either a static method or from a global variable, we're creating direct coupling to the global state. This type of coupling is particularly bad for a number of reasons. First, it makes setup for testing quite difficult. For each test we must explicitly create and populate the object that is storing the global state or set the global variable to an expected test value before we can run a test that accesses this variable. Second, since there can only be a single instance of a global state variable in memory, this means that we can no longer run our unit tests in parallel. If we set a global state variable to an expected test value for our first test, then start our second test, which changes the value before the first test completes, then our first test will appear to fail because it was expecting a different value. In the world of software development, this phenomenon is referred to as spooky action as a distance. Changes to a class in one part of the application should generally not affect classes in other parts of the application in unpredictable ways. In general, action at a distance is considered a software design anti-pattern. However, it's equally problematic for software testing. I've seen this problem occur on numerous software projects that I've consulted on over the years. The unit tests run perfectly most of the time, but every now and then a test will mysteriously fail for no apparent reason. Then, every time the test is run on its own in order to diagnose the problem, the problem magically disappears. The problem is that two tests are modifying each other's global state values

while testing in parallel. The first test sets a shared variable to the test value it's expecting, as a second test sets the shared variable to the value that it's expecting. Then the first test reads the second test value rather than the value it had previous set that it was expecting. However, when you run these tests one at a time, they are no longer changing each other's expected values, so the test passed as expected. If we are attempting to run an isolated test, we shouldn't expect anything outside of our isolated test to be able to modify the outcome. A test that is directly coupled to global state is no longer an isolated test. There are several symptoms of these problems that we should be on the lookout for. First, in programming languages that allow programmers to create global variables, if we see someone reading from, or writing to a global variable in their method, this is clearly a sign that they are directly coupling to global state. Second, more commonly, static method calls on classes with static fields may be a sign of direct coupling to global state, but only if these static methods are reading from or writing to a static field in the class. Once we are accessing data in static fields, we have direct coupling to global state. Third, use of the singleton design pattern from the book Design Patterns: Elements of Reusable Object-Oriented Software, is a symptom of coupling to global state. This design pattern is commonly referred to as the Gang of Four singleton, in reference to the four authors of the book. The Gang of Four singleton pattern is a way to construct an instance of an object that guarantees that only one instance can exist within an application. While this might seem like it's better, because it's an instance of an object rather than a static class, it's still creating global state, it's just not as obviously about doing so. We'll learn about the Gang of Four singleton in our upcoming demo. Finally, a symptom that occurs quite often is unit tests that randomly fail for no apparent reason when they're run in parallel, but always pass when you run them in isolation. When this occurs, the culprit is likely because a test is reading from global state, which is being modified by a second test while the first test is still running. The solution to these problems is to first keep state local if possible. You should always keep the scope of your variables as narrow as possible. If the data is only needed in a single method, then the scope should be that method. However, if it's needed in multiple methods within a class, then its scope should be the class. Keeping scope local where possible helps us to avoid spooky action at a distance in our tests. Second, if you have a static method that is part of a third party framework or library that is accessing global state, wrap this static method in a wrapper class that implements an interface and inject this wrapper class into the calling class. This way, you can substitute the wrapper class with a test double to decouple from the global state while testing. Please note that we already saw how to use a wrapper class in this way in the demo in module 2, so we won't show how to do this a second time in our upcoming demo. Finally, rather than using the Gang of Four singleton design pattern, replace it with a dependency injection singleton instead. A dependency injection

singleton is a singleton created by the IoC container. We do this by explicitly telling the IoC container which binding should return the same object for each request, rather than a new instance for each request. This way, the IoC container is controlling which classes behave as singletons versus which classes behave as normal object instances. Now let's see how to refactor code that is directly coupled to global state via Gang of Four singleton into code that is easily testable using a dependency injection singleton.

Demo

Now let's see a demo where we'll refactor code that is coupled to global state via Gang of Four singleton in order to make it easy to test. In our last demo, the office managers asked us to implement a feature that would allow them to retroactively discipline wasteful employees who print too many invoices. Unfortunately for the office managers, or fortunately for the employees, this strategy has completely failed to identify any wasteful employees. Paper costs, which the office managers are responsible for, are still at an all-time high. Despite the fact that the data we collected showed no identifiable patterns of wasteful behavior, the office managers are still convinced that the problem is probably just a handful of wasteful employees. So, they've asked us to restrict access to the printing of invoices to only office managers so that all employees must go through an office manager in order to print an invoice. They believe that this will help them to identify who keeps printing all these invoices, or at least deter them from printing unnecessary invoices if it requires an office manager to do so. Personally, we think this is a terrible idea, however, our cautionary advice is rejected once again, and our development team is required to implement the new feature as specified. So, let's see an example where this new feature has been implemented in a way that isn't easily testable, then we'll refactor it into a much more testable design. Let's start again with the PrintInvoiceCommand. Another developer on our team has been selected to implement the new restricted invoice printing feature. This developer recently read a book called Design Patterns and learned about the Gang of Four singleton pattern. They thought that this might be a good place to try out this new pattern, so they implemented the new security class as a Gang of Four singleton. As we can see, the command looks pretty much like it did in our previous demo, however, there are a few new lines of code. First, there is a call to a static method called GetInstance on the new security singleton class. The GetInstance method returns an instance of a security object. Because this is a singleton class, any calls to the GetInstance method will always return the same object, no matter when or where it was called within the application. This is the essential function of a singleton. A singleton is guaranteed to only ever return a single instance of an object. We'll dig through how the singleton class is implemented

shortly, after we finish walking through the changes in this class. With the security object in hand, the code checks to see if the user is an admin or not. If the user is not an admin, then the code throws a UserNotAuthorizedException. This will trigger a message to the user in the user interface, informing them that they need to contact an admin, that is an office manager, in order to print an invoice for the employee. So, all regular employees will need and office manager to log in and print the invoice for them. The rest of the code behaves pretty much as it did previously. However, it now uses the username from the security class to log the last person to print an invoice before saving the changes to the database. Next, let's take a look at the security class itself. This class implements the Gang of Four singleton design pattern. This means that the GetInstance method is guaranteed to return the same instance of the security object every time it is called from anywhere in the application. Please note that there are various ways to implement a Gang of Four singleton, and the way this class has been implemented is not representative of best practices for multi threading or security. I'm just using it as an example of how a traditional singleton might be implemented. First, the class contains a static field that holds a security object called instance. Because this field is static, there will only ever be a single instance of this variable in the application. We also have two private fields contained within this class, that is a string holding the username and a Boolean field indicating whether the logged in user is an admin or not. These two fields are not static, so they will be part of the security object instance created by the singleton class. The GetInstance method of this class is a static method. This means that we call the GetInstance method on the class and use it to retrieve an instance of the security object. Inside of the method, it checks first to see if the static instance field is null. If so, it populates the field with a newly created security object instance, then the method returns the object instance to the caller. As a result, this method is creating an object instance if it doesn't exist yet, then for every other call throughout the application's lifetime, it's returning that single object instance. In addition, the constructor of this class is made private so that no one can create their own instance of a security object. This design pattern guarantees that any class that calls the GetInstance method of this class, is going to get the exact same instance of the security object. The remaining methods are instance methods used by external classes, get the current user's username, and determine whether or not the user is an admin. The PrintInvoiceCommand is tightly coupled to the global state that this class contains. So how do we decouple it from this global state? The first thing we need to do is eliminate the Gang of Four singleton design pattern used in our security class. We'll replace this with a plain old class that generates normal object instances. We'll eliminate all of the code that produces a singleton. However, we'll keep the two instance fields for the userName and the isAdmin variable. In addition, we'll keep the three-instance methods SetUser, GetUserName, and IsAdmin. We'll also have this class implement an

interface called ISecurity that defines these three methods. Next, let's refactor our PrintInvoiceCommand so that it uses an object that implements the ISecurity interface, just like we've seen in our previous demos. We'll inject this object into the constructor and assign it to a private field. Then, in the Execute method of this class, we'll use the object to determine if the user is an admin or not, and we'll use it to get the current user's username as well. This should all seem pretty familiar by now. However, you might be asking yourself, how do we guarantee that the security object that gets injected into this class is always the same security object and not a new object instance each time? This is where our dependency injection framework comes into play. Modern dependency injection frameworks allow us to specify the scope of each binding in the IoC container. This is sometimes referred to as object lifetime management. By default, most IoC containers will create a new object instance for each object requested from the IoC container. However, we can override this behavior to tell the IoC container to return only a single object instance each time the object is requested. So rather than having the class itself control whether it's a singleton or not, we let the IoC container do this work instead. We refer to this type of singleton as a dependency injection singleton, as opposed to a Gang of Four singleton when we implement the singleton this way. I should probably note that there are typically other scopes that exist for IoC containers as well. For example, one object instance per HTTP request, one per thread, and more. However, this is a topic that is better suited for a course on dependency injection. So let's see how we'd refactor our Gang of Four singleton into a dependency injection singleton in our demo application. First, at the root of the application, we'll construct an IoC container. Next, we'll set up the default bindings by scanning the assembly and using the standard naming conventions to bind interfaces to their default implementations, then we'll override the default binding between the ISecurity interface and the security class that implements this interface. We'll do this by explicitly specifying a binding between the ISecurity interface and the security class. However, we'll specify that it should be done InSingletonScope. Essentially, we're telling the inject to only return a single object instance each time our code is asked to resolve a binding. Finally, at the root of our application, we'll use our IoC container to resolve the PrintInvoiceCommand, which will inject new instances of the database, security, and invoice writer objects. However, if we ask the IoC container to resolve this command a second time, the IoC container would inject a new database object and a new invoice writer object. However, it would inject the exact same instance of the security object each time. I should probably point out, to keep things simple for this demo, I'm neglecting important real-world aspects like code to log in a user and determine their security credentials. In addition, there would generally be multiple commands executed during the life of this application, rather than just a single command. However, this example should be simple enough to communicate the key

concepts of decoupling from global state for testing. Just to demonstrate that a dependency injection singleton does in fact return a single instance of an object for each request to the IoC container, let's take a look at a simple set of unit tests that verify this behavior. The first test will verify that that default binding behavior of Ninject's IoC container will return new instances of objects for each request. First, we create a container, next, we bind the ISecurity interface to the security class using the default scope, that is, the container will create a new instance for each request, then we'll ask the container to resolve the binding two different times. Finally, we'll verify that the first instance is not equal to the second instance. If the two instances are different, then the test will pass. The second test will verify that the singleton binding scope will return a single instance of an object for each request. First, we'll construct a new container, next we'll bind the ISecurity interface to the security class, however, this time we'll do so in SingletonScope, then we'll ask the container to resolve the binding twice. Finally, we'll verify that the first instance is the same as the second instance. If the two instances are the same, then the test will pass. As we can see, the IoC container returns either new instances or a single instance based upon the specified scope of the binding. Finally, before we wrap up this demo, let's quickly see how our PrintInvoiceCommand class is tested now that we've decoupled it from global state. This should all look pretty familiar to you by now, so we'll move through this code a bit more quickly. First, we set up our subject under test, our AutoMoqer, test invoice, and we'll create some test values. Next, we set up the invoice, the AutoMoqer, and any mock dependencies. Please note that we're setting up the security object to return true when IsAdmin is called, and our test username when GetUserName is called. In the first test, we check to see if the Execute method throws a UserNotAuthorizedException when the user is not an admin. First, we'll tell the mock security object to return false when IsAdmin is called, which overrides the previous setup where we told it to return true instead. Then, we'll assert that executing the command should throw an exception with a type of UserNotAuthorizedException. If the command throws this exception, then the test will pass. The second test verifies that the Execute method should print an invoice. We execute the command and verify that the invoice writer's print method has been called exactly once. The third test verifies that the Execute method should set the LastPrintedBy property of the invoice to the current user's username. We execute the command and verify that the invoice's LastPrintedBy property has been set to our test username. In the final test, we verify that execute should save changes to the database. We execute the command and verify that the mock database's save method has been called exactly once. I hope that you're beginning to get a feel for how code that was written with testability in mind is pretty easy to create unit tests for. When we know how to write testable code, many of the difficulties of unit testing and test-driven development all just

seem to disappear. Now let's review what we've accomplished in this demo to ensure that we understand all the concepts.

Demo Summary

In this demo, our print command class used the Gang of Four singleton to get an instance of a security object. The constructor of the security singleton was set to private so the only way to get a security object was to call GetInstance on the object. As a result, the singleton could guarantee that only one instance of the object would ever be created while the application is running in memory. However, this meant that the print invoice command was now directly coupled to global state. So we refactored the print invoice command so that we could inject an object that conformed to the ISecurity interface into the command. Then we created a singleton scope binding in our IoC container so that when we asked for a security object, the IoC container would only ever return a single instance of the security object. This allowed to security object to maintain it singletonness so that all classes reading from or writing to this class would be sharing the same application-wide data. This also allowed us to quickly inject a mock security object while we're testing to quickly test the PrintInvoiceCommand in isolation.

Summary

In this module, first we learned that we should avoid coupling to global state in our applications. Being tightly coupled to global state makes testing more difficult. We learned that we should keep state local when possible. In addition, if we must use global state that is part of a third-part framework or library, we should wrap the global state in a wrapper class and inject it into the calling class like we saw in the demo in module 2. Finally, we should replace Gang of Four singletons with dependency injection singletons instead. Using a dependency injection singleton provides all the same benefits of singletonness without the difficulty of testing due to being tightly coupled to global state. In the next module, we'll learn how adhering to the single responsibility principle improves testability by keeping our test cases small and simple.

Maintaining Single Responsibility

Introduction

Hello again, and welcome to our next module on Writing Testable Code. I'm Matthew Renze with Pluralsight, and in this module we'll learn how maintaining the single responsibility principle helps testability by keeping our tests small and simple. As an overview of this module, first we'll learn about the single responsibility principle and how it applies to writing testable code. Next, we'll learn about problems for testability when we violate this principle. Then, we'll learn how to identify symptoms of single responsibility principle violations. Finally, we'll learn how to refactor these violations to improve testability. So let's get started.

Single Responsibility Principle

Imagine that you're a mechanic at our fictitious automotive parts and service company. As a mechanic, your primary responsibility is the service and repair of automobiles. However, one day your boss tells you that they're short staffed. So, you're now also responsible for checking out customers at the cash register in addition to fixing cars. Oh, and by the way, they're also short a salesperson, so you'll also have to handle customers on the sales floor as well. You start your day by repairing an automobile, but you soon get interrupted by a customer needing to check out at the cash register. You finish the sales transaction and go back to work on your car, just in time to be interrupted by another customer with a sales question. While you're helping the customer on the sales floor, you get interrupted by another customer needing to check out, and this happens all day long, so much so that you do a terrible job in all three roles. In fact, you never even complete a single automobile repair. In the real world, it seems unwise to assign a single person with multiple conflicting roles and responsibilities. However, as software developers, we often ascribe multiple roles and responsibilities to a single class in our software. Much like our auto mechanic/cashier/salesperson, mixing multiple responsibilities into a single class creates similar problems both for software maintenance and software testability. The single responsibility principle is a general design principle in object-oriented software. In the words of Robert C. Martin, the creator of the single responsibility principle, a class should have only one reason to be changed. This principle states that we want to increase the cohesion between things that change for the same reason, and decrease the coupling between things that change for different reasons. This is often paraphrased as a class or a method should do one thing and do that one thing well. For example, we could make an automobile whose spark plugs are also fuel injectors. A single component would both inject fuel into the cylinder and ignite that fuel to create combustion. However, each time one of these hybrid components failed to generate a spark, we'd also have to replace the fuel injector functionality in addition to fixing the spark plug functionality. In addition,

if the component failed to inject fuel, we'd also have to replace its spark plug functionality in order to fix its fuel injector functionality. While the single responsibility principle is valuable for a wide variety of reasons, it's also quite valuable from a testability standpoint. When we violate the single responsibility principle, it creates a few problems for testability. First, classes with multiple responsibilities have more functionality than classes with a single responsibility. As a result, we're forced to test these multiple pieces of functionality within a single test harness for the respective subject under test. This means that we'll likely have more tests in each test harness. Having a large number of tests per test harness is in general more difficult to read, understand, navigate, and maintain, than a small number of tests. In addition, if we have multiple responsibilities in our subject under test, we now have more things to set up before we run our tests. This makes our setup logic more complex and difficult to follow. Finally, if we have multiple reasons to have to change a single class, the likelihood that we'll need to change that class also increases. This means that the likelihood that we'll have to change the test fixture and test cases increases. Ultimately, violating the single responsibility principle means that we'll be making frequent changes to a large number of tests with more complex test setup. This makes it more difficult to create and maintain the test cases of your application. There are several symptoms of this problem that we should keep an eye out for. First, if someone asks you to describe what a class does and your description contains the words and or the word or, it's likely that your class has multiple responsibilities. For example, a class that both prints an invoice and emails an invoice, likely has multiple responsibilities. Second, large class files and methods are usually a pretty good indicator that a class may have multiple responsibilities. Oftentimes large classes and test methods contain several units of functionality that could be refactored into smaller classes and methods. Third, classes that have many dependencies injected into them is often an indication that a class is trying to do too much. While there isn't a strict rule on how many dependencies is too many, in general, the more dependencies that are injected into a class, the greater the likelihood that it's responsible for more than one thing. Finally, if a class or method is changing frequently, it may be an indication that it has more than one reason to change. Classes that have multiple responsibilities generally change more frequently than classes with only a single responsibility. So how do you refactor a class that has violated the single responsibility principle to make it easier to test? First, you identify the independent responsibilities within the class. Look for multiple reasons that a class or method would need to change, multiple people in different roles in the company that would request changes to class, multiple tasks that are being accomplished in the same class or method, and multiple levels of abstraction contained in the same class or method. Second, try to label the responsibilities. When you find a responsibility hiding in another class, it is generally the case that you can give it a name that succinctly

describes what it is doing. Finally, decompose the original class into individual classes, each with a single responsibility. Use the label that you came up with in the previous step to assist you in naming these new classes. So now, let's take a look at a demo of code that is violating the single responsibility principle, and then refactor it to be more easily testable.

Demo

Now let's see a demo where we'll refactor code that has multiple responsibilities into code with a single responsibility that is much easier to test. The feature that we implemented to restrict the printing of invoices to only office managers was a complete disaster. The office managers were totally overwhelmed with requests from employees to print invoices for customers. As a result of this bottleneck, both employees and customers grew increasingly frustrated. In fact, several customers threatened to switch to our competitors because they were tired of waiting in line for their invoices to be printed. In addition, the office managers were completely baffled that they were unable to identify any wasteful employees as they suspected. Office managers were the only ones able to print invoices, but the use of paper went up even more, even after we implemented the new security feature. In fact, based on the new data, the office managers have started blaming one another for being the source of the problem. However, rather than point blame, we offered to do a bit of data analysis to try and get to the root of the problem. As a result of our quick data analysis, we identified the potential source of the increased invoice paper usage. We conclude that the vast majority of invoices that are being printed are in fact necessary and being printed for customers. Since the sales of parts and services are currently at an all time high, the number of invoices being printed for customers has also increased proportionally. Essentially, the problem that the office managers believed was occurring, didn't even appear to be a real problem at all. So as an experiment to help cut down on the amount of paper being used, we suggest adding a new feature to allow customers to choose to have invoices emailed to them rather than being printed on paper. While the office managers were reluctant to agree with our suggestion, they eventually give in and request that we implement the new feature as soon as possible. However, since they're still not convinced by the results of our data analysis, they ask us to keep the restricted printing feature in place just in case. Let's start by looking at a version of the command to print and email invoices that has multiple responsibilities and is difficult to test. Then we'll refactor this code to separate the responsibilities and make the class easier to test. Once again, another developer on our team beat us to implementing this new feature. They decided that it would be easiest to just tack the new email functionality onto the print invoice command. Essentially, if the employee clicks the email button and the customer has a valid email

address, then the invoice will be emailed to the customer. However, if the employee clicks the print button, then the invoice will be printed provided that they're an admin. As we can see, this class has been renamed to PrintOrEmailInvoiceCommand. In addition, this class now has four dependencies that are injected into the class via the constructor. A dependency on the database, a security singleton, a new invoice emailer, which will be used to email the invoice, and the existing invoice writer, which writes the invoice to the printer. As we mentioned earlier, the more dependencies that are injected into the class, the greater the likelihood that it has multiple responsibilities. The Execute method of this command takes two parameters, the invoiceId and a Boolean flag indicating whether we should email the invoice or print it instead. As a quick side note, flag variables like this that change the high-level behavior of a method, are often an indication that a method may have more than one responsibility. In the body of this method, first we get the invoice from the database that corresponds to the specified invoiceId. Next, if the user specified to email the invoice, then the code checks to see if the customer has an email address. If there's no email address, then we'll throw an EmailAddressIsBlankException and display an error to the user. If there is an email address, then we email the invoice using the invoice emailer class. Otherwise, if the user specified to print the invoice, then we execute the same logic to print the invoice that we did before. This command clearly has two responsibilities. It is responsible for emailing invoices, and it is responsible for printing invoices. Not only does this make the code itself more complex than necessary, but it will also make the test setup and test cases more complex as well. This might not be very obvious from the simple example that we've created, but the larger and more complex a class becomes, the more complex and difficult it also becomes to test. This difficulty compounds with each responsibility that a class has, so the increase in testing difficulty generally grows in nonlinear ways. Now let's refactor this class to eliminate the single responsibility principle violation. The solution to this problem is to identify the independent responsibilities for this class, label them, and then decompose the class into multiple classes, each with a single responsibility. In this case, our class has two responsibilities, that is emailing invoices and printing invoices, so the simple solution would be to create two separate classes, one for emailing, and another for printing. We'll create a new class to email invoices, which will be independent of all of the functionality to print invoices. The class will only have two dependencies, a dependency on the database and a dependency on the invoice emailer. We'll inject these dependencies into the class using interfaces and constructor injection. The Execute method of this class will only have a single parameter for the invoiceId. That is, we've eliminated the Boolean flag variable that indicated whether the invoice should be emailed or printed. In the body of this method, first we'll get the invoice from the database, then we check to see whether the user has an email address. If there's no email address, then we throw an exception so we can

display an error message to the user. Finally, if the customer does have an email address, then we use the invoice emailer to send them an invoice. The print command stays identical to what we saw in the previous demo, so we won't spend any time reviewing that code or its test cases. Finally, let's see the test to verify the functionality of the EmailInvoiceCommand. Even though this test fixture and test cases are completely new, you've seen the general pattern enough times now that it will probably seem pretty familiar. Let's quickly walk through the test fixture to see how we're verifying the functionality to email invoices. First, as we've seen in the past demos, we have a subject under test, an AutoMoqer, and a test invoice. We also have two test values that will be used throughout the test fixture. In the setup method, first we create our test invoice and populate the necessary fields. We create an AutoMoqer, then we use the AutoMoqer to set up any dependencies that will be used throughout the test cases. Finally, we use our AutoMoqer to create our subject under test, which automatically creates and injects the mock dependencies. Prior to refactoring this code to a single responsibility, half of the test values, test setup, and test cases would have been used to verify the printing functionality. So, when we're verifying the emailing functionality, these other test values, test setup, and test cases would've just gotten in the way. As a result of our refactoring, our test values, test setup, and test cases will be focused directly on the emailing functionality. This helps keep our test fixture clean and simple. In the first test method, we verified that when we call the execute method with an invoice without an email address, then the email address's blank exception is thrown. First, we set the email address to an empty string, which overrides the test value we set in the setup method, then we verify that when the execute method is called, it throws an EmailAddressIsBlankException. The second test method verifies that calling the execute method should email the invoice. This is referred to as the happy path test case for this command. That is, the path through the logic when the command works as expected. First we execute the command, then we verify that the email method on the IInvoiceEmailer is called with a specified invoice exactly one time. Because we've separated the multiple responsibilities of our previous command into two separate commands for emailing invoices and printing invoices, each of the respective tests is going to be very simple and focused. In addition, there will be roughly half as many test cases since we're only concerned with the emailing functionality in this test case and not the printing functionality. This keeps the number of test cases per test fixture low, which makes the test fixture easy to navigate, understand, and maintain. By now, this style of unit testing should feel pretty familiar to you. In addition to seeing how to write testable code, you've also had quite a bit of exposure to writing simple and effective unit tests. So now, let's review this demo to ensure that we understand what we accomplished.

Demo Summary

In this demo, we started with our PrintInvoiceCommand, which had a simple set of unit tests to verify its functionality. However, another developer added a second responsibility to this class, which made the class more difficult to test and caused the test fixture to increase in size and complexity. So, we refactored this class with multiple responsibilities into two classes, each with a single responsibility. This made the individual classes easier to test, and as a result, this simplified our test fixtures for each respective class. Our new feature that allows customers to receive invoices by email was a huge success. Not only did it create a significant and immediate reduction in paper usage, our experiment also provided further evidence to support our hypothesis that the increase in printing was not caused by employees being wasteful, but rather was the result of normal business operations. As a result, the office managers at our fictitious automotive parts and service company finally agree with the results of our data analysis and request that we remove the code which prevents regular employees from printing invoices. Our customers, fellow employees, and even the office managers are all very happy with the invoice printing and emailing features that our testable code now provides.

Summary

In this module, first we learned about the single responsibility principle and that we should attempt to limit our classes to a single responsibility for testability. Next, we learned that if we see a violation of the single responsibility principle, we should first attempt to identify the independent responsibilities, then we should try to give these responsibilities labels that properly describe each responsibility. Finally, we learned that we should decompose our independent responsibilities into individual classes, each with a single responsibility. In the next module, we'll learn about test-driven development and where to go for more information on writing testable code.

Next Steps

Introduction

Hello again, and welcome to our final module on Writing Testable Code. I'm Matthew Renze with Pluralsight, and in this module we'll learn about test-driven development and where to go next to learn more about writing testable code. As an overview of this module, first we'll start with a brief

introduction to test-driven development. Then, we'll discuss where to go next to learn more about writing testable code. Finally, we'll wrap things up for this module and for the course as a whole. So let's get started.

Test-driven Development

In this course, we learned how to write testable code. However, learning how to write testable code is just the first of three steps in the bigger picture of creating high-quality, fully-tested software. It's also important that we learn how to write effective unit tests. Oftentimes developers find it difficult to create fully-automated tests because they have not invested time in learning good testing practices. So learning how to write effective unit tests can make this process significantly easier. We saw a bit of information on how to write effective unit tests through the demos that we saw in this course. However, an in-depth coverage of this topic could easily fill an entire course. Finally, once you understand how to write testable code and effective unit tests for your code, it's important that you learn test-driven development. While we don't have time for an in-depth coverage of this topic, I feel that this topic is important enough to writing testable code that it at least deserves an introduction. I want to help you see the benefits that test-driven development has to offer and show you why it is so important for writing testable code. My hope is that this will get you interested in learning more about test-driven development. Also, please keep in mind that you don't need to learn these three skills in any specific order. They all feed into one another and reinforce each other as well. Test-driven development is a software practice where we create a failing unit test before we write any production code, then we use this unit test to drive the design of the code. Essentially, we write the code necessary to get the test to pass. We refer to this three-step process of test-driven development as red, green, and refactor. First, we start by creating a failing unit test for the simplest piece of functionality that we need to create. This is the red step in the TDD process. Next, we implement just enough production code to get that failing test to pass. This is the green step in the process. Then, we refactor our existing code. That is, we improve both the test and the production code to keep the quality high. This is the refactor step of the process. Finally, we repeat this cycle for each piece of functionality in order of increasing complexity in each method and class until the entire feature is complete. By using the test-driven development practice, we're creating a comprehensive suite of tests that cover all code paths of importance in our application. In addition, by using TDD, the design of each of these classes and methods is being driven by the testing process. This means that all classes and methods will be testable by virtue of the fact that the tests are driving the design of the design of the production code. In addition, TDD forces you to have to think about testability

first. As a result, you generally create classes that are not just testable, but easily testable. This is a reason why TDD is so important for writing testable code. If you're creating code using the TDD process, you're creating code that is guaranteed to be testable, and you increase the likelihood that you'll be creating code that's easily testable as well. In addition, this coincidentally makes our classes and methods more maintainable because of an interesting parallel in the physics of cohesion and coupling with both testability and maintainability. Essentially, by using the test-driven development process, not only are we creating testable code, but we're also creating more maintainable code. Finally, this comprehensive suite of tests created by the TDD process eliminates fear. Fear that making changes to our code will cause regression errors in our software. Eliminating this fear when we're changing or refactoring code increases the likelihood that we'll continuously refactor our code and keep the quality high. Essentially, there are significant benefits to using test-driven development. As a result, I highly recommend that you invest time to learn how to create code using TDD. Of all the software practices I've adopted in my career as a software developer, TDD was the skill that had the most impact on the quality of the code that I produced. In fact, test-driven development starts to become a bit addictive once you get over the hurdle of learning this new practice. Once you get hooked, you can't imagine writing code any other way. So, now let's learn where to go to learn more about this topic and the other topics we've discussed in this course.

Where to Go Next

Now let's discuss where to go next in order to learn more about the topics we discussed in this course. I have a great list of books, courses, and websites that will help you go beyond what we've covered in this course and take your skills to the next level. First, regarding recommended books, Mark Seemann's book, Dependency Injection in. NET, is a great place to start if you want to learn more about dependency injection and how to create loosely coupled and easily testable code. Next, Michael Feathers book, Working Effectively with Legacy Code, is a great source of information if you're working with an existing code base and you'd like to learn how to refactor it to make it more easily testable. Finally, if you'd like to learn more about test-driven development, Kent Beck, the inventor of TDD, created an excellent book called Test-Driven Development: By Example. For additional, more in-depth courses on Pluralsight, the following courses may be of interest to you. First, if you're new to dependency injection, Jeremy Clark's course, Dependency Injection On-Ramp, can get you up to speed with dependency injection quickly. Next, Misko Hevery discusses the psychology of testing large JavaScript applications in test-driven development in his course on Code Testability. Finally, to learn more about test-driven

development, the two-part course on Test-First Development by David Starr and Scott Allen is a great place to start. Third, I have a few recommended websites that you might want to check out as well. First Misko Hevery's blog on software testability is one of the best resources I've found for information on how to write testable code. In addition, Google's testing blog contains many great articles on testable code practices. Finally, my website contains additional information on writing testable code as well. I have articles, videos, presentations, open-source sample projects, and links to additional resources available. So be sure to check out all of these excellent sources of information to learn more.

Course Summary

Before we wrap things up for this course, feedback is very important to me. I use your feedback to improve each and every one of my courses, so please be sure to take a moment to rate this course, ask questions in the discussion board if there's something in this course that you didn't understand or would like me to clarify, leave comments to let me know what you liked about this course of if there's things that you think I could improve upon for future courses, and feel free to send me a tweet on Twitter if you like this course and you'd like to provide me with feedback in public as well. You can find me on Twitter @matthewrenze. And if you found this course valuable, please encourage your coworkers and peers to watch as well. It will be much easier to get everyone on board with writing testable code if everyone has an in-depth understanding of this topic. In this course, we learned how to write testable code. First, we learned that we should create seams in our code to make our code testable. Seams allow us to alter the behavior of the program without manually editing the code in that place. This means that we can replace runtime dependencies with mock dependencies for testing. Next, we learned that we should simplify the construction of our objects to improve testability. We do this by avoiding logic in our constructors, not mixing object construction and behaviors, and not mixing newables and injectables. Then, we learned how to work with our dependencies in ways that make testing easier. We do this by following the Law of Demeter, injecting only the dependencies that we need, and making our dependencies explicit. Next, we learned that we should decouple our code from global state. We do this by keeping state local where possible, wrapping static methods that access global state, and using dependency injection singletons instead of Gang of Four singletons. Then, we learned how maintaining the single responsibility principle benefits testability. We learned that we should identify individual responsibilities, give each of them a label, and then decompose the individual responsibilities into classes that each have a single responsibility. Finally, we learned that we should use test-driven development to help us write

testable code. We learned that the TDD process drives the design of testable classes and methods. In addition, it forces us to have to think about testability before we write our production code. Thank you for joining me for this course on Writing Testable Code. I hope that you've learned some valuable new skills that you'll be able to put to great use, and I hope to see you again in another Pluralsight course in the future.

Course author

Matthew Renze

Matthew is a data science consultant, author, and international public speaker. He has over 17 years of professional experience working with tech startups to Fortune 500 companies. He is a...

Course info

Level	Beginner
Rating	★★★★½ (228)
My rating	★★★★★
Duration	2h 2m
Released	5 May 2017

Share course