APPLIED ARCHITECTURE - ARCHITECTING THE DOMAIN LAYER

Presented by LearnVisualStudio.NET

Abstract

This document is a companion to the video series presented by Bob Tabor at LearnVisualStudio.NET. It is the second series in an ongoing learning path around architecture, principles, patterns and practices in the .NET world. This series takes on a Hands-On-Labs style to reinforce the ideas presented in the Application Architecture Fundamentals series, also available on LearnVisualStudio.NET.

Written by Bob Tabor

Contents

Introduction to Version 1.0.1 2
Version History 3
Chapter 1: Analyzing the Problem and Defining a Solution Architecture 4
Lesson 1 - Series Introduction 5
Lesson 2 - Discovering the Problem Domain 9
Lesson 3 - Analyzing the Problem Domain 14
Lesson 4 - Reviewing the User Stories 18
Lesson 5 - Designing a Solution Architecture 25
Chapter 2: Beginning the Architectural Spike 33
Lesson 6 - Preparing for the Architectural Spike 34
Lesson 7 - Creating the first Unit Test 43
Lesson 8 - Refactoring the first Unit Test 51
Lesson 9 - Creating a Stubbed Persistence Class and Data Transfer Object class 57
Lesson 10 - Using Dependency Injection to Break Layer Dependencies 67
Lesson 11 - Introducing Fakes and FakeItEasy to better isolate our domain 77
Lesson 12 - Introducing AutoMapper to Automatically Map Objects 90
Lesson 13 - Arrange, Act, Assert 96
Lesson 14 - Handling Exceptional Cases 104
Chapter 3: Implementing the First User Story 119
Lesson 15 - Adding a new Unit Test for Creating an Order 120
Lesson 16 - Creating the Exceptional Case for Creating an Order using an Expired Contract 133
Lesson 17 - Adding Order Items to the Order - Part 1 136
Lesson 18 - Getting Parts 139
Lesson 19 - Adding Order Items to the Order - Part 2 152
Lesson 20 - Adding Order Items to the Order - Part 3 161
Lesson 21 - Getting a List of Order Items 168

Introduction to Version 1.0.1

I realize that not everyone likes to watch video to learn new ideas. Heck, I have a confession to make … I don't like watching other people's screencast videos. Instead, I like to read. I simply don't like to listen or watch someone else. Ironic, I know.

This material is hard because it is very hands on. I'm changing something one minute, changing it back the next. I decided to that breaking down the steps in written form with copious amounts of screenshots might help to make sense of the difficult nature of the material to help those who want to follow along more closely.

Many have asked for a study guide to go with the videos. I never really could figure out what format that should take. Should it be extra material in addition to the video lectures / demonstrations? Should it have assignments?

Here's what I've decided to produce … a document that in many ways is a textual version of the videos. It should help those who have a disability, those who are not native English speakers, and those who want to slow down and listen and watch as well as read my words. I imagine you could follow along in both, pausing the video to catch up in the book and vice versa. Hopefully, it will help you study this material.

It's absolutely NOT a perfect 100% word-for-word transcription. There will be a few places where the code in the video doesn't match the code in this book. The fact of the matter is that it was more difficult to accomplish this than first anticipated. However, I already keep copious notes on what I want to say … why not polish them a bit and release them with the videos?

I relied heavily on Microsoft Word to help with spell checking, grammar and formatting. Reviewing this document, I can see that I may have missed some instances of both incorrect spelling and grammar usage. Perhaps there will be spots where the formatting is off as well.

And from a content / technical perspective, there may be some text that could have been explained better.

If you would be so kind, please send me your corrections for Version 1.1 or 2.0 or whatever comes next. I'll list you in the document as a Technical Reviewer. I'm not sure it will mean much on your resume, but hey … it couldn't hurt. Send your corrections to: bob@learnvisualstudio.net

Version History

1.0.1 – Includes 45 fixes from Nathan who said "Don't worry about listing me as a technical reviewer. Glad to help (Matthew 6:3)." But I included him anyway. ☺ I also found the Spell Check and Grammar feature in Word 2013 (which they hid in plain sight) and as a result I was able to more easily comb through the document to fix other small spelling and formatting problems.

Chapter 1: Analyzing the Problem and Defining a Solution Architecture

Lesson 1 - Series Introduction

Hello, and welcome to this series on applying the concepts you learned about in the Application Architecture Fundamentals series to a real project.

If you're watching this series PRIOR to the Application Architecture Fundamentals series on LearnVisualStudio.NET I would ask you to put this series on the back burner for a day or two and watch that series first.

The goal of this series is to demonstrate those ideas we learned about in the Application Architecture Fundamentals series in their proper context. As a result, it's different from any series I've created before.

For starters, what we'll do in this series is focus on the first steps of analysis and design of the solution, and will build the domain layer of an architectural spike. I've named this series:

Applied Architecture - Architecting the Domain Layer

... There will be at least three other series that will come afterward, namely:

- Applied Architecture - Architecting the Persistence Layer featuring the Entity Framework
- Applied Architecture - Architecting the Presentation Layer featuring ASP.NET MVC
- Applied Archtiecture - Architecting the Web Services Layer featuring the Web API

... And perhaps more. The fact of the matter is that there's too much material to cover in just a single series. It helps me to break things up a little.

Another way in which the Applied Architecture series will be different from everything I've done in the past is that each will combine many different concepts into a single series. There will be a number of occassions throughout this series when I mention an idea in passing or use a technology without explaining the fundamentals of that technology in any depth. I'll create a number of mini-series that will elaborate on those ideas.

For example, I will talk about a third-party tool called FakeItEasy later in this series. Instead of stopping, teaching you everything you need to know about FakeItEasy, then resuming this series, I'll just relegate that to its own mini-series. Now, granted, at the time I'm recording this, I've not yet recorded any of those mini-series, so I expect this to be a lengthy process. But by the end, we should have what many have asked for -- a large application that that is more substantial than what most books or videos cover, all related and applied to a single problem.

More than covering the API's or the tools and the tactics required to build this large project, the focus of these related series will be employing a thought process to solve a problem. If you miss that, you'll miss the point of what I want you to learn. The tools and technologies change often, but the thought process will be applicable years from now and as I have said in other places, you should invest your time in things that are permanent and borrow those things that are likely to change.

You'll also witness the process real developers go through to architect enterprise scale apps. What you'll see is a lot of angst and second guessing. I'll try to verbalize my inner dialog so you can tell what I'm thinking. There's a lot of uncertainty as I struggle to apply the principles I understand -- things we learned in the Application Architecture Fundamentals series -- to a new problem domain I'm not yet familiar with and am just coming to better understand.

And that brings me to talking about the next lesson ... In the next lesson I'll lay out a scenario of a fictitious company from the recesses of my imagination called ACME International and the application they have hired you and I and our consulting company to create.

So, once we hear about their business and a particular problem they've asked us to solve, we'll then begin to perform analysis on the requirements and design a solution architecture. We'll then perform analysis and talk about candidate architectures. We'll turn our focus then to the domain layer and begin to identify candidate classes that will comprise our domain layer, and we'll talk through the process of delegating responsibilities to each class.

When we're finished with all of these series, we'll merely have an architectural spike. In other words, I'm not going to create the entire application. Enterprise scale applications require months or years to build and one or more teams of developers working together. Clearly, that's not possible in a video series. However, I can say "we're going to tackle a single function of the application and drive the development all the way through from presentation to persistence." After we design and build the architectural spike, most of the hard work will be done ... after that, it will merely be a matter of writing more code in a similar style to flesh out the rest of the application.

So you and I? We're going to do a little role playing ... we'll play the part of architects on the project and work together to drive an architectural spike through the entire project from presentation through to the domain and then to the persistence layer and back. Throughout this and the rest of the Applied Architecture series, we're going to practice what I preached in the Application Architecture Fundamentals series, namely...

- We'll defer technology decisions as long as possible, which means we'll focus on the domain problem, the domain layer first. If we can get the domain layer correct, we know we can build an adequate user interface and decide on which database technology to use later.
- We'll use Unit Testing to drive our design decisions. We'll use Unit Testing as a design exercise. In other words, as we try to figure out what methods and properties each class should have, and what responsibilities to delegate to each class, we'll write the code that CONSUMES those classes and methods and properties FIRST, the way we would want to use them, THEN we'll go back an actually implement those classes and methods and properties in our production code. This has a secondary benefit ... we now have a collection of unit tests that can give us confidence to make changes later and rely on the unit tests to let us know when we've broken something and why. A third benefit is that we will stay organized throughout the entire development process and be able to move quickly as we progress. Remember: unit testing doesn't slow us down, it ultimately speeds us up. When we stay organized and when we're confident in our code, we can be more agile. We'll go more slowly at first, but we'll go faster later in the project and won't have to rely as much on debugging tools in Visual Studio to isolate the defects in our software.
- We'll keep the layers decoupled by adding interfaces and relying on dependency injection.
- We'll use fakes fakes to help isolate the domain layer during development so that we can focus on the domain layer by itself. This will ensure that our unit tests are testing the domain layer, and not code from the other layers like the persistence layer code.

There will be a lot of fun things to learn and exercise in this series. This is as close to real development as you're going to get in a video series.

And so, as a result, the code, the project, the solution will be messy at times. We'll finish a video and the code may not compile or unit tests may not pass.

Once we have something working one way, we'll rip it apart in the name of making it more decoupled, more flexible. Like I said, in the Application Architecture Series, writing software is more like molding clay than chiseling stone. We will mold and refactoring our code into shape.

I can't emphasize this enough ... while we will utilize specific .NET technologies like the Entity Framework and ASP.NET MVC and others, the key takeaway is the THOUGHT PROCESS I utilize. You should be discovering the thought process of separating concerns, removing coupling, applying patterns, starting with unit tests, starting with the domain, and so on. You should not be as concerned about the particulars of my code. I guarantee I will not be happy with the code I write ... I never am ... there's always something I could have done better, and I am always learning better ways to do things.

So, while the code I write is irrelevant, and will change over time as I learn more, I believe the thought process I employ is timeless ... it will always help you identify and build reliable software that focuses its energies on solving the heart of the complexity in the business problem first.

It will help you build reliable software that you can be confident in because it is supported by hundreds or thousands of tiny tests.

And finally this process will help you build software that will be resilient to change because it avoids coupling and consistently separates the major concerns of the application.

I suspect that by the end of this series of lessons, some of you may be thinking — "I could have accomplished the exact same thing you did in fewer lines of code! And you don't even have an entire working application!"

Believe you me, I could hack this application together fast if all I needed to do was knock something out quickly. However, this customer wants an application that can withstand the changes that will come for the next 15 or 20 years. Tell me, which database will they use 10 years from now? How will they consume the application 10 years from now? On their Windows computer? On their iPhone? Probably not ... neither of us know.

But I assure you ... if you start the application development process by choosing the ASP.NET Web Forms or ASP.NET MVC project template and if you then proceed to mix all the concerns -- the data access, the business rules, the presentation all in a single project, you've just limited the future of this application. You'll be doing the customer a grave dis-service.

Just like I said in the previous series — ADMITTEDLY -- not all problems require this degree of Separation of Concerns and the like. Some problems are smaller in nature, are departmental, are disposable, and so on. If the organization wants the application finished tomorrow and could care less about the impact of change and they just need need a stop gap measure that is only expected to be useful for a couple of years — that's their decision and the organization entrusted them to make those calls.

Furthermore, you may be creating this as a start-up company and so you may decide to create what some call a "Production Mockup". I take that to mean that the timeline is measured in weeks instead of months and you decide to "throw something against the wall to see if it sticks" rather than spend time creating something bulletproof. You may decide there's a certain level of quality you're willing to accept in exchange for something you can put in a customer's hands very quickly and "fail fast" if the solution is not marketable. That's your decision, as in all things, "it depends".

Depending on your needs, you may decide to keep it simpler. That's fine, but at least I want you to know how to meet complexity with an approach equal to the job. You can always remove layers, forego test driven development, worry about the domain rules -- if any -- later. But my goal is to help you know where to turn if you need a strategic approach to developing an enterprise system.

I think we have a moral obligation as experts and craftsmen to lay out the options to the stakeholders and management -- or if this is something we're creating for ourselves or a startup -- and be honest upfront about the ramifications of short term thinking. It's not right or wrong, good or bad, it just is a fact ... as the old programmer expression goes: "Good, cheap, fast ... pick any two."

What I'm going to demonstrate is for those scenarios where quality matters. If the stakeholders are strategic thinkers and want something that can be maintained and nurtured for years to get the most out of their software development expenditure, then you and I need to employ the mindset and techniques that will deliver the system they seek: a robust, high quality work product. And that is what I’ll be demonstrating in this series. At least, that's the plan. Wish me luck.

Lesson 2 - Discovering the Problem Domain

In this lesson, I want to describe a scenario that we'll utilize throughout this series as well as the other Applied Architecture series. The scenario will be fictional, but it is representative of the types of applications that I've encountered throughout my programming career. The problem domain is enterprise in nature — it describes a business critical process that spans departments and organizations. Our job is to create a system to support this process and replace it's predecessor — a 35 year old mainframe system. As you'll learn, the organization is currently limited by the system because the developers who first created it are retiring, there have been so many updates to the system that the architecture is barely comprehensible — there's no seeming organization — just one hack on top of another. As a result, it's been a thorn in the side of the organization for years and they've now received the funding to replace it. While the primary goal of the project is to support a fundamental change in their business processes around sales and support, our main stakeholder sees this as an opportunity to both re-write the application in such a way that it can be more thoughtfully maintained going forward, and even an opportunity to consolidate other subsidiaries into a more centralized system that will help the world-wide organization as a whole.

Now, your job in this lesson is to listen to the problem domain. I use the term "problem domain" to mean the scope of the project. What is it that we're trying to do here? What's the business problem we've been asked to help solve? So, as you listen (or read) about the problem domain I want you to be thinking about the architectural ramifications. Then in the next lesson, we'll revisit what we learn here and talk about how the goals of the initiative and the requirements that are expressed to us will impact a candidate solution architecture.

Our Client

Our client is ACME International, or more specifically, the North American subsidiary of ACME International's CNC Sales and Maintenance group. ACME International is a multi-national Robotics company based in Japan best known for their CNC machines that are used by large manufacturing customers however the success in this business has allowed them to expand dramatically into other related products and services. CNC stands for Computer Numerical Control, and it's used in large and small manufacturing for a variety of different purposes. In a nut shell, a CNC automates manufacturing processes, like drilling, tapping, cutting, laser cutting, carving, welding, grinding, lathing, and so on. A CNC usually consists of a computer terminal which engineers will program using a domain specific language to automate a series of tasks across one or more machines that are linked to the CNC. You can learn more about CNC machines on Wikipedia, however more details is not all that important for our purposes.

The Business Problem

When our client sells a CNC machine to one of their customers, they also sell a service contract. The service contract is important for the ongoing maintenance of the machine as well as those situations where the machine breaks down and requires repair.

So, when a CNC machine breaks down at a customer's factory, one of ACME's service technicians is dispatched to the factory to assess the situation. They evaluate the problem and ideally the customer has the required parts on hand. The customer is required by the terms of the contract to keep a ready supply of parts that are likely to require maintenance — motors, circuit boards — things that can burn out in the course of their use.

The customer agrees to an audit by an account manager to ensure they have a certain count of each required part on hand. When the parts are below a threshold, they should be re-supplied. However, during the course of a year, a given part may become depleted and due to oversight, the part may not be re-ordered.

So, supposing that the customer doesn't have the necessary parts on hand — the field service technician will contact ACME's parts desk in Chicago to order the required parts. The Chicago office is the centralized point of contact and will communicate with other warehouses to locate the part in one of ACME's six warehouses in North America. Ideally, the parts desk representative can find the required parts as close to the customer as possible. If no parts can be located, then then the parts desk representative will check the inventories of other customers to see if they have those parts on hand and temporarily "borrow" the part until it can be replaced from manufacturing in Japan. In a pinch, if the exact part cannot be located, an engineer at the Chicago office can suggest alternatives that can be modified to work until the replacement can be acquired. The absolute worst case scenario is that it would have to be ordered from Japan which could take days.

Unfortunately, the legacy systems in each of the warehouses are not centralized — each warehouse shares the same system, but the data is not centralized in a single database so for all practical purposes they might as well be six totally different systems. As a result, a lot of time is spent making phone calls from the Chicago office to the other parts desk representatives in each of the warehouses. Furthermore, at those warehouses, it's not always clear if a part is on premises — sometimes the parts desk representative must go into the warehouse and seek out the part to ensure that it is indeed available and not already promised to another customer.

This process can take hours and a lot of time is wasted waiting for employees to return phone messages.

ACME's legacy system was created over 35 years ago, but this past year ACME International's headquarters in Japan has funded an initiative to build a new system. The organization hopes for several outcomes:

First, they hope the new workflow and software system will allow for better response times for replacement part acquisition. A centralized data store should reduce all the phone calls and wait time between the warehouses. It will allow the field service technician to make a determination on the spot as to the closest location of the part geographically and even create the requisition — all performed directly from a phone or tablet device or any computer via a web interface or possibly a native client app.

Second, by enacting this new workflow, they hope to reduce the required workforce by eliminating the parts desk representative position

Third, due to the nature of ACME's CNC business as part of other company's manufacturing process, new government regulationsare on the horizen which will require ACME to collect more audit information about who inside of ACME is looking at and modifying replacement part data. Let's pretend that in light of the fact that many countries use ACME's CNC's in their manufacture of weapons or to create weaponized-grade uranium, the US wants audit trails. This type of functionality would be nearly impossible to add to the existing software system.

Goals of the New System

So, let's write these as a series of goals. The business goals of the new system:

(1) To support ACME's new Sales and Maintenance business process

(2) To create a solid code base that can be maintained and will be responsive to business opportunities

(3) To unify the processes and data across all ACME subsidiaries across the world

(4) To centralize the data to allow for better analysis, tracking and reporting across the entire supply chain

(5) To bake-in a high degree of auditing in anticipation of government regulations on data related to manufacturing of robotics machinery that could be used in the manufacture of weapons **

(** Now, truth be told, I just made this up ... I don't know of any such regulations, but I don't suppose it's all that far fetched in today's environment)

Ok, so these are the goals of the system from a business perspective. Some of these have obvious ramifications from a technology perspective and others are less obvious. Let's elaborate on these goals ... once we talk in specific terms, hopefully you can analyze these high level requirements and determine the ways in which they will impact the architecture of the system.

(1) To support ACME's new Sales and Maintenance business process

The first goal of the system is to support the new Sales and Maintenance business process.

The new business process itself is pretty straight forward. In a nutshell, a field service technician will locate the part and create an order. In this initial phase of the system, we'll only be concerned about the inventory management and the creation of orders. Furthermore, as we build the architectural spike, we'll merely be concerned with a sliver of that functionality. So, at first, we'll be biting off a very small amount of functionality.

In the next few lessons, you'll learn about User Stories, which are bite-sized descriptions of features we'll add into the system. We'll wind up creating dozens of User Stories, but the very first User Story will define the individual steps of the process required to locate a part and create an order ... as you'll come to learn, there are many business rules around re-order levels, customer service contracts, etc.

Supporting the new business process is the primary objective. Getting it right is very important to the business because it is supporting a fundamental change in how they respond to maintenance calls. As architects, how will this influence our thinking about the composition of the system?

(2) To create a solid code base that can be maintained and will be responsive to business opportunities

Next, the system must not handcuff the business, but rather allow it to grow and expand.

The previous system has been in place for 35 years ... this is an organization that seeks to optimize their technology investments by nursing systems along. While the previous system did not have a long-term view and was not resilient to change, they don't want to make this mistake again. The old system required lots of work arounds and hacks and limited the types of new products and services that could be offered.

In our conversations with the stakeholders, we hear the same message repeated over and over: the new system needs to stay flexible as new opportunities and technologies emerge. The hope is to keep a maintenance team engaged to support the application throughout it's life in the organization.

Here again, as architects, what decisions can we make to help achieve this goal?

(3) To unify the processes and data across all ACME subsidiaries across the world

Third, the system must unify the business. This requires a little explanation.

During our initial meetings with key sponsors and stakeholders, we learn that there's a vision for the system beyond the needs of their individual organization.

ACME International has acquired several different companies through the years that manufacture different types of industrial and manufacturing equipment. Despite the fact that there are a number of different product lines and each product line subsidiary is autonomous, they all happen to support a similar maintenance model to what we've already come to learn about ... ACME's account managers sell the equipment and the maintenance contracts. When problems occur, the ACME subsidiary dispatches field service agents to handle malfunctions. Then, the field service agent contacts a parts desk representative. Again, different subsidiaries, different product lines, but a similar business model.

There are over a dozen different ACME Sales, Maintenance and Parts subsidiaries in North America each specializing in a particular product line, and there are at least four other regions in the world where ACME International operates. This represents a fantastic opportunity for our client, the North American CNC business, to share the cost of developing this system with the other ACME subsidiaries to support their own similar needs. The hope is that it might even lead to a more integrated company, allowing for new sales channels given each subsidiaries existing customer base.

However, that will undoubtedly mean that we will need to take into account different regional differences ... at a minimum, we will not want to couple a specific language to our application. In addition to English, we will want to support German, Spanish, Mandarin, French, Russian and of course, Japanese.

Furthermore, with an eye to "selling" this application internally to the other subsidiaries, they may already have technology investments they can't abandon. Until now, each subsidiary operated as their own entity and made technology choices independent of each other. So, a polling of the different Subsidiaries found that we may need to support SQL Server, Oracle, and be prepared to support cloud-based architectures as well.

Ok, so in this case, the goal seems innocuous at first. However, upon further examination, we can see that this may in fact account for the majority of the anticpated change in the system. We know the user interface will change. We know the persistence technology will change. As a result, how does that impact our architectural approach?

(4) To centralize the data to allow for better analysis, tracking and reporting across the entire supply chain

The fourth goal for the new system is that it needs to be centralized. What that means from a technology perspective is yet to be determined, and while we don't want to make final decisions too early in the development process.

However, at a minimum, we do know that all replacement parts from all warehouse inventories will be stored in a single database. All parts from all customers will be stored there as well (although this is a less reliable source since it is not under our direct control).

Furthermore, while you and I are reluctant to make any final decisions about the presentation layer, that doesn't stop the client from thinking ahead. They are convinced that the system should be available via a web interface initially. Since each field service technician can make their own device or computer decisions they feel that by starting with a web interface this will allow all of their field service technicians to use the system immediately. In the future, however, they may consider other native clients, whether for Windows Desktop, Windows Store Apps, or possibly even iOS and Android. That decision has not been finalized yet.

Given these new goals (or I suppose you could call them non-functional requirements), how will that impact our architectural decisions?

(5) To bake-in a high degree of auditing

The final goal of the new system is to accommodate auditing. What the stakeholders describe is the need for each read and write to the database should be logged into some storage mechanism. Let's think about the ramifications of auditing for a moment. Typically, there are a lot of writes to the database ... each time the user takes some action in the system it will generate a record of data, including at a minimum the date and time, the user's identification, the action the user is taking, and the specific data they are acting on. Wow, that means a LOT of data will be generated. Furthermore, we don't want the generation and the storage of that information to negatively impact the performance of the system. We would like to "fire and forget", sending the data "over the wall" with some confidence that it won't be lost.

As architects we also know that a second characteristic of auditing data is that it is rarely read or reported on. You only need to review it when there's a problem or a review. We'll need more information about this later, but for now, at a high level, how will this impact our architectural decisions?

Assignment: Analyzing the Problem Domain and Identifying How it Impacts Our Architecture

As we learned about the problem domain through the goals that ACME has outlined, I asked a similar question each time: how does this goal impact our architectural decisions? By that I hope you understood me to be asking "how do we translate this goal into the design of the system? How does this goal impact the system we design?" We'll answer that in the next lesson.

Lesson 3 - Analyzing the Problem Domain

In the previous lesson, we learned about the ACME and their current pain points and plans for a new system. We discovered the five goals for their project and uncovered the rationale behind those goals.

In this lesson, we will talk about how those goals impact the decisions we'll be making as we begin to formulate a candidate architecture.

At this point, all of our discussions should share two traits:

(A) They are at a very high level, and...

(B) We understand them to NOT be set in stone.

Let's examine each of the goals from the previous lesson and talk about the way in which that will impact our architectural choices.

(1) To support ACME's new Sales and Maintenance business process

This suggests that the core of the system is the problem domain ... locating the parts and creating an order. As I said in the previous lesson, we'll flesh out the business rules around these business processes in a series of user stories which we'll talk about in the next lesson. However, we did identify the fact that there may be a lot of business rules related to who can create an order and how we find parts. What was clear to me was that getting those business rules right will ultimately spell the success or the failure of the project.

How does this impact our architectural decisions? Knowing that there are many business rules, and that this system will be supporting a brand new business process moves me closer to deciding that we should focus our efforts on allowing the domain layer to drive the development process. We want to get clear on the domain layer because it will be THE model of the business going forward.

(2) To create a solid code base that can be maintained and will be responsive to business opportunities

This goal suggests that ACME is concerned with code quality, maintenance, and having the confidence to make changes to the code base in the future. There will be a number of decisions that will need to be made regarding creating a maintenance team that is regimented in how it adds and deploys new features, versions the code base, creating a code review process and so on, there will need to be some architectural decisions made up front that will faciliate a healthy code base that can be modified in confidence.

So, how does this goal impact our architectural decisions? Knowing that the client wants to ensure maintainability, that pushes me towards utilizing unit tests. This doesn't necessarily mean that unit tests should drive the process, although I may prefer that for other reasons, I'm thinking that a solid architecture coupled with a set of unit tests could be handed over to the maintenance team and they would be able to quickly ascertain the purpose and intent of each method in each class in the system. It will also help as they go to make changes -- if they're not familiar with the code base, they don't know what they might be breaking by making changes. A good collection of unit tests should warn them immediately of the impact of their changes. This seems like an easy decision to me, personally.

(3) To unify the processes and data across all ACME subsidiaries across the world

If you'll recall, the idea here was to get the other subsidiaries to sign on to this new system. Not only would this help create a more integrated organization, but in the short term, it would help justify the expenditure as our stakeholders attempt to sell this application internally.

The main areas impacted by this were:

(1) Multiple language support, and...

(2) Existing persistence technology commitments

How does this impact our architectural decisions? Automatically I'm leaning towards a layered architecture. Even if all the layers are in a single project, and even if the layers are anemic, meaning that they don't do a whole lot ... they're there due to a formality, not because they necessarily add value, I'm still committed to at least thinking about every project in terms of layers. So, this goal just reinforces that I'll have at least three layers: a presentation layer, a domain layer and a persistence layer.

Furthermore, this goal reinforces what I already am inclined to design into the system ... namely, that the layers should be completely abstracted away from one another. In other words, I will seek to separate the concerns and I'll do so using the tools I have at my disposal like C# Interfaces and dependency injection.

Admittedly, there are ASP.NET specific ways to handle globalization on the persistence layer. However, it occurs to me that other parts of the system may be impacted by this need as well. For example, reporting (which is not in scope for this phase, but I have to have some foresight and do a little thinking ahead). That means I may want to pay special attention to the storage, the handling, and the presentation of date values, money values, and the like.

(4) To centralize the data to allow for better analysis, tracking and reporting across the entire supply chain

We learned that the stakeholders were already thinking in terms of a centralized data base and a web-based user interface with other potential native clients to be made available in the future for iOS, Android, Windows Phone 8, etc. It's not uncommon for clients to have strong opinions about the presentation layer. Everyone thinks they're a design expert. Oh well, as the expression goes: "Let the baby have his bottle". It may be a bit pejorative, but utlimately the design of the user interface is something the client will work with every day.

More importantly, how does a centralized data store and a web-based UI, with the possibility of native clients impact our archtiectural decisions?

Well, to me, it continues to reinforce the need for a layered approach. The layers themselves do not change, but I now begin to think about their deployment. This is shaping up to be physically deployed in a single location accessible via a web browser. While I don't want to necessarily make any decisions that will back me into a corner, I start to think about things like authentication and authorization over the public internet as opposed to utilizing Windows Authentication, whether I'll use ASP.NET MVC or Web Forms, will I rely on a more traditional interaction between the web server and the web client, or opt for an emerging set of Single Page Application JavaScript libraries. While I may not have included a Web Services layer in my initial thoughts of the solution, I begin to consider how that might be useful in supplying native client applications with the data they need to display and accept data. Perhaps it could be useful in a Single Page Application scenario as well.

(5) To bake-in a high degree of auditing

Finally, the auditing requirement is somewhat troublesome because of the potential performance impact it has on the system. Just think: each time we touch the database, we'll need extra code somewhere that formulates an audit message. The sheer amount of data we'll need to collect could be mind boggling. Also, I would like to "fire and forget", and not allow the writing of that audit record into a data store to block the application.

I suppose we could rely on features built into our data storage persistence tool, like SQL Server or Oracle. However, is that a truly cross platform solution? By delegating that responsibility to one of these tools, we'll be tying the business to that tool forever, or until the next re-write. I shudder at that possibility.

While I don't know all the answers, this is a big constraint on the system and definitely impacts my architectural decisions. At a minimum, I would want to keep this part of the system decoupled so I could change my mind in the future. I don't like any of my options, and that means I don't want to tie my solution to a single option for all eternity. Second, my own biases push me towards using Microsoft Azure, specifically Azure Queues for the "fire and forget" aspect that queues afford, and then use a series of Worker Processes to grab the audit messages off the queue, format them appropriately or validate them or whatever, then write them into a more appropriate permanent storage. It would probably be cheap if I could utilize Azure Blobs or Tables. I think I would want to avoid Azure SQL Server because I don't need the relational capabilities. Honestly, at this stage, I'm probably drilling down too much, but in the back of my mind, I want to think to myself why an Azure-based auditing solution wouldn't work, keeping it appropriately decoupled, of course.

What Did We Accomplish?

The purpose of this lesson was to talk through the impact of the goals, requirements and constraints of the project on our architectural thinking.

As you and I collaborate on the requirements in the next few lessons, we start drawing diagrams and having discussions about what the system should look like conceptually at a high level.

We agree to defer technology decisions as long as possible. It’s too early to talk about which presentation technology and which persistence technology we will implement for the first and subsequent versions of this app. Nonetheless, it has already begun to creep up in our thinking. I suspect we already know the answers to those questions in the backs of our minds, but let’s pretend for now like we don’t really care to think about them. That liberates us to focus on the part of the system that matters: we agree to focus on the problem domain first.

As the architects on this project, you and I need to start diving deeper into the detailed requirements of the system. That means we need to review any User Stories that ACME's business analysts already created -- or worst case scenario, you and I will need to create these ourselves. Remember from the previous lesson ... User Stories are simply agile descriptions of tasks users should be able to perform in our system. These will lead us to discovering much more detail about the problem domain. You'll learn how these flexible little tools will be leveraged throughout the planning, analysis and development process. While I don’t plan on talking about them in earnest in this series, I will be talking about Agile Software Development and Visual Studio Team Services in a dedicated mini-series so when it is available on LearnVisualStudio.NET you can see the huge role they play in our project.

For now, we’ll mainly use them to help drive the development of our architectural spike, which we’ll begin talking about in the next lesson.

Lesson 4 - Reviewing the User Stories

In this lesson you and I will begin the process of architecting a solution for ACME. We know the high level goals, we've begun to formulate a high level design in our minds -- or at least, we've considered how the goals of the project will impact our architectural decisions, and we have decided we will start with the domain layer. Many details will need to be fleshed out over the next few weeks and months, but in order to gain some clarity, we'll need to start diving into the details of a small sliver of the overall functionality required by the system. In the Application Architecture Fundamentals series we called that an "architectural spike".

The "architectural spike" serves several purposes. At this stage in the project, it helps us gain clarity as to what responsibilities to delegate to each of the layers. We'll use it to begin identifying key classes that will comprise our domain layer. And perhaps most importantly, it will help us identify issues that we need we need to solve or need more information about.

Admittedly, it feels a little early to start writing code ... but my goal is not necessarily to begin writing production worthy code but rather I think what this helps do is drive the process forward. When we realize we don't have the information that we need, we'll have to stop, find that information, then push forward. So, our work is to drive forward and find all the roadblocks and face them sooner than later.

Later on, the architectural spike will help the rest of the development team see what design decisions we've made, in other words, it will help them see the manner in which layers and objects collaborate to perform a task, they'll see examples of the naming conventions we've chosen, they'll be able to see examples of the technologies we've chosen -- not just the technologies we've used for Presentaton and Persistence layers, but also the logging technology, the isolation framework, the unit test technology and more.

Requirements and User Stories

The first thing we'll need prior to building the architectural spike is some requirements. There are entire careers built around defining formal requirements, so there's no way I can do that topic justice in this series. It's definitely not my area of expertise so you would be better off finding that information elsewhere. However, in my experience, anecdotally, I've seen one of several scenarios:

(1) No one has taken the time to write down the requirements for the new system. It's in the heads of the stakeholders and users.

(2) Some effort has been taken to define the requirements, but they are woefully inadequate.

... OR ...

(3) A LOT of effort has gone into creating the requirements, to the point where it's overwhelming. Hundreds of pages of documents and diagrams.

In my experience, I've seen scenario #1 and #2 much more often than I've seen #3. To keep things simple, we'll pretend that ACME has not spent a lot of time codifying their requirements.

Regardless of whether you have a great set of requirements or no formal requirements, we'll want to translate the requirements into a format that we can use as developers. In the past, Use Cases were a popular format for developer-centric requirements. More recently, a lighter-weight style, an agile style of developer-centric requirements has emerged called User Stories. I described them earlier as bite-size descriptions of a task that the system must perform. We'll use them for planning and estimating, but also to help drive the discovery process to learn more about the specifics of the system.

Once I have some initial user stories defined, my goal is to analyze the user stories and begin the process of identifying candidate classes — or rather, those key classes representing major concepts or entities in the solution that will collaborate to enable business processes. At that point, we'll be moving away from a high-level analysis to a more low-level detailed analysis.

Let's pretend that we work with the stakeholders to capture a few very basic user stories. We can and will add many more throughout the course of the project as new details emerge. But, let's look at the first, most basic user story we came up with:

1. As a field service technician, I want to locate the customer's service contract in the system so that I can create a new order for the customer.

A "user story" is just a simple description of a scenario for the purpose of scoping out the work required to build the functionality to support that scenario. It is the basis for further conversation and requirements gathering. Based on this most basic explanation, the project manager would ask developers how long they think it would take to build this functionality into the system.

A user story should be in the form:

> As a [role] I want to [action] so that [outcome].

So, as you'll learn in the series about Agile Development and Visual Studio Team Services (which will be a series that we'll create, user stories are combined together to determine what can and can't be accomplished during a given iteration. They also help the team understand the intent of the system and ask better questions about the details of the system.

They are short ... in fact, short enough to be written on an index card. This keeps their level of detail intentionally at a high level. Developers are encouraged to make assumptions about the scenario based on what they already know about the client, the existing system, their experience as software developers and so on. Ideally, more experienced developers will recognize those areas that should be easy and those areas that may represent a risk — where more information is required before they can provide an estimate of the time required to complete the user story in code.

User Stories in the context of a Project Planning Tool

Mainly, User Stories are used as a project estimating and planning tool. The project manager asks the developers to give their best estimates on how long it would take to build the functionality described in the user story. If the user story is unclear, the developers can discuss assumptions and raise questions to help get more clarity on the matter. This would prompt the team to work with the customer to get more detailed information about the scenario. They may get so much information that it would require more user stories to be added to those that are being considered for inclusion in the iteration. Again, more about that in another series.

Let’s fast forward in time AND let's assume for a moment that we're not architects ... I’ll come back to that in a moment ... instead for now, let’s pretend that we're two developers on the team who will be working on this project and we're past the architectural spike phase of the project. We're sitting in a room with the project manager. She has scheduled this to time play a round of “planning poker” ... each developer is given a deck of cards with numbers like:

http://en.wikipedia.org/wiki/Planning_poker

0

1/2

1

2

3

5

8

13

20

40

100 (A really long time)

? (I have no idea)

Coffee Cup (I need a break)

The project manager reads a user story, and she encourages you to pick a card that represents our estimate. We all show our cards at the exact same time. The developers who pick the lowest estimate and the highest estimate have a soap box — they have to explain the rationale for their decisions.

Even though we've fast forwarded in time, and we've already got some functionality of the application finished, there’s a lot unknown. However, in this case it just so happens that I have some experience with the project, the client and the functionality that's described by the user story. So, I make some broad assumptions about those parts I don't fully understand and I throw out a number ... 4 days. This would allow me to create the unit tests and production code as well. It also gives me some leeway for those parts I don't yet understand.

We're still early in the development process and I expect that this estimate will be off the mark, but it's a start and once I know more, and the rest of the team knows more about the problem domain and the state of the code, we can give better estimates as time goes on.

My estimate is on the upper end of all the estimates — most developers pick either 1 or 2 days. That’s fine ... I'm given the soap box, in other words, an opportunity to explain my rationale for the high-end estimate. I explain how it was implemented in the old system, I give a few examples of the issues and unknowns and while there is some discussion, the other developers tend to agree that this particular user story may uncover other details we're not already aware of. The project manager takes a few notes and continues on.

We continue playing this game for all the other user stories and the project manager has a pretty good feel for the amount of work ahead, or at least, a rough estimate. We'll talk about the project manager's use of these estimates when we talk in more detail about Agile Planning, Estimating and Development with Visual Studio Team Services.

User Stories in the context of an Architecture Spike

For now, I want to stay focused on this from an architectural perspective.

So allow me to switch gears back ... we're going back in time to the architectural spike. We're raclaiming our roles of the architect on the project, and we'll pretend the planning poker never happened.

Before we get the rest of the development team involved, we need to make some key architectural decisions around the composition of the layers, the major classes that will be delegated the responsibilities of the system, and so on. So, our aim as architects is to pick one or two of the most fundamental user stories off the deck and tackle them in an effort to create an architectural spike through the system.

Once we create our architectural spike, we can provide better estimates, we can decide on conventions, we get the initial project setup *and* ready for the other developers to join us. We identify new questions and issues that need to be resolved, and most importantly, we provide a good template / starting point for the other team members. We can say “Here’s the interaction between the layers ... you’ve been assigned this user story, so you can follow the pattern here ... do that like I did this.”

Fleshing out the Details of the User Story

User Stories start out as very high level descriptions of the functionality we need to add to the system. However, as we begin to tackle that functionality and encode it into the system, we'll need to discover more details and flesh out the user story we're currently working on. This requires the developer who will write code based on that user story has access to the client liason, stakeholders, end users or anyone else who may have more details. If there are too many details to complete in the current iteration, perhaps that will spawn new user stories that can be added to the stack of user stories and tackled in future iterations of the project.

So, let's pretend that through conversations with the client liaison, we learn more about this first user story.

As a field service technician, I would probably already know the contract numbers — we have those stored in a Customer Relationship Management tool and part of the dispatcher’s request will be an email with all the important details. The first step should probably be for me to copy and paste the Contract ID into the new system to find the customer’s service contract and kick off the ordering process.

There’s a series of checks that happen on the contract ... is it valid? If not, I would need to notify the account manager that the contract is expired. Ideally, that should never happen — but the customer may have been dragging their feet on renewing, so this will definitely be the impetus to drive a contract renewal.

For this first user story, we'll just make sure the contract expiry date is in the future. If not, the field service agent is not authorized to create an order on behalf of this customer and it becomes an account manager issue.

Assuming the contract is not expired, an order will be created at that moment with a status of "New".

As a field service technician, I'm going to type in a part number and click a button to search for that part. In different user stories, perhaps we'll be able to look up the machines owned by the customer and more easily identify the exact part number required. For now, we'll just type in the part number. It's encoded in a bar code on each part, so it should be relatively easy for the field service technician to find and type in manually.

The system will return a list of those parts and what region they're located in. In a different user story, they should be sorted by geo-location in relation to the customer. Also in a different user story the field service technician should be able to see nearby customers that also have this part in their personal inventories.

To create order items, I will first select a replacement part from a given region and then enter a quantity. As I add items to the order, I want to see the running subtotal. I want to be able to easily review the items already added to the order. Once finished adding items to the order, I want to set the status of the order to "Open" and provide the customer with an Order Confirmation Number.

Assignment - Designing the Layers and Candidate Classes

So per our agreement in the previous lesson, we'll be allowing the domain layer to drive the development of the application. This will allow us to focus on the most important parts ... getting the business rules around the ordering process correct. We can worry about the user interface and the database access code later.

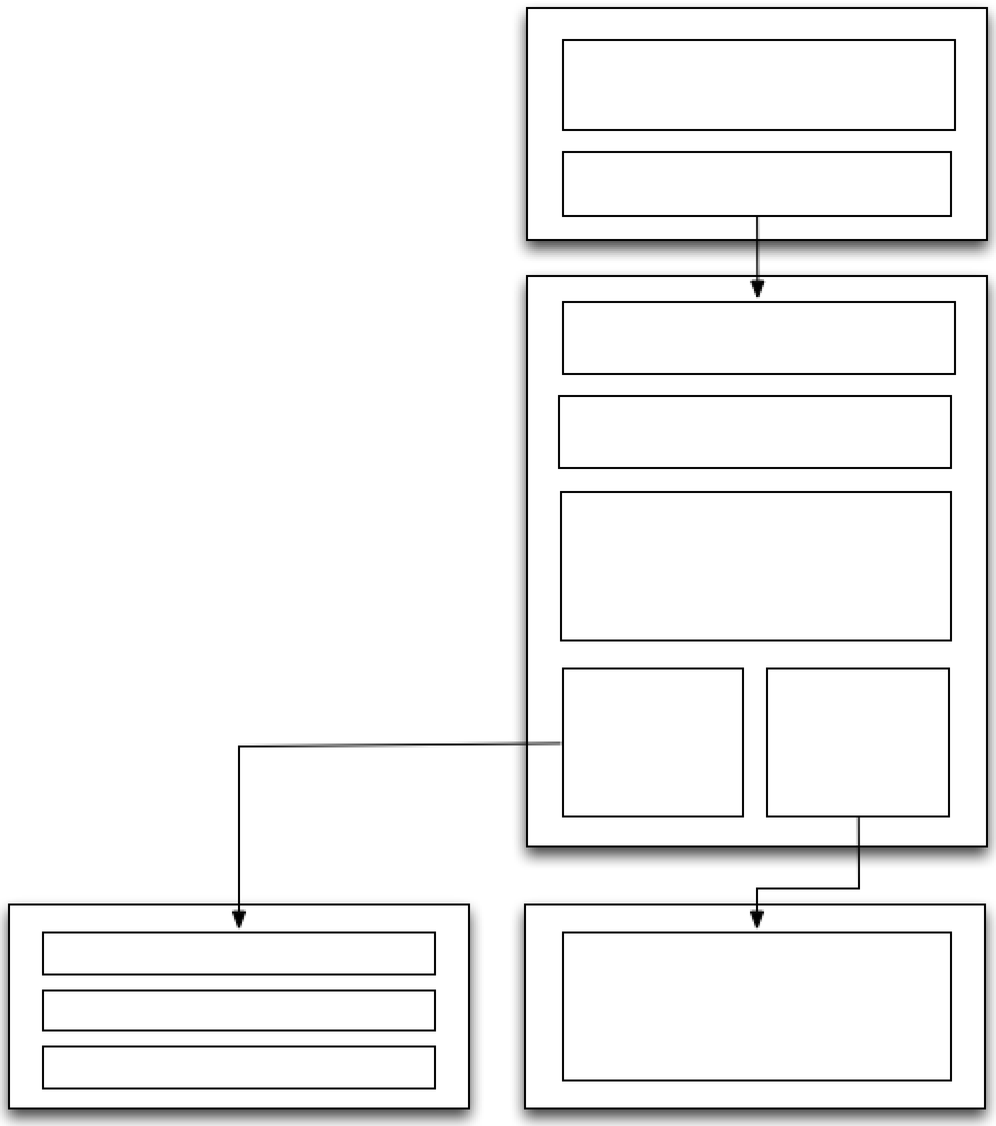
For now, getting the domain right is our first priority because it is a model of the business.

So, when I think of the domain, I think of the major classes and their interactions required to accomplish the user story as I understand it right now.

Let me get you involved at this moment. We're just about at the end of this lesson, so answer the following question ... based on your current understanding of the problem domain...

(1) Create a high level architectural diagram for a web deployment scenario ... map the layers in our application to a physical architecture, or in other words, how will we distribute the different layers on to physically different systems. Just use big boxes ... a big outer box to indicate a physcal tier, and by that I mean a machine or a farm of machines. Then draw boxes inside that represent the various conceptual layers of our architecture -- for the most part, I'm talking about the layers that we learned about in the Application Architecture Fundamentals series.

The following is an example of the diagram I want you to draw ... I've left out the words that each of these boxes represent ... that's your job.



Note: This diagram is just an example. It is not the solution or a hint about this first exercise. This is NOT a fill-in-the-blanks exercise.

Remember: DON'T FORGET THE AUDITING REQUIREMENT as detailed in the previous lesson.

(2) Create a second high level architectural diagram for a native device scenario ... this will look a lot like the first diagram, but here's a hint ... the presentation layer will look different, and at the very least you'll want to add another layer to enable many native devices ... phones, tablets, Windows Store apps, etc.

(3) Moving on to the domain layer, what major classes would you envision being a part of our solution? Can you draw a simple UML Class diagram showing the classes, some of the properties you'll create, and their relationships?

Note: You should remember how to create UML Class Diagrams from the Absolute Beginner's C# series on LearnVisualStudio.NET.

Please do take the time to perform this exercise. It doesn't matter if you get it wrong — I'm not even sure at this point that there's a wrong answer! Just look over both the User Story as well as the additional information I've captured from ACME's client liaison and let's start identifying some candidate classes, attributes and their relationships.

Lesson 5 - Designing a Solution Architecture

Ok, hopefully you took the time to draw a diagram I assigned you to create in the previous lesson and give this some thought. I'll explain how I would go about it.

Let me start off by saying that you may have thought I was contradicting myself by asking you on the one hand to ignore technology decisions, and on the other hand go ahead and draw a diagram that includes technology decisions. There's an expression in the Bible that says with regard to giving that you should not let your "left hand know what your right hand is doing". There's a sense in which that is true with regards to how I would approach architecture. On the one hand, I constantly say "I'm not ready to make implementation decisions yet". Which is true, because I don't want it to unduly impact my design decisions in the domain layer. However, there's another sense in which I want to give some thought to how this application will ultimately be deployed.

The goal of this exercise of not thinking about technology and deployment is an effort to think about the domain layer in isolation. That's all. As technology people, it's impossible to not at least think a little about how it will be deployed. This is just part of examining a problem from all angles.

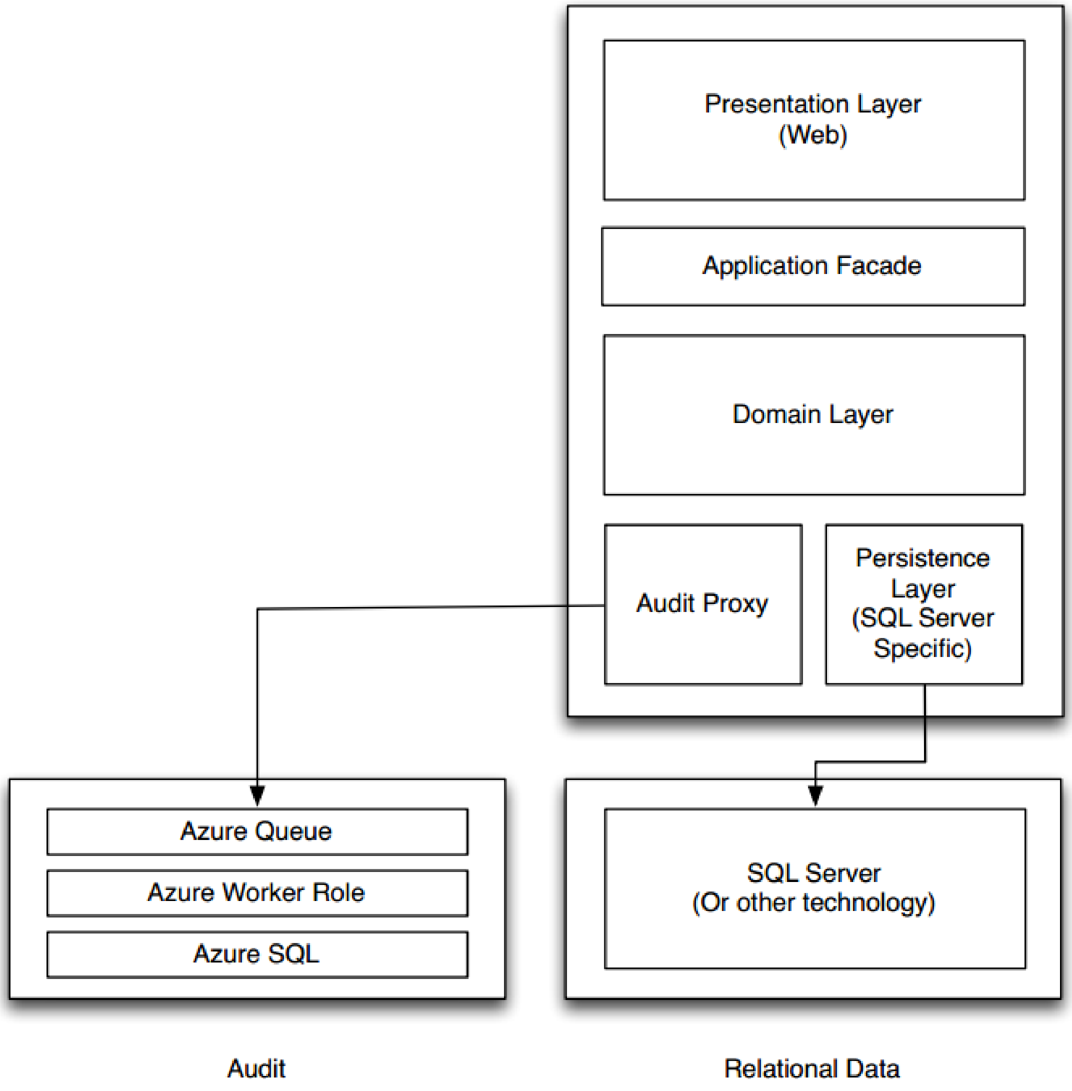
So, with that in mind, let's see what you came up with.

Considering Physical Web Deployment

The first assignment was:

(1) Create a high level architectural diagram for a web deployment scenario ... map the layers in our application to a physical architecture, or in other words, how will we distribute the different layers on to physically different systems. Just use big boxes ... a big outer box to indicate a physcal tier, and by that I mean a machine or a farm of machines. Then draw boxes inside that represent the various conceptual layers of our architecture -- for the most part, I'm talking about the layers that we learned about in the Application Architecture Fundamentals series.

This is what I came up with:



So, as you can see, when I began to diagram the web deployment scenario, I put many of the layers in the same physical tier. The main exception is the persistence layer which typically will live in its own physical tier, for example, a clustered SQL Server farm dedicated to this supporting this project complete with roll-over redundancy and so on.

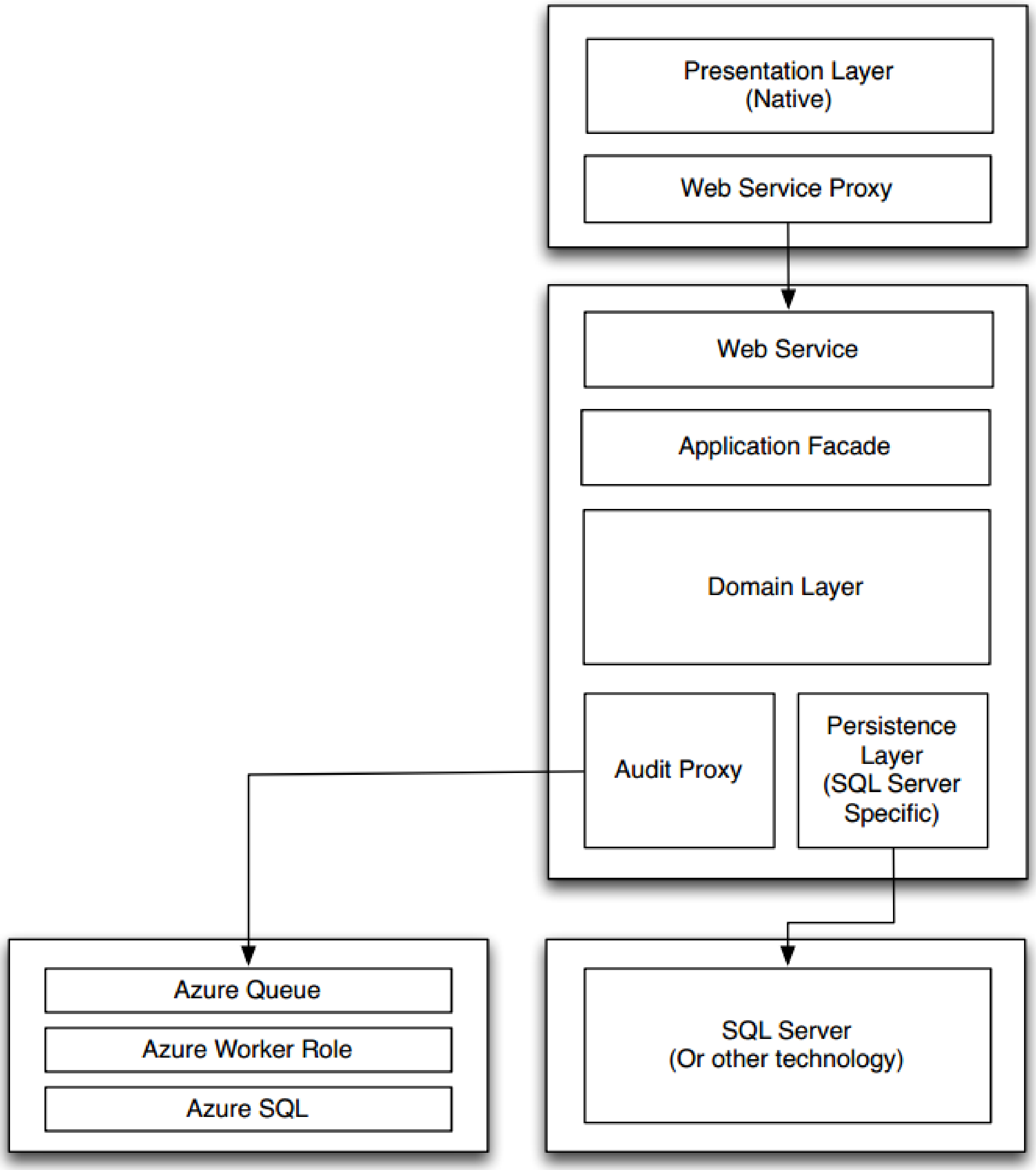
The physical tier that contains the layers ... notice, I have them all deployed to the same box or server farm. I suppose I could separate some of these layers to their own physical tiers, but I don't see the point. We don't anticipate any processor intensive operations like image processing or the like and I certainly don't want to distribute my layers and add all the network latency. No, it's better that these layers live on the same physical tier.

You may notice the Audit Proxy ... a proxy is just a go between ... this allows us to replace implementations of the auditing functionality in the future. It will allow us to decouple our domain layer and persistence layer from the auditing functionality. Obviously, I was already leaning towards Azure, and I've just codified that in this diagram.

Considering a Native Client Deployment with a Web Services Layer

(2) Create a second high level architectural diagram for a native device scenario ... this will look a lot like the first diagram, but here's a hint ... the presentation layer will look different, and at the very least you'll want to add another layer to enable many native devices ... phones, tablets, Windows Store apps, etc.

Here's what I created:



So, the only thing I added here was a Web Services layer. Also, the top-most box represents the Presentation Layer, a native client app. The Web Services Layer and its corresponding Proxy on the client have the shared responsibilities of serializing and deserializing the data transfer objects that will be sent back and forther between the Presentation Layer and the Application Facade.

UML Class Diagram Assignment

The third assignment was:

(3) Moving on to the domain layer, what major classes would you envision being a part of our solution? Can you draw a simple UML Class diagram showing the classes, some of the properties you'll create, and their relationships?

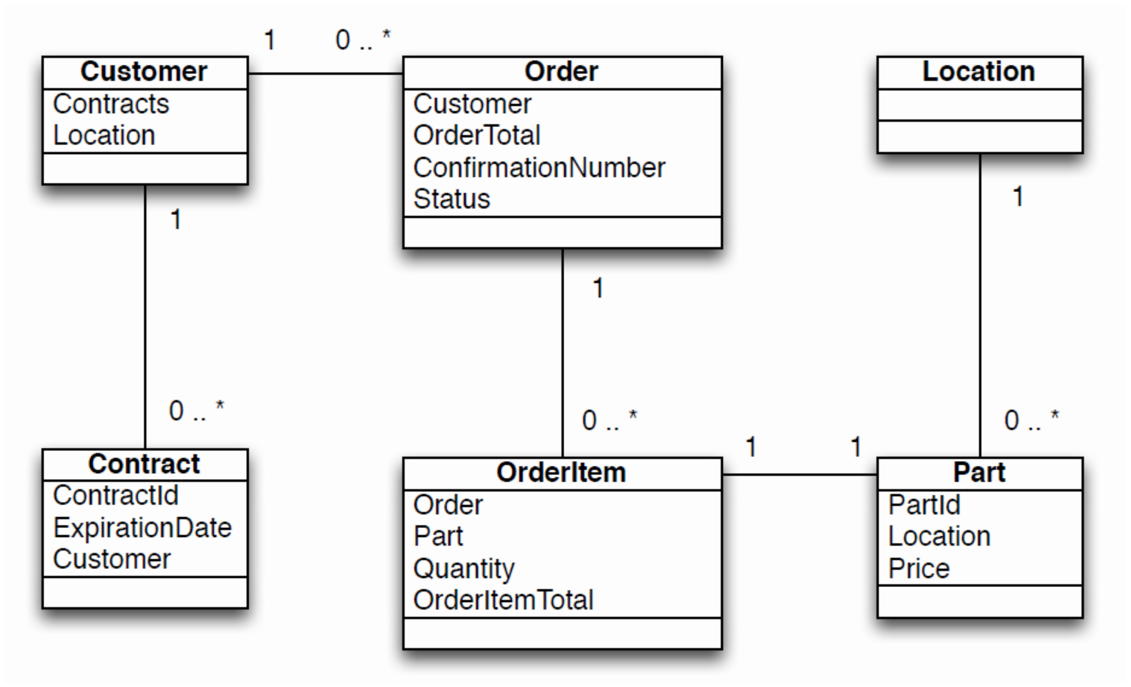
So, I begin with envisioning the major classes:

- Customer
- Contract
- Order
- Order Item
- Part
- Location (This might factor out into warehouse vs. Customer factory ... not sure about this part yet ... We'll cross that bridge later once we work on that user story)

As you can see, I start decomposing the user story picking out the major nouns from the user story itself as well as the additional clarification I've gained from talking with the client liaison.

These nouns become candidate classes. The more I think about these nouns, the more I come to a better understanding of whether they are a class unto themselves, or just a property of an existing class. The Contract ID ... is that a class by itself? Or it is a property of another class? What about Order Item ... is that a property of another class? Or does it deserve it's own class with it's own properties.

I begin to commit them to a diagram and make design decisions. Here's what I came up with.



While I was designing the classes, their properties and relationships, I had to make several key decisions around what should be a class and what should be an attribute of a class.

For example, the Contract ID feels like it is simply an attribute of a Contract. A Contract feels like a class unto itself because I can imagine it has all sorts of attributes we'll need to know, not the least of which is the Expiry Date ... And Contract seems like it should be related to a Customer ... In fact, the more I think about it, one customer could have many contracts throughout the lifetime of their relationship to the ACME organization. So, in a way, we can use the Contract ID as a way to identify a specific customer. So, I'm thinking we'll have a Customer class and it will have a collection of related Contracts.

The Order Item feels like it is an entity unto itself. An order item is like a line item in an order ... each line item will reference a Part at a specific Location. We'll also need some way of ascertaining the price of the Part. I'm going to assume that the part will cost the same no matter which Location is selling it to us. However, I also assume that the Part's price may change over time. I could be wrong, and I would want to validate that with a subject matter expert — the client liaison at ACME. So an Order Item should also include the Part price at a particular moment in time. The line item will have it's own line total ... the quantity times the price. An Order will own a collection of line items. When you add up all the line totals, this will create the sub total for the entire order.

Does that make sense? Did you come up with the same sorts of design decisions?

Now, for me, this is an obvious organization of classes and their properties. It may be because I've done this so many times that I don't have to think about it any more. It's difficult for me to break down the micro-steps, and frankly, while some have suggested a series of steps you can take to decompose a problem and come up with the right assignment of classes, properties and relationships, I generally just use logic — is this an object? Or just an attribute of an object?

Most of the time I get it right, but occasionally, once new information comes in, I may find that I was off the mark a bit ... regardless, I try to stay flexible. These are candidates, not my final decisions. It's like a foggy morning ... I begin to make out big shapes through the fog ... as I get closer or focus more, I can make out more details. As I become more familiar with a concept that I've identified as a candidate class I can see more detail, and it might change my perception of what it is. I'm doing that here, too. I suspect as I get closer to these candidates, my perception will change slightly but overall I feel pretty good about what I've got so far.

Design Decision: Identifiers in the System

One thing that I may need some more clarification on ... ID's. I'll assume that they're not going to change appreciably.

For now, based on what I know about the client's existing system, I'm going to assume that the ID's for each entity -- the contract, customer, the order, the parts -- are either GUIDs or some fabricated number that has some meaning encoded into it.

For example, here's a Contract ID:

ALLIED-13-MIL-07122005

... It may be a combination of company name, geographical region, classification, a date/time stamp of when the customer's record was originally created and other factors. This may be a relic from a legacy mainframe system -- those crafty developers were always trying to slip data into identifiers in some cryptic way.

I'll get clarification of that later. For testing, I'll simply use strings. When I need to generate an ID for some purpose, I'll create a GUID and convert it into a string. This will make it unique, and I will be sure to isolate the ID generation code into it's own replaceable module so I can revisit it in the future once I know more about the qualities and makeup of the ID's.

Preparing for an Architectural Spike

So, we've made some preliminary design decisions, both high and low level. But, where should I start? I'm looking at essentially a blank page. I know I'll have a few classes. I know a few properties I'll need for each class. But that's it.

Well, we're going to take a test driven design approach. That means, I'm going to let unit tests drive the process of designing the classes, methods, properties and their interactions.

So, my first order of business is to create an order for an existing customer. I need to get a customer object and create an order for that customer. At first, the order will be empty -- it will have no Order Items associated with it, it will have a status property set to "New" ...

So, the most basic thing I could do to kick this off is to:

(1) Given a valid Contract ID, retrieve a Contract.

(2) I'll want to determine whether the Contract's Expiration Date is valid. If not, notify the User.

(3) Orders are related to Customers, so grab the Customer for the current Contract.

(4) Create a new Order for the Customer. The Order should be initialized to a valid state (I.e., Status = "New".)

(5) Type in a Part ID. Show any matches, including Quantity on Hand.

(6) The User will select one Part-Location combination, will select a Quantity, and will create a Order Item.

(7) The Order's contents will be displayed on screen and allow the User to search for another Part.

(8) The User can choose to finish ordering. This will set the Order's status to "Open".

Now, honestly? I usually don't spell out what I'm going to do quite in this much detail. I may have an internal dialog as I read the user stories and my existing code and think to myself "What needs to happen next?" But this is how I would decompose the problem, identify the classes I'll need, delegate responsibilities to the classes, and so on — if not out loud, then to myself.

And so, given this list of steps in the process to enable the user story, my next task is to create unit tests that force me to write the production code to make these scenarios work.

Let's take the very first step:

(1) Given a valid Contract ID, retrieve a Contract.

I would further break that down in a unit test like so:

(1a) create a new instance of a Contract class

(1b) I would call a GetById method, passing in a Contract ID that the user types in.

(1c) I would expect that the Contract's properties — it's Expiry Date — to be populated in the Contract.

(1d) I would call a Validate() method to perform a check on the Expiry Date and determine if the Contract is in good standing or not.

(1e) If not, I would probably raise an exception to whoever called the Validate() method.

(1f) If the Contract isn't in the database, perhaps the user typed it in incorrectly. I'll raise an exception to whoever called the FindById() method.

So, right there ... I've got a few individual steps all wrapped up in that single task. I would probably create at least 5 unit tests to make sure each of these steps worked well. These unit tests will force me to create Classes and methods on those classes. I anticipate that it will force me to also create custom exceptions. But the payoff is that, from this point on, I'll have the confidence that finding a Contract works and if it ever does break, I'll have a pretty good idea of what broke it.

But as I begin to further decompose this User Story into UnitTests, this is when things start getting messy, especially at first because everything is in a gelatinous state trying to congeal. We'll follow the mantra "Red, Green, Refactor" and we'll shape and mold the code into a more congealed state.

I'll be introducing new unit tests and new production code in the domain layer. I'll be introducing new layers and interfaces and fakes, each time adding more uncertainty, then shaping and working with it until it all looks good again.

And while I do this all of those topics we talked about in the Application Architecture Fundamentals should be coming back to you over the next several lessons.

The next step is to get the Solution set up, create the Domain Layer project and the Unit Test for the Domain Layer. We'll do that in the next lesson.

Chapter 2: Beginning the Architectural Spike

Lesson 6 - Preparing for the Architectural Spike

In this lesson, I'll begin the process of setting up my solution so I can get started building the architectural spike.

I'll add two projects:

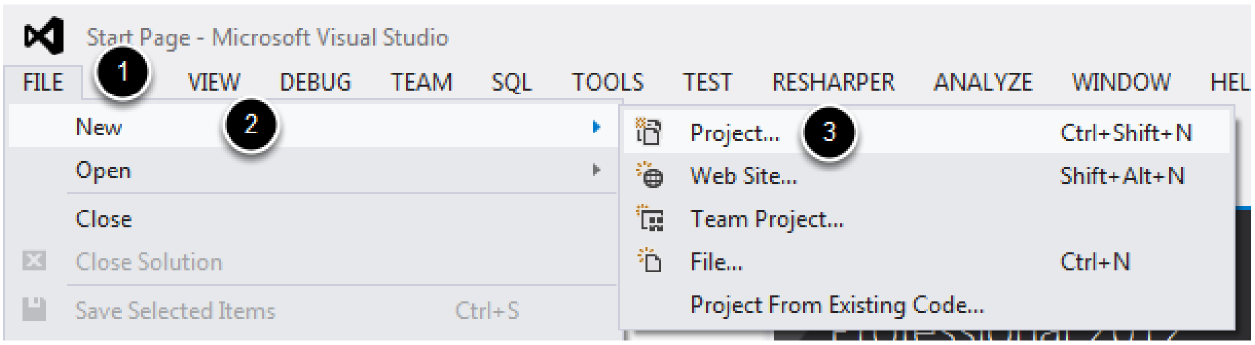
- a Class Library project for my Domain Layer classes
- a Unit Test project where I'll test my domain layer class methods.

Throughout this project, I'll be adding more projects to enable more layers ... but this will get us started.

I'm going to be working with Visual Studio Professional 2012, you can use any edition of Visual Studio as long as it allows you to create Unit Tests. I think you can follow along in Visual Studio 2012 Express for Web, but I'm not entirely sure. You might be better off downloading the 90 day trial of Visual Studio 2012 Professional or greater to make sure you can follow along.

Also, I'm pretty sure you'll be able to follow along in Visual Studio 2010 Professional, but I don't think you can use the 2010 versions of the Express edition -- if that describes you, try out the 2012 Express for Web -- I think you should be able to follow along. Again, worst case -- just download and install the trial version of Professional 2012.

1. Create a Blank Solution called ACME.Maintenance



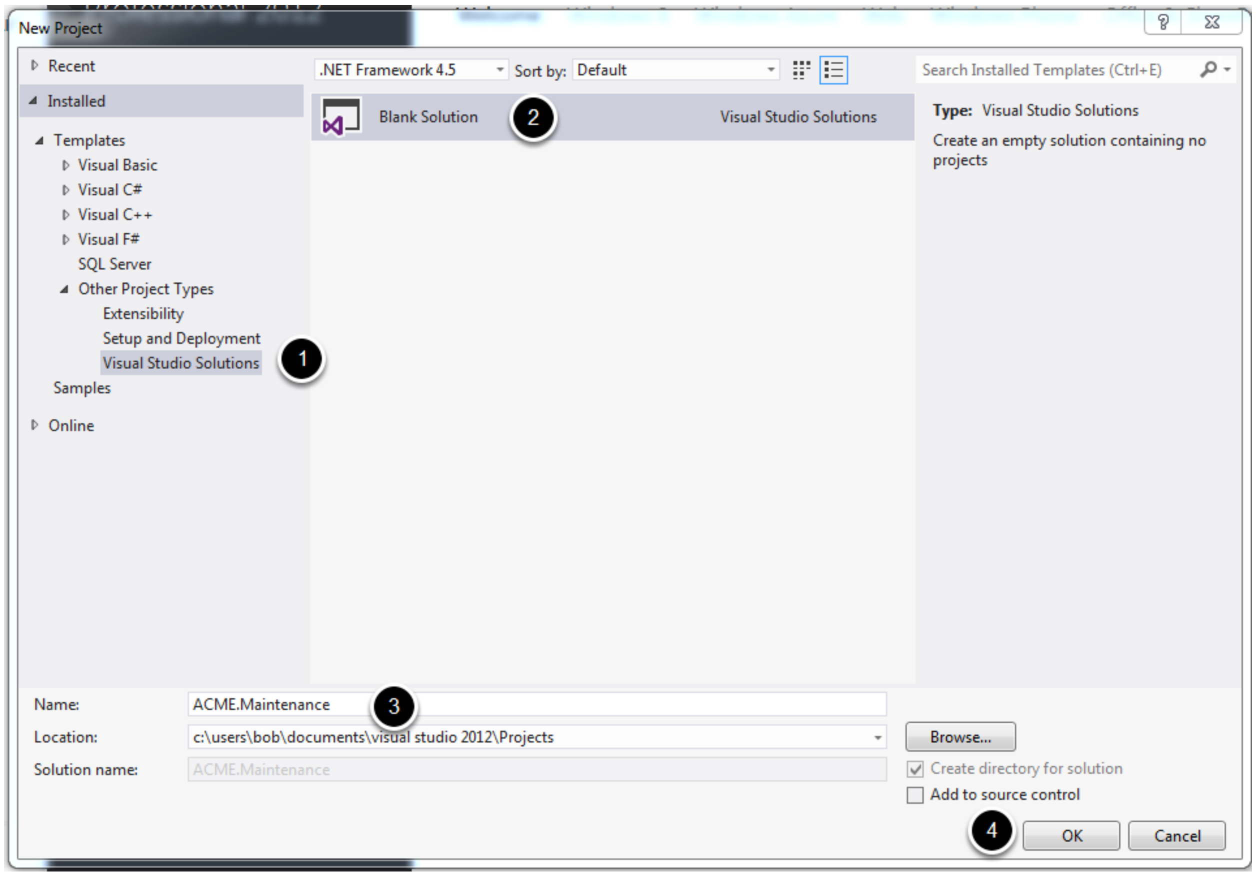
Select:

(1) File

(2) New

(3) Project ...

In the New Project dialog...



Type: Visual Studio Solutions

Create an empty solution containing no projects

Name: ACME.Maintenance

Location: c:\users\bob\documents\visual studio 2012\Projects

Solution name: ACME.Maintenance

[✓] Create directory for solution

[ ] Add to source control

(1) Navigate to Templates | Other Project Types | Visual Studio Soluitions.

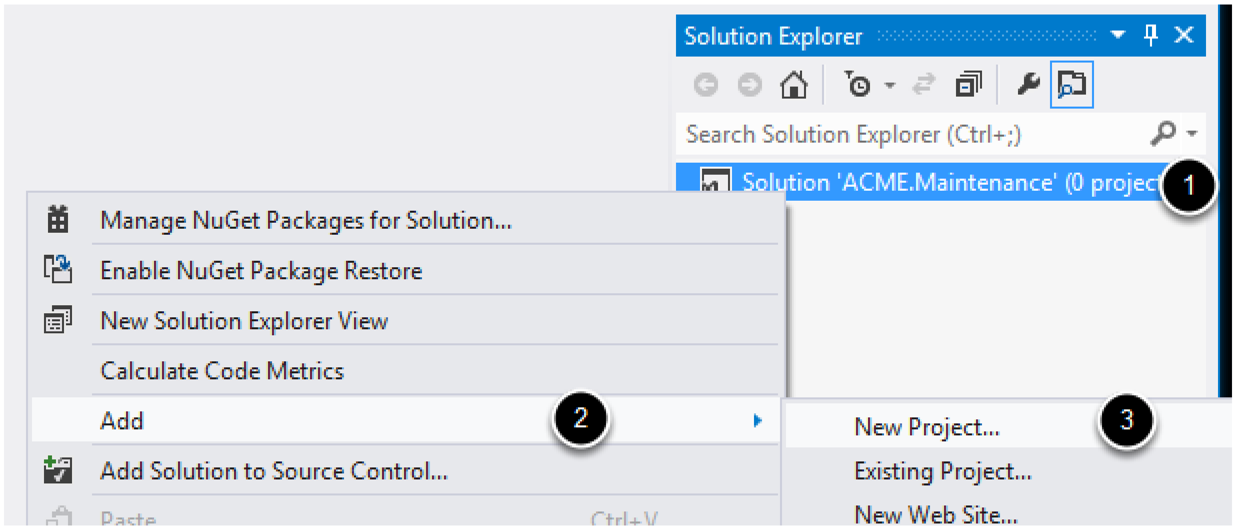
(2) Select the Blank Solution template.

(3) Rename to ACME.Maintenance

(4) Select OK

2. Add a new Class Library project called ACME.Maintenance.Domain

In the Solution Explorer...

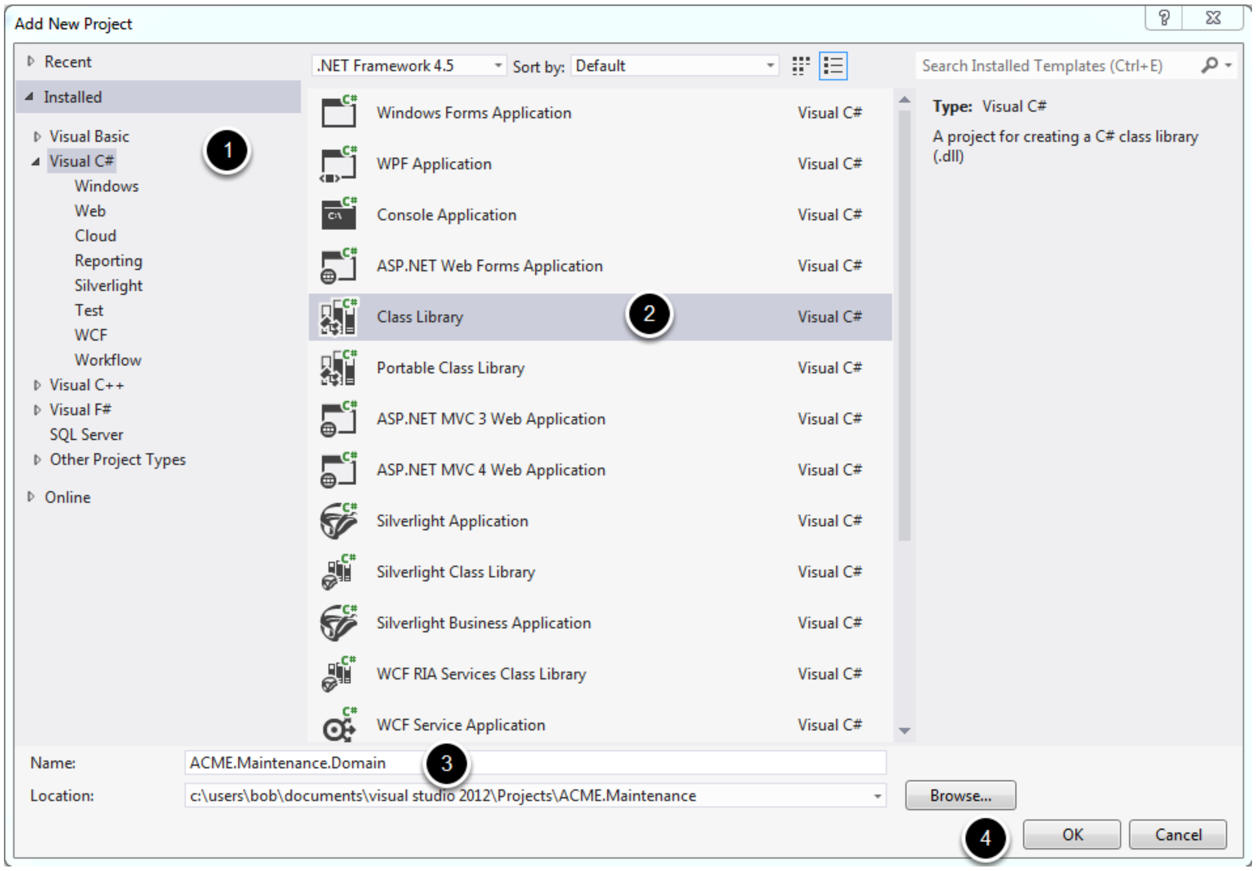

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ACME.Maintenance' (0 projec
1
Manage NuGet Packages for Solution...
Enable NuGet Package Restore
New Solution Explorer View
Calculate Code Metrics
Add
2
Add Solution to Source Control...
Paste
Ctrl+V
New Project...
3
Existing Project...
New Web Site...

(1) Right-click the Solution 'ACME.Maintenance'

(2) Select Add

(3) New Project ...

In the Add New Project dialog...


Add New Project
Recent
Installed
Visual Basic
Visual C#
Windows
Web
Cloud
Reporting
Silverlight
Test
WCF
Workflow
Visual C++
Visual F#
SQL Server
Other Project Types
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
ASP.NET Web Forms Application Visual C#
Class Library Visual C#
Portable Class Library Visual C#
ASP.NET MVC 3 Web Application Visual C#
ASP.NET MVC 4 Web Application Visual C#
Silverlight Application Visual C#
Silverlight Class Library Visual C#
Silverlight Business Application Visual C#
WCF RIA Services Class Library Visual C#
WCF Service Application Visual C#
1
2
3
4
Browse...
OK
Cancel

Type: Visual C#

A project for creating a C# class library (.dll)

Name: ACME.Maintenance.Domain

Location: c:\users\bob\documents\visual studio 2012\Projects\ACME.Maintenance

(1) Navigate to Installed | Visual C#

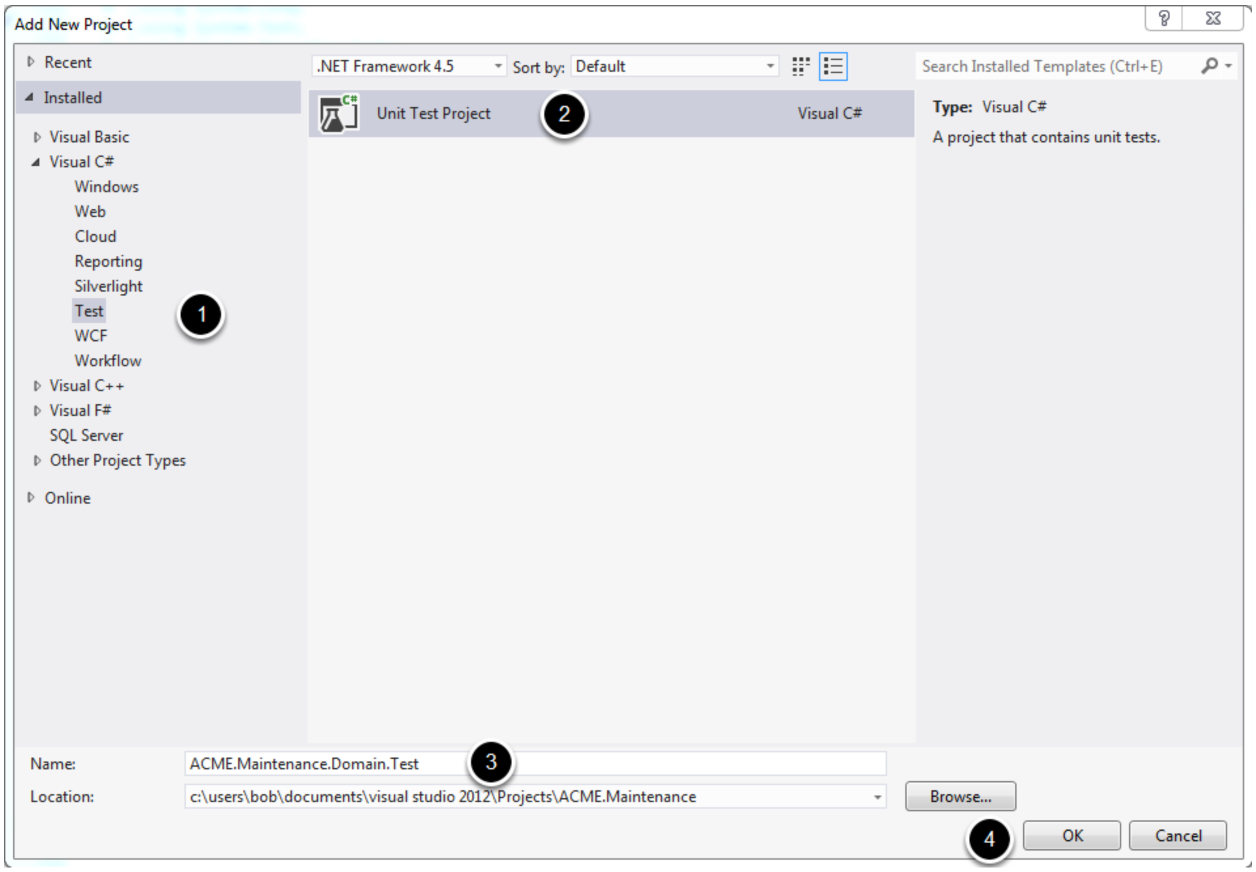
(2) Select the Class Library template

(3) Rename to: ACME.Maintenance.Domain

(4) Select OK

3. Create a new Unit Test Project called ACME.Maintenance.Domain.Test

Start by using the same steps to Add a New Project to the Solution (step 2 above), however in the Add New Project dialog…



Type: Visual C#

A project that contains unit tests.

Name: ACME.Maintenance.Domain.Test

Location: c:\users\bob\documents\visual studio 2012\Projects\ACME.Maintenance

(1) Select Visual C# | Test

(2) Select the Unit Test Project template

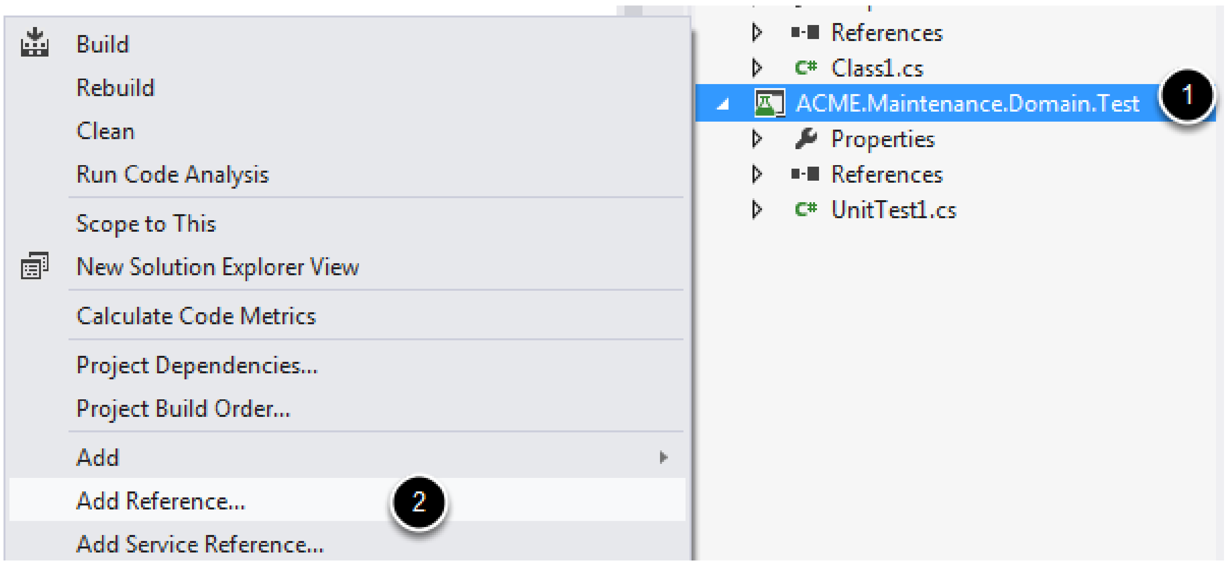
(3) Rename: ACME.Maintenance.Domain.Test

(4) Select OK

4. Create a Reference from the Unit Test project to the Domain Project

Our goal is to make sure that the Test project can "see" the Domain project. We'll create a reference from the Test project to the Domain project.

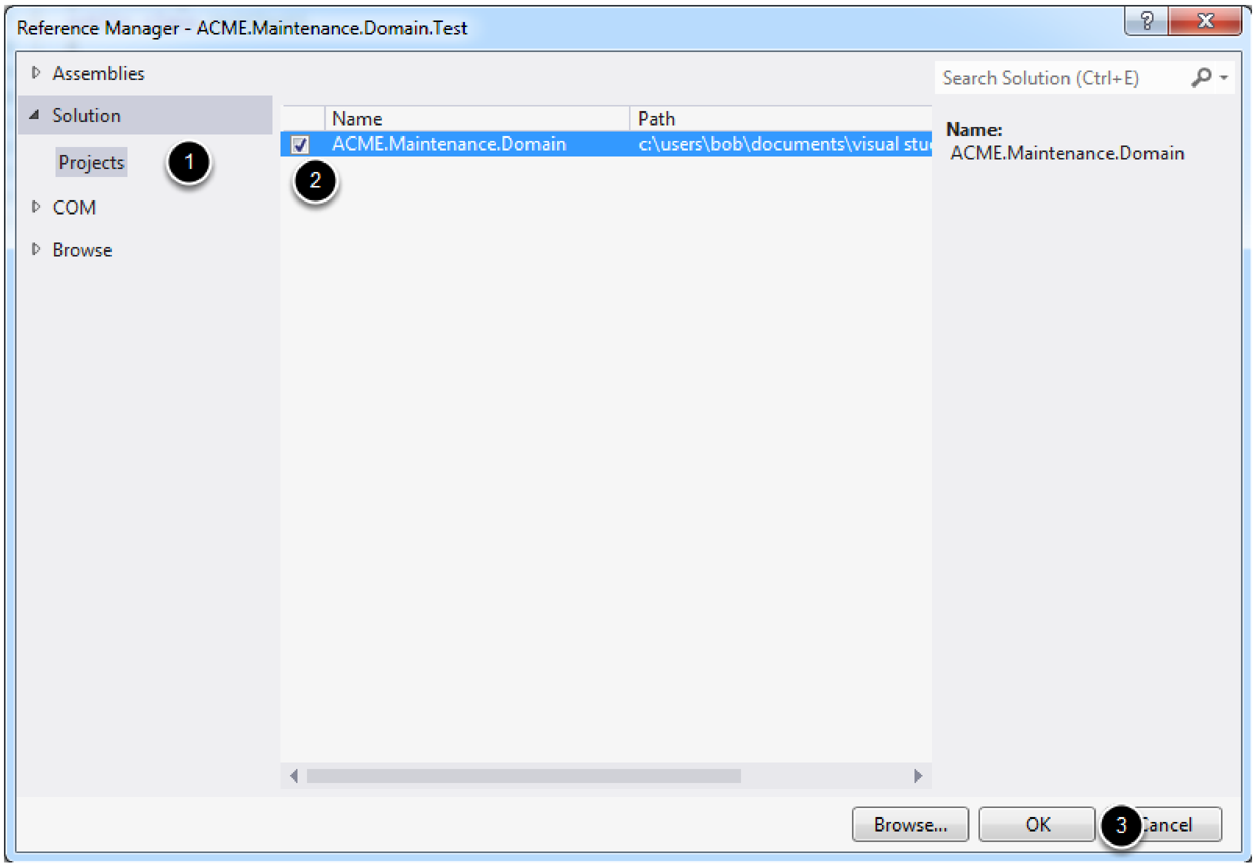
In the Solution Explorer...



(1) Right-click the ACME.Maintenance.Domain.Test project

(2) Select Add Reference ...

In the Reference Manager dialog...



	Name	Path
✓	ACME.Maintenance.Domain	c:\users\bob\documents\visual stu

Name:
ACME.Maintenance.Domain

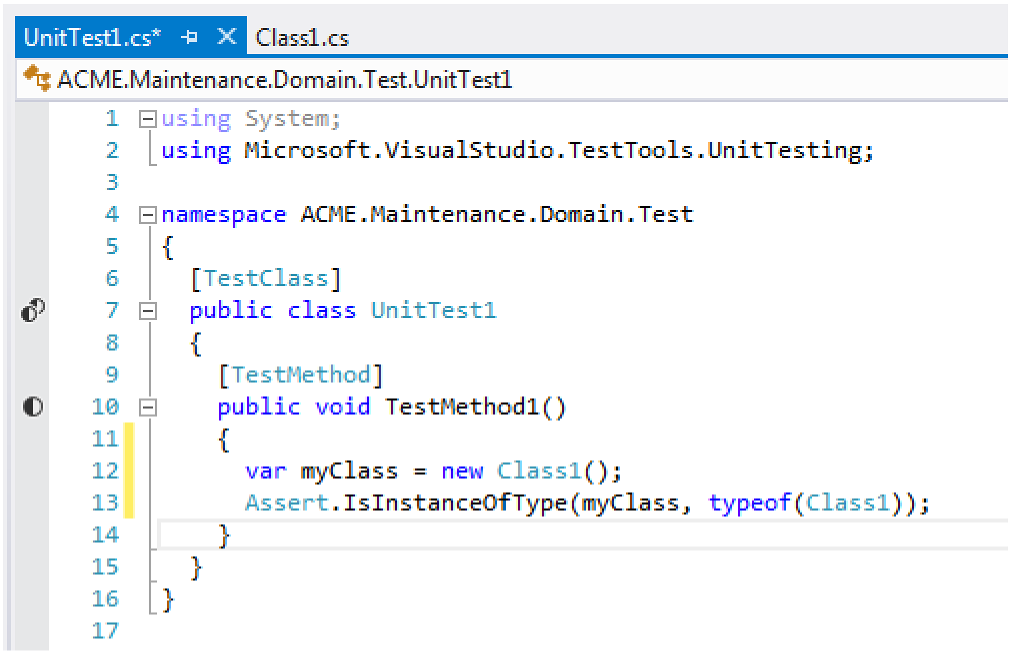
(1) Make sure Solution | Project is selected

(2) Check the checkbox next to the ACME.Maintenance.Domain project

(3) Select OK

5. Test to make sure the Unit Test project can see the Domain project

To verify that the Unit Test project can see the Domain project, I'll write a simple test. I write the following code in the UnitTest1.cs file:

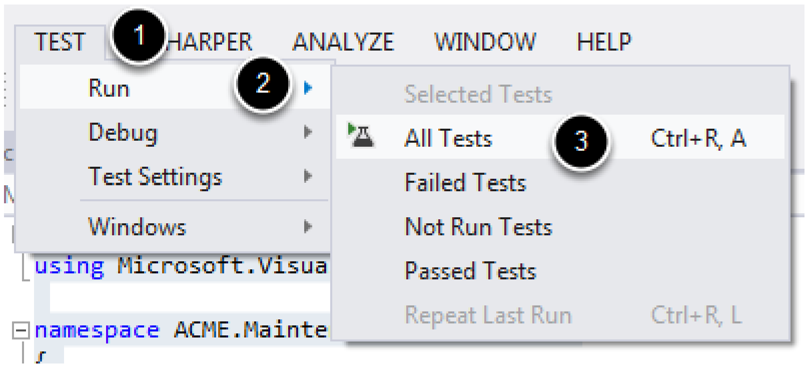


```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ACME.Maintenance.Domain.Test
{
  [TestClass]
  public class UnitTest1
  {
    [TestMethod]
    public void TestMethod1()
    {
      var myClass = new Class1();
      Assert.IsInstanceOfType(myClass, typeof(Class1));
    }
  }
}
```

Copy code from:

https://gist.github.com/LearnVisualStudio/fb7f76442380a6771856

6. Run the unit tests

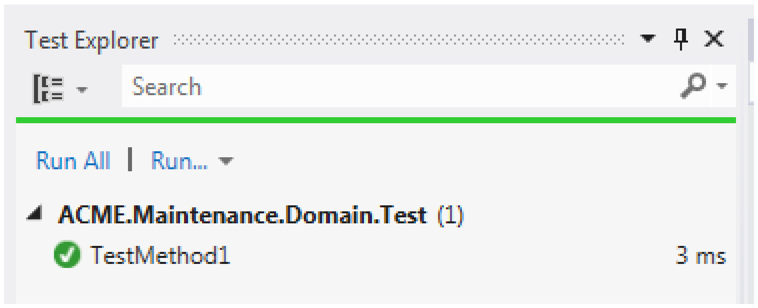


(1) Select Test menu

(2) Run

(3) All Tests

7. Verify that the test passed



```
ACME.Maintenance.Domain.Test (1)
  TestMethod1                                3 ms
```

A green checkmark next to the Unit Test name tells us we've succeeded.

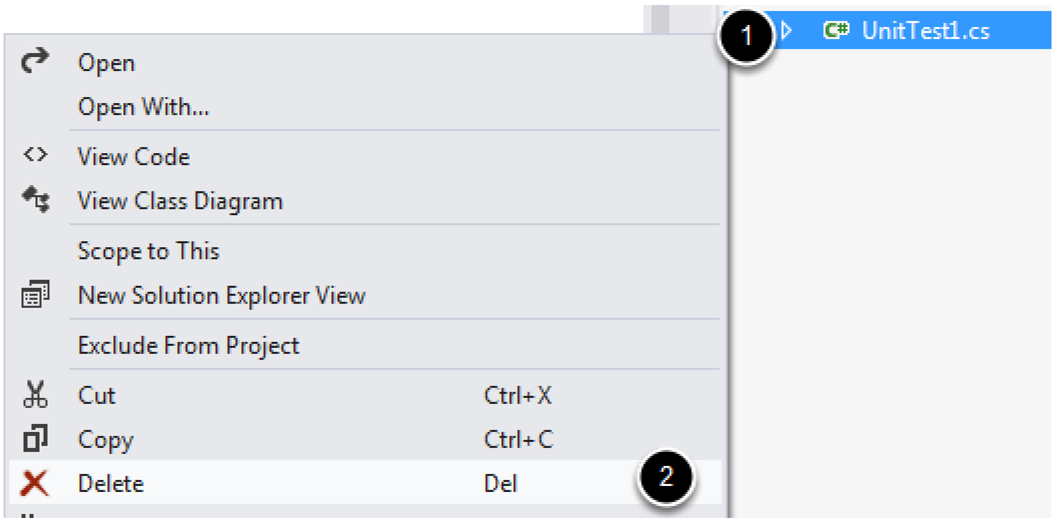
Now that we have everything wired up, we can start writing real tests. We'll do that in the next lesson.

Lesson 7 - Creating the first Unit Test

Now that we know the Test project can see the Domain project, I will delete the class files that are added by default to each project. I could rename each of them, but it's just a workflow preference.

1. Delete the UnitTest1.cs and the Class1.cs files

In the Solution Explorer, in the ACME.Maintenance.Domain.Test project...



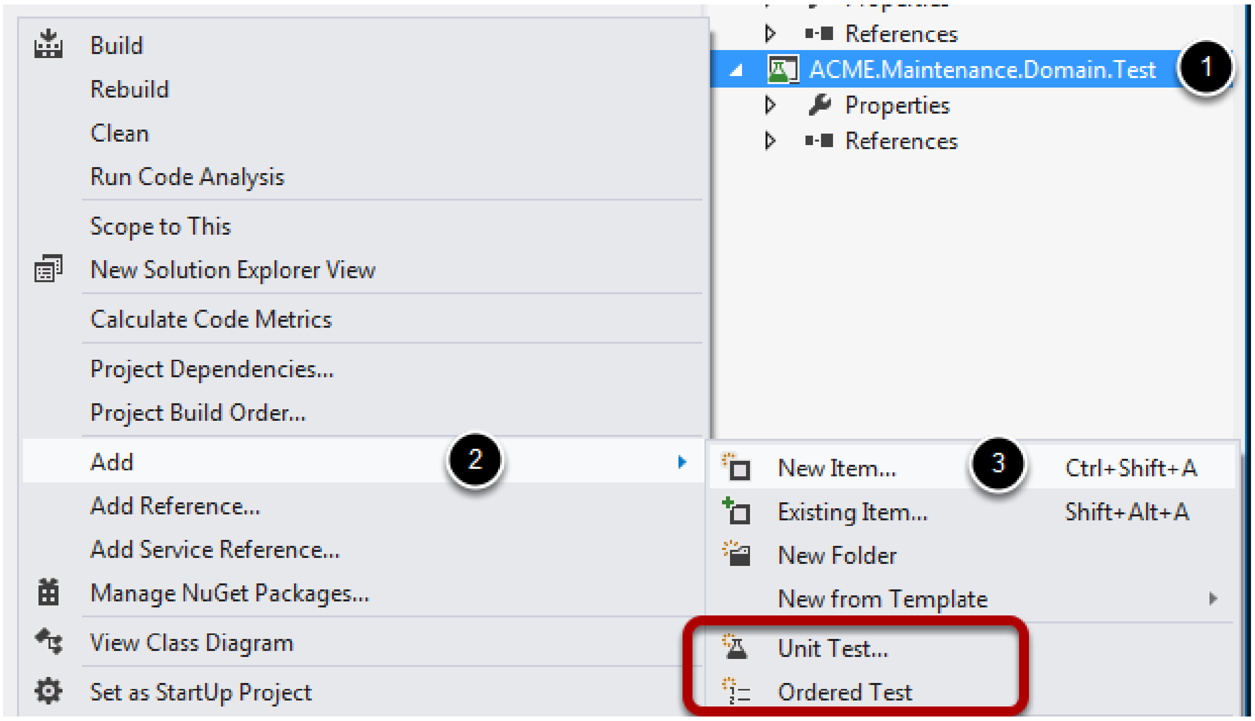
(1) Right-click the UnitTest1.cs file

(2) Select Delete

Repeat this process to delete the Class1.cs file in the ACME.Maintenance.Domain project.

2. Add new Unit Test class file to the Test project

In the Solution Explorer...



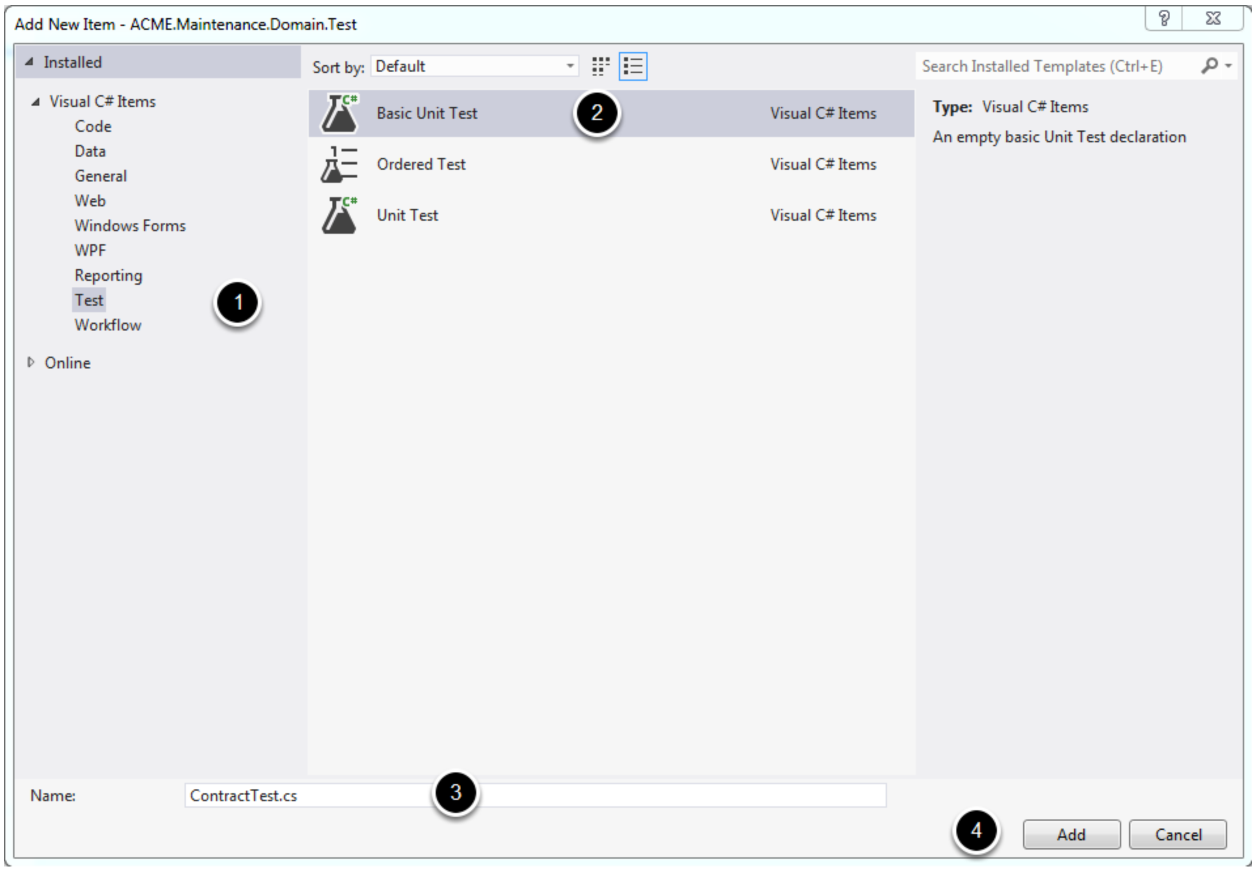
(1) Right-click the ACME.Maintenance.Domain.Test project

(2) Select Add

(3) New Item ...

Note ... I've put a red box around two other options ... throughout this series I do not choose these because (a) you have to rename the file and the class name, and (b) it doesn't provide me the template that I want to use. Feel free to experiment and learn the differences. Ultimately, here again, this is a workflow preference. Use which ever approach you prefer.

In the Add New Item dialog…


Add New Item - ACME.Maintenance.Domain.Test

Installed
- Visual C# Items
  - Code
  - Data
  - General
  - Web
  - Windows Forms
  - WPF
  - Reporting
  - Test
  - Workflow
- Online

Sort by: Default

Basic Unit Test	Visual C# Items
Ordered Test	Visual C# Items
Unit Test	Visual C# Items

Search Installed Templates (Ctrl+E)

Type: Visual C# Items

An empty basic Unit Test declaration

Name: ContractTest.cs

(1) Select Visual C# Items | Test

(2) Select Basic Unit Test template

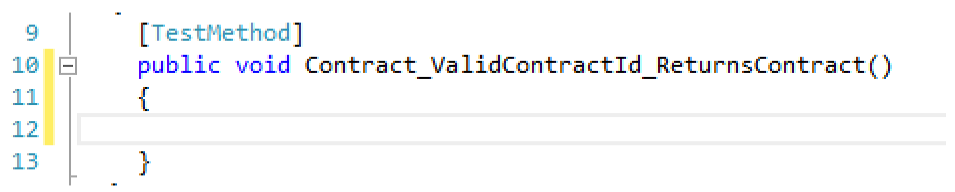
(3) Rename to: ContractTest.cs

(4) Select Add

3. Rename the Unit Test

I choose to rename the unit test:

Contract_ValidContractId_ReturnsContract()



```
[TestMethod]
public void Contract_ValidContractId_ReturnsContract()
{

}
```

We'll talk about the naming convention in another video, but in a nutshell, I follow this convention:

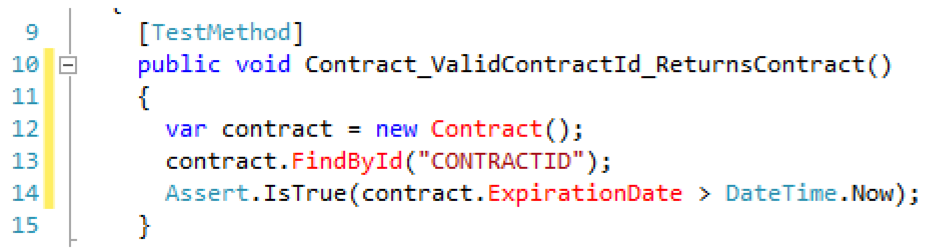
MethodName_StateUnderTest_ExpectedBehavior

In this case, I've named the Unit Test "Contract_" because I thought I would be implementing most of the object instantion code and logic in the constructor of the Contract class. As I work through this example, you'll see I deviate from this and later in this series I'll be forced to rename this method.

4. Write the Unit Test in an effort to design the Contract class

The point of this exercise is to write code that consumes the Contract class EVEN THOUGH WE HAVEN'T CREATED THE CONTRACT CLASS YET. We'll write the code designing how we would expect to use the Contract class. This means (a) we'll be fighting Intellisense and auto-completion (use the ESC key once Intellisense appears to avoid it from happening) and (b) we'll likely be modifying this in the future once we see it working. It's rare to get this right on the first try. However, we have to start somewhere.

Here's what I come up with:



```
 9      [TestMethod]
10      public void Contract_ValidContractId_ReturnsContract()
11      {
12        var contract = new Contract();
13        contract.FindById("CONTRACTID");
14        Assert.IsTrue(contract.ExpirationDate > DateTime.Now);
15      }
```

In this example, I would create a new instance of the Contract class, then call the FindbyId(), passing in the contractId. I'm hard-coding this value for now. I would expect something in the FindById() method to work with the database (or rather, the Persistence Layer) to retrieve the contract's data associated with the contractId and then populate the Contract object instance's properties.

Finally (line 14) I want to ensure that we have a valid contract, meaning that the ExpirationDate property is set to a value in the future. There are probably more (and better) checks we could also make, but we'll start here.

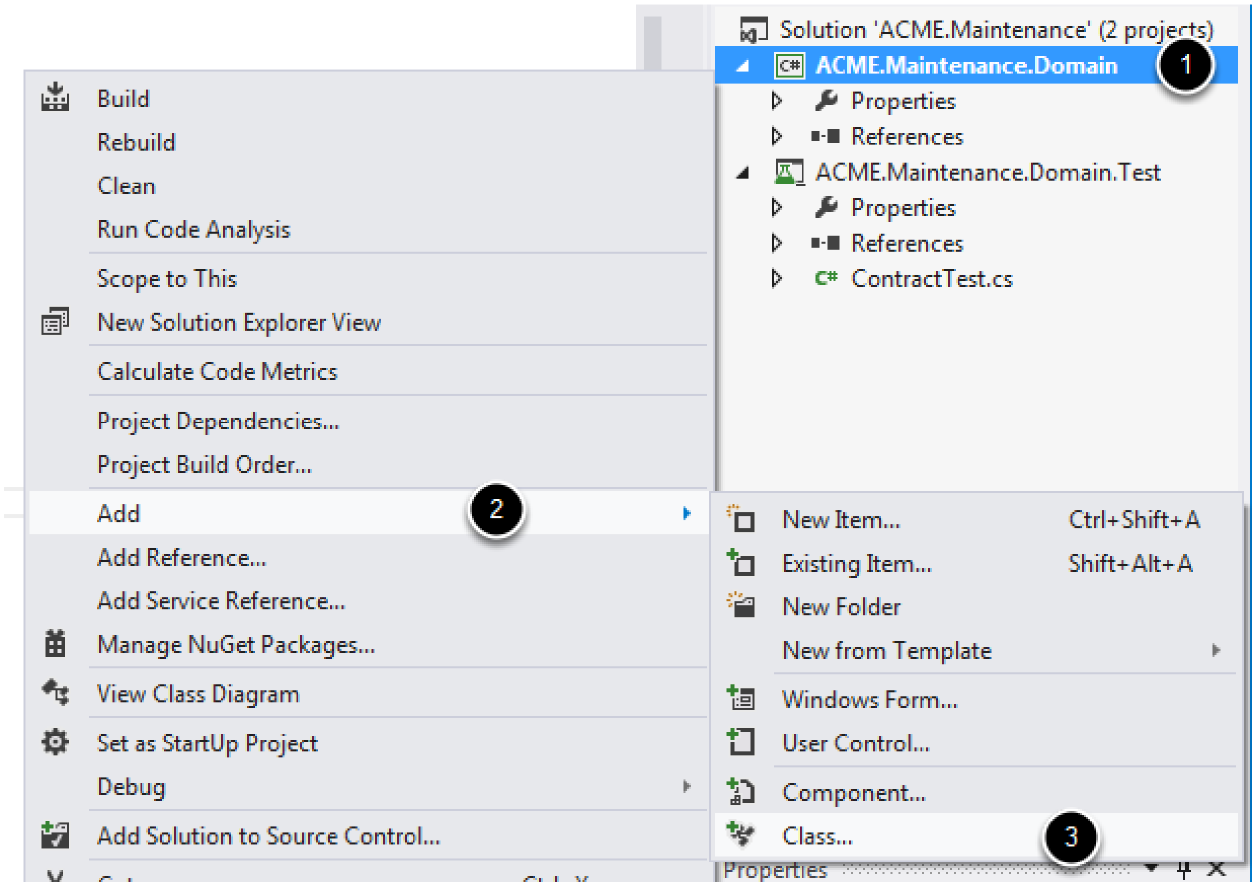
Get this code:

https://gist.github.com/LearnVisualStudio/a5e45929bd2bd3f0edbe

5. Create the Contract class to implement the design

Now that we have a Unit Test, we need to write the production code to satisfy the Unit Test. We'll create the Contract class in the Domain Layer project.

In the Solution Explorer ...

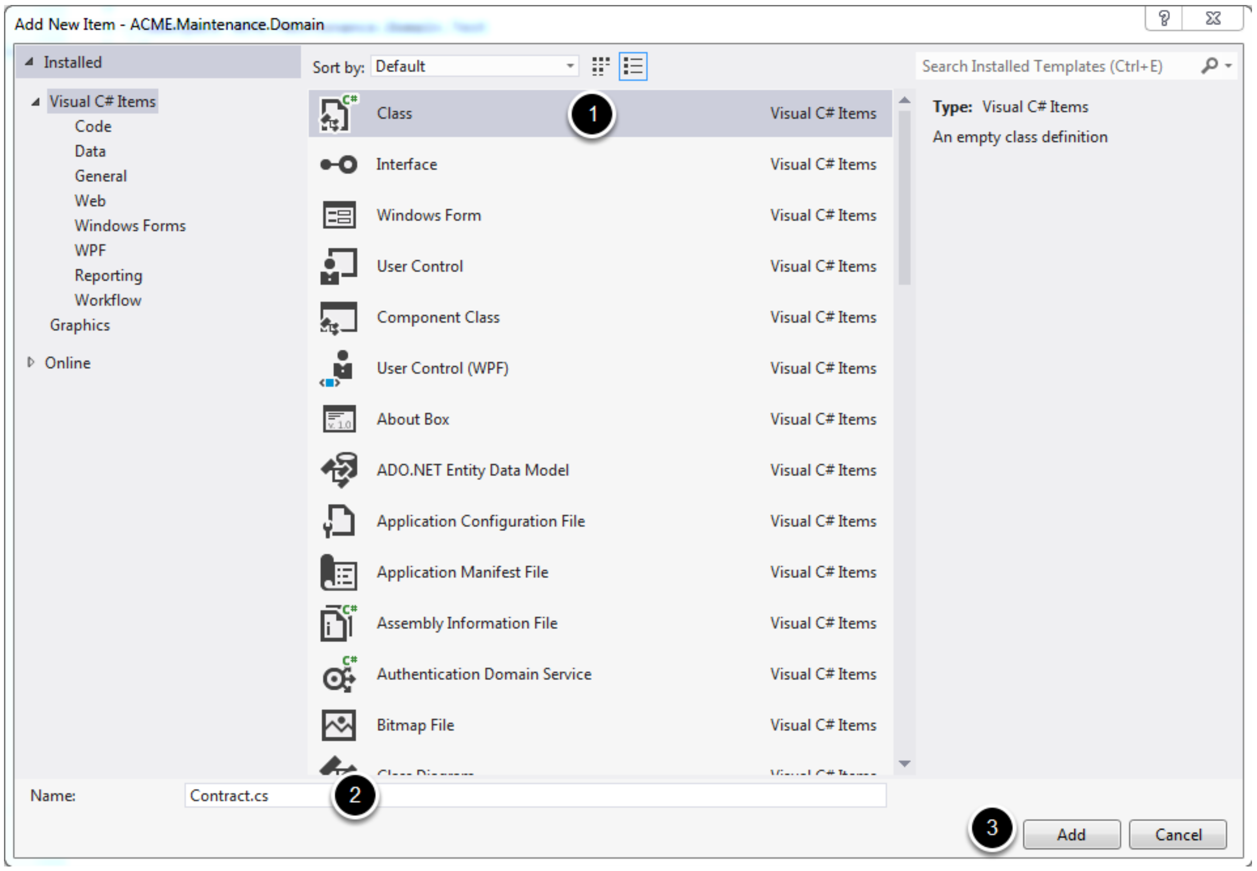


(1) Right-click the ACME.Maintenance.Domain project

(2) Select Add

(3) Class ...

In the Add New Item dialog…


Add New Item - ACME.Maintenance.Domain
Installed
Visual C# Items
Code
Data
General
Web
Windows Forms
WPF
Reporting
Workflow
Graphics
Online
Sort by: Default
Class — Visual C# Items
Interface — Visual C# Items
Windows Form — Visual C# Items
User Control — Visual C# Items
Component Class — Visual C# Items
User Control (WPF) — Visual C# Items
About Box — Visual C# Items
ADO.NET Entity Data Model — Visual C# Items
Application Configuration File — Visual C# Items
Application Manifest File — Visual C# Items
Assembly Information File — Visual C# Items
Authentication Domain Service — Visual C# Items
Bitmap File — Visual C# Items
Search Installed Templates (Ctrl+E)
Type: Visual C# Items
An empty class definition
1
2
3
Add
Cancel

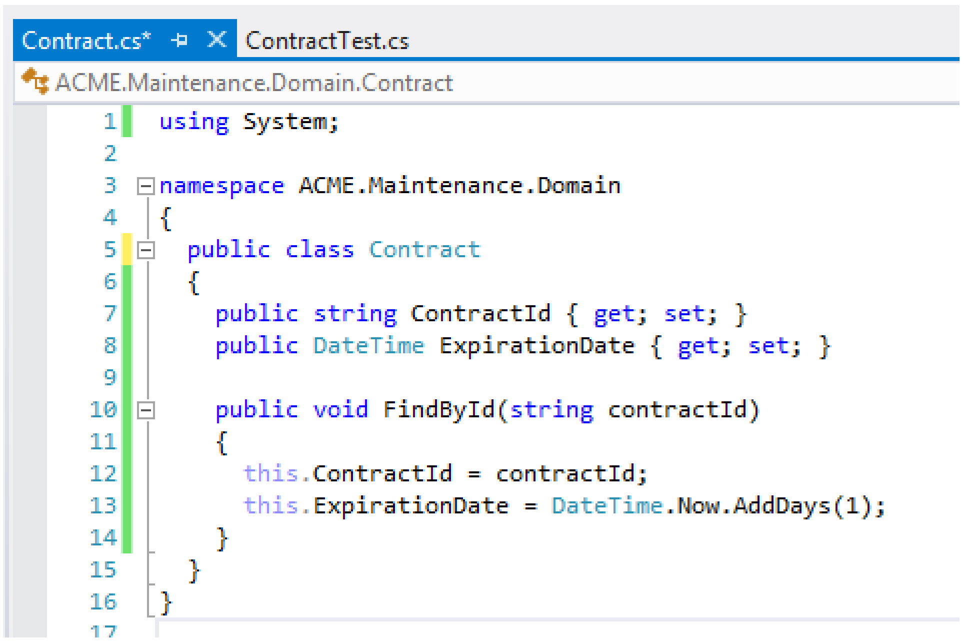
Name: Contract.cs

(1) the Class template should already be selected. If not, choose it.

(2) Rename to: Contract.cs

(3) Select Add

I then write the code in the Contract.cs to satisfy the unit test. This is what I come up with:



```
using System;

namespace ACME.Maintenance.Domain
{
  public class Contract
  {
    public string ContractId { get; set; }
    public DateTime ExpirationDate { get; set; }

    public void FindById(string contractId)
    {
      this.ContractId = contractId;
      this.ExpirationDate = DateTime.Now.AddDays(1);
    }
  }
}
```

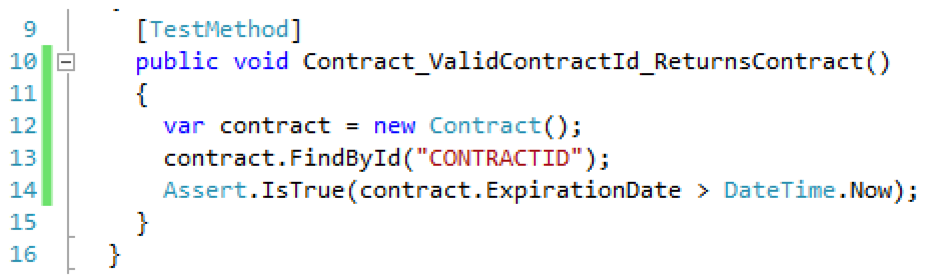
Hopefully this is self-explanatory. The FindbyId() method is simply hardcoded at the moment ... I do just enough to satisfy the unit test. I want to get my "Red" non-passing unit test to a "Green" passing unit test. I'll then continue to work on it until I get the production code quality I desire. That will hapen in the next lesson.

Get this code:

https://gist.github.com/LearnVisualStudio/588cdd56b4e72a4ba4ec

6. Observe the Unit Test

I briefly look back at the unit test to see if it is "satisfied" by the addition of the Contract class and its members.

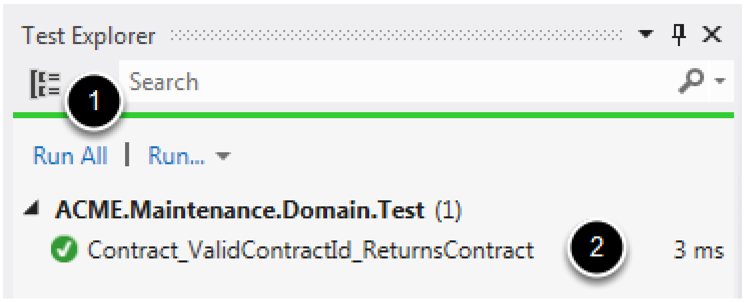


```
    [TestMethod]
    public void Contract_ValidContractId_ReturnsContract()
    {
      var contract = new Contract();
      contract.FindById("CONTRACTID");
      Assert.IsTrue(contract.ExpirationDate > DateTime.Now);
    }
  }
```

Notice that by adding the Contract.cs class and its implementation, the Unit Test should now see it and Intellisense no longer "complains".

7. Run All Unit Tests

In the Test Explorer...



ACME.Maintenance.Domain.Test (1)
Contract_ValidContractId_ReturnsContract 3 ms

(1) Click the Run All link

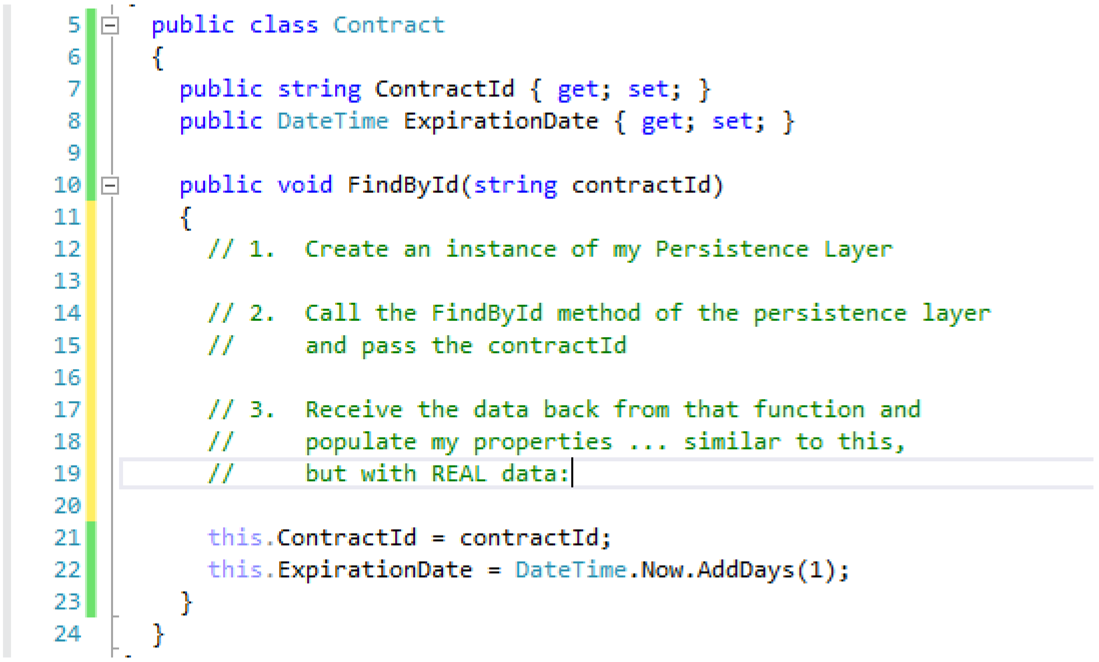
(2) The Unit Test should succeed (green check mark)

We're following the Unit Testing mantra of "Red, Green, Refactor". While we've gotten the Unit Test to pass, we've merely hard coded our production code (the method under test, namely, the Contract). Merely to safisfy the test. At this point, we need to refactor the method under test. Furthermore, the Unit Test itself needs to be refactored. Now that we see it, we have to ask ourselves, is this the implementation we really want? What are the ramifications of this style of object creation?

Lesson 8 - Refactoring the first Unit Test

When we left the previous lesson, we successfully got the Unit Test to work. However, everything is hard coded. It's time to refactor and really think about the implementation. Is this what I want? It is simple, but I foresee a problem. I will need to create a new instance of the persistence layer class, then immediately call FindById() passing in the ContractId I want to search for. While I've not implemented this yet, I would expect to make a call into a persistence layer to retrieve the data. What I retrieve from the persistence layer, I would then populate in the properties of my object instance instance.

I've added some comments to explain what I anticipate will be the process I will need:



```
public class Contract
{
  public string ContractId { get; set; }
  public DateTime ExpirationDate { get; set; }

  public void FindById(string contractId)
  {
    // 1.  Create an instance of my Persistence Layer

    // 2.  Call the FindById method of the persistence layer
    //     and pass the contractId

    // 3.  Receive the data back from that function and
    //     populate my properties ... similar to this,
    //     but with REAL data:

    this.ContractId = contractId;
    this.ExpirationDate = DateTime.Now.AddDays(1);
  }
}
```

The problem with this approach is twofold:

(1) I'm relying on a certain order of operations. If someone were to create an instance of my Contract class AND NOT CALL FINDBYID, but instead reference the ExpirationDate ... it would raise an exception. It's not a good idea when creating an API (which is essentially what we're doing by starting with a Class Project type and creating methods and properties that implement our problem domain) to *not* rely on a certain order.

(2) I've delegated too much responsibility to the FindById() method ... and probably to the entire class! The Single Responsibility Principle applies here.

Question: can you identify where we are violating the Single Responsibility Principle (SRP)? Hint: this is a subtle but important distinction.

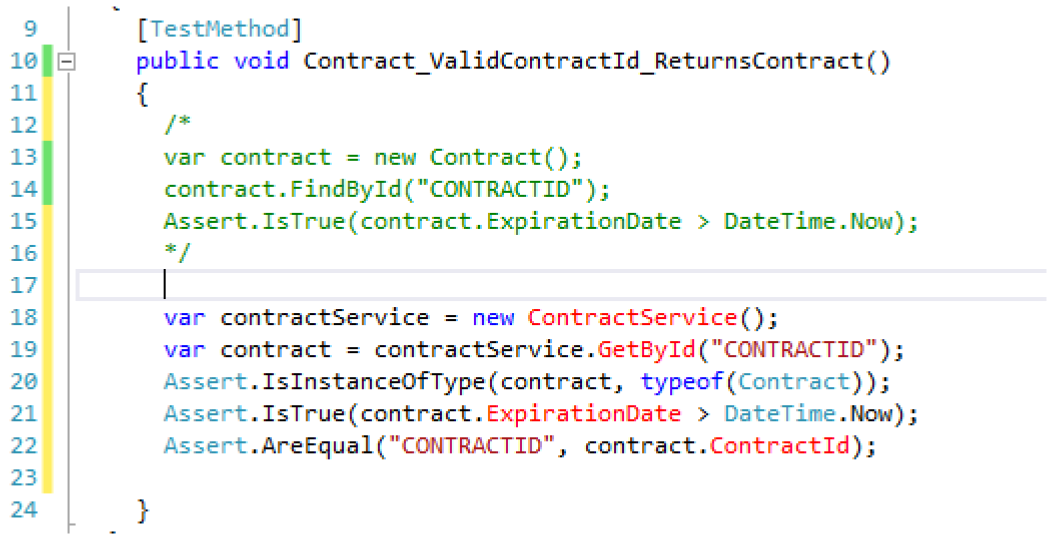
Answer: In my mind, the two responsibilities are at a class level, and are conceptual. They are (a) creating and calling into the persistence layer, and (b) performing all the functions and responsibilities of being a domain entity ... In other words, if I were to continue down this road, there would be way too much emphasis on instantiation. I would like to take those responsibilities and give them to another class.

So, I'm going to create a ContractService. It will have the responsibility of creating an instance of a Contract, and will be able to work with the persistence layer to get the data needed at creation and populate the new Contract's properties. Once we have the state of the new Contract ready, the pure Contract object can be used solely for its intended purpose: to represent a Contract in our problem domain.

To right this wrong, I'm going to re-do the Unit Test to include the notion of a ContractService.

1. Refactor the Unit Test to Re-design the Method Under Test

I decided to comment out the previous code and re-design the unit test. The real goal is to re-design the production code based on my new thinking on the matter. Here's what I come up with:



```
[TestMethod]
public void Contract_ValidContractId_ReturnsContract()
{
  /*
  var contract = new Contract();
  contract.FindById("CONTRACTID");
  Assert.IsTrue(contract.ExpirationDate > DateTime.Now);
  */

  var contractService = new ContractService();
  var contract = contractService.GetById("CONTRACTID");
  Assert.IsInstanceOfType(contract, typeof(Contract));
  Assert.IsTrue(contract.ExpirationDate > DateTime.Now);
  Assert.AreEqual("CONTRACTID", contract.ContractId);

}
```

I've broken it dramatically, but this more closely represents what I ultimately want. Let me implement the new ContractService class and refactor the Contract as well, removing the old FindById method.

Get the code:

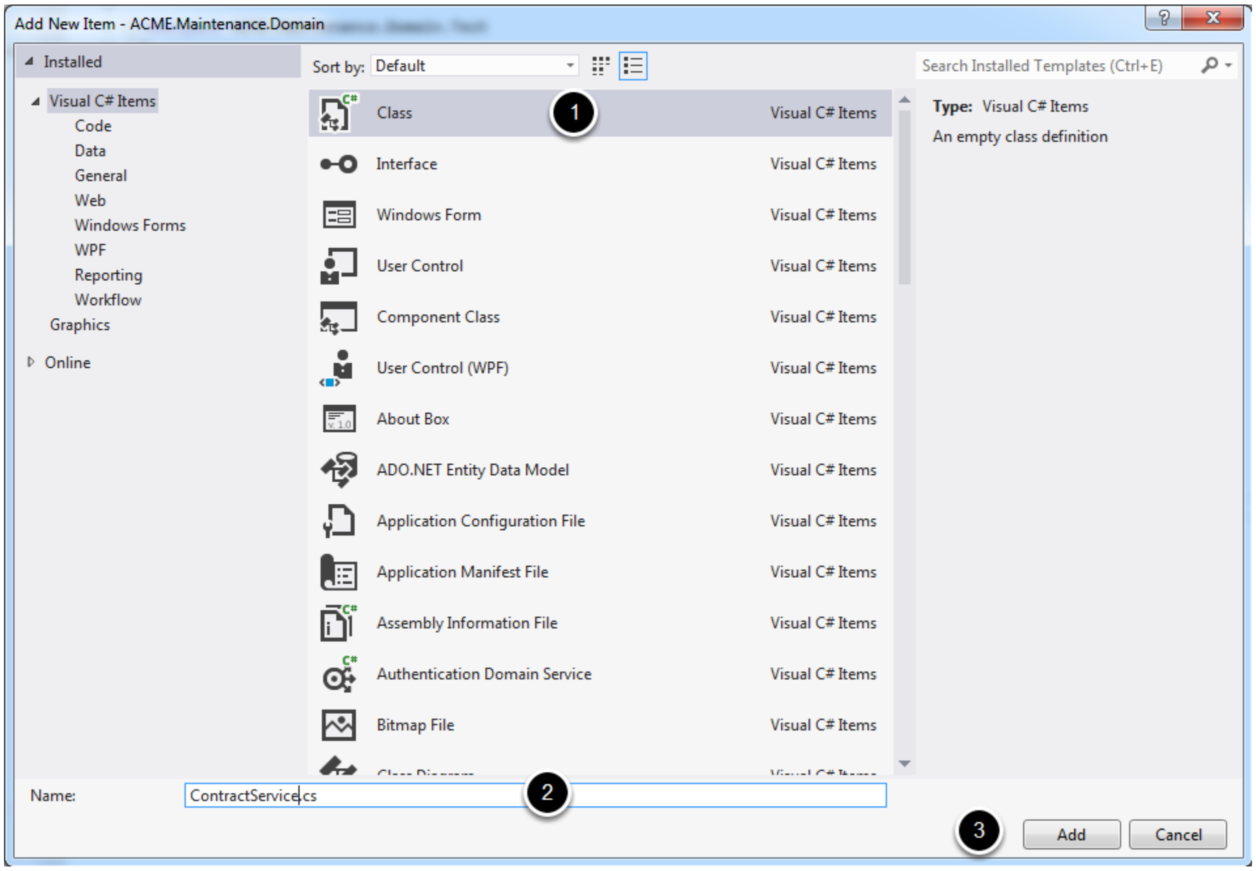
https://gist.github.com/LearnVisualStudio/1a0bb9788e256e17d072

2. Add the new ContractService Class

Now that I have a new design, I'll need to implement it in production code, in the Domain Layer project. I begin by creating a new ContractService.cs class.

In the Solution Explorer, right-click the ACME.Maintenance.Domain project and select Add New | Class from the context menu.

In the Add New Item dialog...

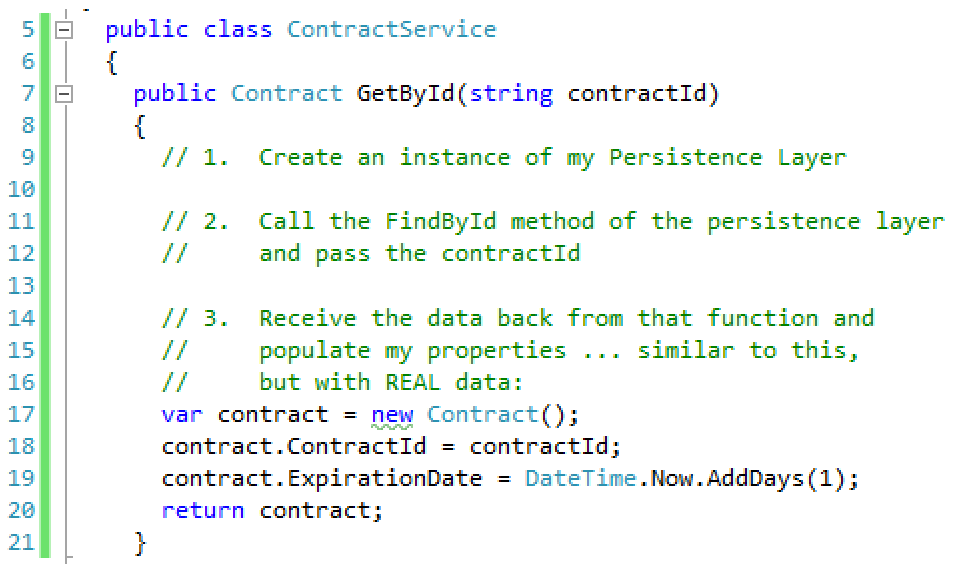


(1) Make sure the Class template is selected

(2) Rename to: ContractService.cs

(3) Select Add

Next, I implement the GetById() method. I use the Contract.FindById() method as a template (truth be told, I copy, paste and edit it. Shhh. Don't tell anyone I copied and pasted.)



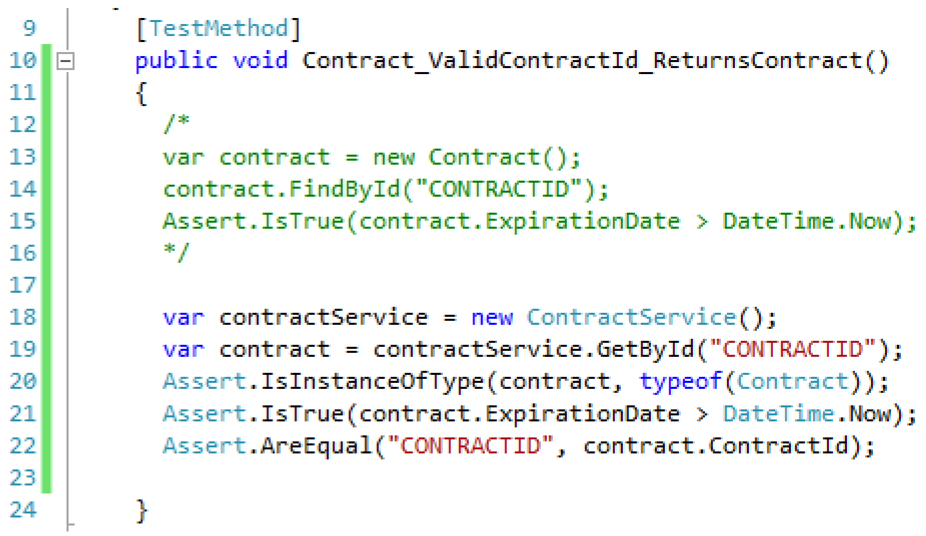
```
public class ContractService
{
  public Contract GetById(string contractId)
  {
    // 1.  Create an instance of my Persistence Layer

    // 2.  Call the FindById method of the persistence layer
    //     and pass the contractId

    // 3.  Receive the data back from that function and
    //     populate my properties ... similar to this,
    //     but with REAL data:
    var contract = new Contract();
    contract.ContractId = contractId;
    contract.ExpirationDate = DateTime.Now.AddDays(1);
    return contract;
  }
```

As you can see, the new GetById method closely resembles the FindById from our Contract class. The difference? Here we're creating an instance of Contract, populating it's values, and returning it.

Get the code:

https://gist.github.com/LearnVisualStudio/b111ea1bf0fcf7324229

I briefly glance back at the unit test code to make sure it is "satisfied" with my changes.

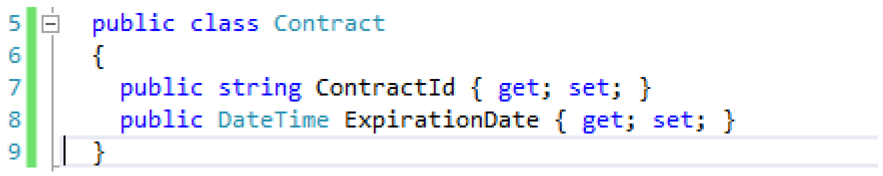


```
[TestMethod]
public void Contract_ValidContractId_ReturnsContract()
{
  /*
  var contract = new Contract();
  contract.FindById("CONTRACTID");
  Assert.IsTrue(contract.ExpirationDate > DateTime.Now);
  */

  var contractService = new ContractService();
  var contract = contractService.GetById("CONTRACTID");
  Assert.IsInstanceOfType(contract, typeof(Contract));
  Assert.IsTrue(contract.ExpirationDate > DateTime.Now);
  Assert.AreEqual("CONTRACTID", contract.ContractId);

}
```

As a result of my changes, the unit test now has what it needs. There are no red-squiggly lines.

3. Modify the Contract class

Next, I'll need to remove the FindById() method from



```
public class Contract
{
  public string ContractId { get; set; }
  public DateTime ExpirationDate { get; set; }
}
```

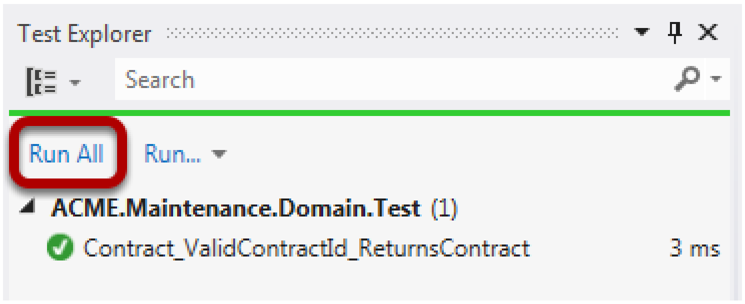
I also simplify the Contract class, removing the FindById method.

Get the code:

https://gist.github.com/LearnVisualStudio/d39732f4e2d32d055083

4. Run All Unit Tests

In the Test Explorer, I click the Run All link in the upper-left-hand-corner:



```
ACME.Maintenance.Domain.Test (1)
  Contract_ValidContractId_ReturnsContract    3 ms
```

And the green checkmark icon tells me it's working again.

So, we made an improvement to the creation of the Contract object by delegating the responsibility of its creation to a new class ... we have encapsulated instantiation in the ContractService class.

Admittedly, we have two classes where once we had one class. That does add a little complexity -- you have to look in two places instead of one to understand what you need to know about creating and working with Contracts. Generally, this is acceptable because each class is now more cohesive ... they have a distinct responsibility, aka, the Single Responsibility Principle. Moreover, when we get a new instance of a Contract now, that Contract instance will be in the proper state -- we don't have to worry about the Contract's properties not being populated yet because we forgot to call a method on that object instance.

Also, another potential concern is that these two classes are coupled together and I said that coupling is bad. However, what I actually said was coupling is bad *between layers*. Inside of a given layer, you cannot avoid some coupling.

Now, the majority of complexity is in the ContractService.GetById method. It's not quite right and needs to be refactored. Can you look at it and determine what the problem is? I'll address it in the next lesson.

Lesson 9 - Creating a Stubbed Persistence Class and Data Transfer Object class

In the previous lesson I asked if you could determine the problem with our ContractService.GetById. The problem is that we're still working with hardcoded values. Obviously this is not a production-worthy class yet. Ideally, we can pull values from a persistence class, or rather, a Class in another layer that will actually return data to us from a datastore. Once we get that data, we can populate the properties of our new Contract instance and return it to the caller.

But I don't want to worry about persistence just yet. I'm not ready to think about it. So, what I'll do is create a stub. Martin Fowler wrote a now-famous article: "Mocks Aren't Stubs"

http://martinfowler.com/articles/mocksArentStubs.html

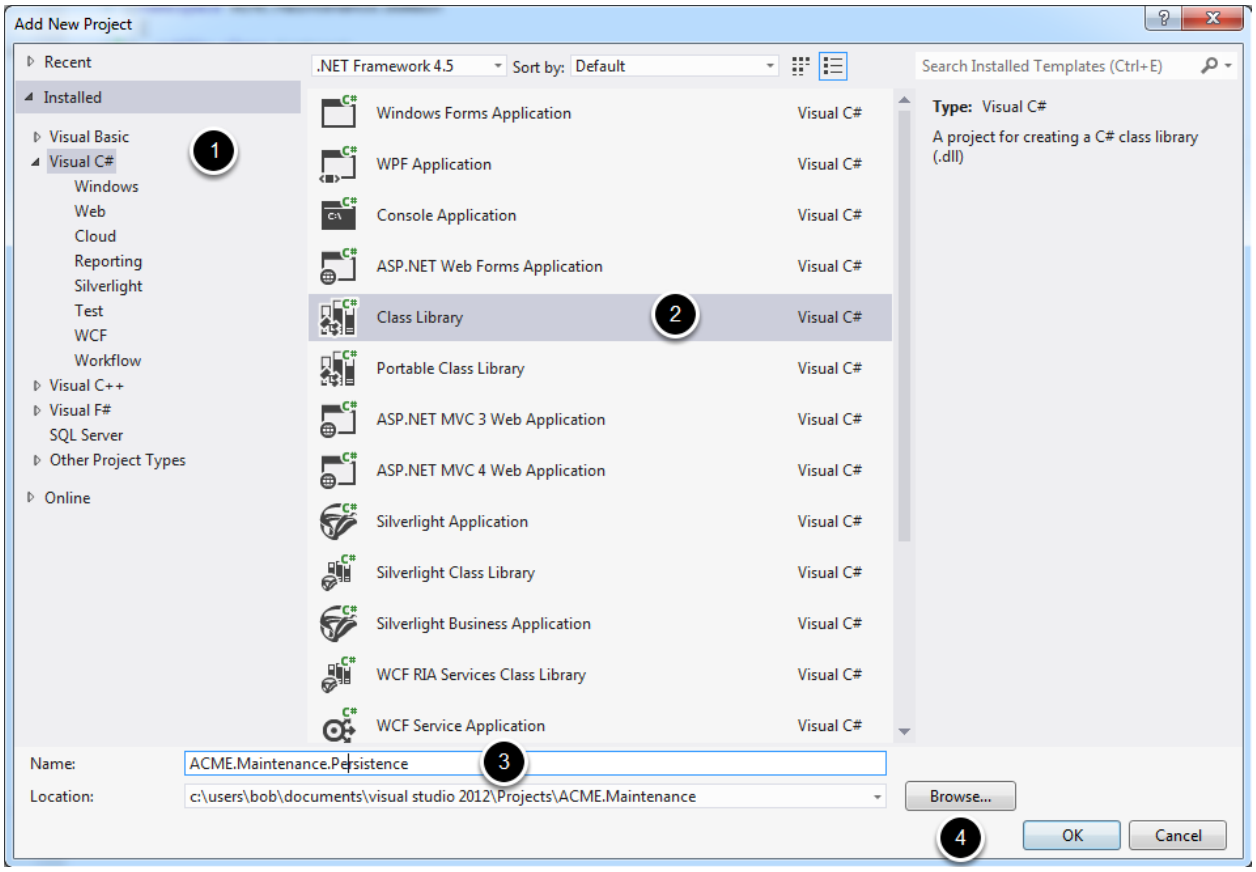
And he defines a "stub" as the following: "Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'."

So, what we need is a class that can act like a real persistence class, but we can hardcode the values we want in there. Yes, we're pushing this hardcoding off to another layer to worry about it another time, and that's fine.

So, I'm going to create a new project called ACME.Maintenance.Persistence and I'll implement a class here that will hardcode the values ... but I'll give you a little foreshadowing ... I won't be satisfied with this. Hardcoding may work with this particular use case we've created, but I can foresee that we'll want a more flexible type of stub later and this won't do. However, I'm taking this step so you can see the thought process. I'll wind up deleting that class, and then re-using this new project much later to implement the REAL persistence layer. Ok, pretend you never heard all that. Just follow along and let's pretend we're thinking this through and it hasn't occurred to us yet that this stub isn't going to be what we ultimately want.

1. Create a new ACME.Maintenance.Persistence project representing the Persistence Layer

In the Solution Explorer, right-click the ACME.Maintenance solution and select Add New | Project ... context menu option. When the Add New Project dialog appears ...


Add New Project
Recent
Installed
Visual Basic
Visual C#
Windows
Web
Cloud
Reporting
Silverlight
Test
WCF
Workflow
Visual C++
Visual F#
SQL Server
Other Project Types
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
ASP.NET Web Forms Application Visual C#
Class Library Visual C#
Portable Class Library Visual C#
ASP.NET MVC 3 Web Application Visual C#
ASP.NET MVC 4 Web Application Visual C#
Silverlight Application Visual C#
Silverlight Class Library Visual C#
Silverlight Business Application Visual C#
WCF RIA Services Class Library Visual C#
WCF Service Application Visual C#
Browse...
OK
Cancel
1
2
3
4

Type: Visual C#

A project for creating a C# class library (.dll)

Name: ACME.Maintenance.Persistence

Location: c:\users\bob\documents\visual studio 2012\Projects\ACME.Maintenance

(1) Select Visual C#

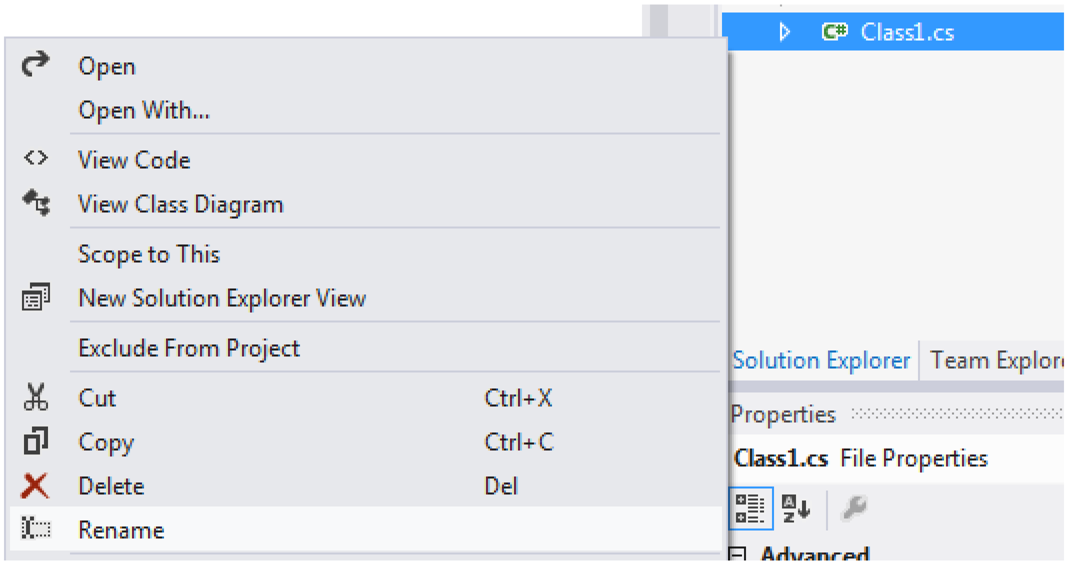
(2) Select Class Library project template

(3) Rename to: ACME.Maintenance.Persistence

(4) Select OK

2. Rename Class1 to ContractRepository

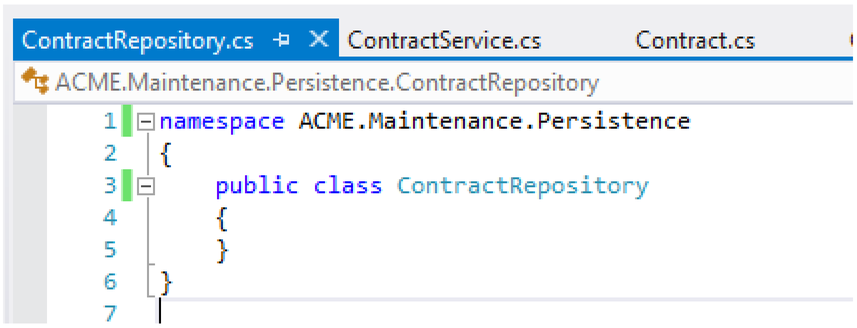
Rename Class1.cs to ContractRepository.cs



(Right-click the Class1.cs file in the ACME.Maintenance.Persistence project, select Rename)

Rename the class inside of that file to:

ContractRepository



```
namespace ACME.Maintenance.Persistence
{
    public class ContractRepository
    {
    }
}
```

... when I use the term Repository I have a very specific style / pattern in mind. Instead of getting sidetracked on that right now, I'll explain it in a mini-series called Repositories. You can watch it now, or watch it after this series to learn more about the specifics of Repositories.

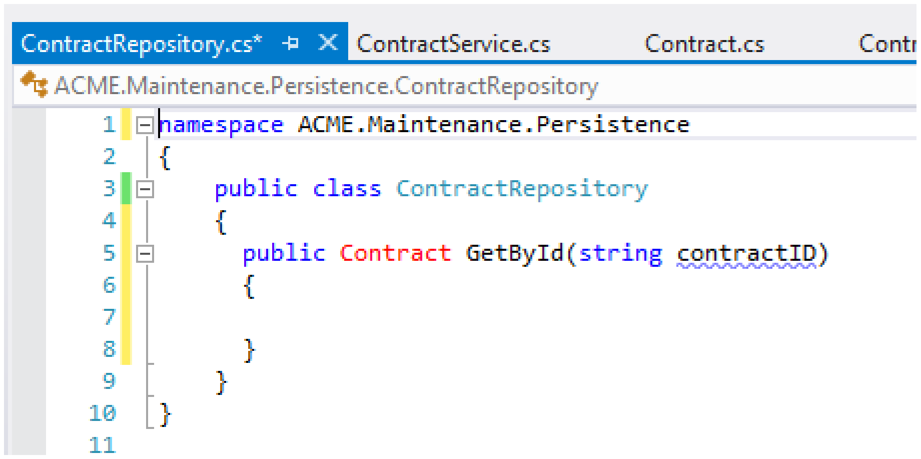
In a nutshell, the repository class will have the following methods:

- GetById()
- Other GetBy___() methods
- Save()
- Update()
- Delete()
- GetListBy____() methods

... all of these are optional, and the key is: what you include in here depends on what your Domain class needs. The domain class will drive the interface of the Repository class. That's all I'm going to say about Repositories for now. It's a term that means "the class that handles data related operations for a single domain class". As with all things, there's variation and disagreements on how a Repository should operate. I'll explain more in the mini-series on Repositories.

Next, I need to implement the GetById() method and consider how it will work / interact with the domain layer.

I begin by designing the method signature of GetById():



ContractRepository.cs* ContractService.cs Contract.cs Contr

ACME.Maintenance.Persistence.ContractRepository

```
namespace ACME.Maintenance.Persistence
{
    public class ContractRepository
    {
      public Contract GetById(string contractID)
      {

      }
    }
}
```

This is what I would like to create in my stubbed class ... I want to return a Contract. But I don't have a reference to the ACME.Maintenance.Domain class.

I have a decision to make here. Should I return an instance of Contract here? Or should I return a data transfer object? "It depends".

In this simple case, I could easily just handle the database access, grab the record in the database that corresponds with the contractId input parameter, create a new Contract, and pass it back. But I don't feel like this is the right course of action.

Creating an instance of Contract in this Repository class suffers from the violation of the Single Responsibility Principle. It's not only grabbing the data from the database, but will also be responsible for creating an instance of the object. That's already the job of the ContractService on the Domain layer.

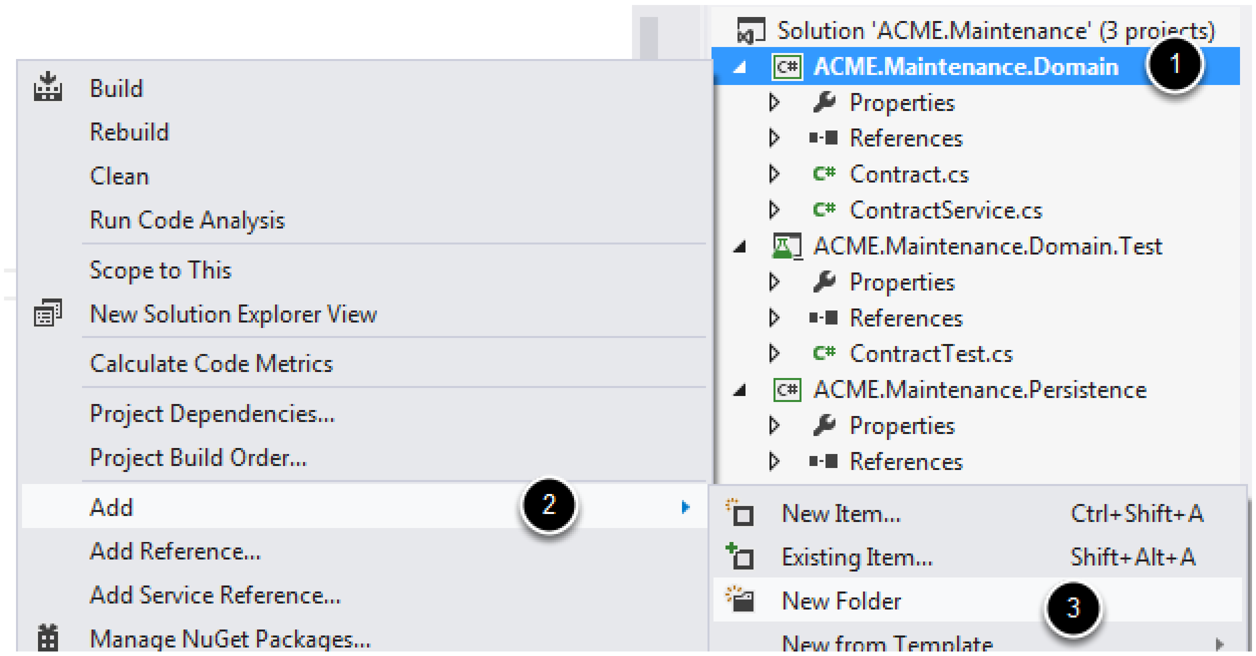
The other option is to create a data transfer object. We talked about these in the Application Architecture Fundamentals series. On the one hand, using a DTO makes sense in this case because it allows us to pass something back -- a simple data container -- with just the data we need. If we were to ultimately distribute this application across physical tiers in our enterprise, we would want to minimize the chatter between layers, and could use DTO's as a means of sending just the right data back to minimize round trips and data sizes.

On the other hand, in this pithy example, it seems like overkill because the properties of the DTO will match the properties of the actual Domain class. But for that matter, this entire architecture may be overkill -- again, I'm asking for you to use your imagination a bit here ... we want to pretend like this is enterprise scale, so keep in mind that if you have a simpler application, going to all this work of separating responsibilities and delegating them to different classes and layers may be over engineering things. I want to show you how, but you need to use your judgement on when to scale back based on your own application.

Ok, so we've decided to introduce data transfer objects to pass data back from our Persistence layer to our Domain layer. Where do we define these?

3. Add a DTO folder in the Domain Layer Project

We have a couple of options, but for simplicity's sake, we can put them in our domain layer. I would probably add a folder to keep all the DTO's together and in their own namespace.

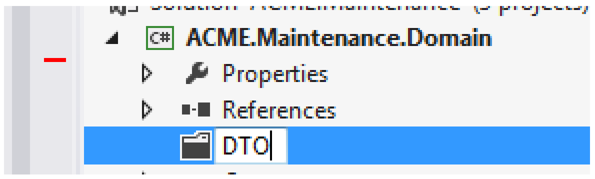


(1) Right-click on the ACME.Maintenance.Domain project

(2) Select Add

(3) New Folder

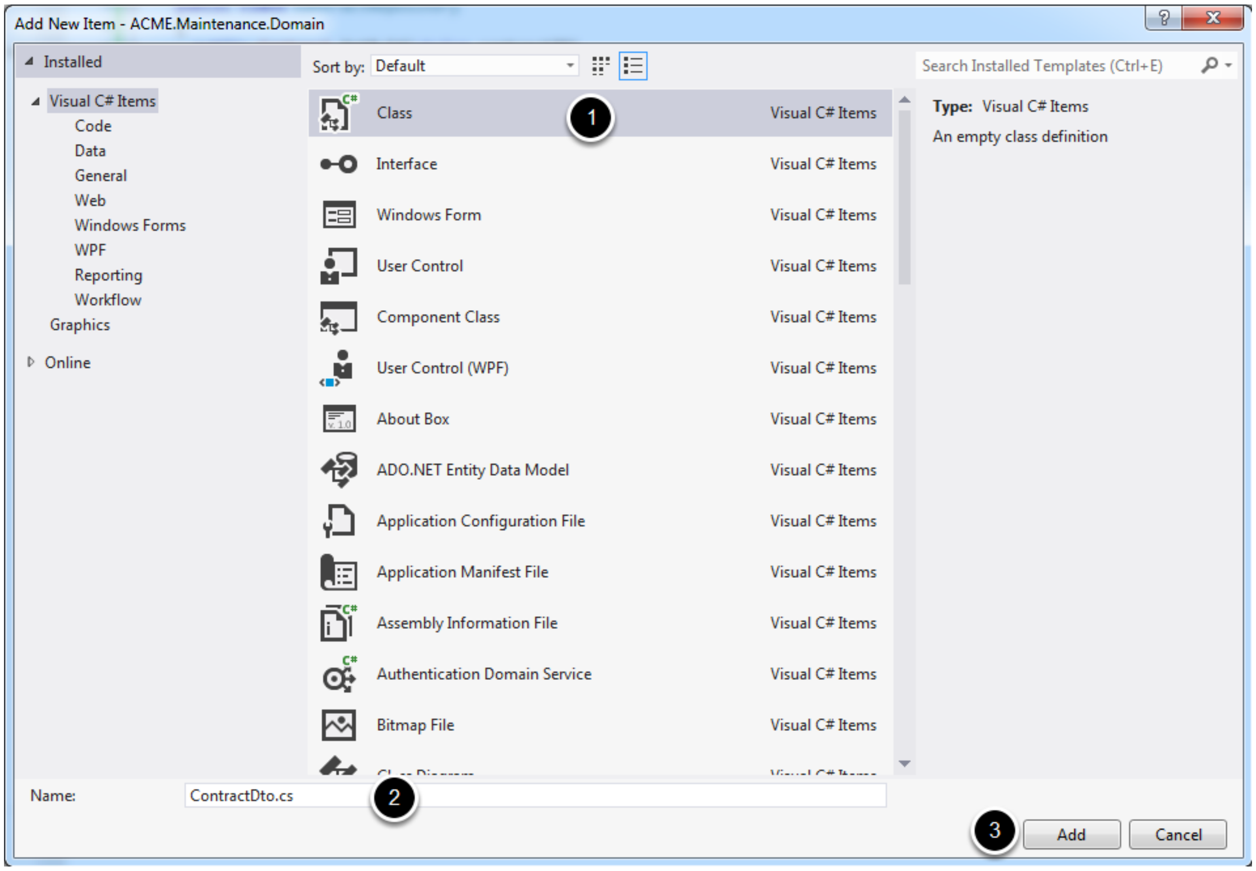
Right-click the new Folder1 and select Rename from the context menu. Change the name to: DTO



4. Add ContractDto.cs to the DTO Folder

Right-click the new DTO folder, select Add | Class...

In the Add New Item dialog...



Add New Item - ACME.Maintenance.Domain
Installed
Visual C# Items
Code
Data
General
Web
Windows Forms
WPF
Reporting
Workflow
Graphics
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
Class — Visual C# Items
Interface — Visual C# Items
Windows Form — Visual C# Items
User Control — Visual C# Items
Component Class — Visual C# Items
User Control (WPF) — Visual C# Items
About Box — Visual C# Items
ADO.NET Entity Data Model — Visual C# Items
Application Configuration File — Visual C# Items
Application Manifest File — Visual C# Items
Assembly Information File — Visual C# Items
Authentication Domain Service — Visual C# Items
Bitmap File — Visual C# Items
Type: Visual C# Items
An empty class definition
1
2
3
Add
Cancel

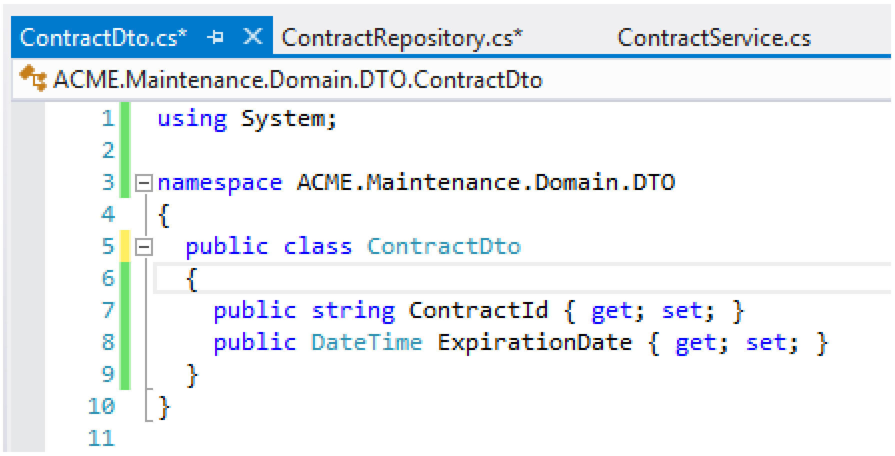
Name: ContractDto.cs

(1) Make sure Class file template is selected

(2) Rename to: ContractDto.cs

(3) Select Add

In this simple case, the ContractDto should have the same properties as the domain Contract class.



```
using System;

namespace ACME.Maintenance.Domain.DTO
{
    public class ContractDto
    {
        public string ContractId { get; set; }
        public DateTime ExpirationDate { get; set; }
    }
}

```

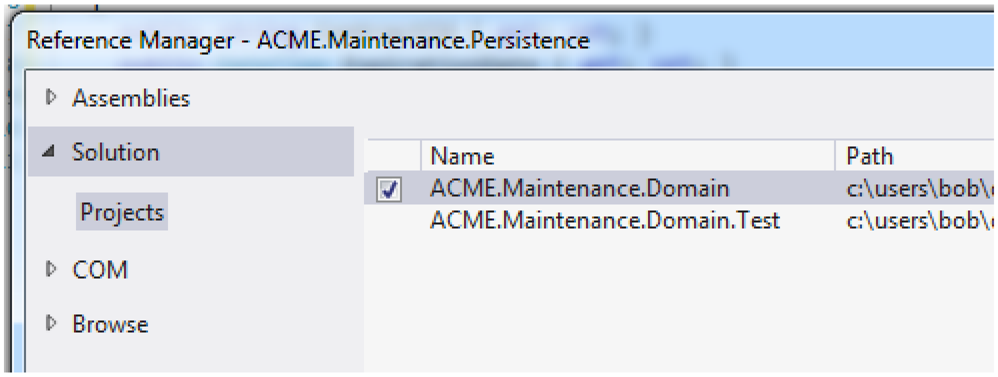
Get this code:

https://gist.github.com/LearnVisualStudio/46313ba66a780f327af4

5. Add Project Reference from Persistence Layer Project to Domain Layer Project

Right-click the ACME.Maintenance.Persistence project, select Add Reference from the context menu.

In the Reference Manager dialog, check the check box next to: ACME.Maintenance.Domain

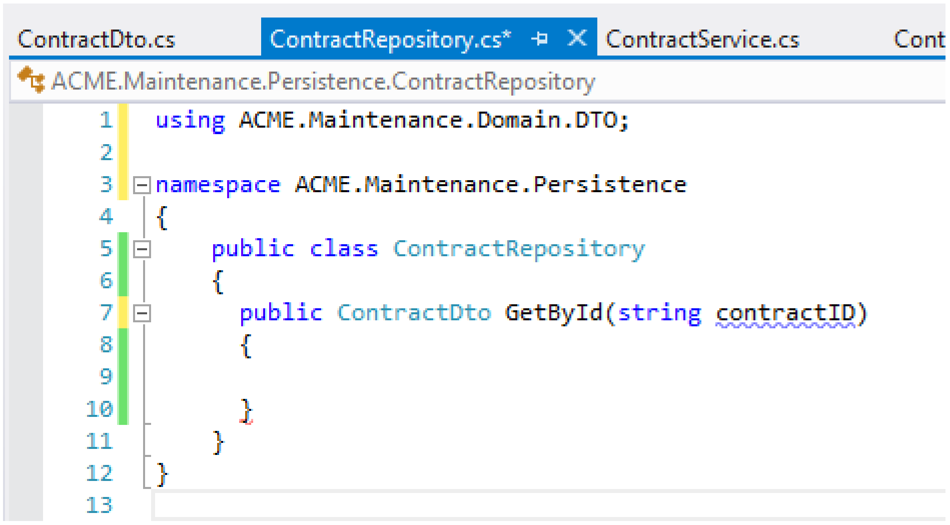


	Name	Path
✓	ACME.Maintenance.Domain	c:\users\bob\
	ACME.Maintenance.Domain.Test	c:\users\bob\

Close the Reference Manager dialog.

6. Modify the Return Type from the ContractRepository's GetById() method

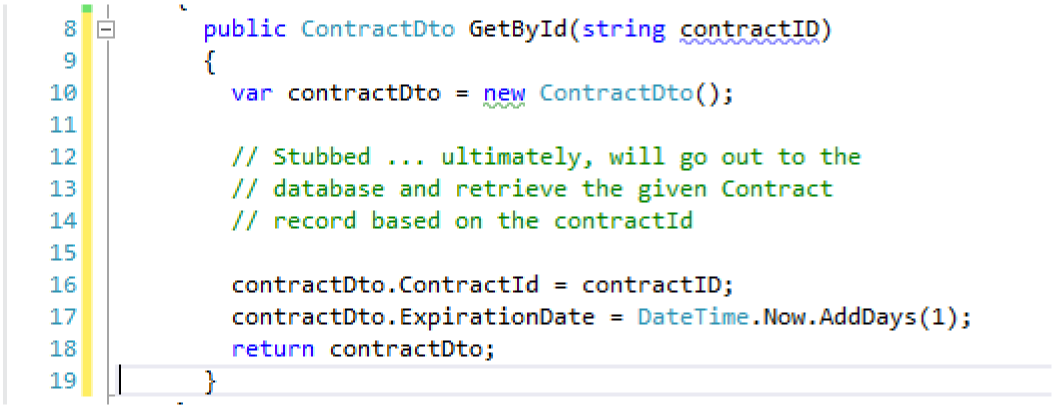
Now it's time to refactor the GetbyId() method's signature to return ContractDto.



```
using ACME.Maintenance.Domain.DTO;

namespace ACME.Maintenance.Persistence
{
    public class ContractRepository
    {
      public ContractDto GetById(string contractID)
      {

      }
    }
}

```

7. Implement the body of the ContractRepository's GetById() method

I'm now ready to implement the stubbed out version of GetById(). Again, it's hard coded to specific values, but this is in an effort to isolate the domain layer for testing. We'll come up with a better way to accomplish this later. But for now, here's the code I write:



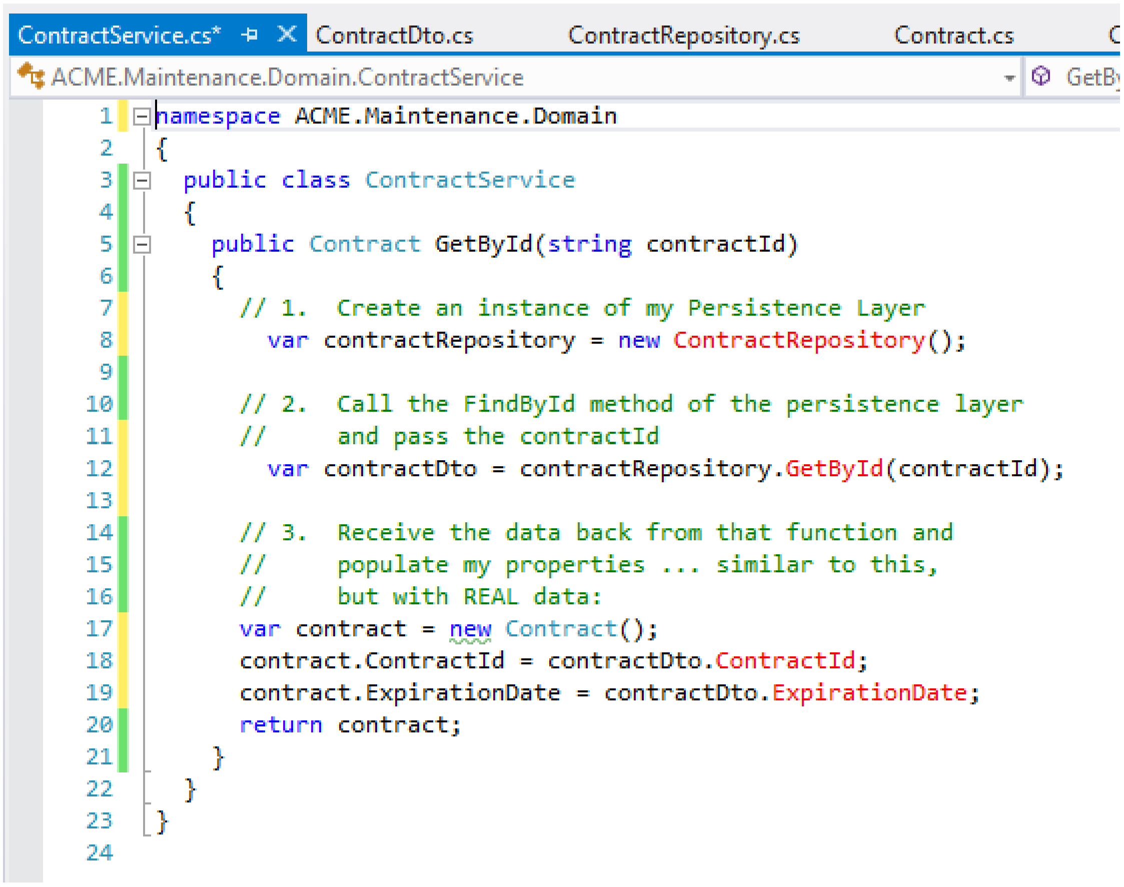
```
      public ContractDto GetById(string contractID)
      {
        var contractDto = new ContractDto();

        // Stubbed ... ultimately, will go out to the
        // database and retrieve the given Contract
        // record based on the contractId

        contractDto.ContractId = contractID;
        contractDto.ExpirationDate = DateTime.Now.AddDays(1);
        return contractDto;
      }
```

Get this code:

https://gist.github.com/LearnVisualStudio/0e893f5714838ccbe657

8. Modify the ContractService's GetById() method to utilize the new ContractRepository

Back in the ContractService class, I update the code to work with the stubbed ContractRepository.



```
namespace ACME.Maintenance.Domain
{
  public class ContractService
  {
    public Contract GetById(string contractId)
    {
      // 1.  Create an instance of my Persistence Layer
        var contractRepository = new ContractRepository();

      // 2.  Call the FindById method of the persistence layer
      //     and pass the contractId
        var contractDto = contractRepository.GetById(contractId);

      // 3.  Receive the data back from that function and
      //     populate my properties ... similar to this,
      //     but with REAL data:
      var contract = new Contract();
      contract.ContractId = contractDto.ContractId;
      contract.ExpirationDate = contractDto.ExpirationDate;
      return contract;
    }
  }
}
```

Get this code:

https://gist.github.com/LearnVisualStudio/558e127a376ecb9ca386

As you can see, I've not yet created a reference from my domain layer project to my new (stubbed) persistence layer project. If I were to do this, based on what you know from the Application Architecture Fundamentals series, what problem will I be introducing into my application? What will this prevent me from doing in the future? Think about this and we'll discuss it in the next lesson.

Lesson 10 - Using Dependency Injection to Break Layer Dependencies

In the previous lesson, we created a stubbed Persistence layer with a Respository class. Now, we need to call the method in that Repository class to retrieve a given Contract. However, I hesitated a moment ... I stopped short of creating a reference from the Domain layer project to the Persistence layer project. Do you know why?

If we implement the interaction between the layers as I've started doing by creating an instance of ContractRepository from the ContractService class that will create a hard dependency between the layers. I will never be able to use any other Persistence layer unless I re-write the code in the ContractService class. This will limit me in the short term as I try to Unit Test and in the long term as I want to change out the persistence layer per the requirements that we support other persistence mechanisms other than SQL Server.

Another way of looking at this is that my dependencies will go from the Domain Layer to the Persistence Layer. Ideally, I want the Persistence layer (as well as the Presentation layer) to have their dependencies pointing TOWARDS the Domain layer. I want everything ELSE to depend on the Domain layer, and the Domain layer should have its dependencies passed to it. This is known as dependency inversion, and so we'll use the Dependency Injection pattern to make that happen.

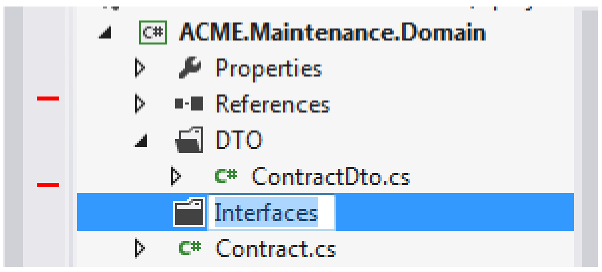
From a practical perspective, we'll want the client of our ContractService to tell it what it should use for Persistence. The ContractService should define an interface and say "I'm willing to work with any Repository class that supports these methods."

So, this will be a three step process. First, I'll create an interface "on behalf" of the ContractService class defining the methods it needs to get its work done. After I've defined that interface, I'll implement that interface in the Persistence layer ... I'll make sure my ContractRepository conforms to that interface. Finally, in the unit test, I'll create an instance of the stubbed ContractRepository and pass it to the ContractService. When I finish, my dependencies will be pointing in the right direction and my ContractService will operate like what I envision it should.

1. Create Interfaces folder in the ACME.Maintenance.Domain

In the Solution Explorer, right-click the ACME.Maintenance.Domain project, select Add | New Folder ...
Right-click the new folder, select Rename ... rename to:

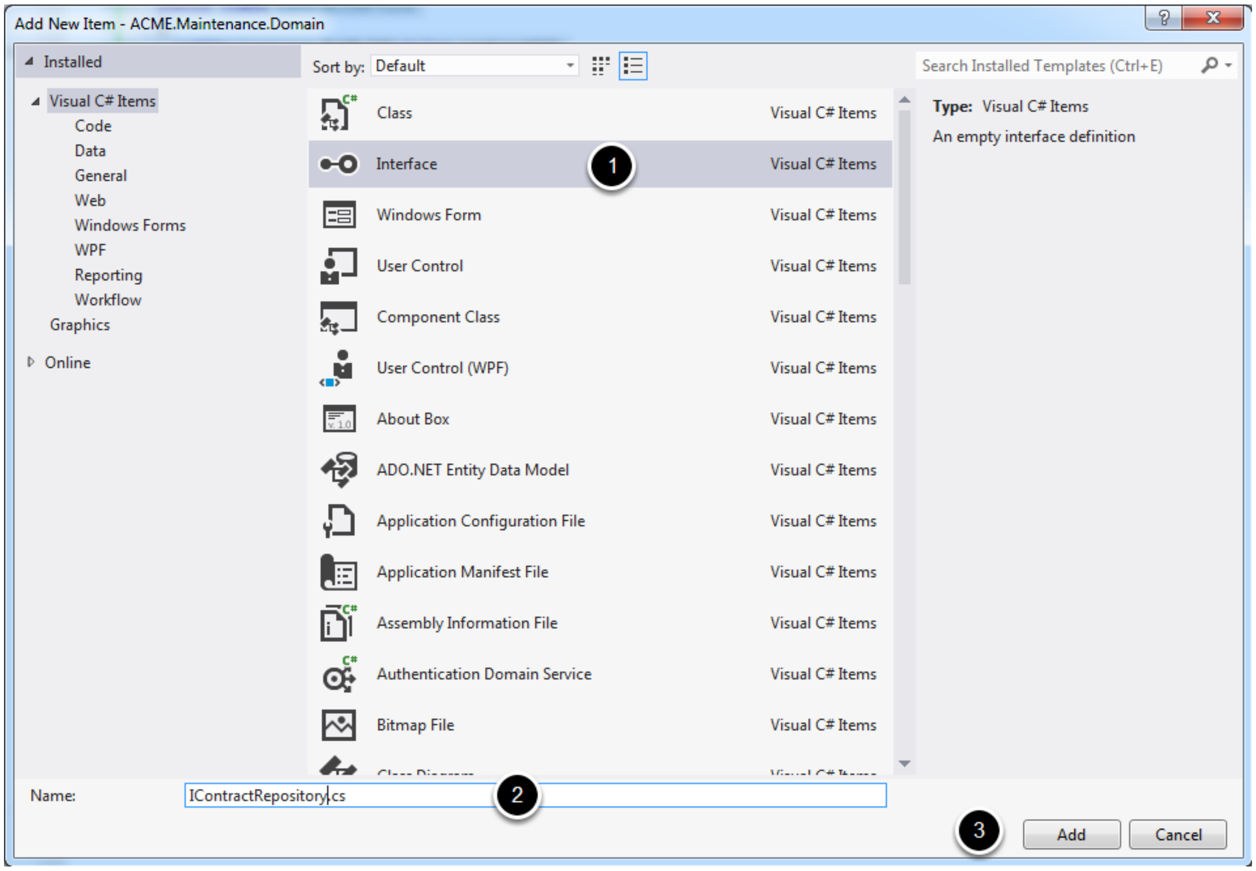
Interfaces



2. Add IContractRepository.cs to Interfaces folder

Right-click the new Interfaces folder, select Add | New Item...

In the Add New Item dialog...


Add New Item - ACME.Maintenance.Domain
Installed
Visual C# Items
Code
Data
General
Web
Windows Forms
WPF
Reporting
Workflow
Graphics
Online
Search Installed Templates (Ctrl+E)
Class
Interface
Windows Form
User Control
Component Class
User Control (WPF)
About Box
ADO.NET Entity Data Model
Application Configuration File
Application Manifest File
Assembly Information File
Authentication Domain Service
Bitmap File
Visual C# Items
1
2
3
Add
Cancel

Sort by: Default

Type: Visual C# Items

An empty interface definition

Name: IContractRepository.cs

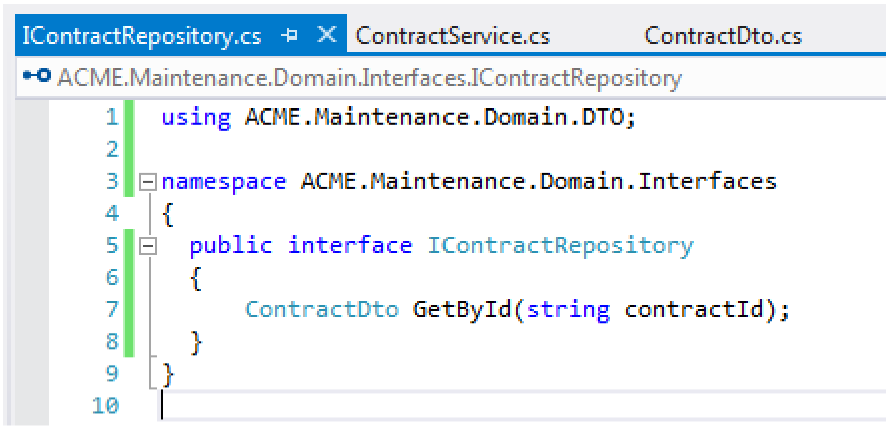
(1) Select the Interface template

(2) Rename to: IContractRespository.cs

(3) Select Add

3. Create the IContractRepository Interface by defining GetById()

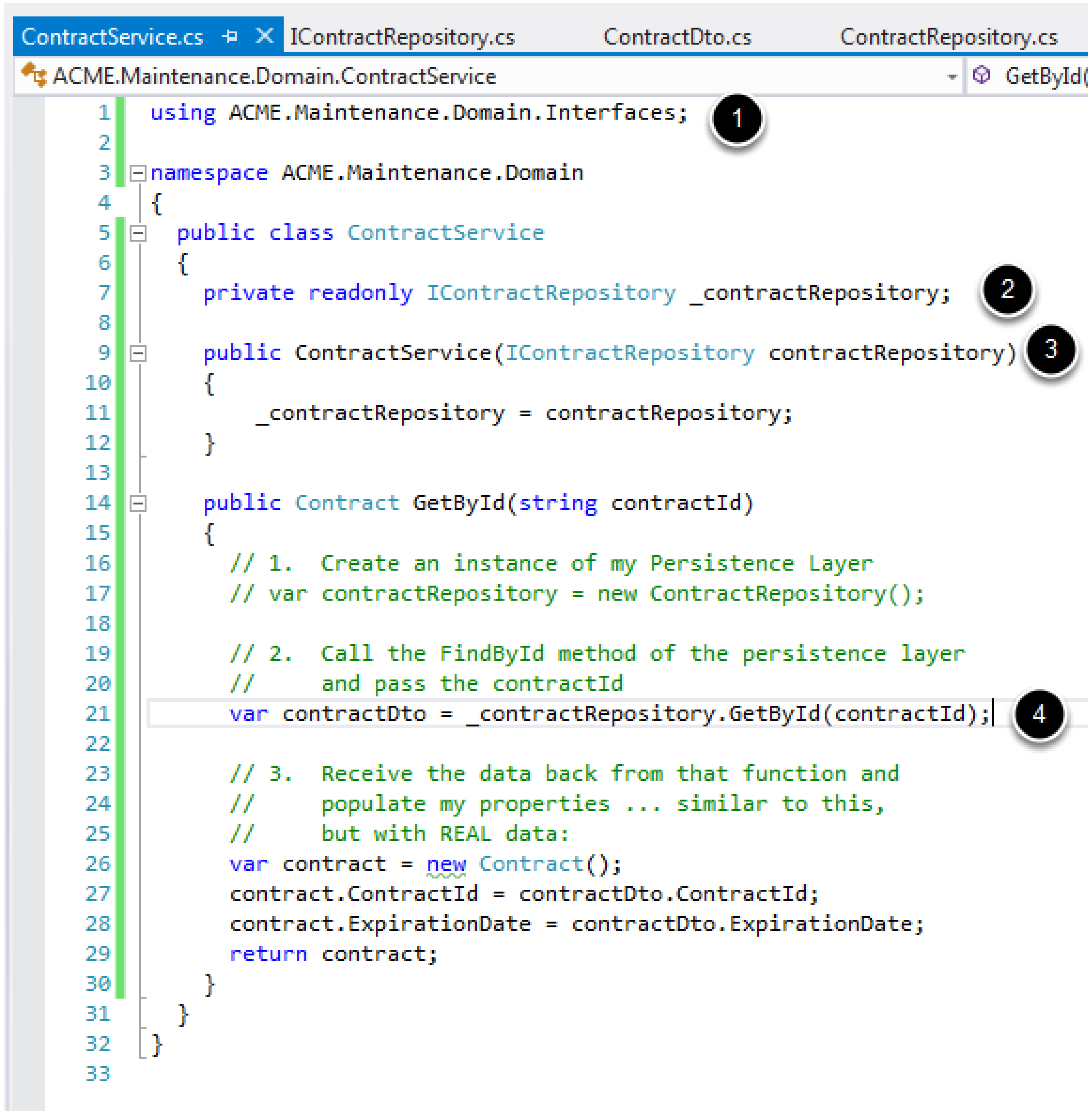
The interface should be simple ... it should merely define the GetById() method with the appropriate signature to accept a contractId and return a ContractDto. Here's what I come up with:



```
using ACME.Maintenance.Domain.DTO;

namespace ACME.Maintenance.Domain.Interfaces
{
    public interface IContractRepository
    {
        ContractDto GetById(string contractId);
    }
}
```

4. Implement Constructor Injection to allow client to pass in a Contract Repository that adheres to the Interface

We modify the ContractService to allow the ContractRepository to be injected dynamically via its constructor.



```
using ACME.Maintenance.Domain.Interfaces;

namespace ACME.Maintenance.Domain
{
    public class ContractService
    {
        private readonly IContractRepository _contractRepository;

        public ContractService(IContractRepository contractRepository)
        {
            _contractRepository = contractRepository;
        }

        public Contract GetById(string contractId)
        {
            // 1.  Create an instance of my Persistence Layer
            // var contractRepository = new ContractRepository();

            // 2.  Call the FindById method of the persistence layer
            //     and pass the contractId
            var contractDto = _contractRepository.GetById(contractId);

            // 3.  Receive the data back from that function and
            //     populate my properties ... similar to this,
            //     but with REAL data:
            var contract = new Contract();
            contract.ContractId = contractDto.ContractId;
            contract.ExpirationDate = contractDto.ExpirationDate;
            return contract;
        }
    }
}
```

(1) Add the using statement for the Interfaces namespace

(2) Create a private member variable that will hold on to the reference of the ContractRepository it is given in its constructor. I make it readonly so that it can only be set at construction (in the constructor) and never changed.

(3) I create the constructor accepting an input parameter of type IContractRepository. Then in the body of the constructor, I set that input parameter to my private member variable which I'll use throughout the body of the class whenever I need to reference the injected ContractRepository.

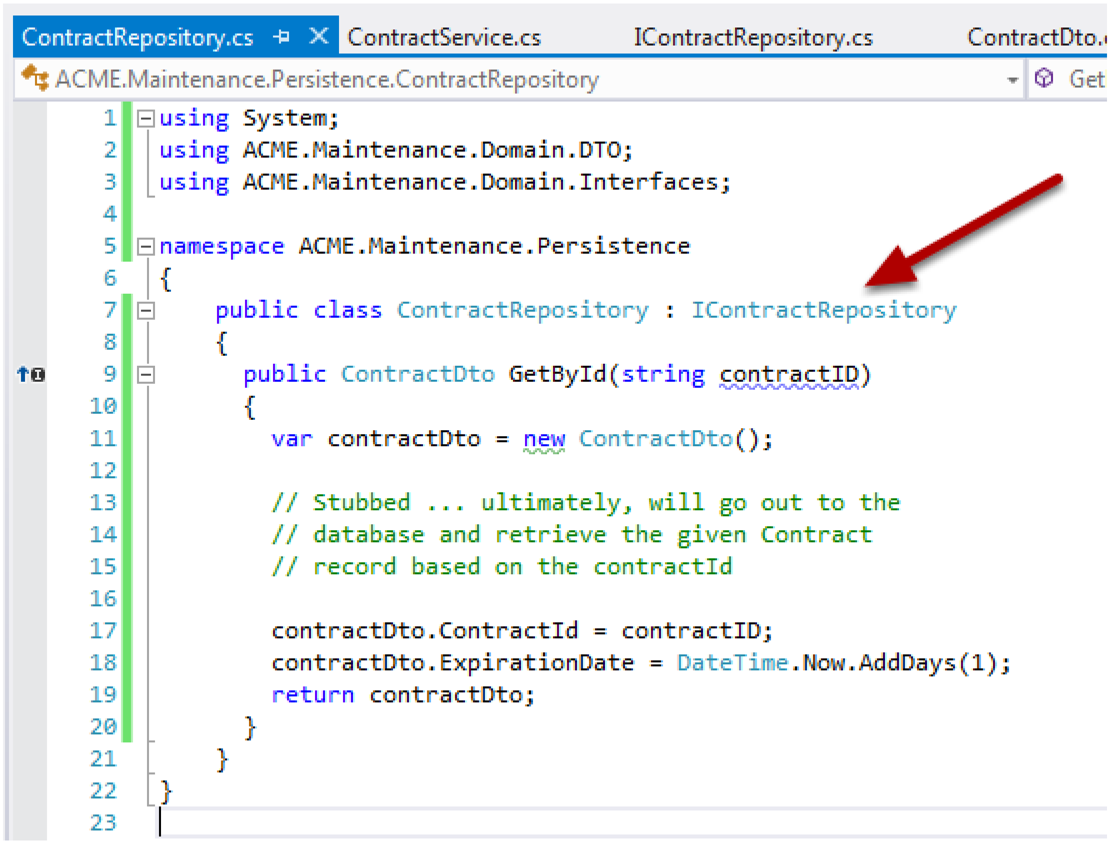
(4) I modify the body of the GetById() method. I comment out the hard coded reference to the concrete ContractRepository (line 17) from the Persistence layer project, and instead use the private member variable (line 21) to reference the injected version.

Get this code:

https://gist.github.com/LearnVisualStudio/32597a63e7de335c89f4

5. ContractRepository should now implement IContractRepository

Next, I modify the ContractRepository so that it implements the IContractRepository interface.



```
using System;
using ACME.Maintenance.Domain.DTO;
using ACME.Maintenance.Domain.Interfaces;

namespace ACME.Maintenance.Persistence
{
    public class ContractRepository : IContractRepository
    {
      public ContractDto GetById(string contractID)
      {
        var contractDto = new ContractDto();

        // Stubbed ... ultimately, will go out to the
        // database and retrieve the given Contract
        // record based on the contractId

        contractDto.ContractId = contractID;
        contractDto.ExpirationDate = DateTime.Now.AddDays(1);
        return contractDto;
      }
    }
}
```

Notice Line 7 where I add the following:

: IContractRepository

... meaning that the ContractRepository should implement the IContractRepository. No other changes to this class are needed because it already implements the GetById() method with the same method signature as defined in the new interface.

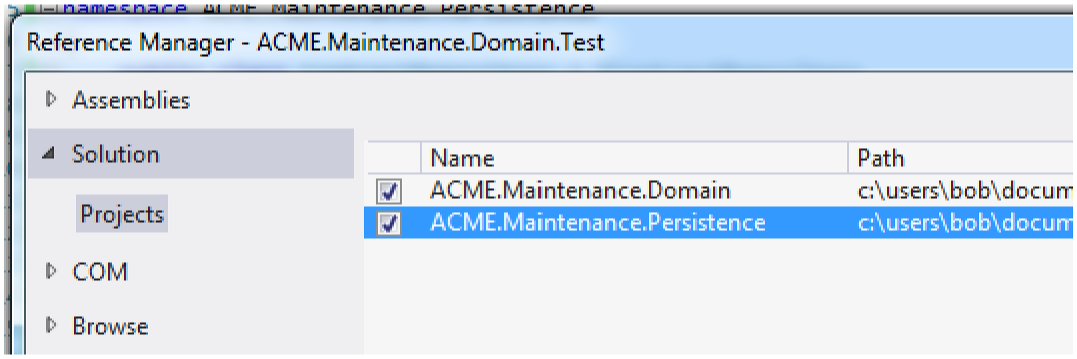
Get this code:

https://gist.github.com/LearnVisualStudio/2633fb34cd702f3fa090

6. In the Test project, add reference to Persistence project

Right-click the ACME.Maintenance.Domain.Test project in the Solution Explorer, select Add Reference...

In the Reference Manager, add a check mark in the box next to ACME.Maintenance.Persistence.

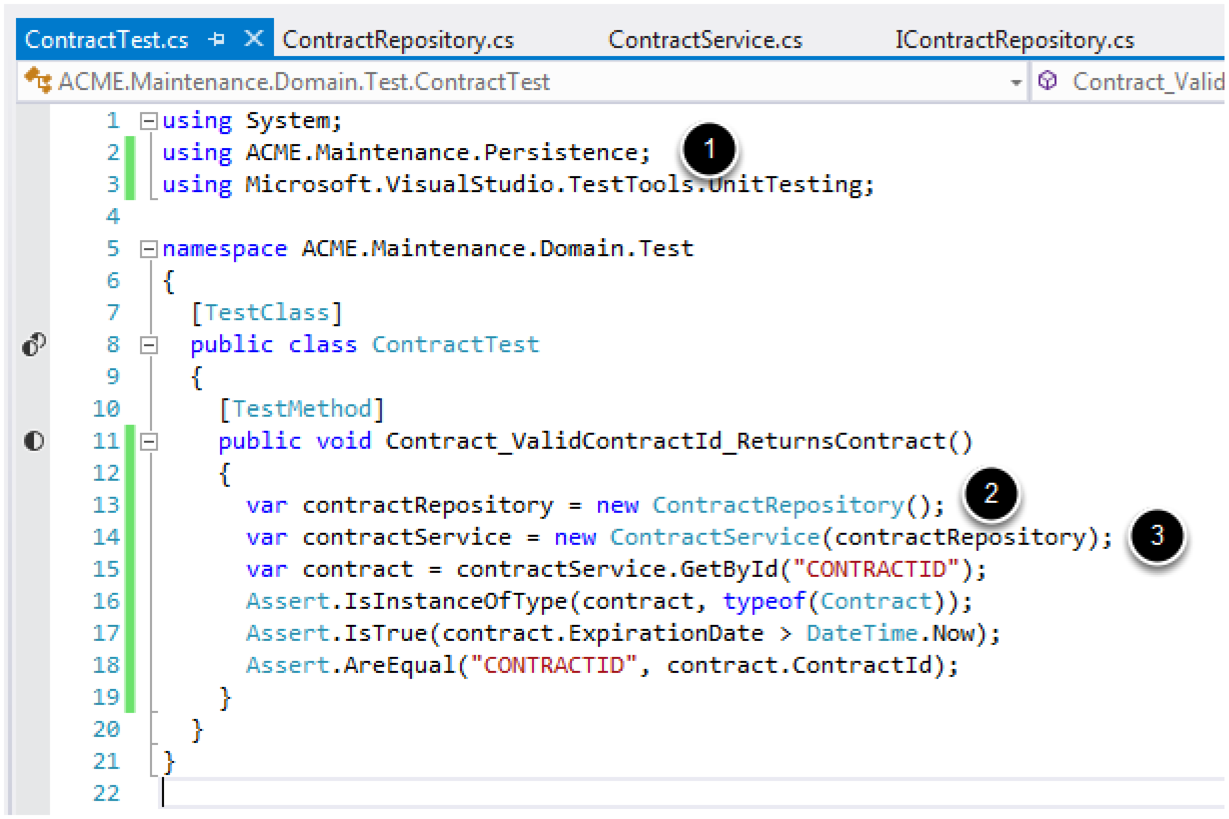

namespace ACME.Maintenance.Persistence
Reference Manager - ACME.Maintenance.Domain.Test
Assemblies
Solution
Projects
COM
Browse

	Name	Path
✓	ACME.Maintenance.Domain	c:\users\bob\docum
✓	ACME.Maintenance.Persistence	c:\users\bob\docum

Why? Because we'll perform the injection of the concrete ContractRepository in the Unit Test. Therefore, we now need a reference to the Persistence Layer project from our Unit Test project.

7. Modify the Unit Test to Inject the ContractRepository into the ContractService

Now, to pull it all together ... I will create a new instance of ContractRepository and inject it into the ContractService's constructor at runtime.



```
using System;
using ACME.Maintenance.Persistence;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ACME.Maintenance.Domain.Test
{
  [TestClass]
  public class ContractTest
  {
    [TestMethod]
    public void Contract_ValidContractId_ReturnsContract()
    {
      var contractRepository = new ContractRepository();
      var contractService = new ContractService(contractRepository);
      var contract = contractService.GetById("CONTRACTID");
      Assert.IsInstanceOfType(contract, typeof(Contract));
      Assert.IsTrue(contract.ExpirationDate > DateTime.Now);
      Assert.AreEqual("CONTRACTID", contract.ContractId);
    }
  }
}
```

(1) I add a using statement to the Persistence namespace

(2) I create a concrete instance of the ContractRepository class

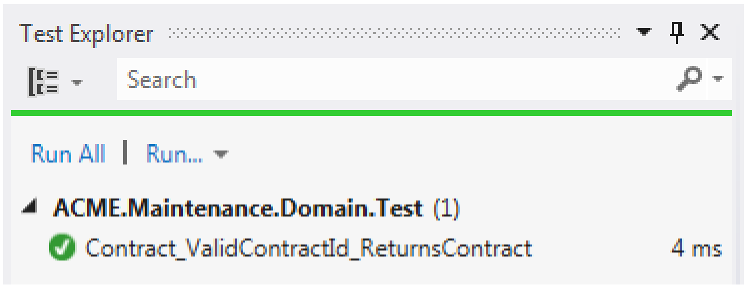
(3) I pass it as an input parameter to the ContractService

Get this code:

https://gist.github.com/LearnVisualStudio/b5d3f3414d604f2805a7

8. Run All Unit Tests

Finally, to confirm it's working, we'll Run All unit tests...



ACME.Maintenance.Domain.Test (1)
Contract_ValidContractId_ReturnsContract 4 ms

It works. We successfully decoupled our layers. This is exactly the type of thing that we need to be vigilant about while we build our application in order to ensure separation of concerns and a host of other related best practices and patterns.

But again, there's a problem that lurks around the corner. What is the problem with this implementation? What decisions have we made that will limit us going forward?

Hint: I already warned you about this! But can you pinpoint what we've done wrong and why?

Another hint: look at the unit test we've created. It's passing in "CONTRACTID" and expecting "CONTRACTID" back. But what if we passed in a different contract id? Will we get the results we expect? We'll figure it out in the next lesson.

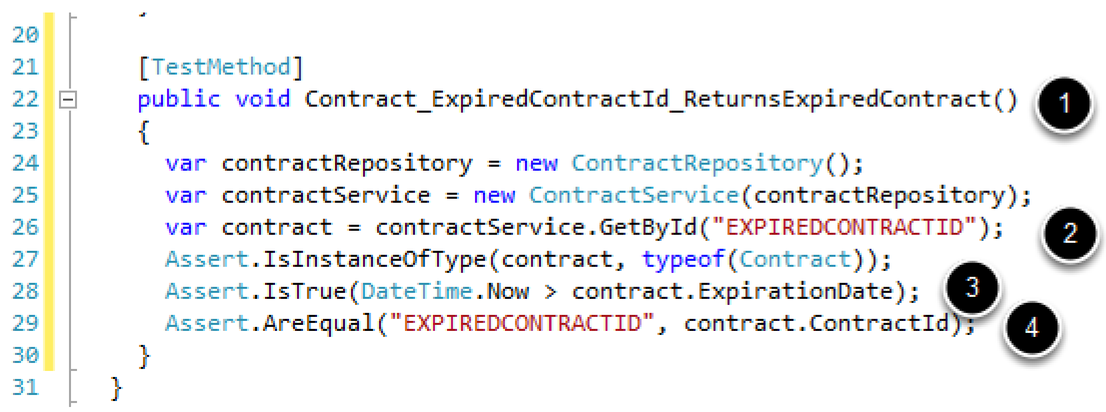
Lesson 11 - Introducing Fakes and FakeItEasy to better isolate our domain

In the previous lesson, I asked you to find another problem in my code. I hinted that, in our current unit test, we're looking for a hard-coded string "CONTRACTID" and expecting "CONTRACTID" back. But what would happen if we tried a different contract id? Would we get the resutls we expected?

To answer that question, in this lesson we'll start by creating a test for expired contracts. So, our ContractRepository stub should be able to return both non-expired and expired contracts. Let's test that.

1. Copy, Paste & Modify the original Unit Test to now test for Expired Contracts

I copy, paste and modify the previous test to check for an expired contract. Here's the code ...



```
        [TestMethod]
        public void Contract_ExpiredContractId_ReturnsExpiredContract()
        {
            var contractRepository = new ContractRepository();
            var contractService = new ContractService(contractRepository);
            var contract = contractService.GetById("EXPIREDCONTRACTID");
            Assert.IsInstanceOfType(contract, typeof(Contract));
            Assert.IsTrue(DateTime.Now > contract.ExpirationDate);
            Assert.AreEqual("EXPIREDCONTRACTID", contract.ContractId);
        }
    }
```

(1) I modify the name of the Unit Test I copied and pasted to communicate the intent of the Unit Test ... namely, to see what happens when I test Expired contracts.

(2) I change the contractId to "EXPIREDCONTRACTID"

(3) I test to make sure that the contract is indeed expired by changing the conditions I'm testing ... in this case, a valid test would find DateTime.Now > contract.ExpirationDate

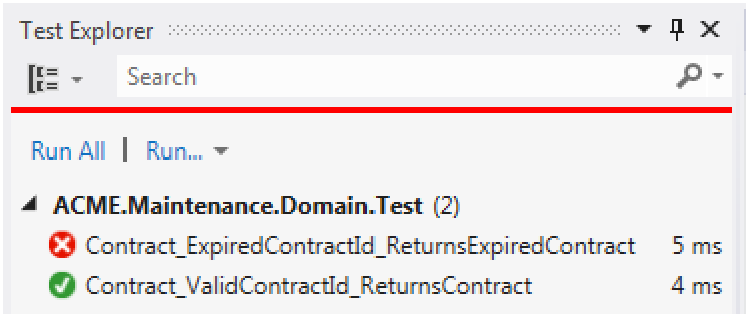
(4) Again, I change the contractId to "EXPIREDCONTRACTID"

Get this code:

https://gist.github.com/LearnVisualStudio/333d17b0f001d4c63684

2. Run All Unit Tests

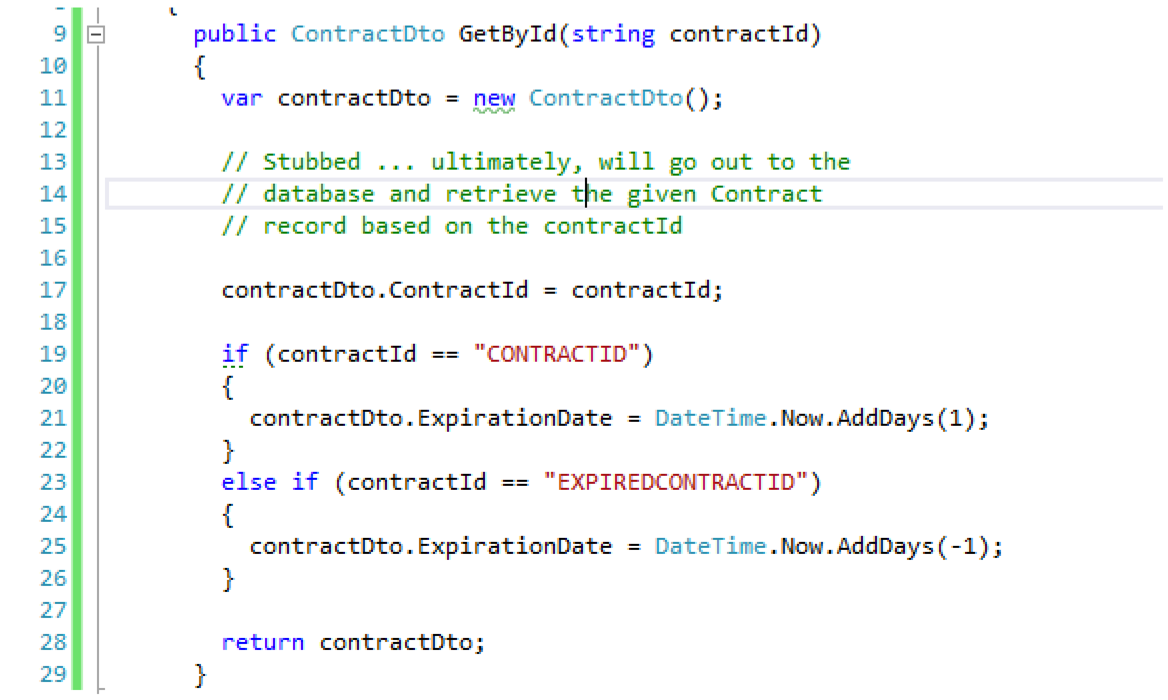
However, when I run the tests, the new unit test fails (see the red x next to the new unit test's name). This is because our current ContractRepository is hard coded to creating contracts with expiration dates in the future.



```
ACME.Maintenance.Domain.Test (2)
  Contract_ExpiredContractId_ReturnsExpiredContract   5 ms
  Contract_ValidContractId_ReturnsContract            4 ms
```

3. Modify the ContractRepository to account for Expired Contracts

Now, we *could* add a little more logic into our ContractRepository to check the contractId and set the date appropriately...

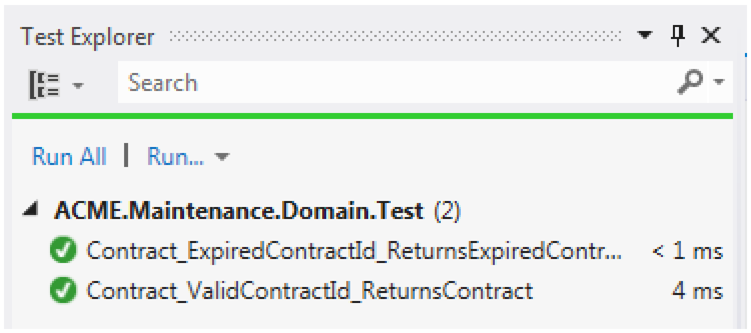


```
public ContractDto GetById(string contractId)
{
  var contractDto = new ContractDto();

  // Stubbed ... ultimately, will go out to the
  // database and retrieve the given Contract
  // record based on the contractId

  contractDto.ContractId = contractId;

  if (contractId == "CONTRACTID")
  {
    contractDto.ExpirationDate = DateTime.Now.AddDays(1);
  }
  else if (contractId == "EXPIREDCONTRACTID")
  {
    contractDto.ExpirationDate = DateTime.Now.AddDays(-1);
  }

  return contractDto;
}
```

Get this code ... you don't really need it ... we'll be removing it later ... but if you want it:

https://gist.github.com/LearnVisualStudio/647d5140bc3ad9710bbb

And just to see if it works, I'll Run All unit tests ...



```
ACME.Maintenance.Domain.Test (2)
  Contract_ExpiredContractId_ReturnsExpiredContr...   < 1 ms
  Contract_ValidContractId_ReturnsContract              4 ms
```

... and while this works, we can already see the fallacy in this approach. Every time we want to test for new conditions, or use new contract id's, we've got to add a new "else if" statement. That's definitely one way to go, but it will only work in the simplest of cases, like the one we have here. Once our contract gets more complicated, we'll need a more robust approach.

So, we need a different type of Test Double ... the Stub has helped us out quite a bit, but it would be cool if we could utilize something with a little more power.

I'm going to use FakeItEasy ... it bills itself as a cross between a Fake and a Mock.

https://github.com/FakeItEasy/FakeItEasy

If you read Fowler's article "Mocks aren't Stubs":

http://martinfowler.com/articles/mocksArentStubs.html

... He uses the definition from Gerald Meszaro's book, "xUnit Test Patterns".

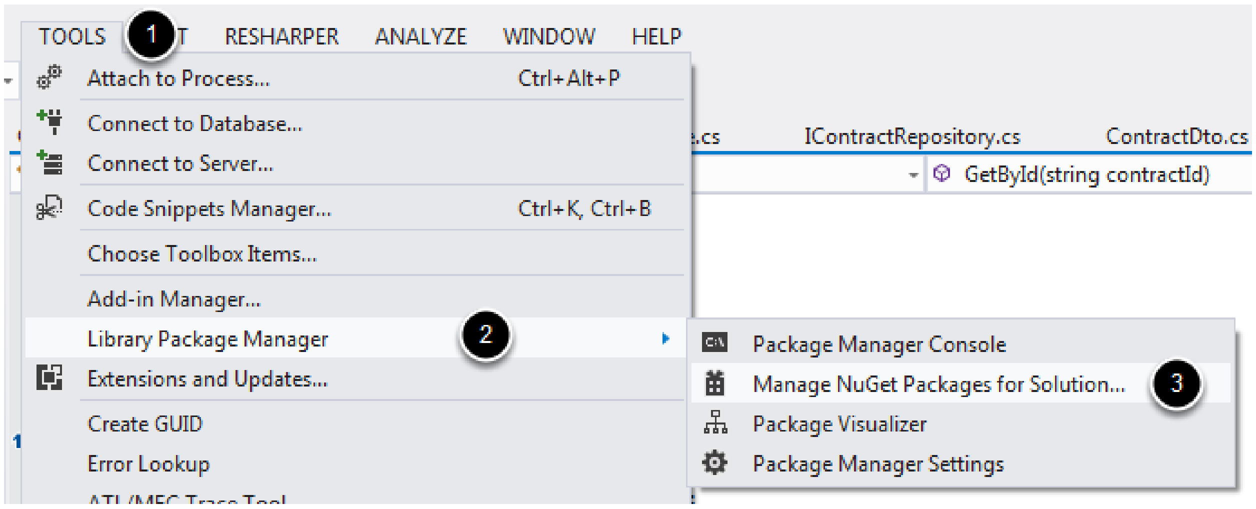
> "Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).
>
> ...
>
> Mocks are what we are talking about here: objects pre-programmed with expectations which form a specification of the calls they are expected to receive."

These are subtle distinctions that honestly I'm not sure I completely understand. However, the great thing about FakeItEasy is that you really don't need to understand it. It will allow you to isolate the exact scenario you want to tackle in your method under test.

4. Add FakeItEasy to the Unit Test Project

We'll use the NuGet to grab the files for the latest version of FakeItEasy and create references to it in our solution.

To open up NuGet (aka, the Library Package Manager):

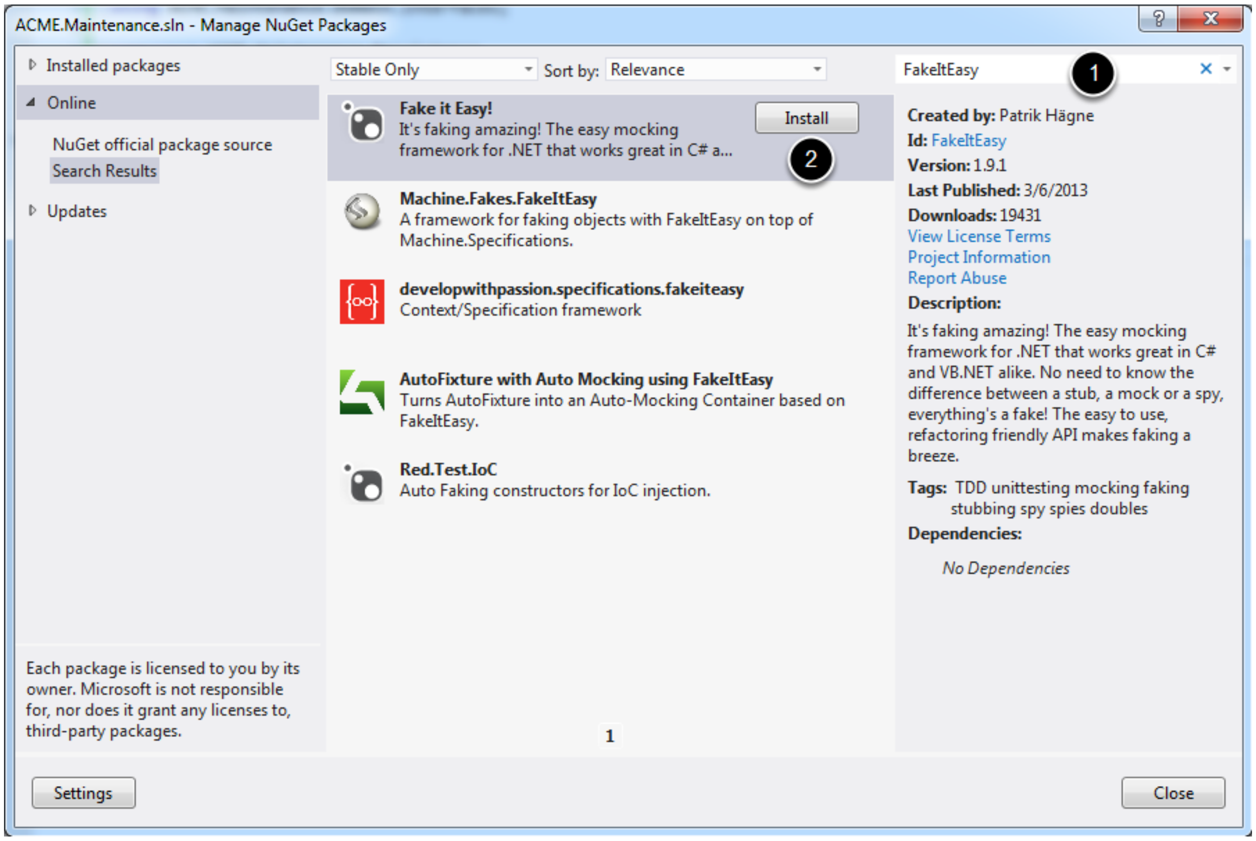


(1) Tools menu

(2) Library Package Manager

(3) Manage NuGet Packages for Solution ...

When the Manage NuGet Packages dialog pops open...



Fake it Easy!
It's faking amazing! The easy mocking framework for .NET that works great in C# a...

Machine.Fakes.FakeItEasy
A framework for faking objects with FakeItEasy on top of Machine.Specifications.

developwithpassion.specifications.fakeiteasy
Context/Specification framework

AutoFixture with Auto Mocking using FakeItEasy
Turns AutoFixture into an Auto-Mocking Container based on FakeItEasy.

Red.Test.IoC
Auto Faking constructors for IoC injection.

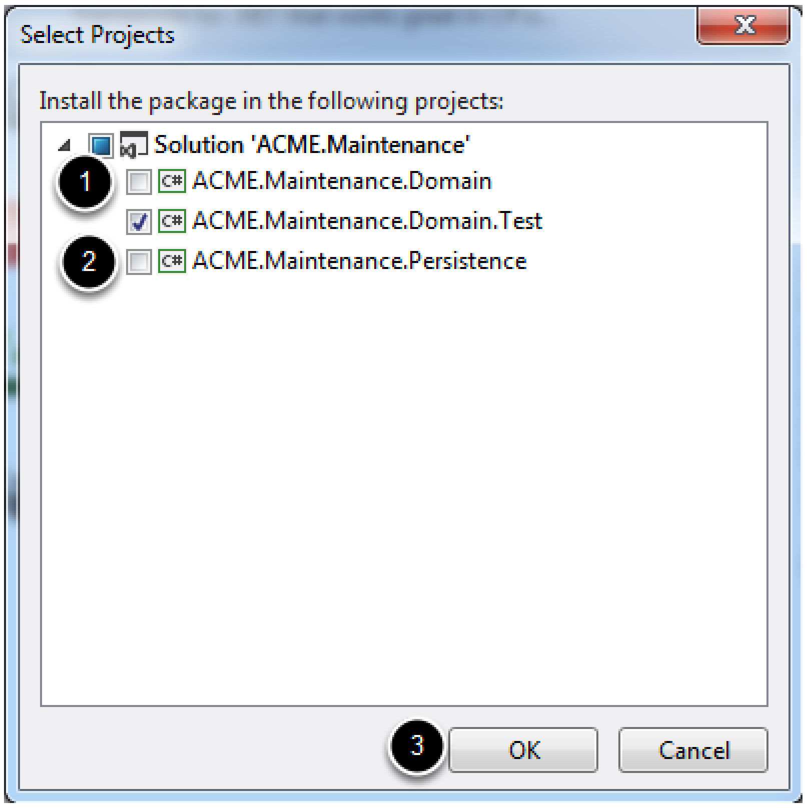
Created by: Patrik Hägne
Id: FakeItEasy
Version: 1.9.1
Last Published: 3/6/2013
Downloads: 19431
View License Terms
Project Information
Report Abuse
Description:
It's faking amazing! The easy mocking framework for .NET that works great in C# and VB.NET alike. No need to know the difference between a stub, a mock or a spy, everything's a fake! The easy to use, refactoring friendly API makes faking a breeze.
Tags: TDD unittesting mocking faking stubbing spy spies doubles
Dependencies:
No Dependencies

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

(1) Type "FakeItEasy" in the Seach box

(2) Click the Install button next to Fake It Easy

The Select Project dialog appears...


Select Projects
Install the package in the following projects:
Solution 'ACME.Maintenance'
1
ACME.Maintenance.Domain
ACME.Maintenance.Domain.Test
2
ACME.Maintenance.Persistence
3
OK
Cancel

(1) Deselect the ACME.Maintenance.Domain project

(2) Deselect the ACME.Maintenance.Persistence project

(3) Select OK

We only need Fake It Easy in our Unit Test project.

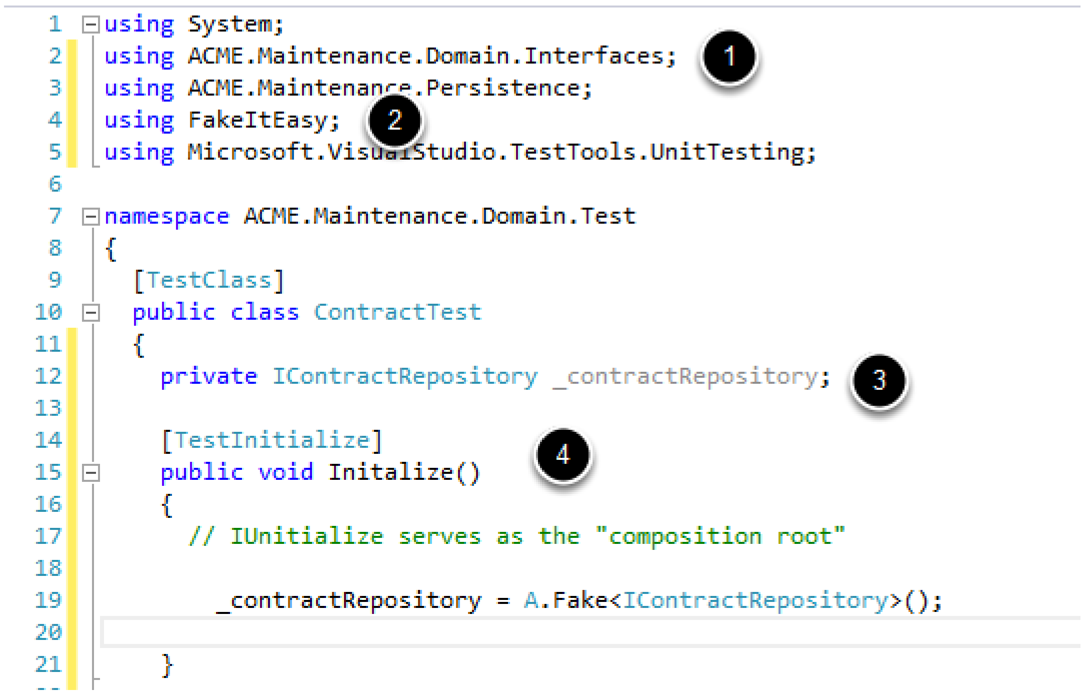
Finally, Close the Manage NuGet Packages dialog.

5. Modify the Unit Test to accommodate Fakes

Now the fun part.

We want to create a fake version of IContractRepository. Fake It Easy can look at the interface, and can dynamically create an object that conforms to that interface. But it's an imposter -- a FAKE because it has no real implementation!

To accomplish this, I write the following code:



```
using System;
using ACME.Maintenance.Domain.Interfaces;
using ACME.Maintenance.Persistence;
using FakeItEasy;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ACME.Maintenance.Domain.Test
{
  [TestClass]
  public class ContractTest
  {
    private IContractRepository _contractRepository;

    [TestInitialize]
    public void Initalize()
    {
      // IUnitialize serves as the "composition root"

        _contractRepository = A.Fake<IContractRepository>();

    }
```

(1) I add a using statement for the interfaces namespace

(2) I add a using statement for FakeItEasy namespace

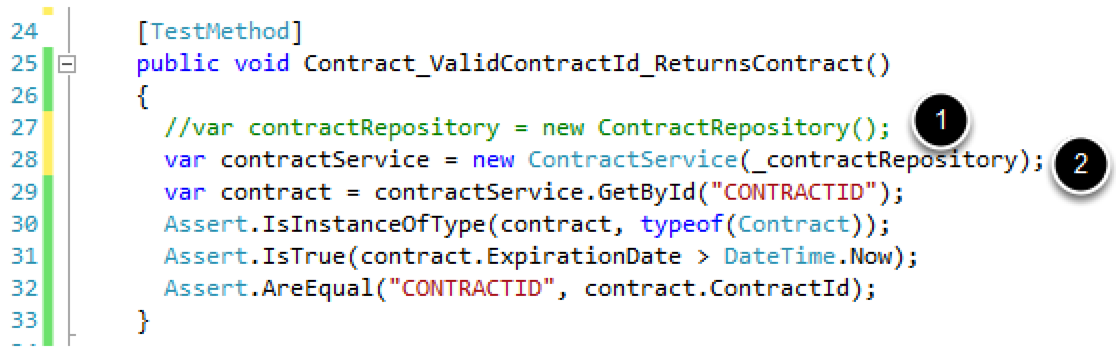
(3) I create a private member variable that will house our (fake) contract repository instance

(4) I create an Initialize method. By adorning it with the [TestInitialize] attribute, we're saying "run this once prior to running any of the test methods". Inside of that Initialize() method, we'll pull off the subterfuge by asking FakeItEasy to create a fake object based on IContractRepository.

Notice the "fuild style interface" of FakeItEasy. It reads like an English sentence:

"Set the contractRepository to a fake IContractRepository."

Furthermore, we'll now need to comb through the rest of the ContractTest.cs to look for concrete instantiation of the old ContractRepository and replace it with our new fake one.



```
[TestMethod]
public void Contract_ValidContractId_ReturnsContract()
{
  //var contractRepository = new ContractRepository();
  var contractService = new ContractService(_contractRepository);
  var contract = contractService.GetById("CONTRACTID");
  Assert.IsInstanceOfType(contract, typeof(Contract));
  Assert.IsTrue(contract.ExpirationDate > DateTime.Now);
  Assert.AreEqual("CONTRACTID", contract.ContractId);
}
```

(1) Comment out the creation of the concrete ContractRepository

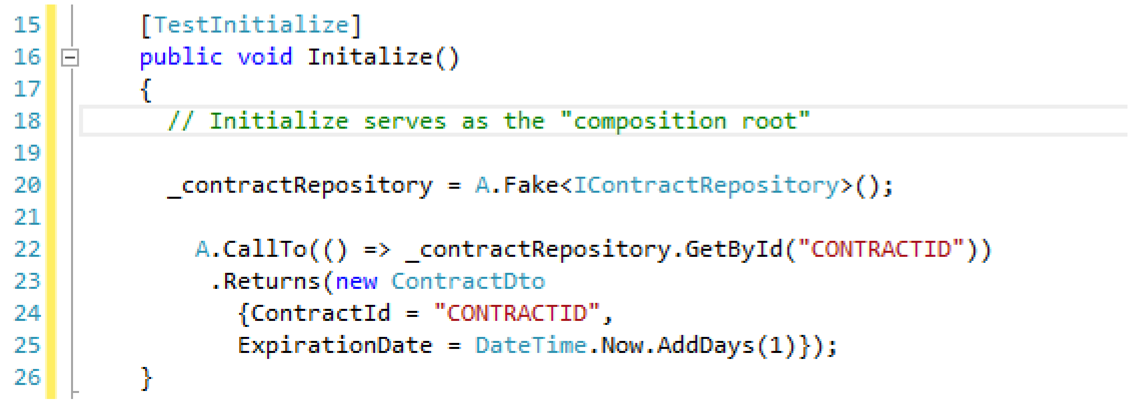
(2) Pass in the fake _contractRepository to the ContractService's constructor

Repeat this for the other unit test as well.

Now, we're no longer referencing the ACME.Maintenance.Persistence project ... we could remove it from the Solution completely and remove the using statement at the top of the file. However, because I know I'll need it again someday, I'll leave it for now. I will, however, remove it's using statement in our ContractTest.cs file.

6. Add an Expectation for the Valid ContractId

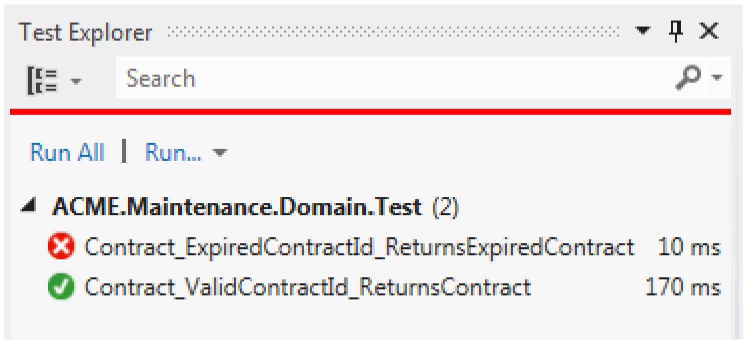
FakeItEasy needs more information to really pull off the deceit. It needs to know how to answer certain requests. So, I'll give it an expectation ... "Expect to be asked for ___. When a caller asks for it, give it _____."



```
    [TestInitialize]
    public void Initalize()
    {
      // Initialize serves as the "composition root"

      _contractRepository = A.Fake<IContractRepository>();

        A.CallTo(() => _contractRepository.GetById("CONTRACTID"))
         .Returns(new ContractDto
           {ContractId = "CONTRACTID",
           ExpirationDate = DateTime.Now.AddDays(1)});
    }
```

In this case, we're faking the Contract Repository, so when a class (most likely, the ContractService) calls the GetById method and passes "CONTRACTID" in, give it back a new ContractDto object that has been initialized with a valid contract id and expiration date. Pretty sneaky!

7. Run All Unit Tests



```
ACME.Maintenance.Domain.Test (2)
  ✗ Contract_ExpiredContractId_ReturnsExpiredContract   10 ms
  ✓ Contract_ValidContractId_ReturnsContract           170 ms
```

Running the Unit Tests works for the first unit test we built, the one testing ValidContractId ... so FakeItEasy was able to help us isolate the Domain Layer project by faking a dependency (the Contract Repository) allowing us to focus on just the code under test, namely, the ContractService. Now, we can be sure that the ContractService's GetById does work in the first case.

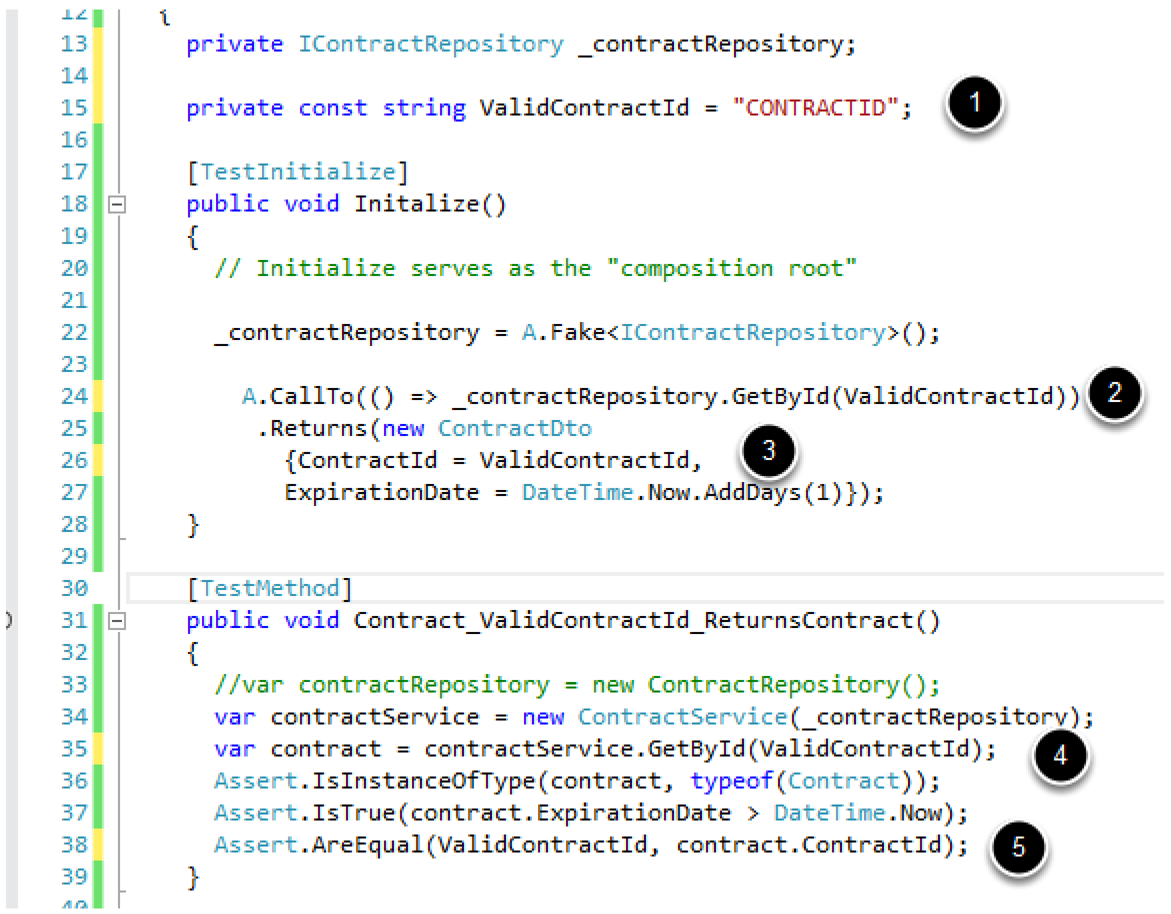
But something is amiss for the second case, the "Expired Contract Id" case. Why does this happen?

Because we've not set up the "Expired Contract Id" expectation in FakeItEasy. In other words, we've not told FakeItEasy to provide the caller with an Expired Contract object when it passes in a contractId of "EXPIREDCONTRACTID".

8. Refactor out the "magic strings" into a constant

Before tackling that, I decide to clean up the code a bit. I don't like having all those "magic strings" ... all of those literal strings. That's a risk. I'm likely to mis-type the text inside of the double-quotes and that will throw my Unit Tests off. I prefer keeping magic strings tucked away in a constant and used throughout the entire class.

Here's my code after refactoring:



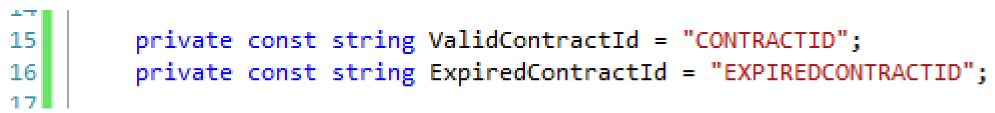
```
{
  private IContractRepository _contractRepository;

  private const string ValidContractId = "CONTRACTID";

  [TestInitialize]
  public void Initalize()
  {
    // Initialize serves as the "composition root"

    _contractRepository = A.Fake<IContractRepository>();

      A.CallTo(() => _contractRepository.GetById(ValidContractId))
       .Returns(new ContractDto
         {ContractId = ValidContractId,
         ExpirationDate = DateTime.Now.AddDays(1)});
  }

  [TestMethod]
  public void Contract_ValidContractId_ReturnsContract()
  {
    //var contractRepository = new ContractRepository();
    var contractService = new ContractService(_contractRepository);
    var contract = contractService.GetById(ValidContractId);
    Assert.IsInstanceOfType(contract, typeof(Contract));
    Assert.IsTrue(contract.ExpirationDate > DateTime.Now);
    Assert.AreEqual(ValidContractId, contract.ContractId);
  }
```

(1) I defined the const for the valid contract id

(2 through 5) I replace the "magic strings" with the new const

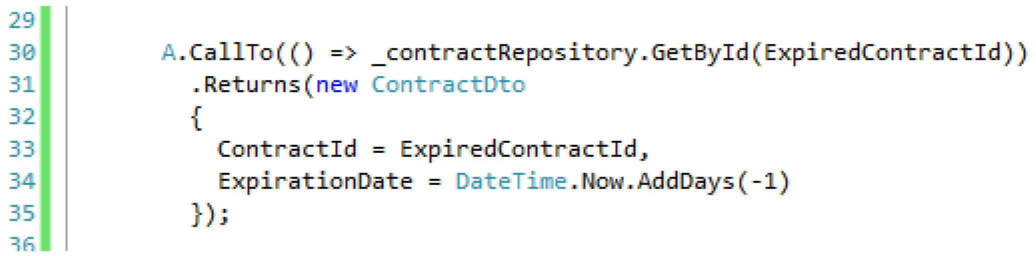
9. Implement the Expired ContractId

Now I'm ready to implement the scenario to handle the Expired contract id. I begin by defining a const immediately (line 16).



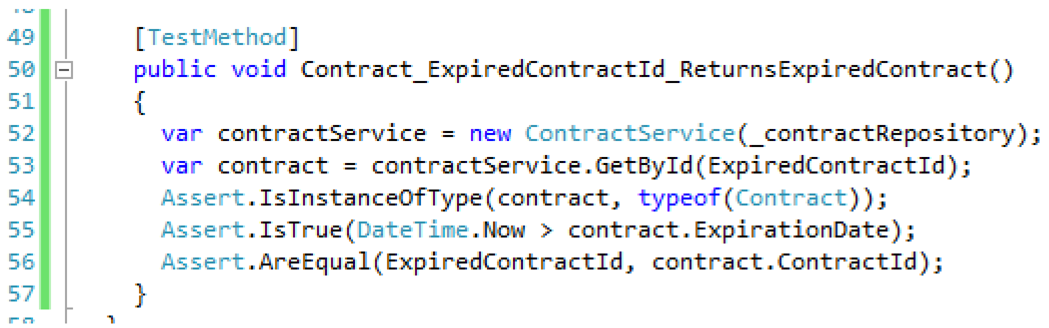
```
15      private const string ValidContractId = "CONTRACTID";
16      private const string ExpiredContractId = "EXPIREDCONTRACTID";
```

Next, in the Initialize() method, I implement an Expectation using my new const, making sure the ExpirationDate is correct (we DO want this to be in the past ... after all, this contract should be expired).



```
        A.CallTo(() => _contractRepository.GetById(ExpiredContractId))
          .Returns(new ContractDto
          {
            ContractId = ExpiredContractId,
            ExpirationDate = DateTime.Now.AddDays(-1)
          });
```

Finally, I modify the Expired version of the Unit Test to use the new const I defined earlier.

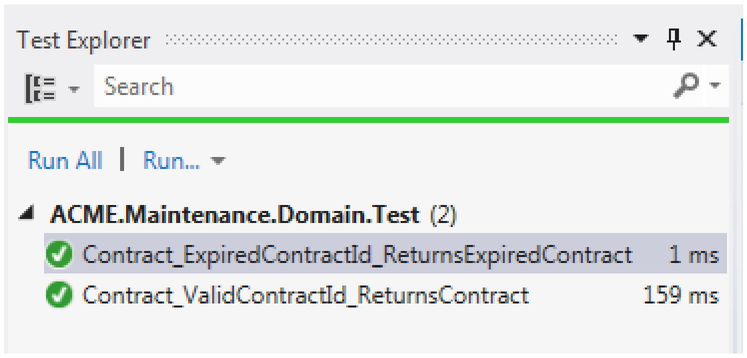


```
      [TestMethod]
      public void Contract_ExpiredContractId_ReturnsExpiredContract()
      {
        var contractService = new ContractService(_contractRepository);
        var contract = contractService.GetById(ExpiredContractId);
        Assert.IsInstanceOfType(contract, typeof(Contract));
        Assert.IsTrue(DateTime.Now > contract.ExpirationDate);
        Assert.AreEqual(ExpiredContractId, contract.ContractId);
      }
```

Get the entire ContractTest.cs code:

https://gist.github.com/LearnVisualStudio/4dfad35d1dc5766d2581

10. Run All Unit Tests



```
ACME.Maintenance.Domain.Test (2)
  Contract_ExpiredContractId_ReturnsExpiredContract    1 ms
  Contract_ValidContractId_ReturnsContract           159 ms
```

Success!

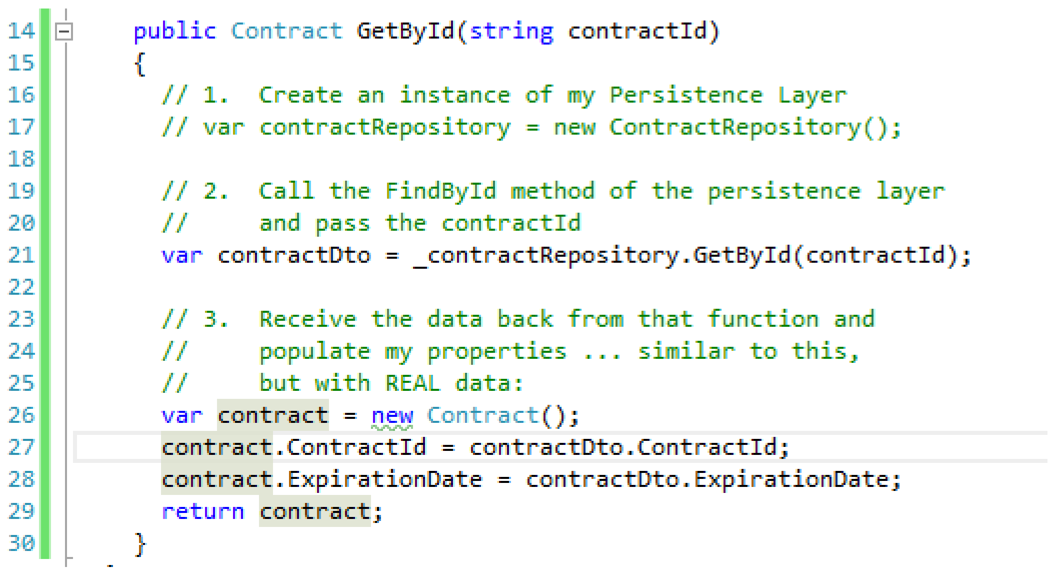
Ok, so now we've introduced FakeItEasy into our unit testing, providing us more options and greater flexibility to set an expectation and then allow FakeItEasy to make "happy noises" that will give our Domain class what it needs to function -- moreover, it allows us to ISOLATE the domain class and just test that, despite it's dependencies on classes in other layers. We've only scratched the surface with it. I'm going to create a mini-series on FakeItEasy and demo most of its functionality. But for now, we've got it configured and working.

Next up, I want to take a new look at our ContractService.cs class ... there's one more improvement I want to make there.

Lesson 12 - Introducing AutoMapper to Automatically Map Objects

Next, I'm going to introduce one more third-party library into our system. This is a bit gratuitous, but this addition will save me a lot of typing in the long run.

Remember from our Application Architecture Series that assuming we choose to use Data Transfer Objects to transfer data between layers, we'll need to constantly be mapping data in to and out of the data transfer objects. This takes the form of what you see in lines 27-28 (below, screenshot).



```
14      public Contract GetById(string contractId)
15      {
16        // 1.  Create an instance of my Persistence Layer
17        // var contractRepository = new ContractRepository();
18
19        // 2.  Call the FindById method of the persistence layer
20        //     and pass the contractId
21        var contractDto = _contractRepository.GetById(contractId);
22
23        // 3.  Receive the data back from that function and
24        //     populate my properties ... similar to this,
25        //     but with REAL data:
26        var contract = new Contract();
27        contract.ContractId = contractDto.ContractId;
28        contract.ExpirationDate = contractDto.ExpirationDate;
29        return contract;
30      }
```

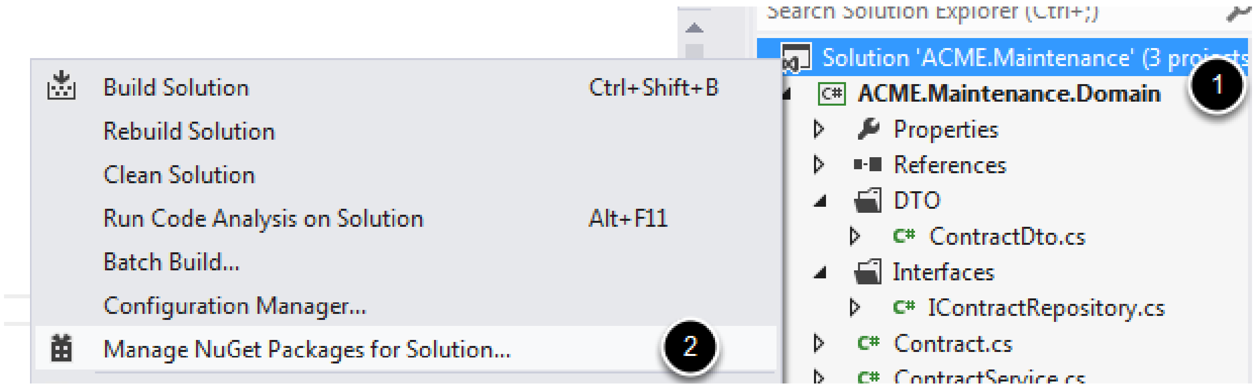
In this particular case, it is barely a problem. However, this is just our first of many mappings since we're just getting started AND it only has two properties. Imagine a dozen or twenty properties. It will require a lot of mind-numbingly boring typing.

To remedy this, we can utilize a third-party library called AutoMapper that, in many cases, will automatically map data between two classes ... and if it can't for some reason, you can tell it to match this property in this class with that property in that class.

1. Add AutoMapper to the Solution

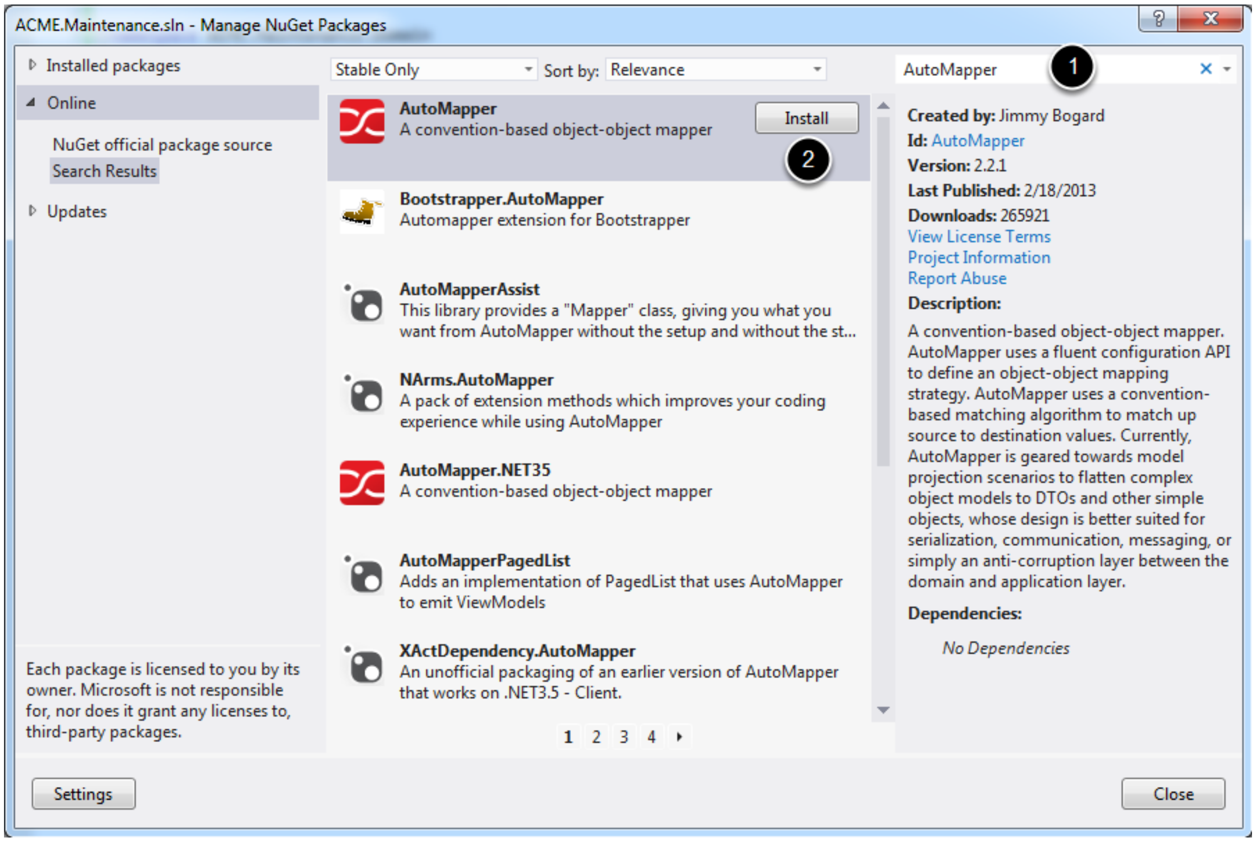
We'll use NuGet again, this time to add AutoMapper.

I'll use a new technique for opening up NuGet ... just to keep you on your toes.



(1) Right-click the Solution in the Solution Explorer

(2) Select Manage NuGet Packages for Solution ...



AutoMapper
A convention-based object-object mapper

Bootstrapper.AutoMapper
Automapper extension for Bootstrapper

AutoMapperAssist
This library provides a "Mapper" class, giving you what you want from AutoMapper without the setup and without the st...

NArms.AutoMapper
A pack of extension methods which improves your coding experience while using AutoMapper

AutoMapper.NET35
A convention-based object-object mapper

AutoMapperPagedList
Adds an implementation of PagedList that uses AutoMapper to emit ViewModels

XActDependency.AutoMapper
An unofficial packaging of an earlier version of AutoMapper that works on .NET3.5 - Client.

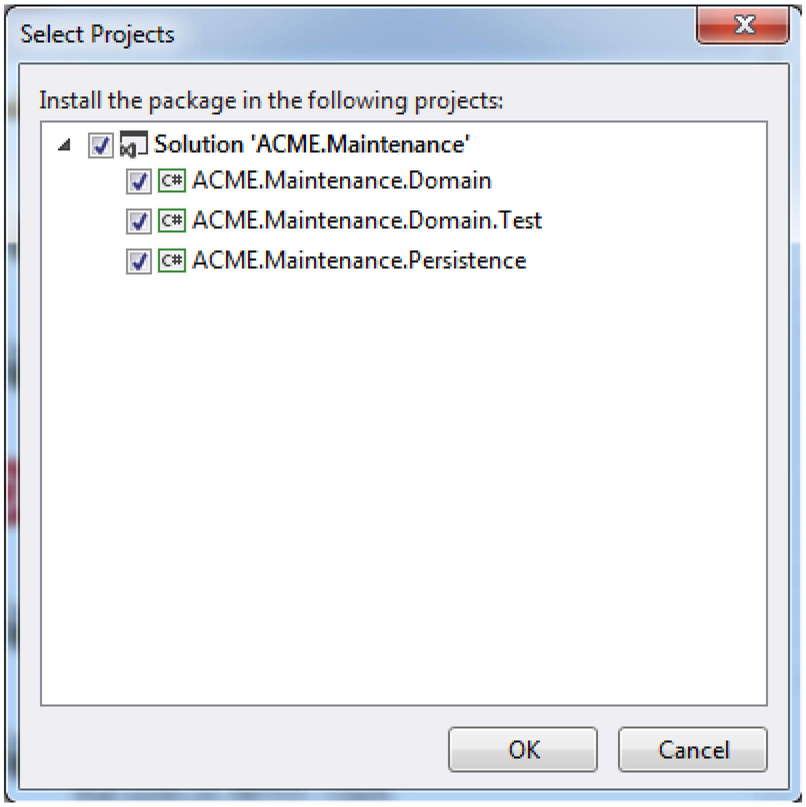
Created by: Jimmy Bogard
Id: AutoMapper
Version: 2.2.1
Last Published: 2/18/2013
Downloads: 265921
View License Terms
Project Information
Report Abuse
Description:
A convention-based object-object mapper. AutoMapper uses a fluent configuration API to define an object-object mapping strategy. AutoMapper uses a convention-based matching algorithm to match up source to destination values. Currently, AutoMapper is geared towards model projection scenarios to flatten complex object models to DTOs and other simple objects, whose design is better suited for serialization, communication, messaging, or simply an anti-corruption layer between the domain and application layer.
Dependencies:
No Dependencies

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

(1) Type "AutoMapper" in the search box, hit enter on the keyboard

(2) Click the Install button

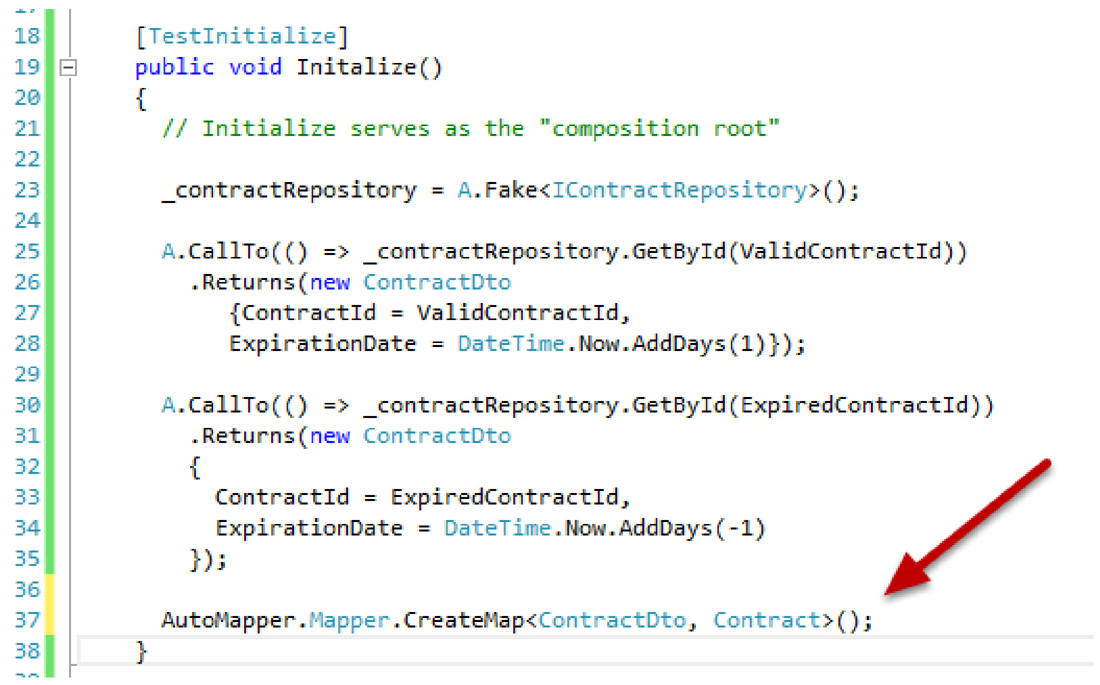
In the Select Projects dialog, make sure that AutoMapper is available to ALL of the projects by checking the check box next to each. Click OK.


Select Projects
Install the package in the following projects:
Solution 'ACME.Maintenance'
ACME.Maintenance.Domain
ACME.Maintenance.Domain.Test
ACME.Maintenance.Persistence
OK
Cancel

Close the NuGet Manager dialog.

2. Setup Automapper in the "composition root"

In the ContractTest.cs file, we'll want to create the mapping. This is simply an instruction that initializes AutoMapper and says "when you see an object of this type, return an object of this type ... and here are all the mappings ... map this property in this object to that property in that object." In our case, since both the Contract and ContractDto classes have properties that are the same names, AutoMapper will INFER the mappings and we don't have to do anything else. However, in more complex scenarios, we would write code here to take control of the mapping process. The good news is that we would only need to do this once for our entire project. From that point on, AutoMapper would know how to map between these two types.



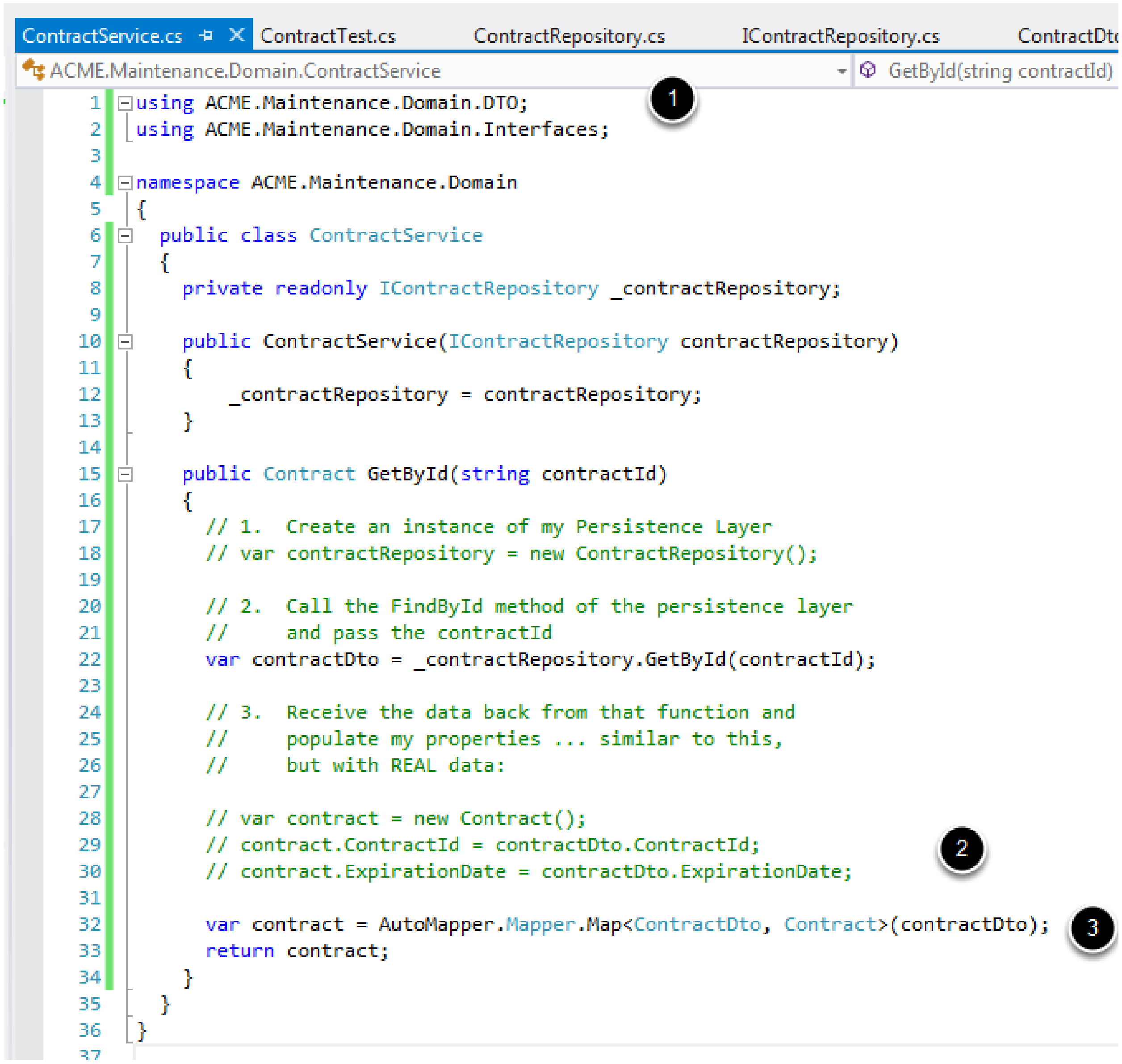
```
    [TestInitialize]
    public void Initalize()
    {
      // Initialize serves as the "composition root"

      _contractRepository = A.Fake<IContractRepository>();

      A.CallTo(() => _contractRepository.GetById(ValidContractId))
        .Returns(new ContractDto
           {ContractId = ValidContractId,
           ExpirationDate = DateTime.Now.AddDays(1)});

      A.CallTo(() => _contractRepository.GetById(ExpiredContractId))
        .Returns(new ContractDto
        {
          ContractId = ExpiredContractId,
          ExpirationDate = DateTime.Now.AddDays(-1)
        });

      AutoMapper.Mapper.CreateMap<ContractDto, Contract>();
    }
```

Get this code:

https://gist.github.com/LearnVisualStudio/06da3fd86c1b51f00bb5

3. Utilize AutoMapper in the ContractService class to perform the mappings between the Domain and DTO objects

Next, I'll replace the code in the ContractService.cs class that AutoMapper makes redundant.



```
using ACME.Maintenance.Domain.DTO;
using ACME.Maintenance.Domain.Interfaces;

namespace ACME.Maintenance.Domain
{
  public class ContractService
  {
    private readonly IContractRepository _contractRepository;

    public ContractService(IContractRepository contractRepository)
    {
        _contractRepository = contractRepository;
    }

    public Contract GetById(string contractId)
    {
      // 1.  Create an instance of my Persistence Layer
      // var contractRepository = new ContractRepository();

      // 2.  Call the FindById method of the persistence layer
      //     and pass the contractId
      var contractDto = _contractRepository.GetById(contractId);

      // 3.  Receive the data back from that function and
      //     populate my properties ... similar to this,
      //     but with REAL data:

      // var contract = new Contract();
      // contract.ContractId = contractDto.ContractId;
      // contract.ExpirationDate = contractDto.ExpirationDate;

      var contract = AutoMapper.Mapper.Map<ContractDto, Contract>(contractDto);
      return contract;
    }
  }
}
```

(1) Add a using statement for the DTO namespace

(2) Comment out the old code that creates an instance of Contract and populates its properties using the contractDto from the ContractRepository's GetById() method

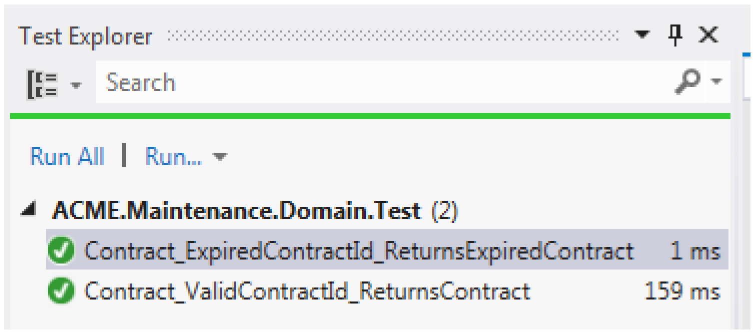
(3) The Map() method both creates a new instance of Contract AND maps properties from the contractDto to the new instance of Contract

In this pithy example, we've eliminated three lines of code. Imagine how many lines could be saved throughout the entire project, especially if there were dozens of properties and dozens of mappings!

Get this code:

https://gist.github.com/LearnVisualStudio/71f288a4f2a5a43ea29c

4. Run All Unit Tests



```
ACME.Maintenance.Domain.Test (2)
  Contract_ExpiredContractId_ReturnsExpiredContract     1 ms
  Contract_ValidContractId_ReturnsContract            159 ms
```

Success! The Unit Tests ensure that we've not introduced any new bugs into our application.

Lesson 13 - Arrange, Act, Assert

Before we move on, I wanted to address two items related to our Unit Tests.

First, recall that I'm choosing the naming convention:

MethodName_StateUnderTest_ExpectedBehavior

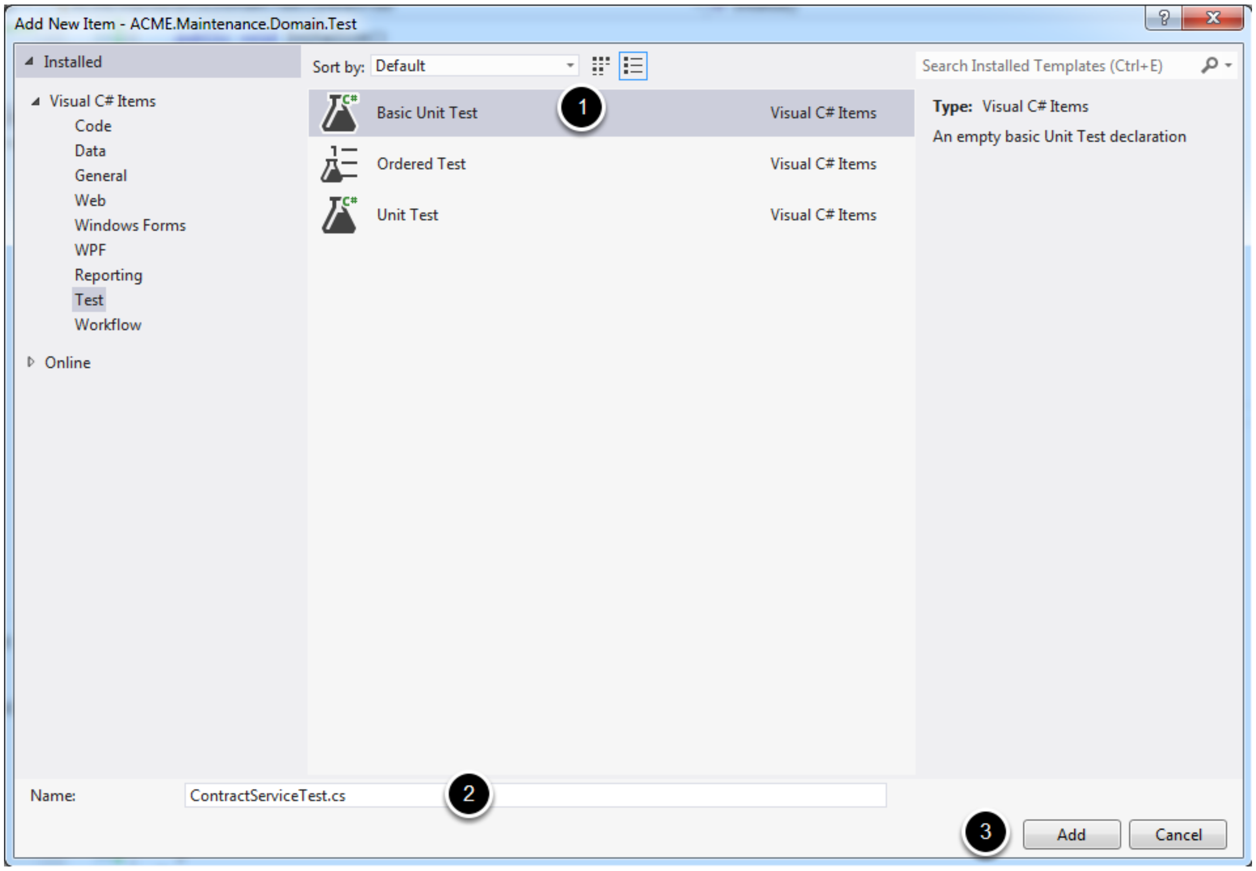
And yet, our method names have changed. In fact, the Unit Test itself is not really named correctly. The Unit Test file and Class name should reflect the Domain class name we're testing. Now that we've refactored a lot of the functionality out of the Contract itself and into the ContractService class, in order to be consistent, I feel like we need to move a few things around and rename a few things.

First, I create a new Unit Test in my Test project called ContractServiceTest.cs.

1. Add new Unit Test class called ContractServiceTest.cs

In the Solution Explorer, right-click on the ACME.Maintenance.Domain.Test project, select Add | New Item...

In the Add New Item dialog...



Type: Visual C# Items

An empty basic Unit Test declaration

Name: ContractServiceTest.cs

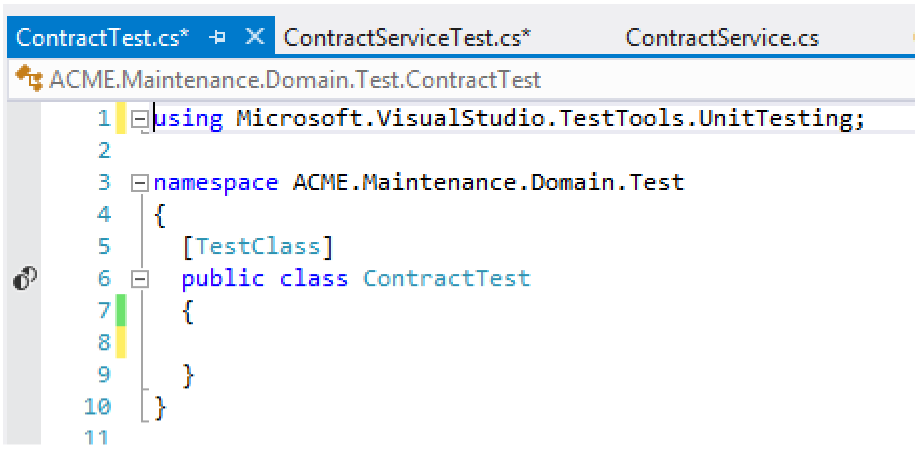
(1) Select the Basic Unit Test template

(2) Rename to: ContractServiceTest.cs

(3) Select Add

2. Cut everything out of the ContractTest class

Next, I cut virtually everything out of the old ContractTest.cs...



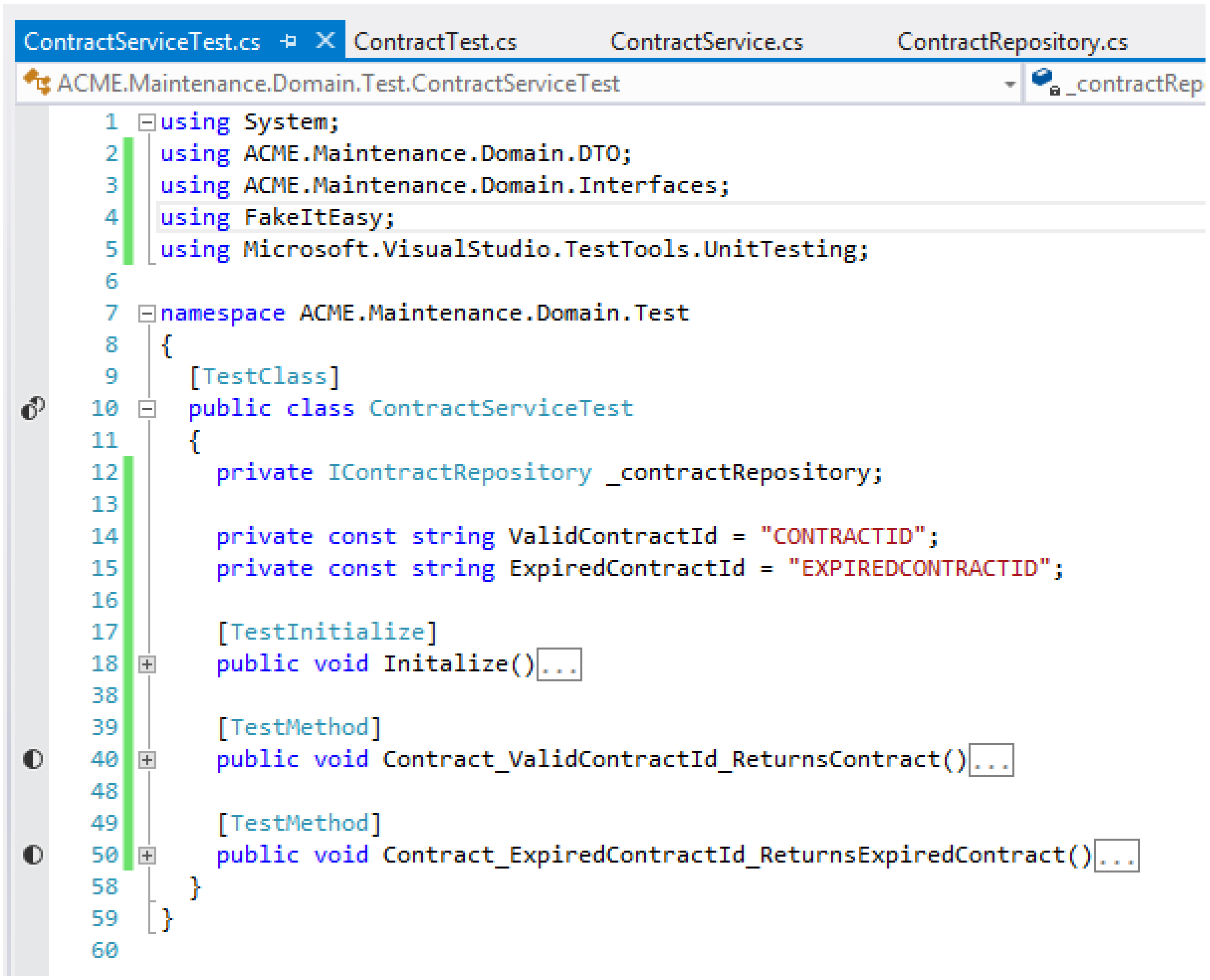
```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ACME.Maintenance.Domain.Test
{
    [TestClass]
    public class ContractTest
    {

    }
}
```

Get this code now that I've cut everything out of it:

https://gist.github.com/LearnVisualStudio/b574ca50bc9ad8cfaf63

3. Paste to the new ContractServiceTest class

... and I paste it into the new ContractServiceTest.cs file.   Here's what it should look like now:



ContractServiceTest.cs | ContractTest.cs | ContractService.cs | ContractRepository.cs

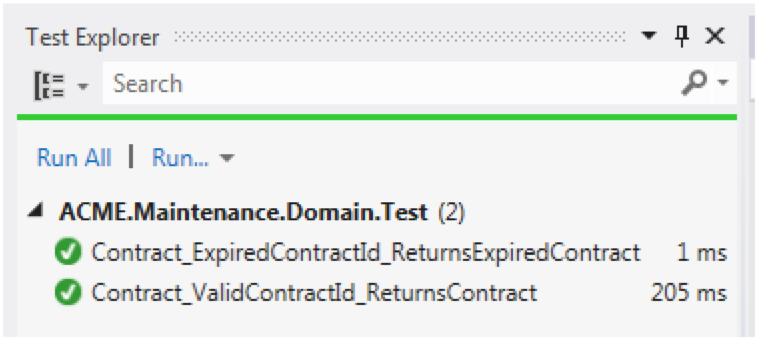
ACME.Maintenance.Domain.Test.ContractServiceTest | _contractRep

```
using System;
using ACME.Maintenance.Domain.DTO;
using ACME.Maintenance.Domain.Interfaces;
using FakeItEasy;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ACME.Maintenance.Domain.Test
{
  [TestClass]
  public class ContractServiceTest
  {
    private IContractRepository _contractRepository;

    private const string ValidContractId = "CONTRACTID";
    private const string ExpiredContractId = "EXPIREDCONTRACTID";

    [TestInitialize]
    public void Initalize()...

    [TestMethod]
    public void Contract_ValidContractId_ReturnsContract()...

    [TestMethod]
    public void Contract_ExpiredContractId_ReturnsExpiredContract()...
  }
}
```

Note that I've rolled up all the Unit Test mehthods so they'll fit nicely in this screenshot.

Also note that when you do this, you'll need to resolve all the references with using statements. You can copy & paste that as well from the ContractTest.cs to the ContractServiceTest.cs.

4. Run All Unit Tests

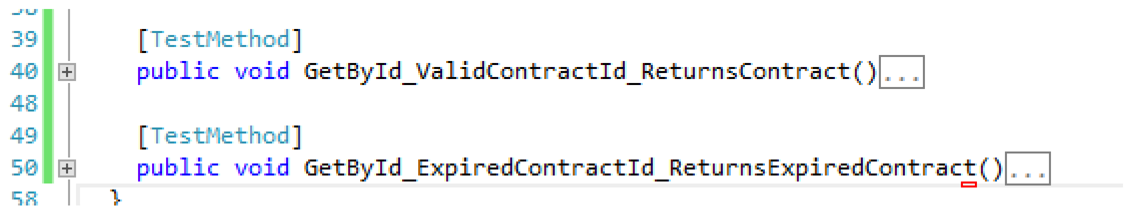


```
ACME.Maintenance.Domain.Test (2)
  Contract_ExpiredContractId_ReturnsExpiredContract     1 ms
  Contract_ValidContractId_ReturnsContract            205 ms
```

I Run All unit tests just to make sure everything is still working.

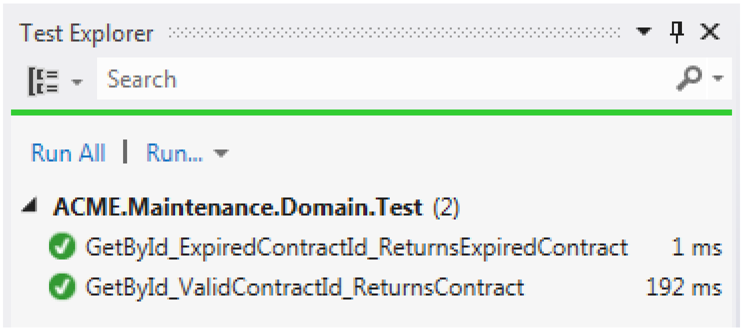
5. Renaming the Unit Tests and Confirming Nothing Broke

Next, I rename the unit tests. We're no longer testing the constructor of the Contract class, but rather, the GetById() method of the ContractService class. So I change the names to reflect that.



```
    [TestMethod]
    public void GetById_ValidContractId_ReturnsContract()...

    [TestMethod]
    public void GetById_ExpiredContractId_ReturnsExpiredContract()...
}
```

... and just to make sure everything is still good, I Run All tests again.



```
ACME.Maintenance.Domain.Test (2)
  GetById_ExpiredContractId_ReturnsExpiredContract     1 ms
  GetById_ValidContractId_ReturnsContract            192 ms
```

It still works.

6. Discovering the Arrange, Act, Assert Pattern

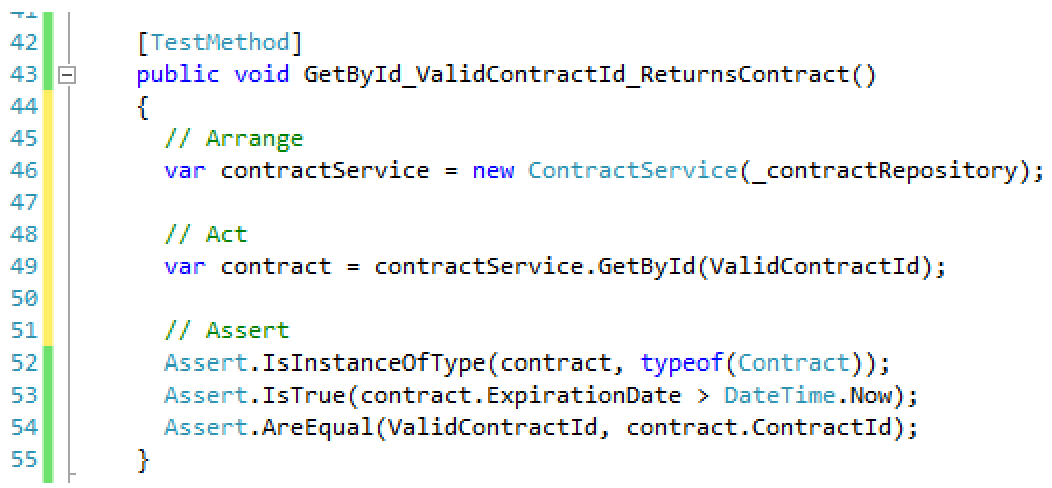
When you look at our two Unit Tests, they follow a pattern (intentionally) of Arrange, Act, Assert.

Arrange - all the necessary preconditions and inputs. In our case, we're arranging the test by creating an instance of the contractService utilizing the _contractRepository from FakeItEasy.

Act - on the object or method under test. In our case, we're calling the GetById. This is what will produce the results.

Assert - that the expected results have occurred. In our case, we're checking few results ... namely that the object returned by the method under test is giving us the correct data type, a Contract, and the new Contract's properties are set to what we expect.

If you're just getting started creating unit tests, this pattern can help you get everything squared away properly. As you can see, I've inserted comments into my code to help me as I write the unit test and see the distinction between the responsibilities of each part.

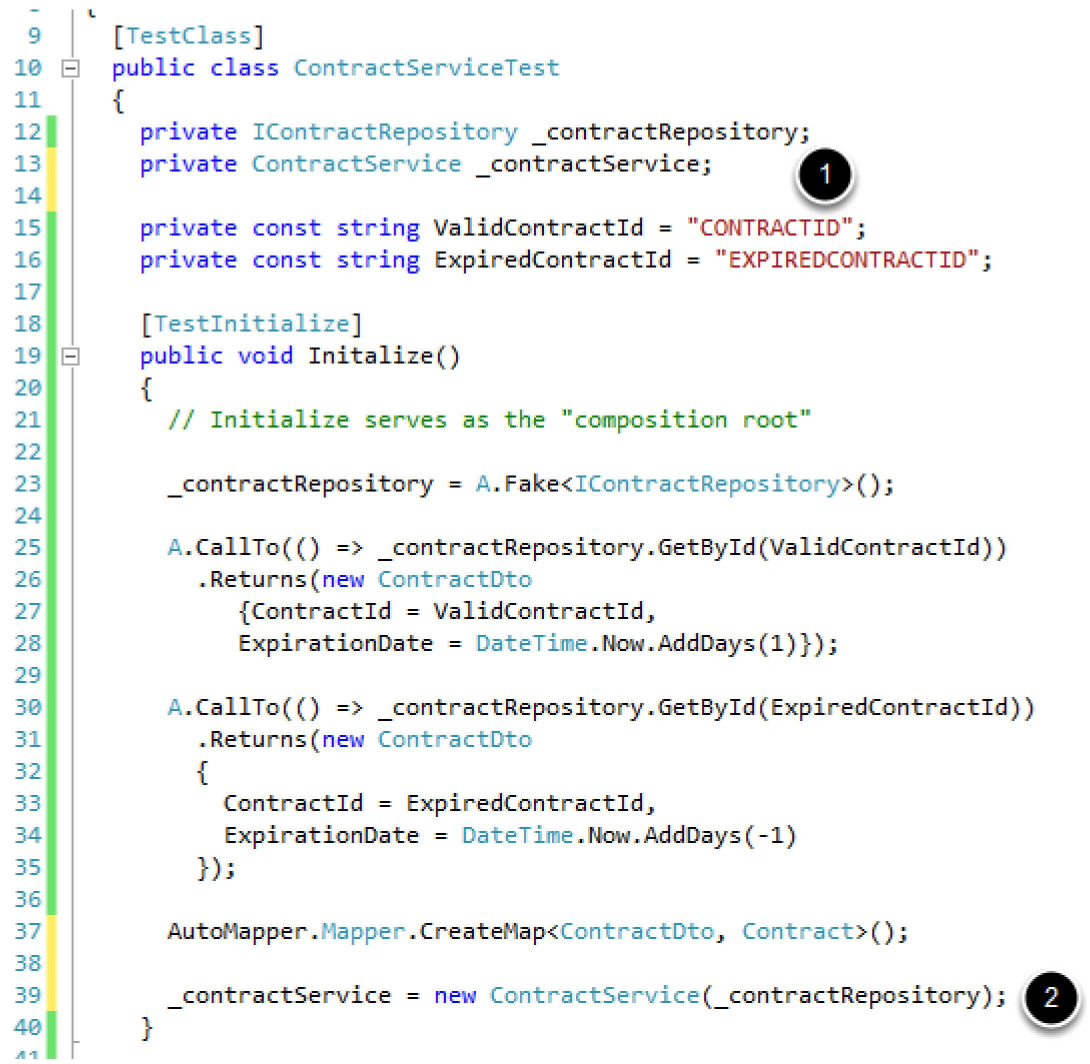


```
[TestMethod]
public void GetById_ValidContractId_ReturnsContract()
{
  // Arrange
  var contractService = new ContractService(_contractRepository);

  // Act
  var contract = contractService.GetById(ValidContractId);

  // Assert
  Assert.IsInstanceOfType(contract, typeof(Contract));
  Assert.IsTrue(contract.ExpirationDate > DateTime.Now);
  Assert.AreEqual(ValidContractId, contract.ContractId);
}
```

As you become more experienced, you will probably want to avoid adding those reminders. Idealy, it will be firmly ensconced in your mind. Until then, it serves as a railing to hold on to when you're uncertain what needs to happen next in your unit tests.

7. Refactoring to consolidate common code across multiple tests

This is a little nit-picky, but I prefer less code to more code. Each of the unit tests were creating a new instance of the ContractService. I'll move that to the Initialize() method and will create a private variable, _contractService which I'll use throughout the unit tests.

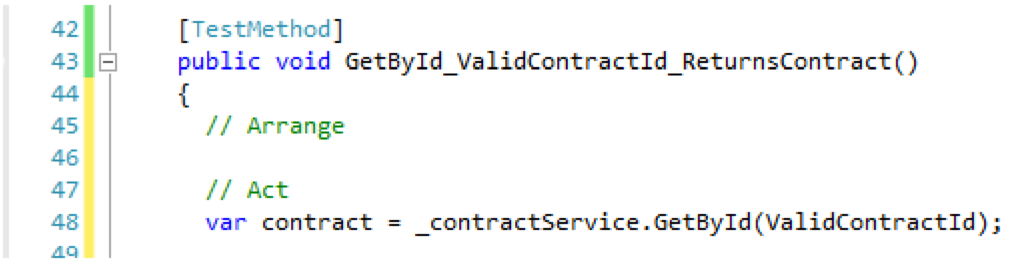


```
[TestClass]
public class ContractServiceTest
{
  private IContractRepository _contractRepository;
  private ContractService _contractService;

  private const string ValidContractId = "CONTRACTID";
  private const string ExpiredContractId = "EXPIREDCONTRACTID";

  [TestInitialize]
  public void Initalize()
  {
    // Initialize serves as the "composition root"

    _contractRepository = A.Fake<IContractRepository>();

    A.CallTo(() => _contractRepository.GetById(ValidContractId))
      .Returns(new ContractDto
         {ContractId = ValidContractId,
         ExpirationDate = DateTime.Now.AddDays(1)});

    A.CallTo(() => _contractRepository.GetById(ExpiredContractId))
      .Returns(new ContractDto
      {
        ContractId = ExpiredContractId,
        ExpirationDate = DateTime.Now.AddDays(-1)
      });

    AutoMapper.Mapper.CreateMap<ContractDto, Contract>();

    _contractService = new ContractService(_contractRepository);
  }
```

(1) I create a private _contractService member

(2) I retrieve the ContractService into the new private _contractService

This will make our Arrange lonely, but that's ok. In fact, I'm going to strip those comments out. If you need to remind yourself at first the form a unit test should take, you can use them. At some point, you'll tire of adding them and other developers should assume you're following that pattern anyway.

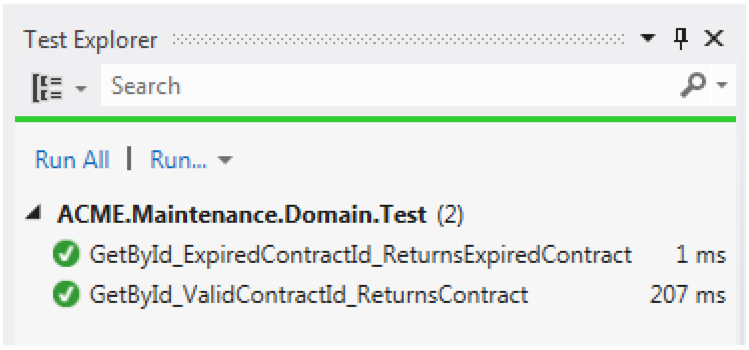


```
    [TestMethod]
    public void GetById_ValidContractId_ReturnsContract()
    {
      // Arrange

      // Act
      var contract = _contractService.GetById(ValidContractId);
```

Get this code:

https://gist.github.com/LearnVisualStudio/7340d5ec64b0a033824f

After the cleanup, I Run All tests and everything is still in order.



```
ACME.Maintenance.Domain.Test (2)
  GetById_ExpiredContractId_ReturnsExpiredContract    1 ms
  GetById_ValidContractId_ReturnsContract           207 ms
```

Ok, so a little moving around and renaming to consistently apply the rules of unit testing to our work. Keeping our work area clean and sticking to our self-imposed rules will help us to stay productive as we add more and more code to our project.

Lesson 14 - Handling Exceptional Cases

Up to now, we've been assuming that we're working with a valid Contract Id. But what if the field service agent fat fingers the contract id and types it in incorrectly? How should our domain layer respond?

Let's think about this for a moment. What we're not asking is how will the user interface notify the user. We're talking about the interaction between the domain layer and the caller ... most likely that will be the application facade. The application facade will in turn pass something back to the presentation layer (or the web services layer).

There are two options, but only one that makes sense in my opinion.

The bad option: we could create an empty contract and pass it back from the ContractService to the caller. In the olden days, programmers would return codes indicating whether an operation was successful or not. The caller of the operation would then check the return code and determine if it was legitimate. That was the source of many bugs in systems as it was not obvious what each function could possibly return. By passing an empty contract back to the caller, we are relying on the caller to know our magic system. I don't like that.

A more explicit way of notifying the caller that the contract id is not associated with a contract in our persistence layer is to return an Exception. I like this because it forces the caller to deal with it and there's no mistaking that it is in response to a bad contract id.

Now, we could send a generic Exception back and pass the explanation in the Exception.Description attribute. However, a better approach would be to create a custom Exception specific to this scenario. We'll create a Unit Test to design this scenario and then implement a custom exception that will satisfy the unit test, just like always.

There's some guidance in the Framework Design Guidelines regarding designing Custom Exceptions:

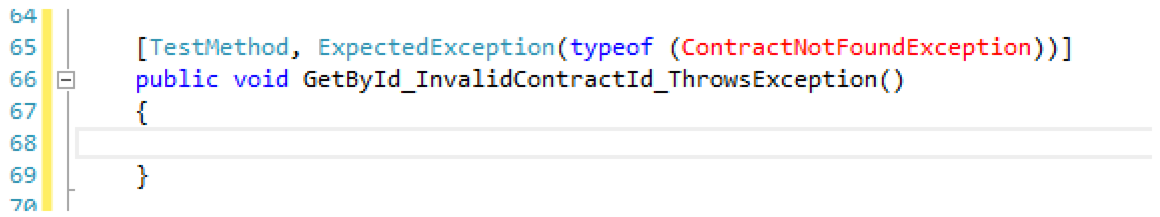
http://msdn.microsoft.com/en-us/library/vstudio/ms229064(v=vs.100).aspx

1. Avoid deep exception hierarchies.
2. Do derive exceptions from System.Exception or one of the other common base exceptions.
3. Do end exception class names with the Exception suffix.
4. Do make exceptions serializable. An exception must be serializable to work correctly across application domain and remoting boundaries.
5. Do provide (at least) the following common constructors on all exceptions. Make sure the names and types of the parameters are the same as those used in the following code example. (See the code on that page)

There are a few others regarding secure information. I won't need those but make sure you reference this (or the book) whenever you set out to design your own custom exceptions.

1. Add new Unit Test for Exceptional case

So, I add the following unit test.



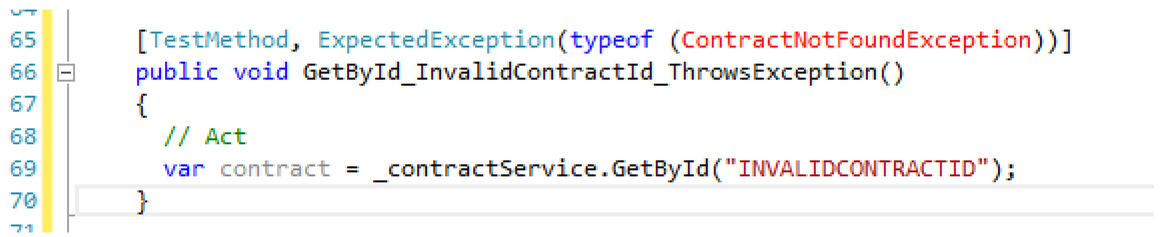
```
    [TestMethod, ExpectedException(typeof (ContractNotFoundException))]
    public void GetById_InvalidContractId_ThrowsException()
    {

    }
```

There are two things of note:

(1) Notice the use of the ExpectedException attribute. This will allow us to say that this unit test SHOULD create an exception, and here's the exception we expect. If we don't get an exception, OR if the exception is not the right one, then we want the unit test to catch it.

(2) I've followed the naming rules of the Framework Design Guidelines. Namely, I've added the suffix Exception. I've also given a clear name to the exception so there's no doubt what went wrong, especially in the context of the GetById() method which is the method under test.

Before I implement it, let me finish out this unit test.



```
    [TestMethod, ExpectedException(typeof (ContractNotFoundException))]
    public void GetById_InvalidContractId_ThrowsException()
    {
      // Act
      var contract = _contractService.GetById("INVALIDCONTRACTID");
    }
```

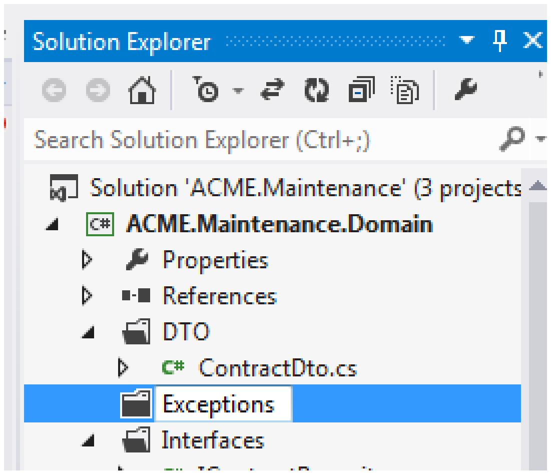
I think this is all that I'll have to do. I may need another line of code ... I'm not really sure. I'll come back to it after I implement the ContractNotFoundException.

2. Add Exceptions folder to Domain Layer project

In the Solution Explorer, right-click the ACME.Maintenance.Domain project, select Add | New Folder ...

Right-click the new folder and select Rename from the context menu. Change the name to:

Exceptions



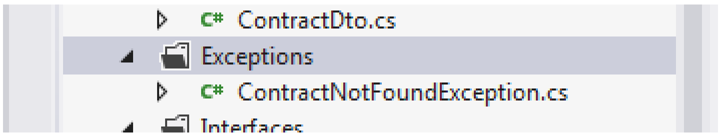
Solution 'ACME.Maintenance' (3 projects
ACME.Maintenance.Domain
Properties
References
DTO
ContractDto.cs
Exceptions
Interfaces

3. Add new ContractNotFoundException.cs class file to Exceptions folder

Right-click the new Exceptions folder, select Add | New Class ...

Name the new class:

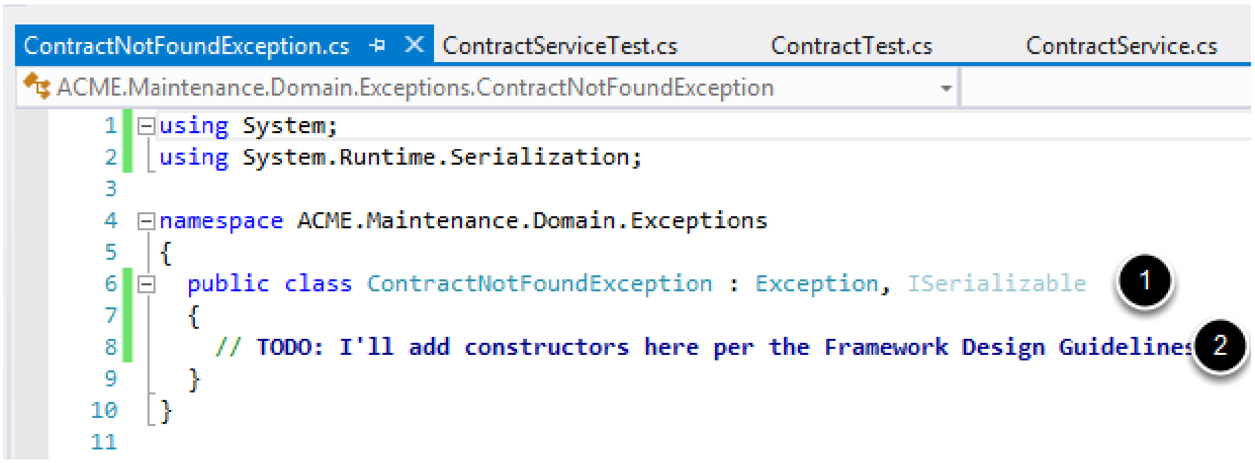
ContractNotFoundException.cs



ContractDto.cs
Exceptions
ContractNotFoundException.cs
Interfaces

4. Make new Exception conform to Framework Design Guidelines

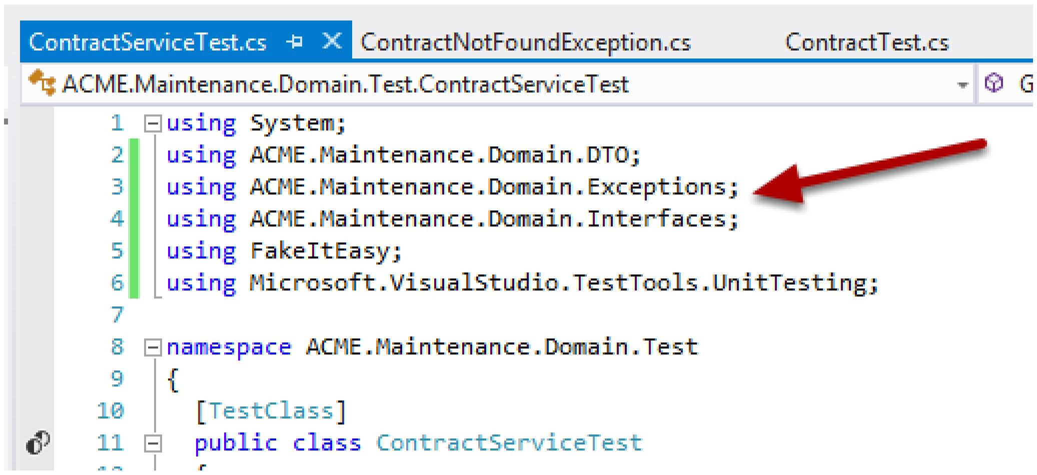
There are two basic steps required to create a custom exception. We'll take the first step now and remind ourselves to take the second step later (to be in compliance with the Framework Design Guidelines):


ContractNotFoundException.cs
ContractServiceTest.cs
ContractTest.cs
ContractService.cs
ACME.Maintenance.Domain.Exceptions.ContractNotFoundException
1
2

```
using System;
using System.Runtime.Serialization;

namespace ACME.Maintenance.Domain.Exceptions
{
  public class ContractNotFoundException : Exception, ISerializable
  {
    // TODO: I'll add constructors here per the Framework Design Guidelines
  }
}
```

(1) The custom exception should inherit from System.Exception and should implement System.Runtime.Serlization.ISerializable

(2) I'll add a comment prefixed with the word: TODO ... I have a very specific reason for this later in this lesson. This will remind me to follow ALL the steps required to properly create a custome exception

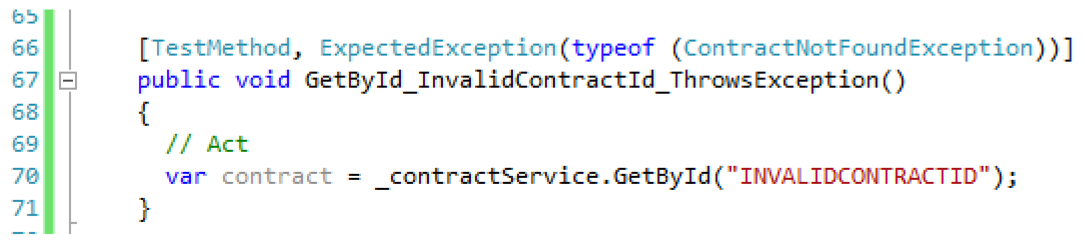

ContractServiceTest.cs
ContractNotFoundException.cs
ContractTest.cs
ACME.Maintenance.Domain.Test.ContractServiceTest

```
using System;
using ACME.Maintenance.Domain.DTO;
using ACME.Maintenance.Domain.Exceptions;
using ACME.Maintenance.Domain.Interfaces;
using FakeItEasy;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ACME.Maintenance.Domain.Test
{
  [TestClass]
  public class ContractServiceTest
```

5. Add using statement to the new Exception namespace in the ContractServiceTest.cs

Back in the ContractServiceTest.cs, in order to get the project to compile, you need to add the following using statement:

```
using ACME.Maintenance.Domain.Exceptions;
```

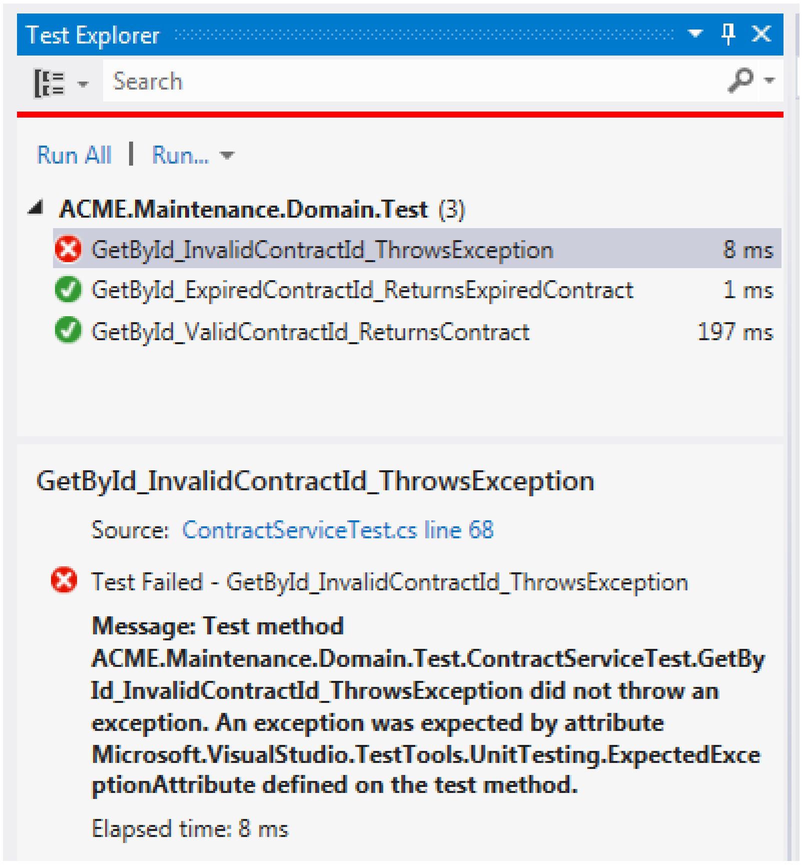


```
    [TestMethod, ExpectedException(typeof (ContractNotFoundException))]
    public void GetById_InvalidContractId_ThrowsException()
    {
      // Act
      var contract = _contractService.GetById("INVALIDCONTRACTID");
    }
```

Now, as a result, the Intellisense should be able to see the type ContractNotFoundException.

When I'm not sure what I need to do next, I Run All unit tests. This will sometimes help focus me on what I may have forgotten.

6. Run All Unit Tests



```
ACME.Maintenance.Domain.Test (3)
  GetById_InvalidContractId_ThrowsException            8 ms
  GetById_ExpiredContractId_ReturnsExpiredContract     1 ms
  GetById_ValidContractId_ReturnsContract            197 ms
```

GetById_InvalidContractId_ThrowsException

Source: ContractServiceTest.cs line 68

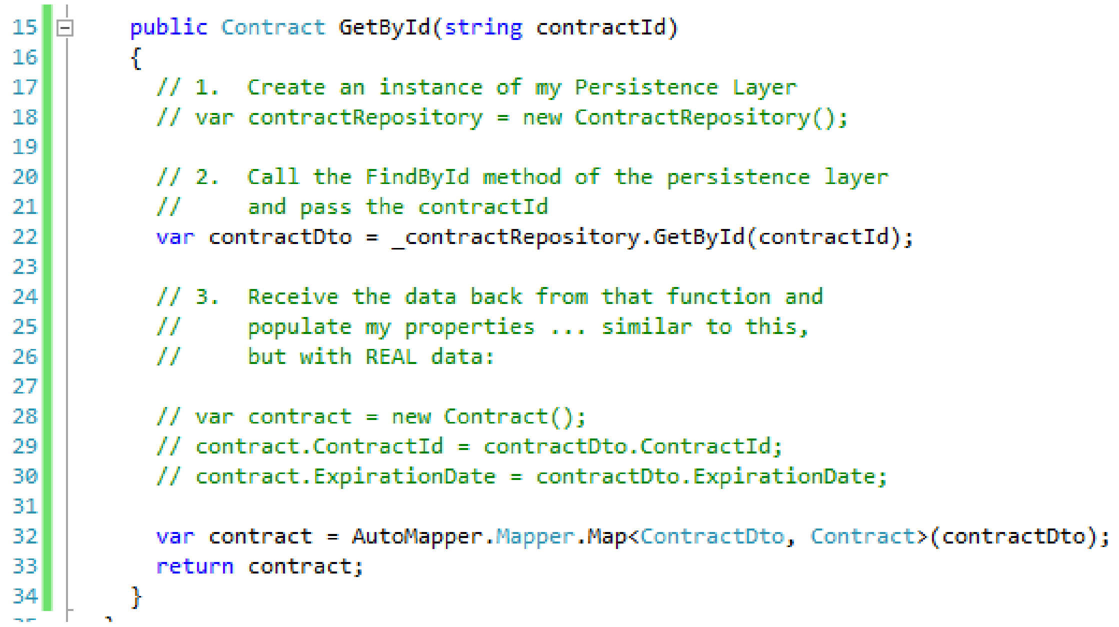
Test Failed - GetById_InvalidContractId_ThrowsException

Message: Test method ACME.Maintenance.Domain.Test.ContractServiceTest.GetById_InvalidContractId_ThrowsException did not throw an exception. An exception was expected by attribute Microsoft.VisualStudio.TestTools.UnitTesting.ExpectedExceptionAttribute defined on the test method.

Elapsed time: 8 ms

The good news is that it compiles. The good news is that we get a "Red", and can now move on to the "Green" phase and get it working.

However, the bad news is that I experience a moment of uncertainty. What do I want to happen here? Here's the current state of the GetById() method:



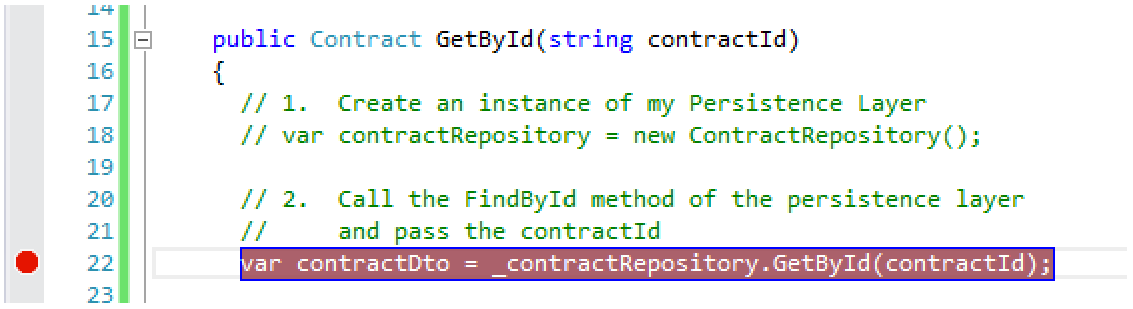
```
    public Contract GetById(string contractId)
    {
      // 1.  Create an instance of my Persistence Layer
      // var contractRepository = new ContractRepository();

      // 2.  Call the FindById method of the persistence layer
      //     and pass the contractId
      var contractDto = _contractRepository.GetById(contractId);

      // 3.  Receive the data back from that function and
      //     populate my properties ... similar to this,
      //     but with REAL data:

      // var contract = new Contract();
      // contract.ContractId = contractDto.ContractId;
      // contract.ExpirationDate = contractDto.ExpirationDate;

      var contract = AutoMapper.Mapper.Map<ContractDto, Contract>(contractDto);
      return contract;
    }
```

I want GetById to throw the exception. But do I want it to be thrown by the Domain Layer? Or thrown by the Persistence Layer and simply allow it to bubble up through the Domain Layer?

Moreover, I'm not even sure what will happen when I call GetById() with a non-existent contract id. I may need to debug to see the current behavior in "slow motion".

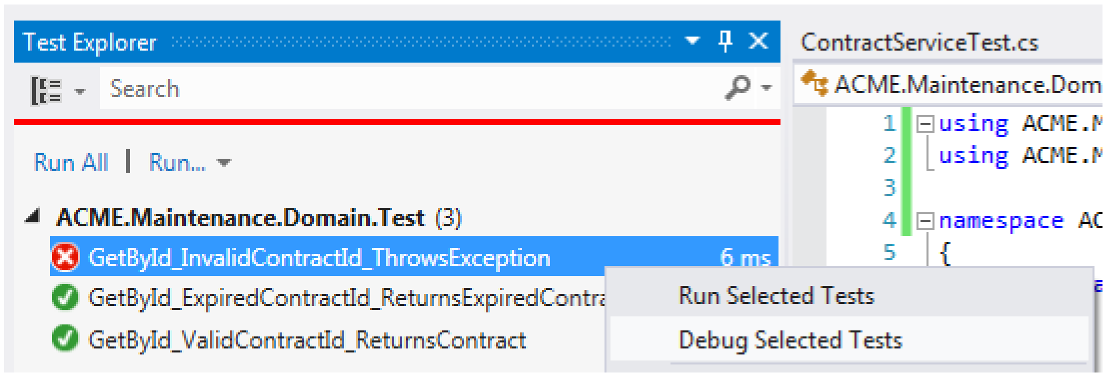
7. Setting a Breakpoint and Debugging the Unit Test

I set a breakpoint on the _contractRepository.GetById() method.



```
        public Contract GetById(string contractId)
        {
          // 1.  Create an instance of my Persistence Layer
          // var contractRepository = new ContractRepository();

          // 2.  Call the FindById method of the persistence layer
          //      and pass the contractId
          var contractDto = _contractRepository.GetById(contractId);
```

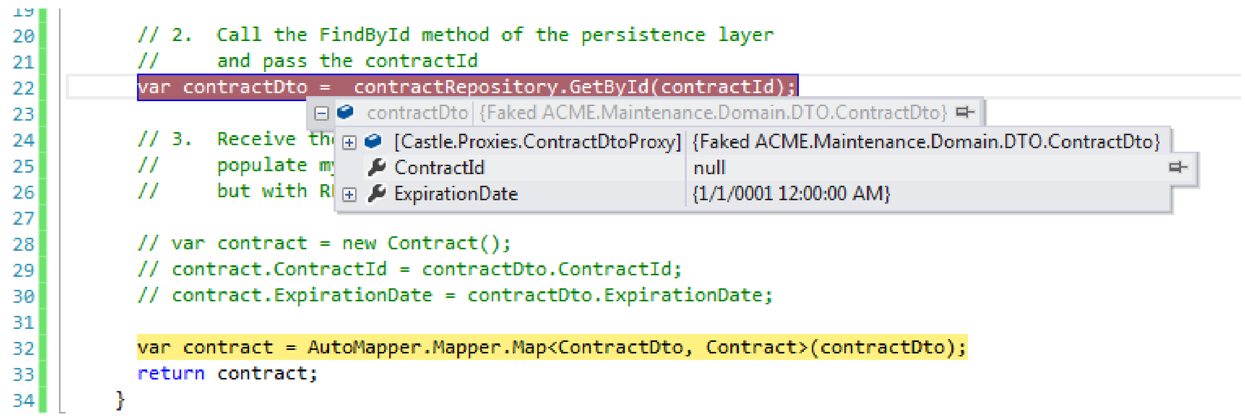
I right-click the exceptional unit test, and select Debug Selected Tests from the context menu.



Test Explorer
Search
Run All | Run...
ACME.Maintenance.Domain.Test (3)
GetById_InvalidContractId_ThrowsException 6 ms
GetById_ExpiredContractId_ReturnsExpiredContra
GetById_ValidContractId_ReturnsContract
Run Selected Tests
Debug Selected Tests
ContractServiceTest.cs
ACME.Maintenance.Dom

```
using ACME.M
using ACME.M

namespace AC
{
```

I Step Into (F11) the GetById method, then Step Over (F10) the break point. I hover my mouse cursor over contractDto and expand out the object ... and see that ContractId and ExpirationDate are both null.



```
          // 2.  Call the FindById method of the persistence layer
          //      and pass the contractId
          var contractDto = _contractRepository.GetById(contractId);

          // 3.  Receive th
          //      populate my
          //      but with R

          // var contract = new Contract();
          // contract.ContractId = contractDto.ContractId;
          // contract.ExpirationDate = contractDto.ExpirationDate;

          var contract = AutoMapper.Mapper.Map<ContractDto, Contract>(contractDto);
          return contract;
      }
```

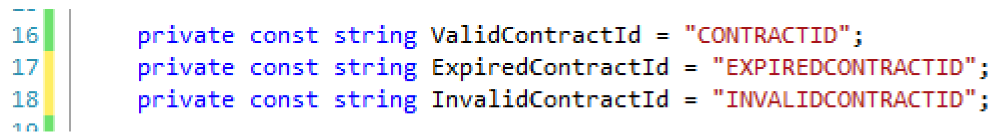
```
contractDto  {Faked ACME.Maintenance.Domain.DTO.ContractDto}
  [Castle.Proxies.ContractDtoProxy]  {Faked ACME.Maintenance.Domain.DTO.ContractDto}
  ContractId                         null
  ExpirationDate                     {1/1/0001 12:00:00 AM}
```

That gives me an idea. I've set expectations for non-Exception scenarios using FakeItEasy ... I wonder if I can use it for Exception scenarios as well?

I stop debugging.

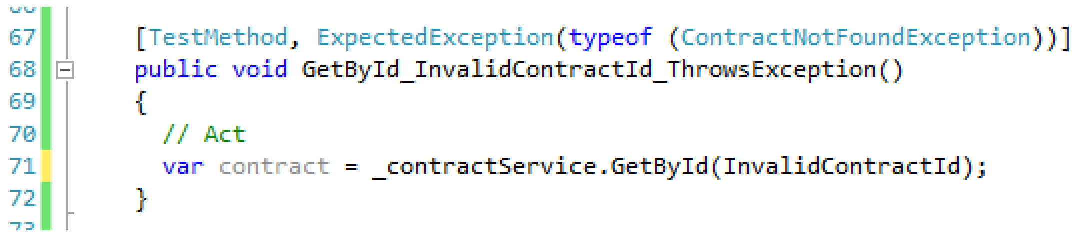
8. Refactor out "magic strings" from the Unit Test

I get started by refactoring the magic string "INVALIDCONTRACTID" into a const.



```
    private const string ValidContractId = "CONTRACTID";
    private const string ExpiredContractId = "EXPIREDCONTRACTID";
    private const string InvalidContractId = "INVALIDCONTRACTID";
```

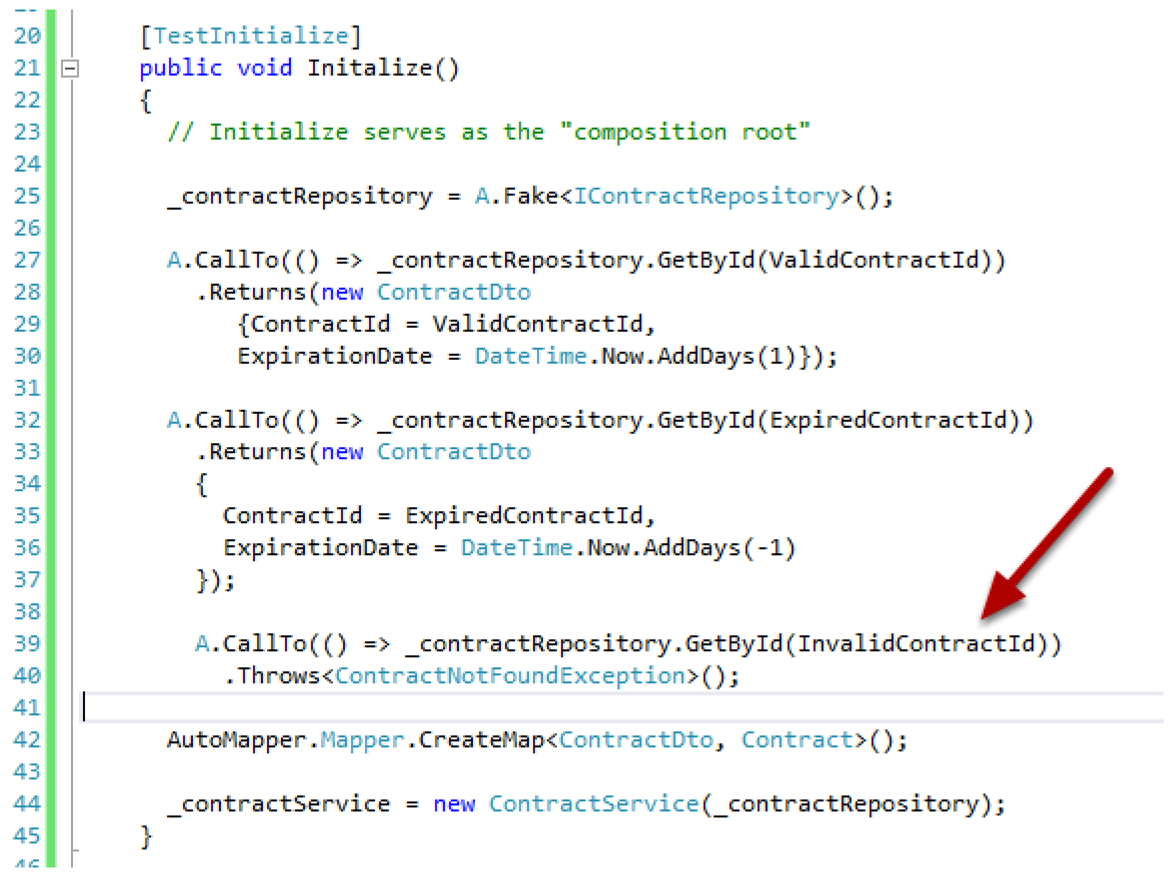
... and I replace it in the body of my newest Unit Test. This doesn't appreciably improve my situation, but it does re-introduce me to the code in with a small win.



```
    [TestMethod, ExpectedException(typeof (ContractNotFoundException))]
    public void GetById_InvalidContractId_ThrowsException()
    {
      // Act
      var contract = _contractService.GetById(InvalidContractId);
    }
```

9. Add an Expectation to FakeItEasy for the Exception

I determine that I need FakeItEasy to throw a specific Exception (our new ContractNotFoundException) when the fake ContractRepository's GetById() method is called. So I write the following code (see arrow below):



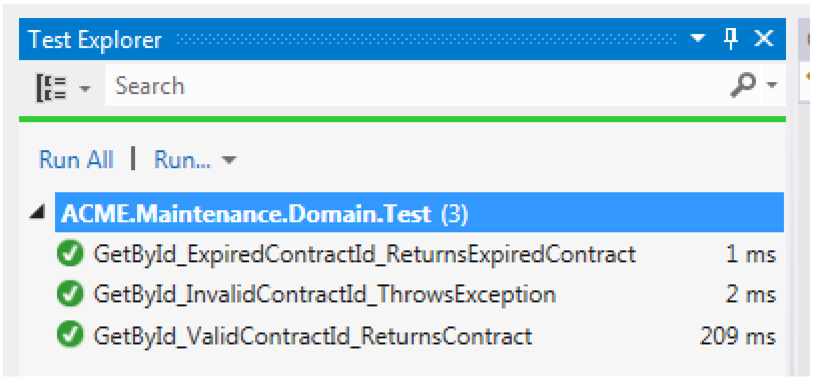
```
20      [TestInitialize]
21      public void Initalize()
22      {
23        // Initialize serves as the "composition root"
24
25        _contractRepository = A.Fake<IContractRepository>();
26
27        A.CallTo(() => _contractRepository.GetById(ValidContractId))
28          .Returns(new ContractDto
29             {ContractId = ValidContractId,
30             ExpirationDate = DateTime.Now.AddDays(1)});
31
32        A.CallTo(() => _contractRepository.GetById(ExpiredContractId))
33          .Returns(new ContractDto
34          {
35            ContractId = ExpiredContractId,
36            ExpirationDate = DateTime.Now.AddDays(-1)
37          });
38
39          A.CallTo(() => _contractRepository.GetById(InvalidContractId))
40            .Throws<ContractNotFoundException>();
41
42        AutoMapper.Mapper.CreateMap<ContractDto, Contract>();
43
44        _contractService = new ContractService(_contractRepository);
45      }
```

In lines 39 and 40 I add a different type of Expectation ... when GetById() is passed an InvalidContractId, I want FakeItEasy to send a ContractNotFoundException.

Get this code:

https://gist.github.com/LearnVisualStudio/b036e2c5163c31b5beaa

Does that fix the problem?



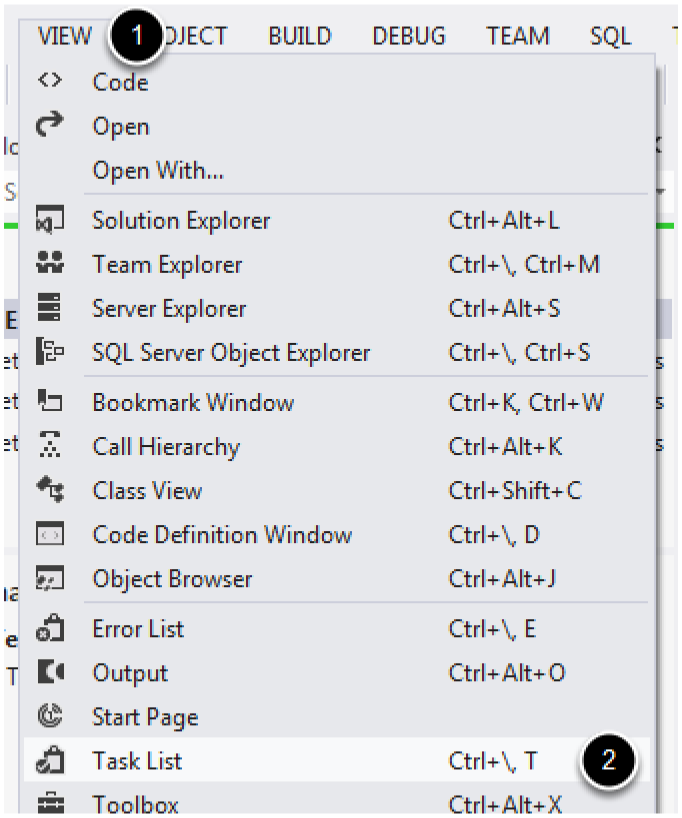
```
ACME.Maintenance.Domain.Test (3)
  GetById_ExpiredContractId_ReturnsExpiredContract     1 ms
  GetById_InvalidContractId_ThrowsException            2 ms
  GetById_ValidContractId_ReturnsContract            209 ms
```

Yes, it fixes the issue! We've successfully added a custom exception and have tested to ensure the correct behavior of the system is to merely bubble up exceptions due to an invalid contract id through our domain layer to the caller (in this case, the unit tests).

10. Working with the Task List to find Unfinished TODO's

I seem to remember adding a TODO ... my past self added that TODO comment to remind my future self of something left undone.

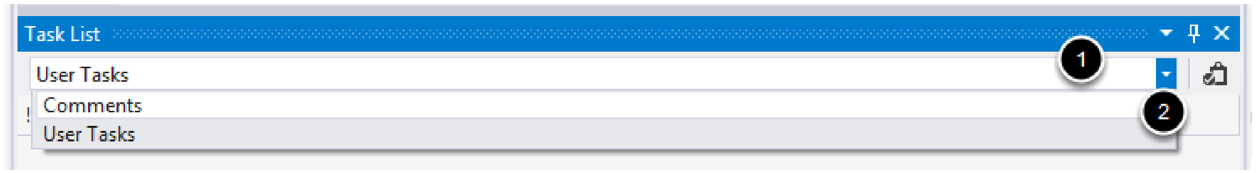
Instead of searching through a bunch of files, I use Visual Studio's Task List.


VIEW
1
OJECT
BUILD
DEBUG
TEAM
SQL
Code
Open
Open With...
Solution Explorer Ctrl+Alt+L
Team Explorer Ctrl+\, Ctrl+M
Server Explorer Ctrl+Alt+S
SQL Server Object Explorer Ctrl+\, Ctrl+S
Bookmark Window Ctrl+K, Ctrl+W
Call Hierarchy Ctrl+Alt+K
Class View Ctrl+Shift+C
Code Definition Window Ctrl+\, D
Object Browser Ctrl+Alt+J
Error List Ctrl+\, E
Output Ctrl+Alt+O
Start Page
Task List Ctrl+\, T
2
Toolbox Ctrl+Alt+X

(1) View menu

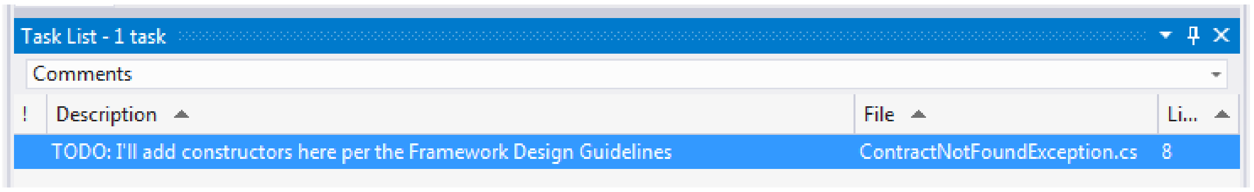
(2) Task List

The Task List docks at the bottom of Visual Studio (by default). I'll need to filter which Tasks are displayed.



(1) Select the drop down list on the right to reveal all list items

(2) Choose the Comments list item



!	Description	File	Li...
	TODO: I'll add constructors here per the Framework Design Guidelines	ContractNotFoundException.cs	8

Filtering by Comments shows all tasks added as a result of a comment prefixed with TODO … or one of the other monikers like HACK or UNDONE.

For more information on this feature:

http://msdn.microsoft.com/en-us/library/zce12xx2(v=vs.80).aspx

Double-click on the task listed in the Task List to view it in the code editor.

11. Fully implementing the Custom Exception Class

My intent all along was to implement the new custom exception according to the guidance in the Framework Design Guidelines.

If you'll recall the following guidelines:

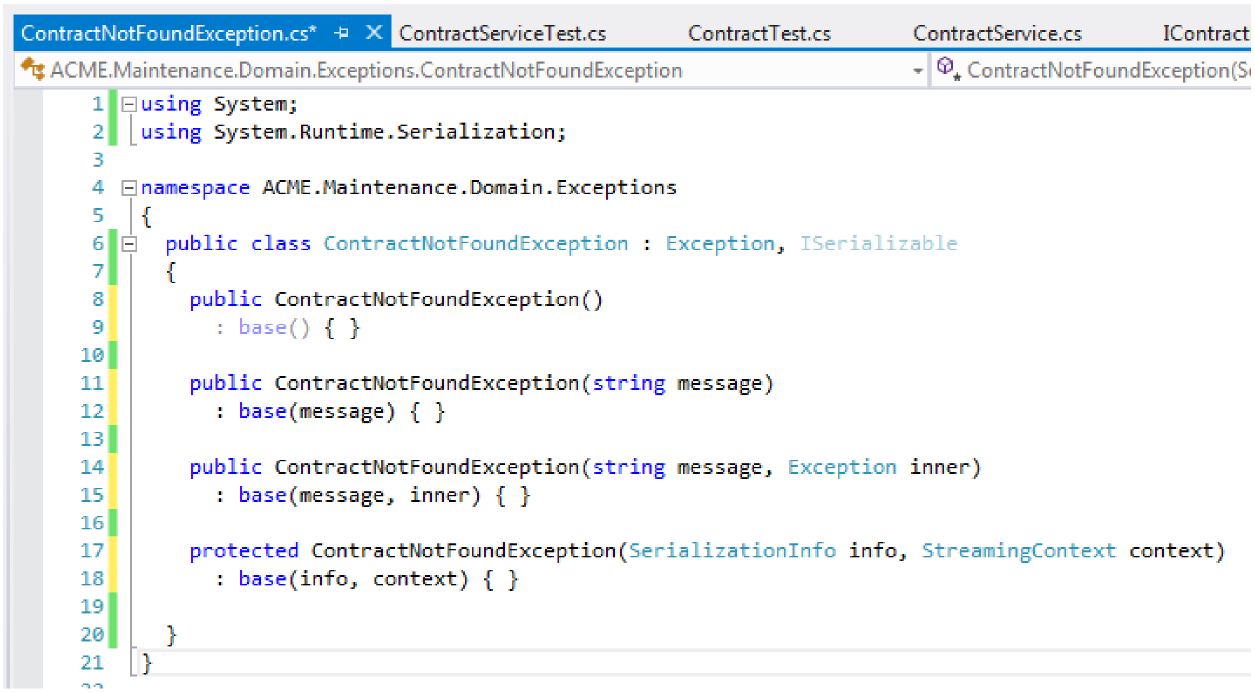
> "Do derive exceptions from System.Exception or one of the other common base exceptions."

... and...

> "Do provide (at least) the following common constructors on all exceptions. Make sure the names and types of the parameters are the same as those used in the following code example. (See the code on that page)"

I set out to implement both. The former I already took care of. With regards to the latter instruction ... the code example that is referenced shows an example custom exception fully implemented. Essentially, it shows four overloaded constructors and their calls into their base class implementations. In this case, we're not overloading to add new functionality. In this case, I merely want to create a custom exception to differentiate the type of exception. Therefore, I'll merely use the base class implementations for all the methods I'm required to create.

This is what I come up with:


ContractNotFoundException.cs*
ContractServiceTest.cs
ContractTest.cs
ContractService.cs
IContract
ACME.Maintenance.Domain.Exceptions.ContractNotFoundException
ContractNotFoundException(S

```
using System;
using System.Runtime.Serialization;

namespace ACME.Maintenance.Domain.Exceptions
{
  public class ContractNotFoundException : Exception, ISerializable
  {
    public ContractNotFoundException()
      : base() { }

    public ContractNotFoundException(string message)
      : base(message) { }

    public ContractNotFoundException(string message, Exception inner)
      : base(message, inner) { }

    protected ContractNotFoundException(SerializationInfo info, StreamingContext context)
      : base(info, context) { }

  }
}
```

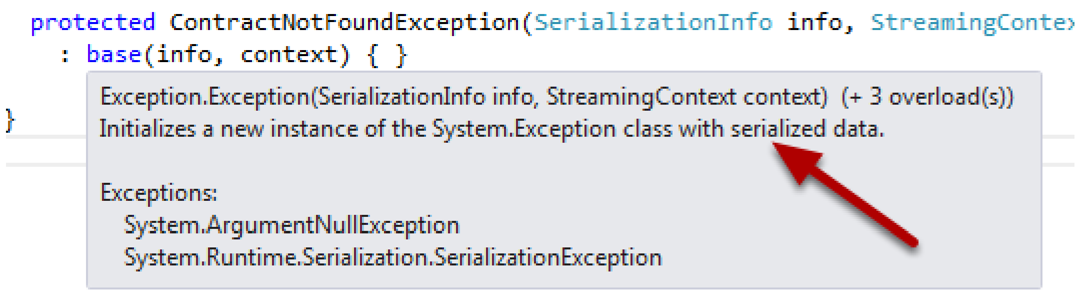
For more about the C# base keyword:

http://msdn.microsoft.com/en-us/library/hfw7t1ce.aspx

In particular, this part at the very top is pertinent (emphasis mine):

> "The base keyword is used to access members of the base class from within a derived class:
>
> Call a method on the base class that has been overridden by another method.
>
> *Specify which base-class constructor should be called when creating instances of the derived class.*
>
> *A base class access is permitted only in a constructor, an instance method, or an instance property accessor.*"

So, I'm (a) accessing the base class Exception in the first three constructors, and I'm doing so to specify which base-class constructor to use when creating instances of the derived class. The last constructor is accessing the base Exception class, but this time it is to implement the ISerializable interface. When I hover over that last "base" keyword, it knows exactly what I'm attempting to do ...



```
protected ContractNotFoundException(SerializationInfo info, StreamingConte›
  : base(info, context) { }

}
```

Exception.Exception(SerializationInfo info, StreamingContext context) (+ 3 overload(s))
Initializes a new instance of the System.Exception class with serialized data.

Exceptions:
System.ArgumentNullException
System.Runtime.Serialization.SerializationException

I probably should create unit tests to ensure this custom exception (1) is designed correctly, and (2) it will work correctly in the future after making changes to it. I'm going to be a little lazy. I believe in making sure we have a high degree of code coverage (i.e., the amount of production code covered by unit tests), but this is one of those cases where the maintenance team (if there is one) can come back and worry about code coverage. Right now I'm spiking, and I've got to move on.

Get this code:

https://gist.github.com/LearnVisualStudio/d6e93cf590fcbd94529f

The good news is that the hard part is over, in my opinion. We got a basic structure in place, have our projects set up, we've put a lot of thought into construction. The rest is just implementation of the user story and following -- in some cases, expanding on -- the direction we're already headed.

Chapter 3: Implementing the First User Story

Lesson 15 - Adding a new Unit Test for Creating an Order

In the previous lessons we've set up the Solution quite nicely by thinking about retrieving a Customer Contract from a data store. I'm not sure we're finished thinking about Contracts just yet, but at least we know we can get a valid contract and an expired contract and we'll throw an exception when an invalid contract is requested. So, we've got those bases covered, and now it's time to focus on the core of the User Story: creating the order.

We'll need to create an empty order and add order items to the order in the form of products. Products are associated with Locations, but we don't know much about Locations just yet. This user story is scoped narrowly, and that's a good thing. Later we'll tackle more features and I'm sure we'll have to account for Locations at that time.

For now, we'll begin with Unit Tests for creating an order.

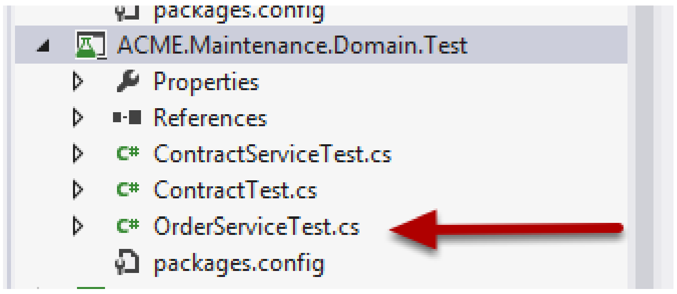
I think I'll start by doing what we ended up doing in the case of contracts, namely, that I'll create an OrderService that will be in charge of creating new Orders (or performing other Order related tasks like eventually retrieving orders from the database).

I want to add a CreateOrder() method that accepts a Contract as an input parameter. This way, I can check to make sure the Contract is valid (ExpirationDate in the future) before I create the order. If it's expired, it should throw an exception.

Creating an order should involve creating a new Order ID. To my knowledge, there's no restriction on Order ID's the way there are on Contracts, so I'm going to simply generate a GUID and use that. If I really wanted to be careful, I would check to ensure that GUID is not already in the database. I will just raise it as a potential issue, otherwise I'll ignore that possibility for now and assume "globally unique" is indeed, "globally unique". Based on the number of orders ACME works through, it's an acceptably small risk.

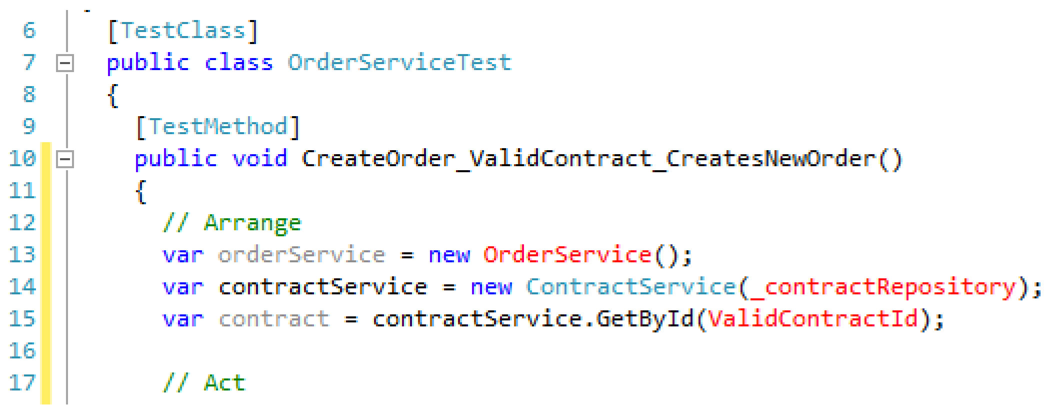
IMPORTANT NOTE: Now that we've walked through several techniques a few times already (and you have the videos to help if you get lost) I'm not going to walk you through all the steps required to add a new project, a new file, new folder, rename a file or folder, etc. I'll expect you know how to perform these tasks already. If not, make sure to go back through these lessons and pay special attention to the tasks you're performing ... never just blindly follow instructions. Make sure you understand what the instructor is teaching you before you move on to the next unit / lesson / chapter / series.

1. Add a new OrderServiceTest.cs file to the Unit Test project



2. Add a Unit Test to Create a New Order

I begin to implement this new unit test with the intent to design the OrderService class' methods, as well as the Order class' properties, etc.



```
[TestClass]
public class OrderServiceTest
{
  [TestMethod]
  public void CreateOrder_ValidContract_CreatesNewOrder()
  {
    // Arrange
    var orderService = new OrderService();
    var contractService = new ContractService(_contractRepository);
    var contract = contractService.GetById(ValidContractId);

    // Act
```

However, I hit a snag immediately. To create an Order, I want to call some method on the OrderService and pass it a contract. That means I'll need a valid contract. Unfortunately, that means I will need to use virtually the exact same Initialize() code (as well as the private member declarations) in both the ContractServiceTest.cs and now the OrderServiceTest.cs.

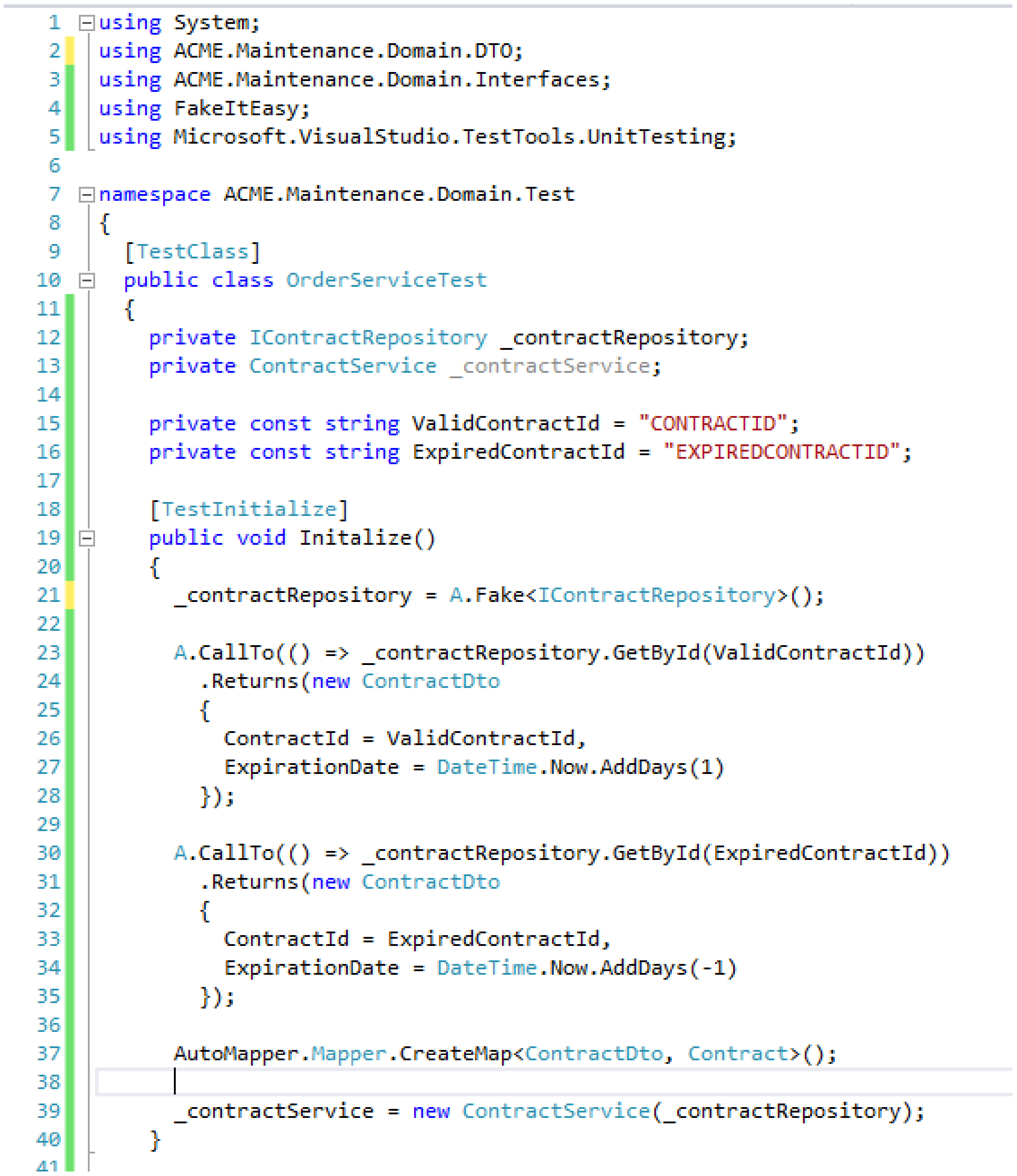
By the way, this doesn't feel right to me. Whenever I begin copying and pasting I start feeling "dumb" and that maybe I'm approaching this all wrong. I'll come to realize the error of my ways later and will refactor this, but for now I decide to just copy and paste and re-think it later.

3. Copy & Paste Initialization code from ContractServiceTest.cs to new OrderServiceTest.cs

I copy and paste most of the private member variable code (i.e., the constants, anything that starts with an underscore character) as well as the Initialize() method FROM the ContractServiceTest.cs TO the OrderServiceTest.cs.

The obvious exception: I leave out the InvalidContractId constant, as well as the FakeItEasy expectation for the invalid contractId scenario.

At this point, my code looks like this:

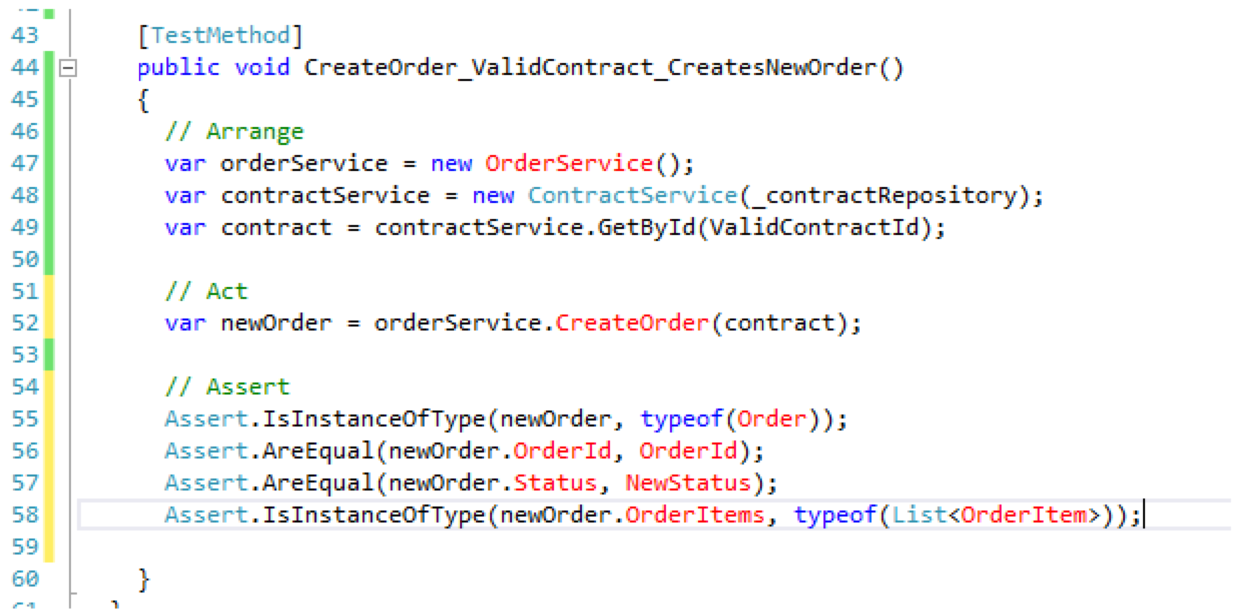


```
using System;
using ACME.Maintenance.Domain.DTO;
using ACME.Maintenance.Domain.Interfaces;
using FakeItEasy;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ACME.Maintenance.Domain.Test
{
  [TestClass]
  public class OrderServiceTest
  {
    private IContractRepository _contractRepository;
    private ContractService _contractService;

    private const string ValidContractId = "CONTRACTID";
    private const string ExpiredContractId = "EXPIREDCONTRACTID";

    [TestInitialize]
    public void Initalize()
    {
      _contractRepository = A.Fake<IContractRepository>();

      A.CallTo(() => _contractRepository.GetById(ValidContractId))
        .Returns(new ContractDto
        {
          ContractId = ValidContractId,
          ExpirationDate = DateTime.Now.AddDays(1)
        });

      A.CallTo(() => _contractRepository.GetById(ExpiredContractId))
        .Returns(new ContractDto
        {
          ContractId = ExpiredContractId,
          ExpirationDate = DateTime.Now.AddDays(-1)
        });

      AutoMapper.Mapper.CreateMap<ContractDto, Contract>();

      _contractService = new ContractService(_contractRepository);
    }
```

4. Implement the Act and Assert of the Unit Test

Step 3 should have resolved several of the missing variable references in my new unit test. Now, I want to finish what I started.

In the Act and Assert, I've introduced a new things...



```
43        [TestMethod]
44        public void CreateOrder_ValidContract_CreatesNewOrder()
45        {
46          // Arrange
47          var orderService = new OrderService();
48          var contractService = new ContractService(_contractRepository);
49          var contract = contractService.GetById(ValidContractId);
50
51          // Act
52          var newOrder = orderService.CreateOrder(contract);
53
54          // Assert
55          Assert.IsInstanceOfType(newOrder, typeof(Order));
56          Assert.AreEqual(newOrder.OrderId, OrderId);
57          Assert.AreEqual(newOrder.Status, NewStatus);
58          Assert.IsInstanceOfType(newOrder.OrderItems, typeof(List<OrderItem>));
59
60        }
```

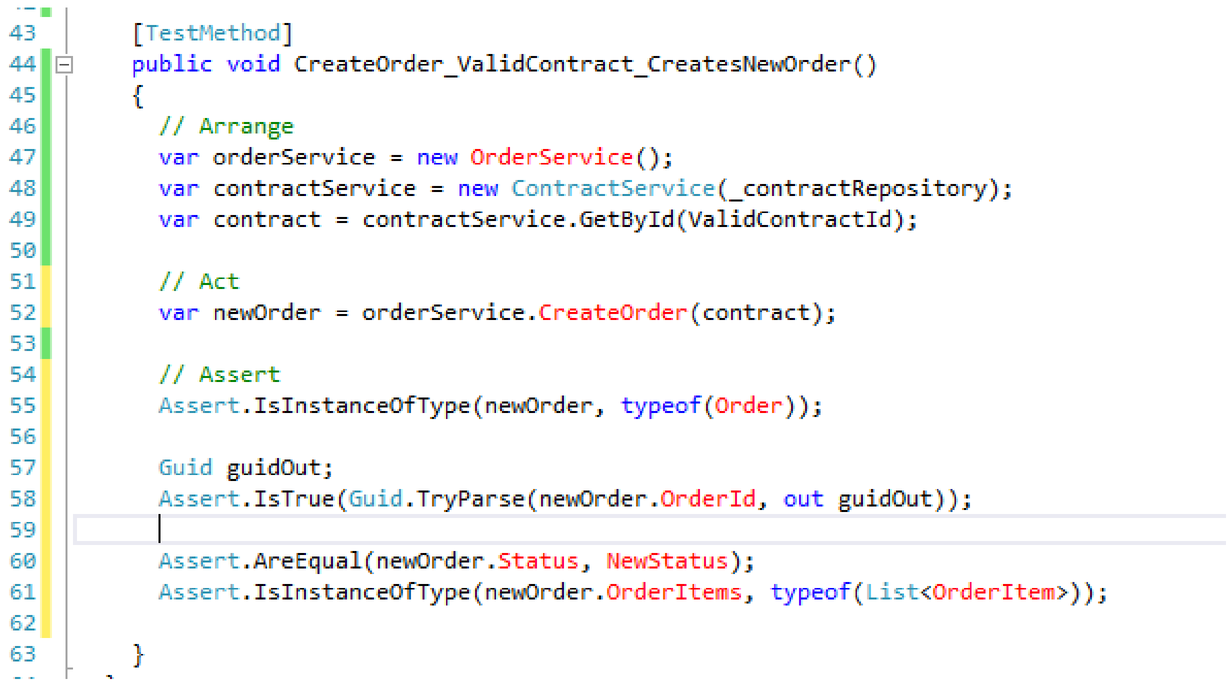
In line 52, I've added a new CreateOrder() method. This was the plan all along. I pass it an instance of a contract.

In lines 56 through 58, I want to check the properties of a brand new order to ensure that they are initialized to the correct values. I'm not sure HOW I'll determine if the OrderId is correct, but I'll worry about that next. I'm planning on creating a NewStatus constant to test for the literal string "New" ... I merely took a shortcut here. Finally, I'll ensure that the OrderItems property is a generic list of type OrderItem. Obviously, I need to create the Order and OrderItem classes.

5. Rethinking how I'll test the new Order's OrderId

The more I looked at how I'll ascertain the correctness of the OrderId, the more I realized there's no way I could know ahead of time (at least, not the way I have it now) what the OrderId SHOULD be. That will be generated automatically by the OrderService's CreateOrder() method. Instead, I devise a way to test to simply ensure that the OrderId is a GUID.

To do this, I use the Guid.TryParse() method. If the OrderId is a guid, then TryParse() will return true. If it's not a GUID, then it will fail. (Look at lines 57 & 58):



```
43      [TestMethod]
44      public void CreateOrder_ValidContract_CreatesNewOrder()
45      {
46        // Arrange
47        var orderService = new OrderService();
48        var contractService = new ContractService(_contractRepository);
49        var contract = contractService.GetById(ValidContractId);
50
51        // Act
52        var newOrder = orderService.CreateOrder(contract);
53
54        // Assert
55        Assert.IsInstanceOfType(newOrder, typeof(Order));
56
57        Guid guidOut;
58        Assert.IsTrue(Guid.TryParse(newOrder.OrderId, out guidOut));
59
60        Assert.AreEqual(newOrder.Status, NewStatus);
61        Assert.IsInstanceOfType(newOrder.OrderItems, typeof(List<OrderItem>));
62
63      }
```

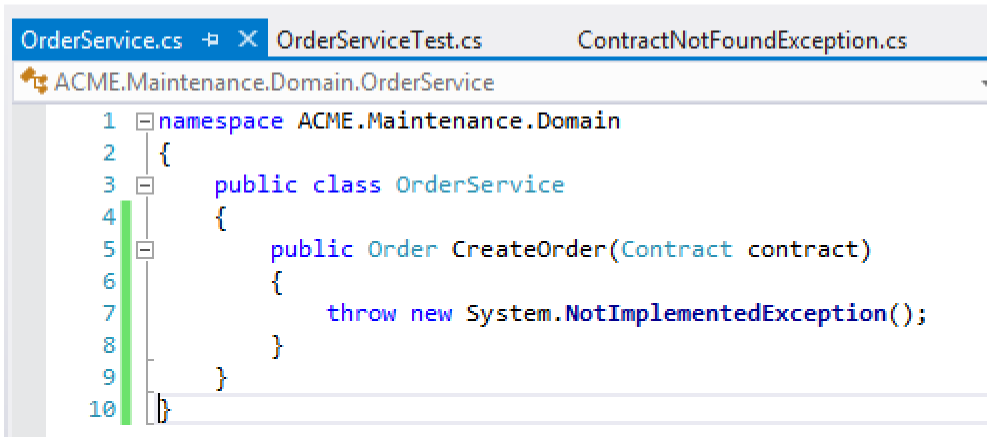
In order to get TryParse() to work, I have to give it a variable to save the GUID into. This is why I added line 57 and why I've added an out parameter ... because TryParse needed it, not because I'm ever going to do anything with the guidOut variable.

6. Implementing (and stubbing out) the OrderService class and CreateOrder method

I decide to stub out the OrderService, Order and OrderItem classes. This will help to get the unit test project to at least compile.

I add a new OrderService.cs to the Domain Layer project.

Also, I partially implement the CreateOrder() method. I'm not ready to fully implement it just yet ... so to remind myself that I've left this undone, I raise a NotImplementedException() exception.

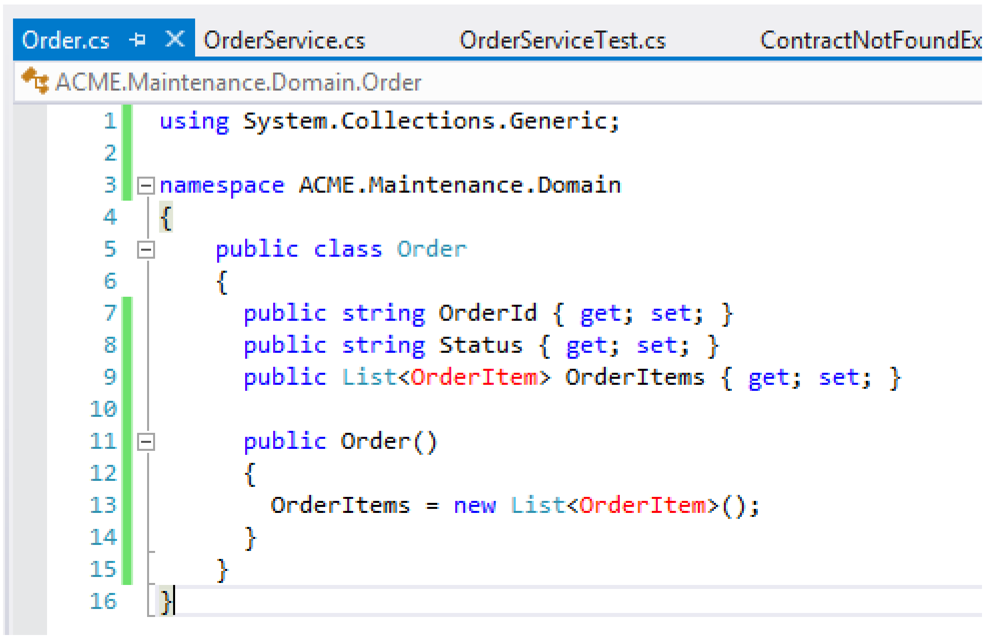

OrderService.cs
OrderServiceTest.cs
ContractNotFoundException.cs
ACME.Maintenance.Domain.OrderService

```
namespace ACME.Maintenance.Domain
{
    public class OrderService
    {
        public Order CreateOrder(Contract contract)
        {
            throw new System.NotImplementedException();
        }
    }
}
```

7. Implementing the Order class

I create the Order object to satisfy the design in the unit test. I do this by adding a new Order.cs to the Domain Layer project.

In the constructor, I initialize the List<OrderItem>().



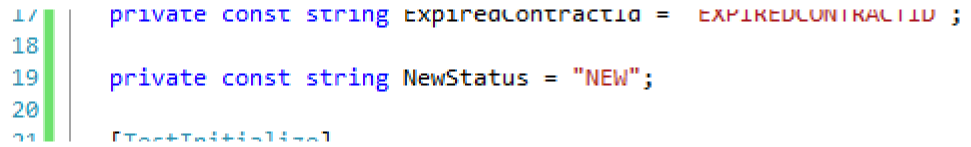
```
using System.Collections.Generic;

namespace ACME.Maintenance.Domain
{
    public class Order
    {
      public string OrderId { get; set; }
      public string Status { get; set; }
      public List<OrderItem> OrderItems { get; set; }

      public Order()
      {
        OrderItems = new List<OrderItem>();
      }
    }
}
```

I immediately have some reservations about this. Giving public access to the List<OrderItem> means that I, the creator of this class have no control over what other developers (even if that other developer is me sometime in the future) who consume this class. In other words, if you can access the List<T> then you can add, remove, etc. items to and from the list. In some cases, that would be acceptable. However, in this case, I may want to take control of the process of adding a new OrderItem to an Order. I may want to use that moment in time to performa a calculation, like adding the new OrderItem's line item total to the Order's subtotal. There may be other "sanity checks" I need to perform prior to committing the new OrderItem to the Order, like reserving the item in the warehouse's inventory.

I consider this, but move on for the moment.

8. Moving the magic string for "New" Status to a constant

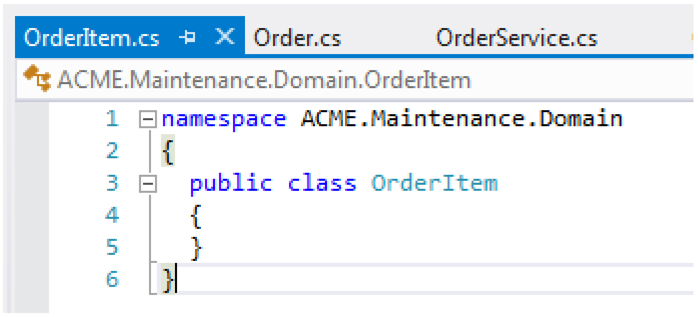
In an effort to clean up the unit test, I take a brief aside and create a constant for the NewStatus. This was a minor improvement, but I dislike "magic strings". I'm more convinced than ever that I'll eventually be replacing this with an Enumeration.



```
private const string NewStatus = "NEW";
```

9. Stubbing out the OrderItem class

In the unit test, I see I still have not created a stubbed out version of the OrderItem class. I create the bare minimum in hopes of getting the unit test to compile. I do this by adding an OrderItem.cs to the Domain Layer project.



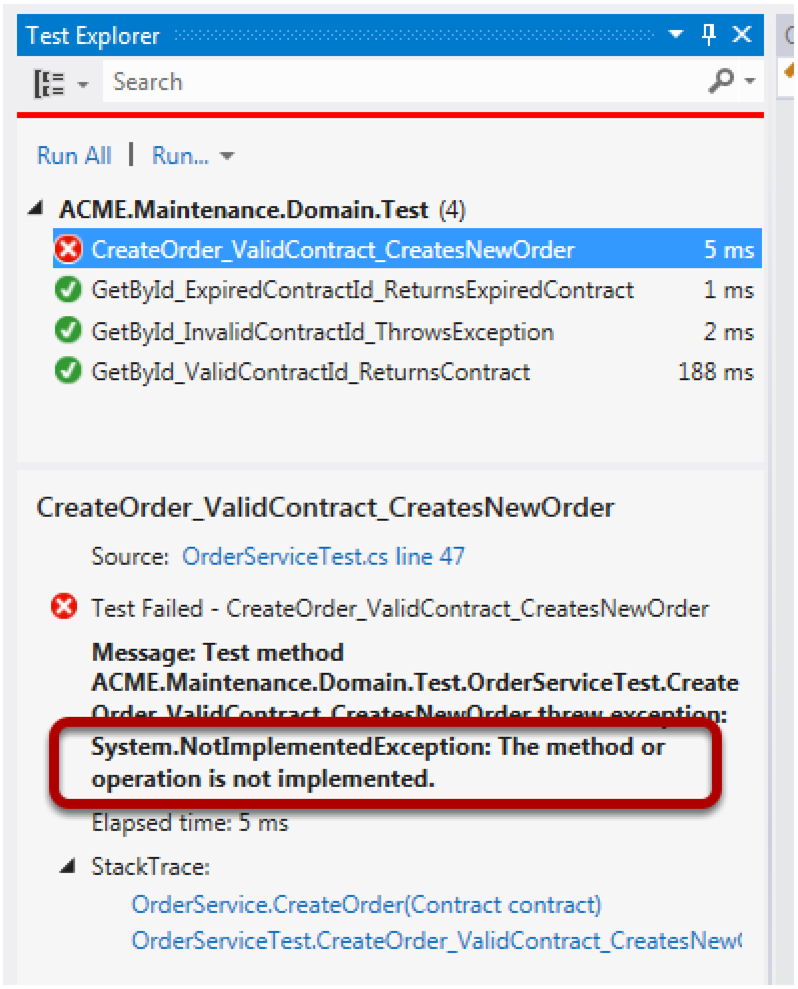
```
namespace ACME.Maintenance.Domain
{
    public class OrderItem
    {
    }
}
```

Get this code:

https://gist.github.com/LearnVisualStudio/a1417e4a5702e6ef718a

10. Run All Unit Tests

Now that I've added new classes, methods and properties to satisfy the design of the new unit test, I decide to Run All unit tests.

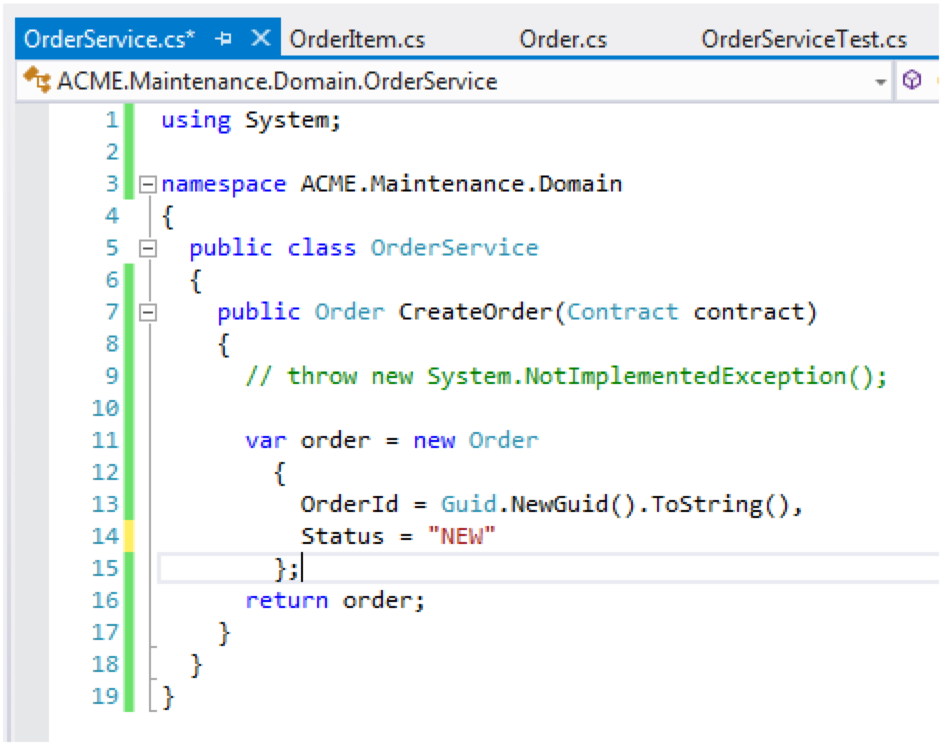


```
ACME.Maintenance.Domain.Test (4)
  CreateOrder_ValidContract_CreatesNewOrder               5 ms
  GetById_ExpiredContractId_ReturnsExpiredContract        1 ms
  GetById_InvalidContractId_ThrowsException               2 ms
  GetById_ValidContractId_ReturnsContract               188 ms

CreateOrder_ValidContract_CreatesNewOrder
  Source: OrderServiceTest.cs line 47
  Test Failed - CreateOrder_ValidContract_CreatesNewOrder
  Message: Test method
  ACME.Maintenance.Domain.Test.OrderServiceTest.Create
  Order_ValidContract_CreatesNewOrder threw exception:
  System.NotImplementedException: The method or
  operation is not implemented.
  Elapsed time: 5 ms
  StackTrace:
    OrderService.CreateOrder(Contract contract)
    OrderServiceTest.CreateOrder_ValidContract_CreatesNew(
```

Doing so shows there's a problem with the new unit test ... at first I'm alarmed, however after inspecting the message, I realize that I intended to raise that exception ... because I did not yet implement the CreateOrder() method. Let's tackle that now.

11. Revisiting the OrderService class' CreateOrder() method

To get the unit test to pass, I need to fully implement the CreateOrder() method.

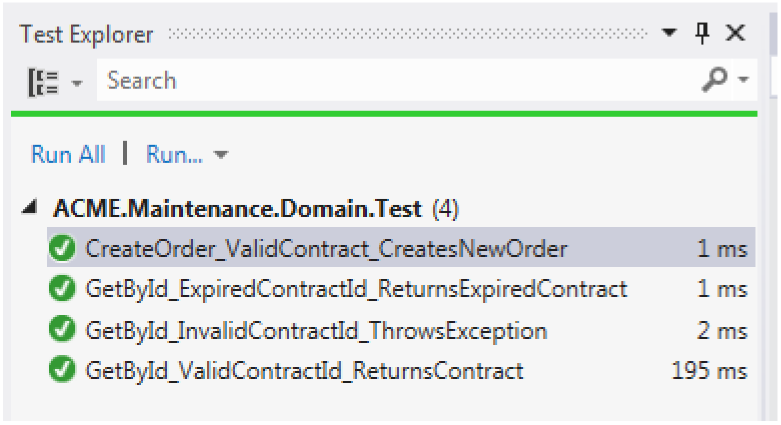

OrderService.cs* OrderItem.cs Order.cs OrderServiceTest.cs

ACME.Maintenance.Domain.OrderService

```
using System;

namespace ACME.Maintenance.Domain
{
  public class OrderService
  {
    public Order CreateOrder(Contract contract)
    {
      // throw new System.NotImplementedException();

      var order = new Order
        {
          OrderId = Guid.NewGuid().ToString(),
          Status = "NEW"
        };
      return order;
    }
  }
}
```

The CreateOrder() method seems simple at this point. All we need to do is create a new OrderId (generating it from the Guid class) and set the Status property to "NEW". Here's another magic string, which I know I'll be refactoring out soon.

Run All unit tests reveals that the implementation of CreateOrder() worked.

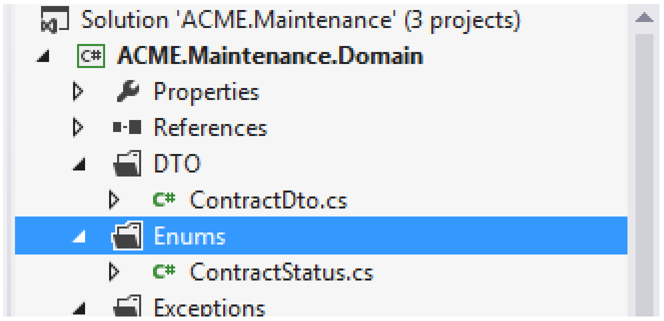


```
ACME.Maintenance.Domain.Test (4)
  ✓ CreateOrder_ValidContract_CreatesNewOrder        1 ms
  ✓ GetById_ExpiredContractId_ReturnsExpiredContract 1 ms
  ✓ GetById_InvalidContractId_ThrowsException       2 ms
  ✓ GetById_ValidContractId_ReturnsContract        195 ms
```

But now we need to think about refactoring, and the "magic string" is on the list.

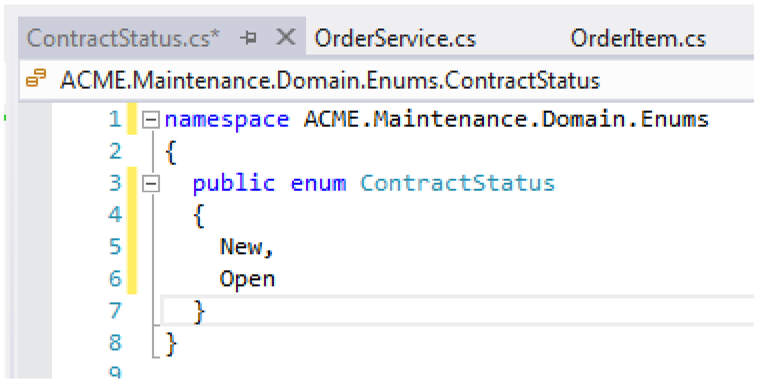
12. Refactoring the Order class' Status property to an Enum

In the Domain Layer project, create a new folder called Enums, and add a new class called ContractStatus.cs to the new folder.



The ContractStatus.cs file should contain the definition for the ContractStatus enumeration as depicted in the screenshot. As of now, we only need one enumeration value (New) but because I'm feeling rebelous I'll add a second one (Open). Ideally, I would not have added that second enumeration until I revealed the need for it in a unit test.

Here's what it looks like at the moment:



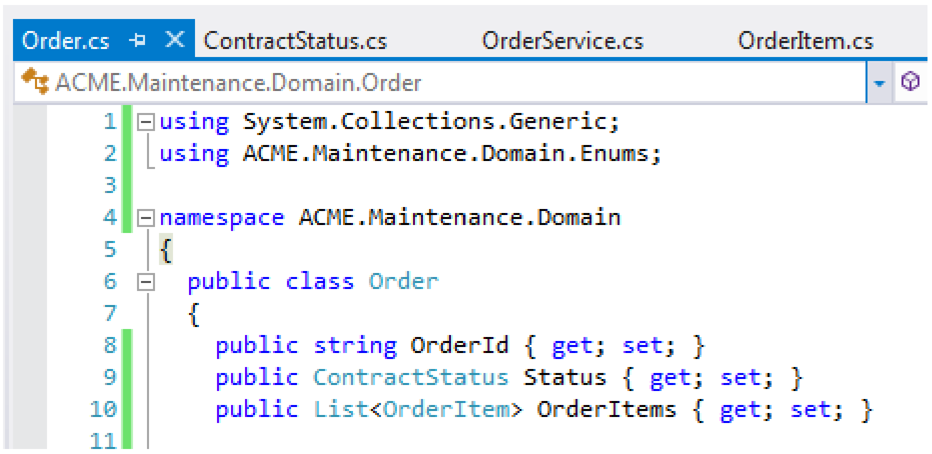
```
namespace ACME.Maintenance.Domain.Enums
{
  public enum ContractStatus
  {
    New,
    Open
  }
}
```

Get this code:

https://gist.github.com/LearnVisualStudio/22ea1a70db4565df06e8

Now, I'll start replacing the magic string "New" everywhere with this new shiny ContractStatus enum.

In the Order.cs, I edit the Status property to be of type ContractStatus.

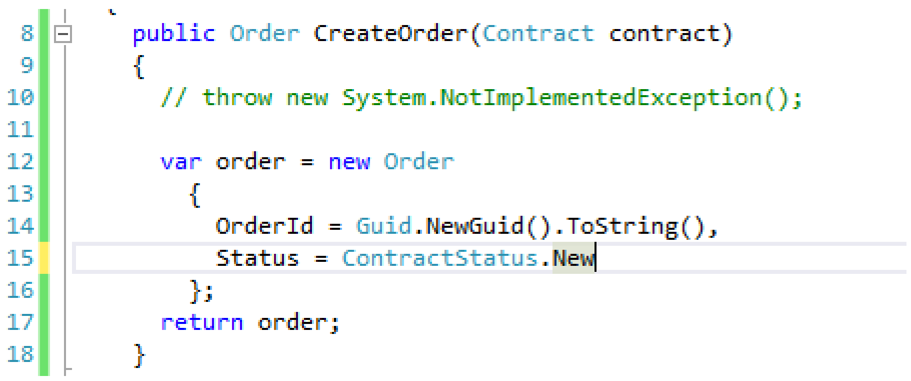


```csharp
using System.Collections.Generic;
using ACME.Maintenance.Domain.Enums;

namespace ACME.Maintenance.Domain
{
  public class Order
  {
    public string OrderId { get; set; }
    public ContractStatus Status { get; set; }
    public List<OrderItem> OrderItems { get; set; }
```

Get the code for the Order.cs (with all changes up this this point):

https://gist.github.com/LearnVisualStudio/a491ee25980e02748db0

In the OrderService.cs, I modify the object initializer for the Status property and set it to the new enumeration.

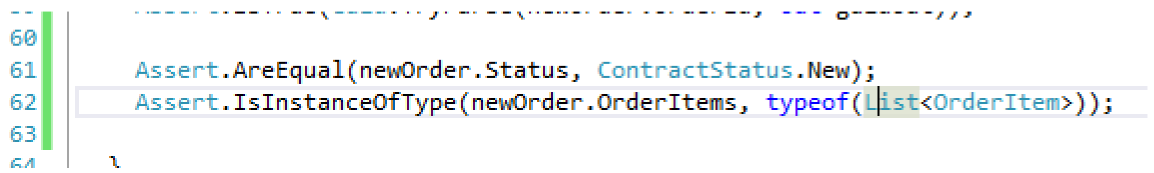


```csharp
  public Order CreateOrder(Contract contract)
  {
    // throw new System.NotImplementedException();

    var order = new Order
      {
        OrderId = Guid.NewGuid().ToString(),
        Status = ContractStatus.New
      };
    return order;
  }
```

Get the code for the OrderService.cs (with all changes up this this point):

https://gist.github.com/LearnVisualStudio/6f40fc400d393a3f15c0

Finally, I modify the unit test itself, removing the NewStatus constant at the top of the file, and the Assert in the unit test itself.

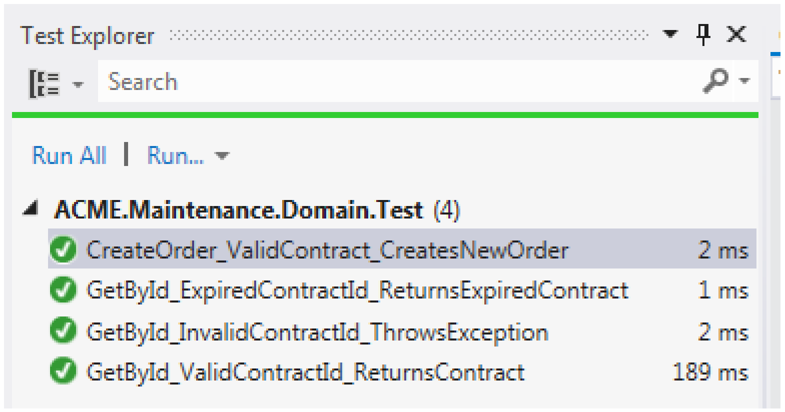


```
    Assert.AreEqual(newOrder.Status, ContractStatus.New);
    Assert.IsInstanceOfType(newOrder.OrderItems, typeof(List<OrderItem>));

}
```

Get the code for the OrderServiceTest.cs (with all changes up to this point):

https://gist.github.com/LearnVisualStudio/f5116d41f4f984c46dc1

Now we're ready to Run All unit tests again ...



```
ACME.Maintenance.Domain.Test (4)
    CreateOrder_ValidContract_CreatesNewOrder              2 ms
    GetById_ExpiredContractId_ReturnsExpiredContract       1 ms
    GetById_InvalidContractId_ThrowsException              2 ms
    GetById_ValidContractId_ReturnsContract              189 ms
```

... and it works.

Lesson 16 - Creating the Exceptional Case for Creating an Order using an Expired Contract

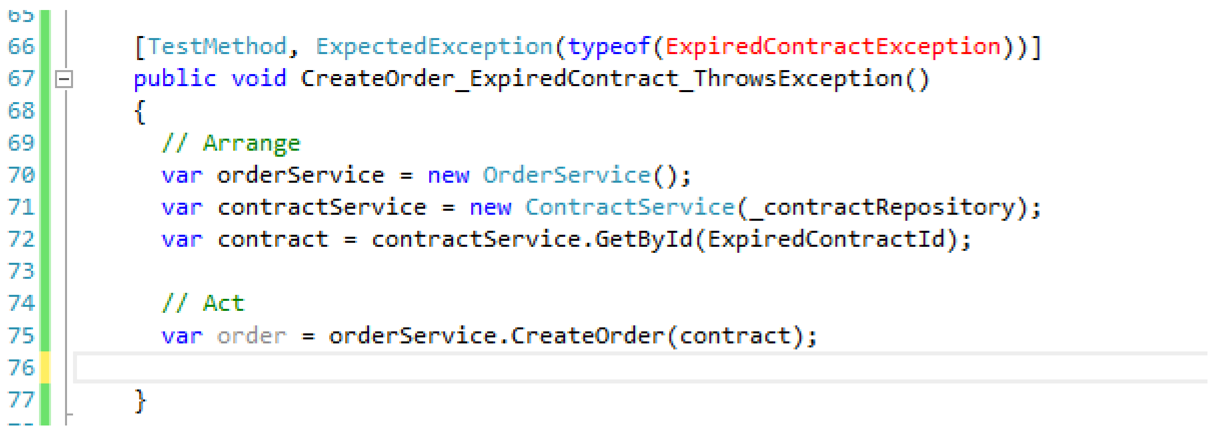
In this lesson, we'll focus on the case where we attempt to create a new Order using an Expired Contract. We would expect an exception to be thrown. We'll create a new Custom Exception to throw from the CreateOrder() method to make sure this happens.

Since we've already created and unit tested custom exceptions, this should go smoothly.

First, I'll create the unit test for the expired contract scenario.

1. Add a new unit test to throw an exception when searching for an expired contract

Here we designate this unit test to expect an exception that we'll call ExpiredContractException. We'll write just enough code to trigger the exception.

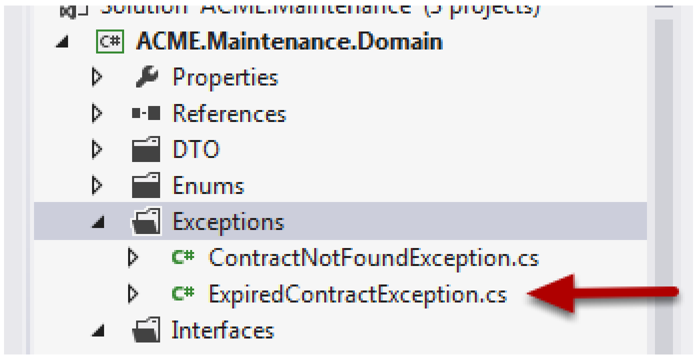


```
    [TestMethod, ExpectedException(typeof(ExpiredContractException))]
    public void CreateOrder_ExpiredContract_ThrowsException()
    {
      // Arrange
      var orderService = new OrderService();
      var contractService = new ContractService(_contractRepository);
      var contract = contractService.GetById(ExpiredContractId);

      // Act
      var order = orderService.CreateOrder(contract);

    }
```

Get this code:

https://gist.github.com/LearnVisualStudio/11861643e670dd80f310

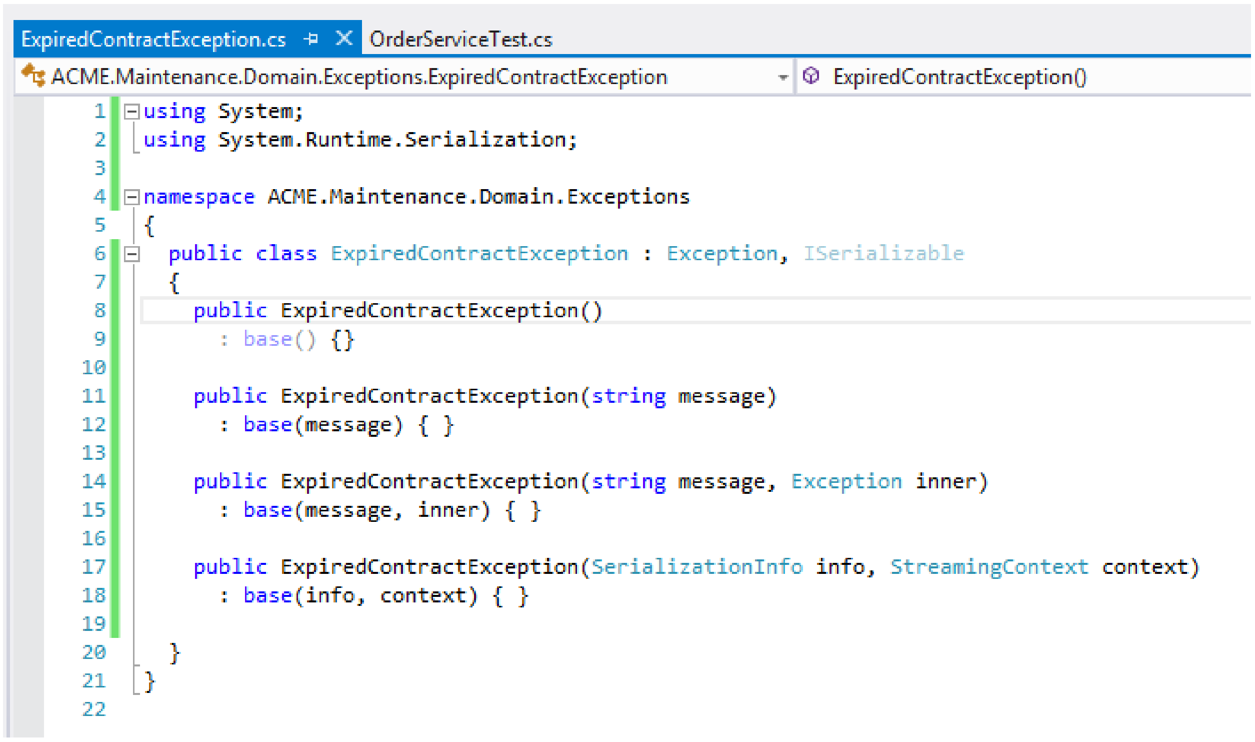
2. Add a new ExpiredContractException.cs to the Domain Layer project's Exceptions folder



```
Solution 'ACME.Maintenance' (5 projects)
◢ ACME.Maintenance.Domain
   ▷ Properties
   ▷ References
   ▷ DTO
   ▷ Enums
   ◢ Exceptions
      ▷ ContractNotFoundException.cs
      ▷ ExpiredContractException.cs
   ◢ Interfaces
```

3. Implement the new ExpiredContractException.cs

We'll implement the custom exception just like we did in a previous lesson.


ExpiredContractException.cs
OrderServiceTest.cs
ACME.Maintenance.Domain.Exceptions.ExpiredContractException
ExpiredContractException()

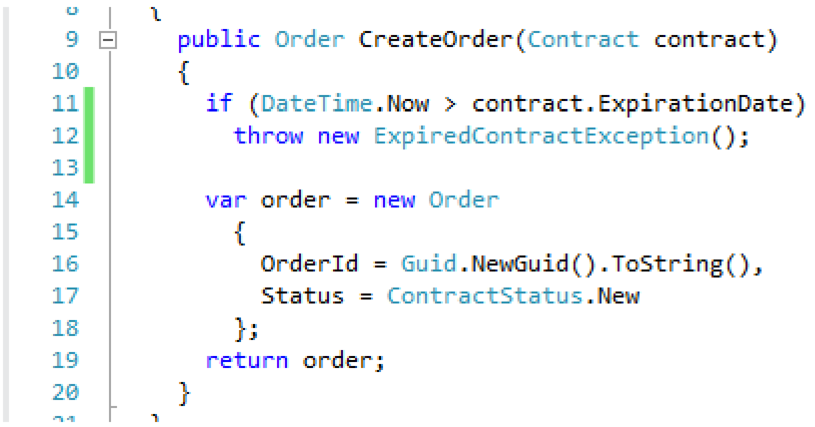
```
using System;
using System.Runtime.Serialization;

namespace ACME.Maintenance.Domain.Exceptions
{
  public class ExpiredContractException : Exception, ISerializable
  {
    public ExpiredContractException()
      : base() {}

    public ExpiredContractException(string message)
      : base(message) { }

    public ExpiredContractException(string message, Exception inner)
      : base(message, inner) { }

    public ExpiredContractException(SerializationInfo info, StreamingContext context)
      : base(info, context) { }

  }
}
```

Get this code:

https://gist.github.com/LearnVisualStudio/f005b4cc26bc69a5727d

4. Revise the OrderService class' CreateOrder() method to check for ExpirationDate

Now, I'll write the "gate" ... checking the contract.ExpirationDate ... if it's in the past, then throw the new custom exception.

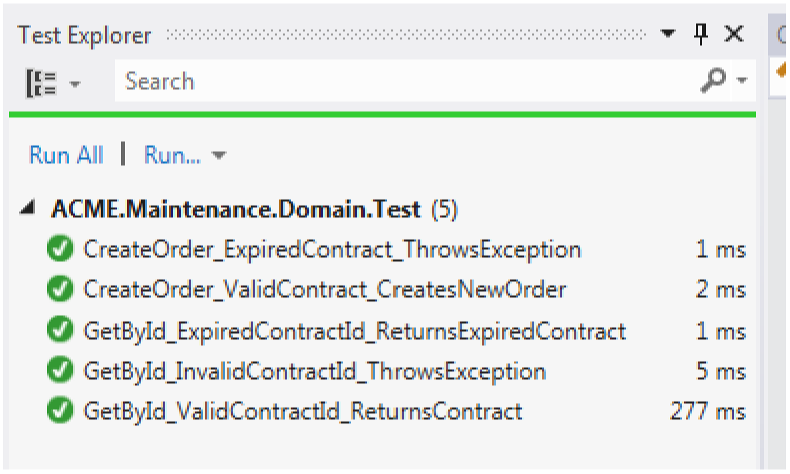


```
    public Order CreateOrder(Contract contract)
    {
      if (DateTime.Now > contract.ExpirationDate)
        throw new ExpiredContractException();

      var order = new Order
        {
          OrderId = Guid.NewGuid().ToString(),
          Status = ContractStatus.New
        };
      return order;
    }
```

Get this code:

https://gist.github.com/LearnVisualStudio/33800180f630d84e353a

5. Run All unit tests



```
Test Explorer
Search
Run All | Run...
ACME.Maintenance.Domain.Test (5)
  CreateOrder_ExpiredContract_ThrowsException        1 ms
  CreateOrder_ValidContract_CreatesNewOrder          2 ms
  GetById_ExpiredContractId_ReturnsExpiredContract   1 ms
  GetById_InvalidContractId_ThrowsException          5 ms
  GetById_ValidContractId_ReturnsContract          277 ms
```

Success!

Lesson 17 - Adding Order Items to the Order - Part 1

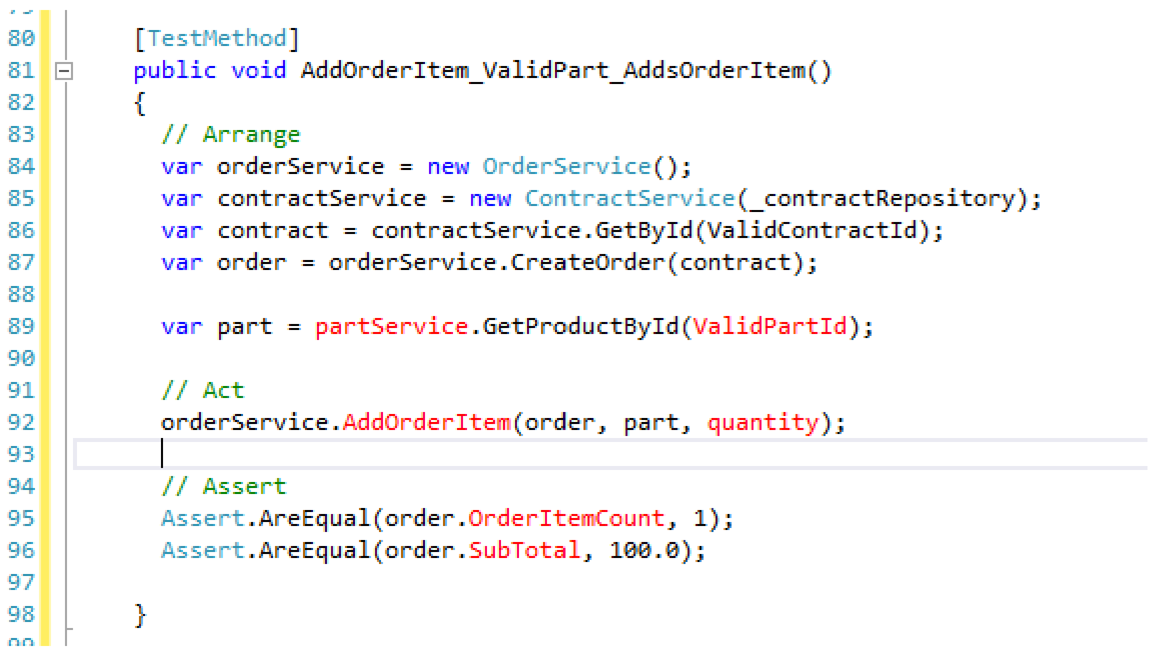
The next step is to add order items to the Order. As it stands right now, we expose a List<OrderItem> as a property of our Order called OrderItems. Since it's a List<T> you could easily add items to the Order. However, I'm not sure that's a good idea. Ideally, we would do some sanity checks and perform OrderItem total and Order total calculations at the moment we add an item to the Order. I'm beginning to think that it was a mistake to make that a List<T>, or at least, it was a mistake to make it a public property. We'll figure that out in a bit.

Let's design the process of adding OrderItems to an Order with a unit test.

I think the OrderService class is still the right class to delegate this responsibility to. In my mind, it should provide all the services the Order class needs to function, so let's start there.

1. Create the Unit Test

I struggle through the process of writing the unit test. I'm scratching my ideas down in code form, but very quickly I start hitting some hurdles and uncertainty. It's at this point I stop and take a screenshot of the current state of my code:



```
    [TestMethod]
    public void AddOrderItem_ValidPart_AddsOrderItem()
    {
      // Arrange
      var orderService = new OrderService();
      var contractService = new ContractService(_contractRepository);
      var contract = contractService.GetById(ValidContractId);
      var order = orderService.CreateOrder(contract);

      var part = partService.GetProductById(ValidPartId);

      // Act
      orderService.AddOrderItem(order, part, quantity);

      // Assert
      Assert.AreEqual(order.OrderItemCount, 1);
      Assert.AreEqual(order.SubTotal, 100.0);

    }
```

My first pass, I'm not so happy about this because it really touches a lot of parts. Ideally, I could keep things really simple. However, as I come to realize, this is really the crux of the system -- being able to add a product to an order. It will naturally touch a lot of moving pieces including the contract, the order, order items, and products.

Next up, I think I may need to back-track a little and implement the ProductService and Product. I may have gotten ahead of myself.

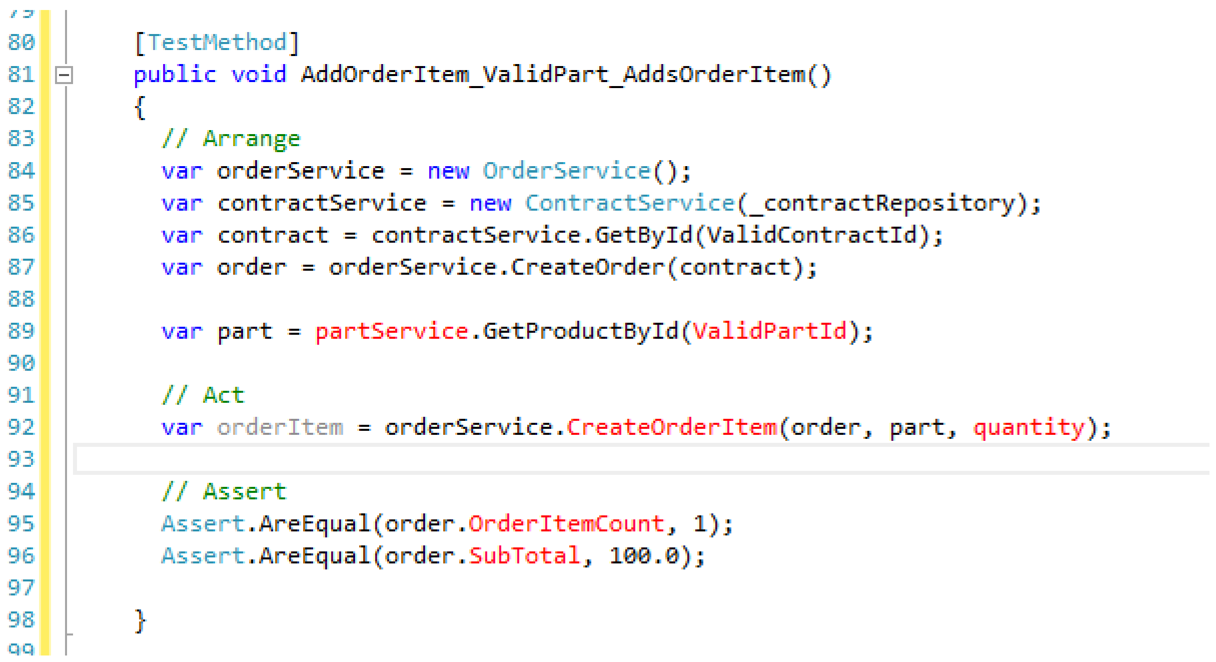
Finally, the way I have it designed in this unit test, I don't have access to the individual OrderItems outside of the Order class. So, for example, currently I can't easily look at the OrderItem to ensure that it's subtotal is correct ... I have no access to the individual items.

Do I really need that? As it turns out, I do. We do have a requirement that says we should display the items in the order onscreen as well as a running total. So, no matter what, this unit test is not going to really be representative of what I want. But what I'm trying to guard against is making the List<OrderItem> public ... List<T> provides too much access to the collection.

So, right now I'm feeling a little confused. I could tackle the parts I know, but I prefer to tackle the parts that trouble me first.

2. Refactor the Unit Test

I set out to right the wrongs of the previous attempt. This is what I come up with:



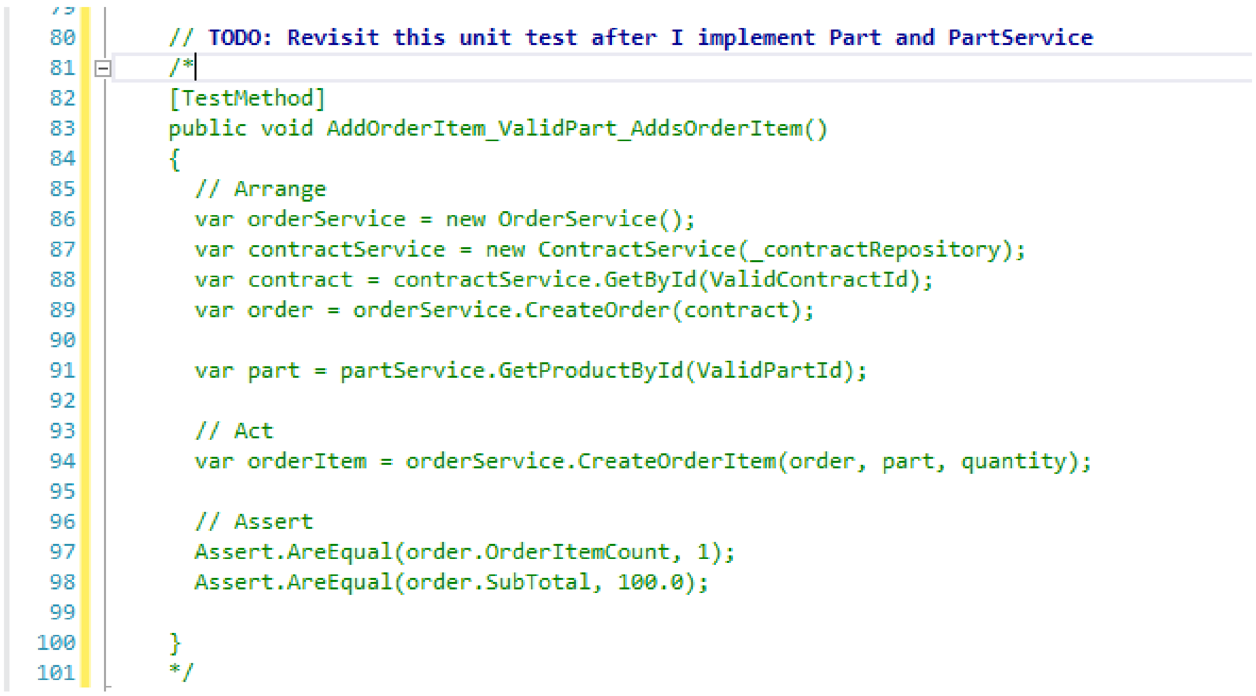
```
[TestMethod]
public void AddOrderItem_ValidPart_AddsOrderItem()
{
  // Arrange
  var orderService = new OrderService();
  var contractService = new ContractService(_contractRepository);
  var contract = contractService.GetById(ValidContractId);
  var order = orderService.CreateOrder(contract);

  var part = partService.GetProductById(ValidPartId);

  // Act
  var orderItem = orderService.CreateOrderItem(order, part, quantity);

  // Assert
  Assert.AreEqual(order.OrderItemCount, 1);
  Assert.AreEqual(order.SubTotal, 100.0);

}
```

I feel better about this second pass ... for two reasons. First, because I mistakenly was calling the Part "Product". Secondly, I was biting off way too much previously. I was trying to both create an OrderItem instance and add it to the Order in one method call. That had a few side effects. Now, the OrderItem is more visible. I still have to worry about the List<OrderItem> being too visible, but that's a concern for another use case. I've re-scoped this one to be all about CREATING an OrderItem, not ADDING it to an Order. We'll worry about that next.

Now that I feel better about it, I can stop for a moment and implement Part and PartService. Of course, I'll do that with a Unit Test. I feel bad that I'm leaving this undone.

3. Comment out the Unit Test

Rather than leave the unit test in an undone state, I decide to merely comment it out and add a TODO comment. I'll have to come back to this later.



```
        // TODO: Revisit this unit test after I implement Part and PartService
        /*
        [TestMethod]
        public void AddOrderItem_ValidPart_AddsOrderItem()
        {
          // Arrange
          var orderService = new OrderService();
          var contractService = new ContractService(_contractRepository);
          var contract = contractService.GetById(ValidContractId);
          var order = orderService.CreateOrder(contract);

          var part = partService.GetProductById(ValidPartId);

          // Act
          var orderItem = orderService.CreateOrderItem(order, part, quantity);

          // Assert
          Assert.AreEqual(order.OrderItemCount, 1);
          Assert.AreEqual(order.SubTotal, 100.0);

        }
        */
```

Get this code:

https://gist.github.com/LearnVisualStudio/3037fba4b0b8535fa9f1

On the one hand, I don't feel like I accomplished much in this lesson -- I don't have any production worthy code to show for my efforts, nor do I have a working unit test. However, I do have a solid direction. In the next lesson, I'll tackle the Parts.

Lesson 18 - Getting Parts

Now that we have some clarity on what needs to happen next, we'll tackle Parts, PartService, part exceptions and so on.

Creating an OrderItem requires locating a Part by its PartId. In this lesson, we'll implement this feature. There's two scenarios to satisfy:

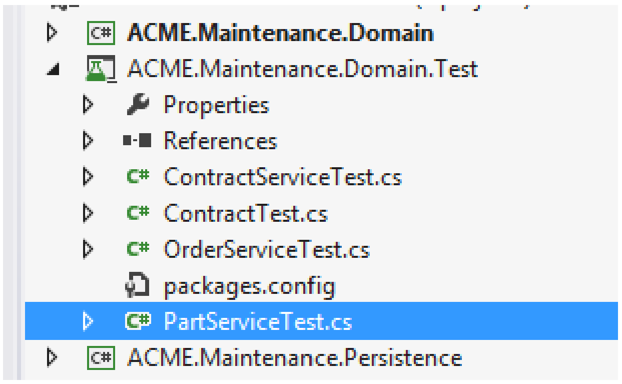
(1) Using a valid Part Id should return a valid Part object

(2) Using an invalid part id should throw an exception

Easy. We've done this a couple of times already.

We start by adding a unit test and allow it to drive the design of the PartService class, et. al.

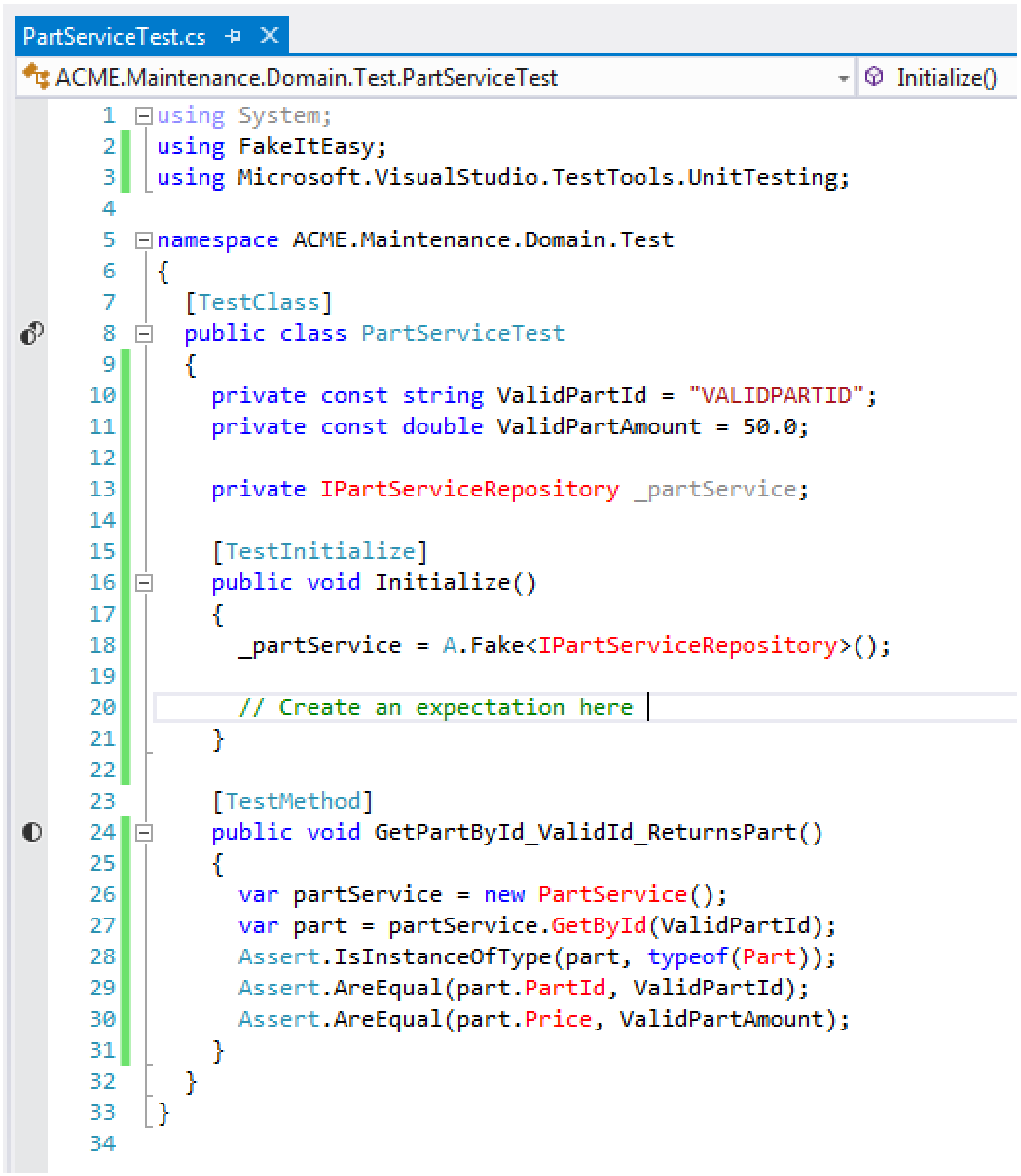
1. Add a new PartServiceTest.cs to the Unit Test project



```
ACME.Maintenance.Domain
ACME.Maintenance.Domain.Test
    Properties
    References
    ContractServiceTest.cs
    ContractTest.cs
    OrderServiceTest.cs
    packages.config
    PartServiceTest.cs
ACME.Maintenance.Persistence
```

2. Begin implementing the PartServiceTest.cs and the first Unit Test

I set out to follow the similar pattern to what I've done before. Here's what I come up with:



PartServiceTest.cs

ACME.Maintenance.Domain.Test.PartServiceTest

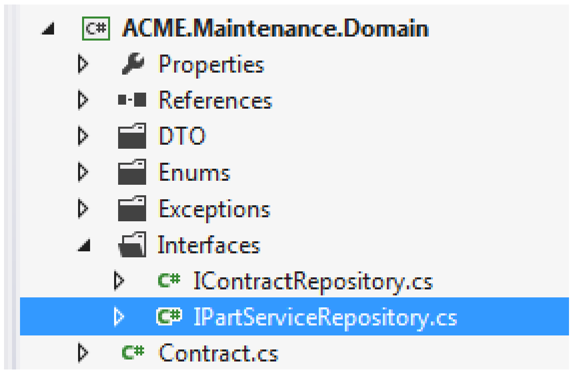
Initialize()

```
using System;
using FakeItEasy;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ACME.Maintenance.Domain.Test
{
  [TestClass]
  public class PartServiceTest
  {
    private const string ValidPartId = "VALIDPARTID";
    private const double ValidPartAmount = 50.0;

    private IPartServiceRepository _partService;

    [TestInitialize]
    public void Initialize()
    {
      _partService = A.Fake<IPartServiceRepository>();

      // Create an expectation here
    }

    [TestMethod]
    public void GetPartById_ValidId_ReturnsPart()
    {
      var partService = new PartService();
      var part = partService.GetById(ValidPartId);
      Assert.IsInstanceOfType(part, typeof(Part));
      Assert.AreEqual(part.PartId, ValidPartId);
      Assert.AreEqual(part.Price, ValidPartAmount);
    }
  }
}

```

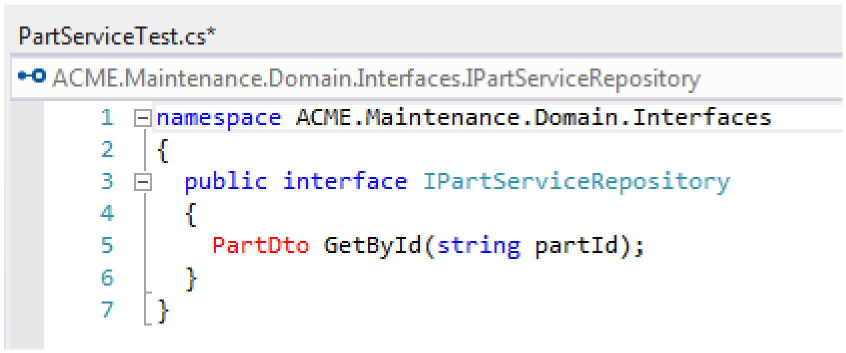
Since I've done this a couple of times, I take a few shortcuts. I know I need an IPartServiceRepository in order to create a fake to supply to the PartService. I write code for a few moments until I have a number of loose ends that need to be resolved ... I'm not yet finished with the Initialize() method, and may need to fix a thing or two in the unit test, but this is a start.

I decide that I need to get the whole dependency injection / fake mechanism working, so I start by adding the new interface for the repository.

3. Add the IPartServiceRepository.cs to the Domain Layer's Interfaces folder



4. Implement the IPartServiceRepository



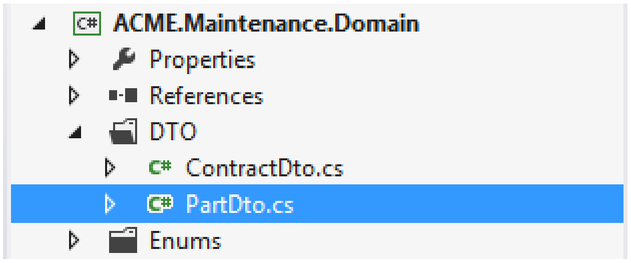
```
namespace ACME.Maintenance.Domain.Interfaces
{
    public interface IPartServiceRepository
    {
        PartDto GetById(string partId);
    }
}
```

From previous experience, I know I want to return a PartDto. I'll add that next.

Get this code:

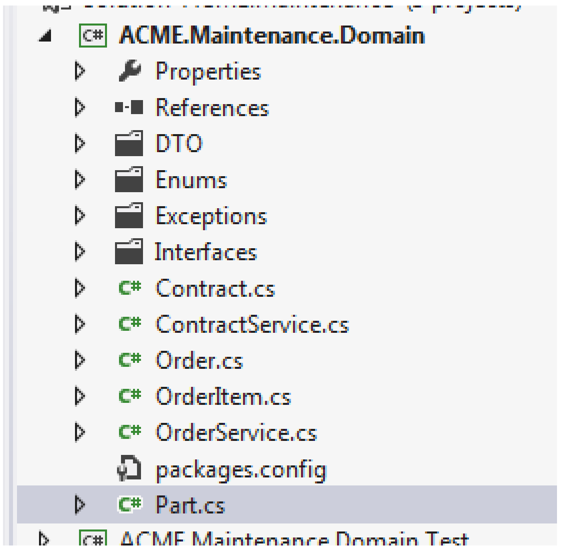
https://gist.github.com/LearnVisualStudio/e939b56f172f88c8240b

5. Add the PartDto.cs to the Domain Layer project's DTO folder



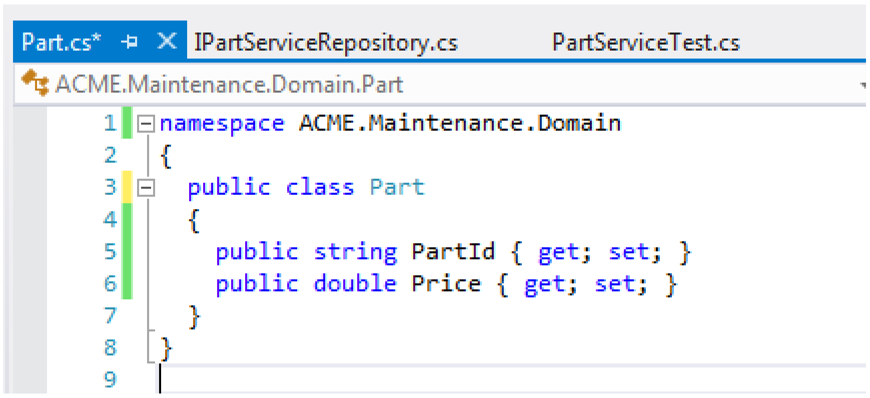
6. Add the Part.cs to the Domain Layer project

While I'm adding files I decide to also add the Part.cs file which will contain the Part class.



The PartDto and Part class will have an identical pair of properties.

7. Implement the Part class



```
namespace ACME.Maintenance.Domain
{
  public class Part
  {
    public string PartId { get; set; }
    public double Price { get; set; }
  }
}

```

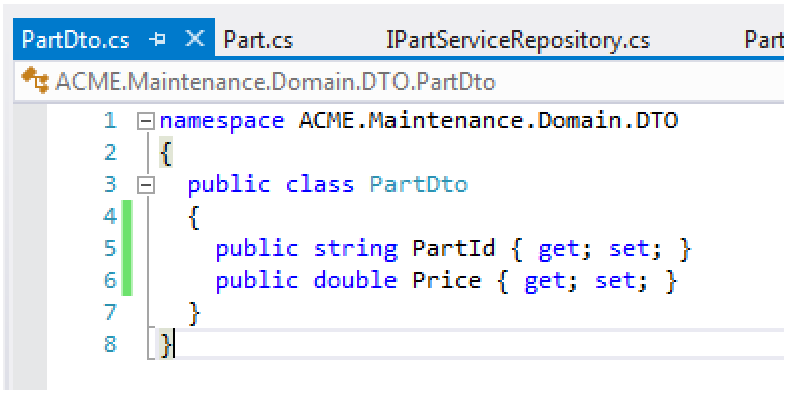
Get this code:

https://gist.github.com/LearnVisualStudio/f3754c85cf70d3eda8b9

Since the PartDto class will be identical, I copy the two properties ...

8. Implement the PartDto class

... and paste them into the PartDto class' body.

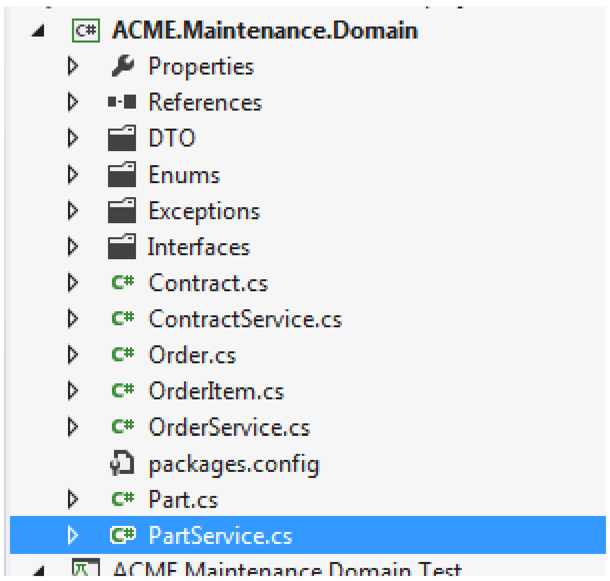

PartDto.cs
Part.cs
IPartServiceRepository.cs
Part
ACME.Maintenance.Domain.DTO.PartDto

```
namespace ACME.Maintenance.Domain.DTO
{
  public class PartDto
  {
    public string PartId { get; set; }
    public double Price { get; set; }
  }
}
```

Get this code:

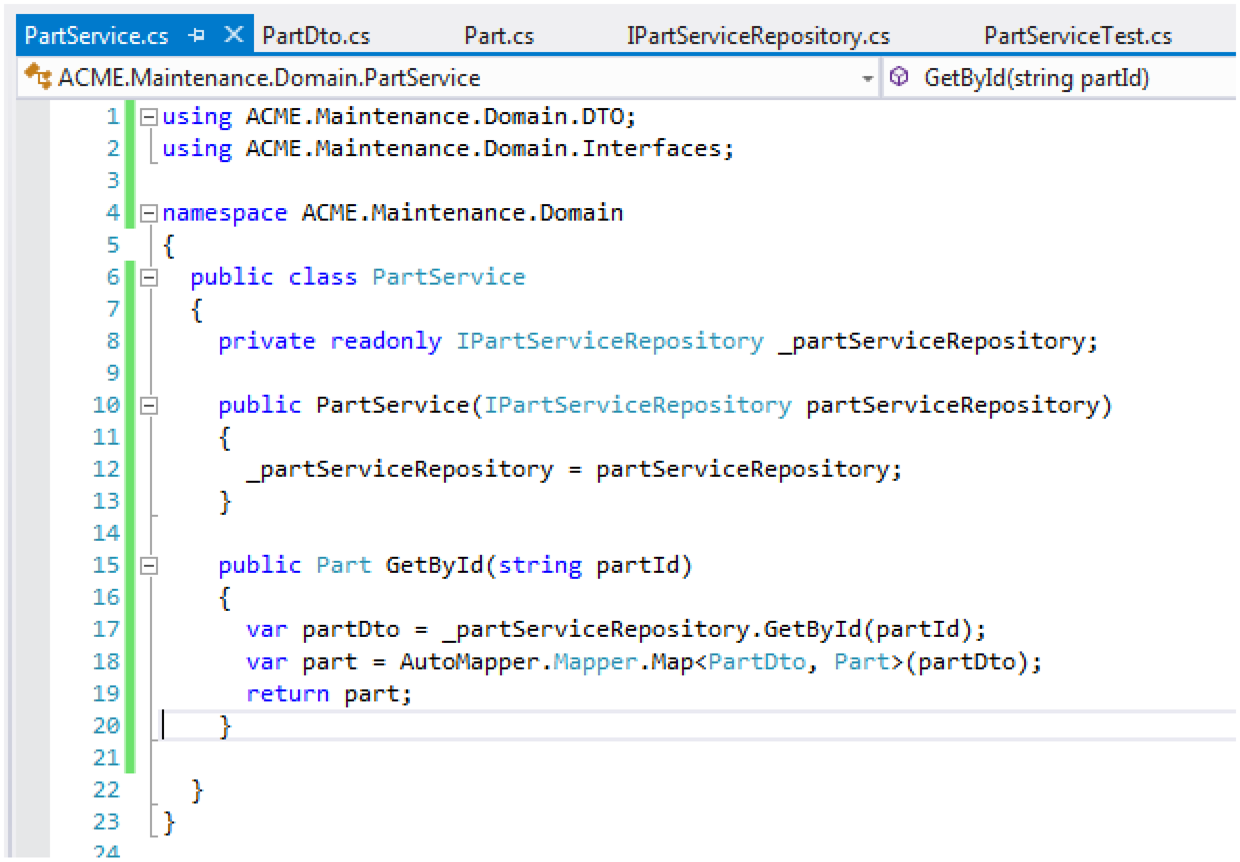
https://gist.github.com/LearnVisualStudio/a9c962df67e9b6ed7008

9. Add the PartService.cs class to the Domain Layer project



10. Implement the PartService class

The PartService class is ultimately what we want to test ... more specifically, the GetById() method. Now that we have all the supporting classes created, we can focus on its implementation.



```
 1  using ACME.Maintenance.Domain.DTO;
 2  using ACME.Maintenance.Domain.Interfaces;
 3
 4  namespace ACME.Maintenance.Domain
 5  {
 6    public class PartService
 7    {
 8      private readonly IPartServiceRepository _partServiceRepository;
 9
10      public PartService(IPartServiceRepository partServiceRepository)
11      {
12        _partServiceRepository = partServiceRepository;
13      }
14
15      public Part GetById(string partId)
16      {
17        var partDto = _partServiceRepository.GetById(partId);
18        var part = AutoMapper.Mapper.Map<PartDto, Part>(partDto);
19        return part;
20      }
21
22    }
23  }
24
```

Lines 8 through 13 allow for constructor injection of a repository. We've seen this a couple of times now, and the pattern should look familiar by now.

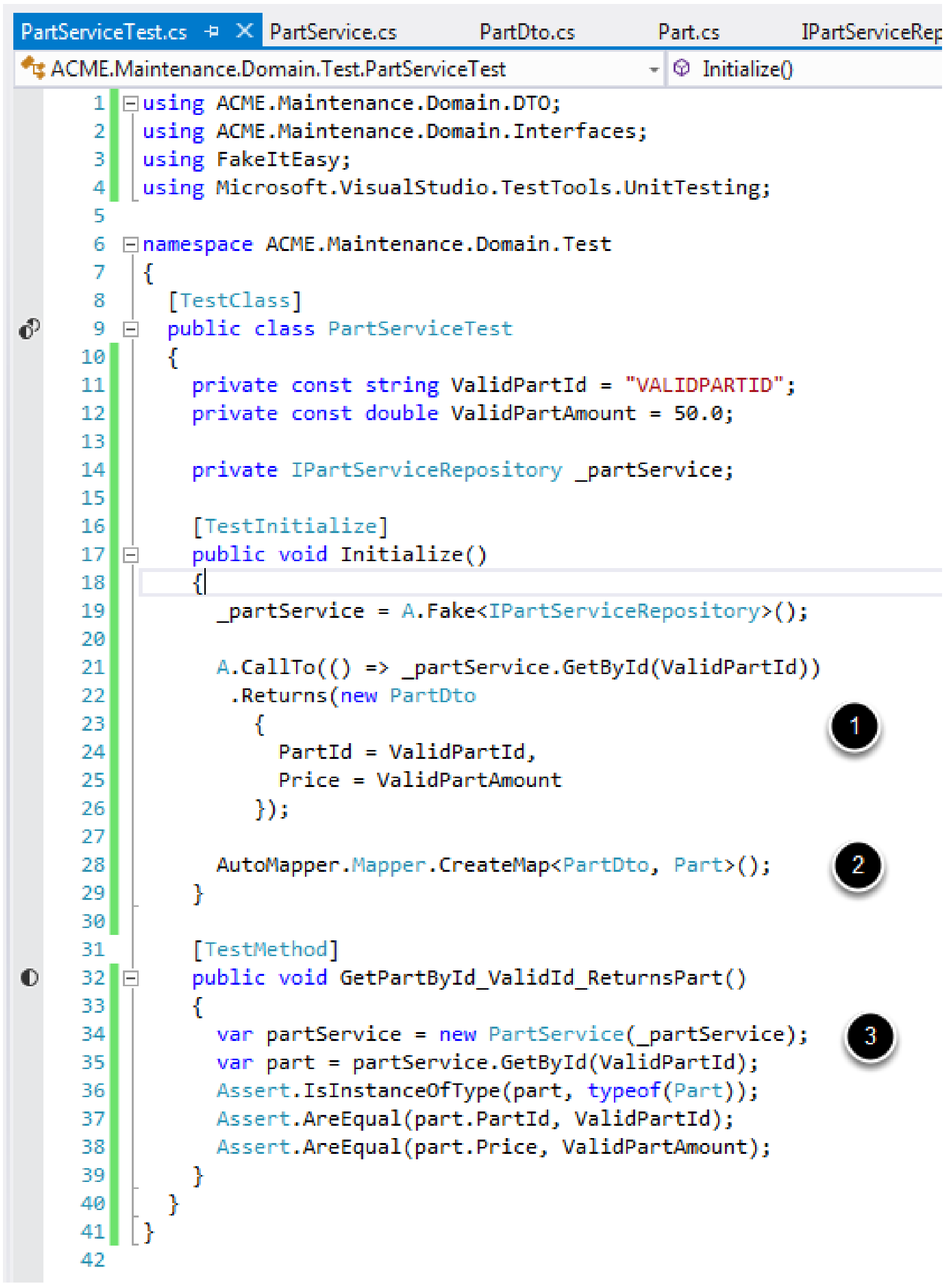
Lines 15 through 20 are where we implement the GetById() method. Here again, we've seen something similar to this before. Besides calling the repository to get the DTO, then mapping it back to the Part class, there's not much going on here.

Get this code:

https://gist.github.com/LearnVisualStudio/335f88394b54de032d50

11. Finish implementing the PartServiceTest.cs

Now that we have implementing a number of the classes, methods and properties required for this test to compile, we're ready to tackle this again. It becomes obvious that I'm missing some key parts:



```
using ACME.Maintenance.Domain.DTO;
using ACME.Maintenance.Domain.Interfaces;
using FakeItEasy;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ACME.Maintenance.Domain.Test
{
  [TestClass]
  public class PartServiceTest
  {
    private const string ValidPartId = "VALIDPARTID";
    private const double ValidPartAmount = 50.0;

    private IPartServiceRepository _partService;

    [TestInitialize]
    public void Initialize()
    {
      _partService = A.Fake<IPartServiceRepository>();

      A.CallTo(() => _partService.GetById(ValidPartId))
       .Returns(new PartDto
         {
           PartId = ValidPartId,
           Price = ValidPartAmount
         });

      AutoMapper.Mapper.CreateMap<PartDto, Part>();
    }

    [TestMethod]
    public void GetPartById_ValidId_ReturnsPart()
    {
      var partService = new PartService(_partService);
      var part = partService.GetById(ValidPartId);
      Assert.IsInstanceOfType(part, typeof(Part));
      Assert.AreEqual(part.PartId, ValidPartId);
      Assert.AreEqual(part.Price, ValidPartAmount);
    }
  }
}
```

(1) I needed to add the expectation for FakeItEasy so that it can supply a PartDto for a valid Part Id

(2) I needed to use AutoMapper to create the mapping between PartDto and Part

(3) I forgot to pass in the fake IPartServiceRepository

Also, I notice that I chose a confusing name for this variable. It shouldn't be:

_partService

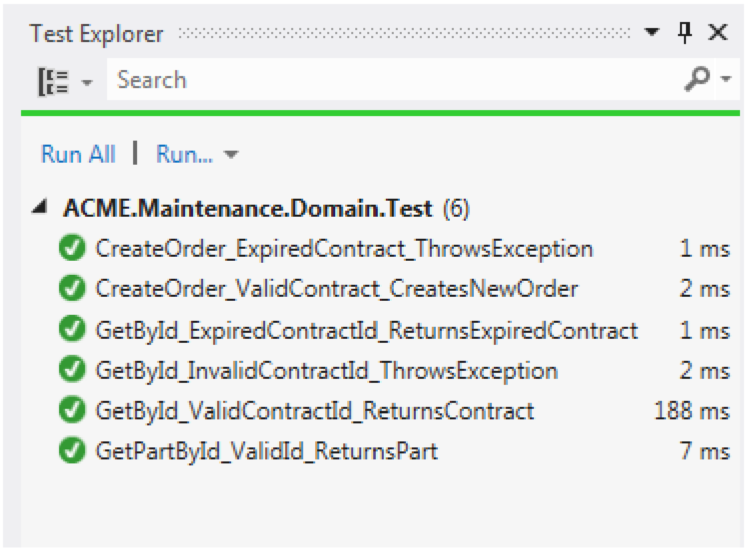
... rather, it should be:

_partServiceRepository

... I'll be sure to fix that soon.

At this point, I want some reassurance that the tests can compile and if not, figure out what I missed.

12. Run All Unit Tests



```
ACME.Maintenance.Domain.Test (6)
  ✓ CreateOrder_ExpiredContract_ThrowsException        1 ms
  ✓ CreateOrder_ValidContract_CreatesNewOrder          2 ms
  ✓ GetById_ExpiredContractId_ReturnsExpiredContract   1 ms
  ✓ GetById_InvalidContractId_ThrowsException          2 ms
  ✓ GetById_ValidContractId_ReturnsContract          188 ms
  ✓ GetPartById_ValidId_ReturnsPart                    7 ms
```

Fortunately, all the unit tests run.

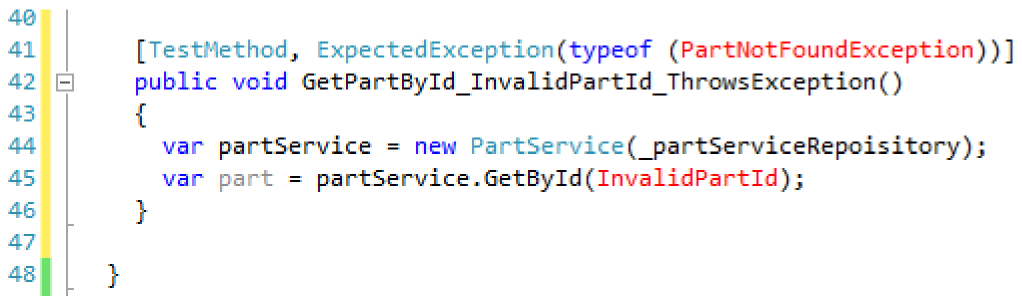
At this point, I know I want to refactor the name of the private variable _partService, and I probably should do that, but I decide to tackle that in the next unit test.

13. Create unit test for the exceptional case where an invalid PartId is submitted to GetById()

Here again, we want to check to make sure the right exception can bubble up from the persistence layer, through the domain layer and to the original caller.

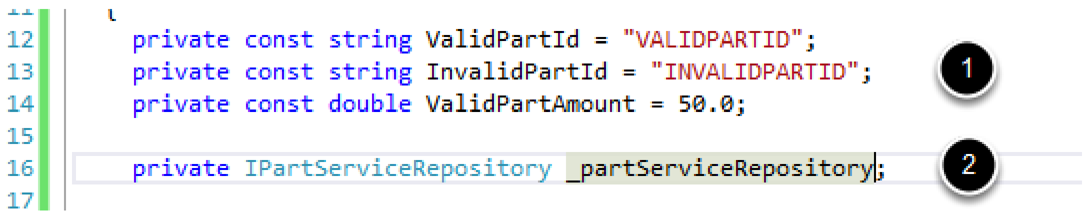
So, I'll create another unit test and tell it that it should expect an exception called PartNotFoundException.

I'll also need to create a constant representing an InvalidPartId. Here's my new unit test:



```
    [TestMethod, ExpectedException(typeof (PartNotFoundException))]
    public void GetPartById_InvalidPartId_ThrowsException()
    {
      var partService = new PartService(_partServiceRepoisitory);
      var part = partService.GetById(InvalidPartId);
    }

}
```

14. Add constant for InvalidPartId

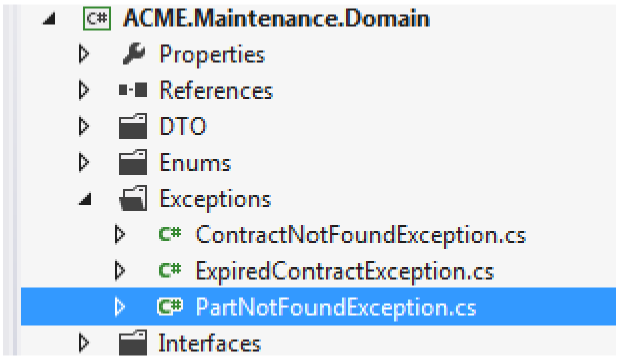


```
12      private const string ValidPartId = "VALIDPARTID";
13      private const string InvalidPartId = "INVALIDPARTID";
14      private const double ValidPartAmount = 50.0;
15
16      private IPartServiceRepository _partServiceRepository;
17
```

(1) In line 12 I add the constant for the InvalidPartId.

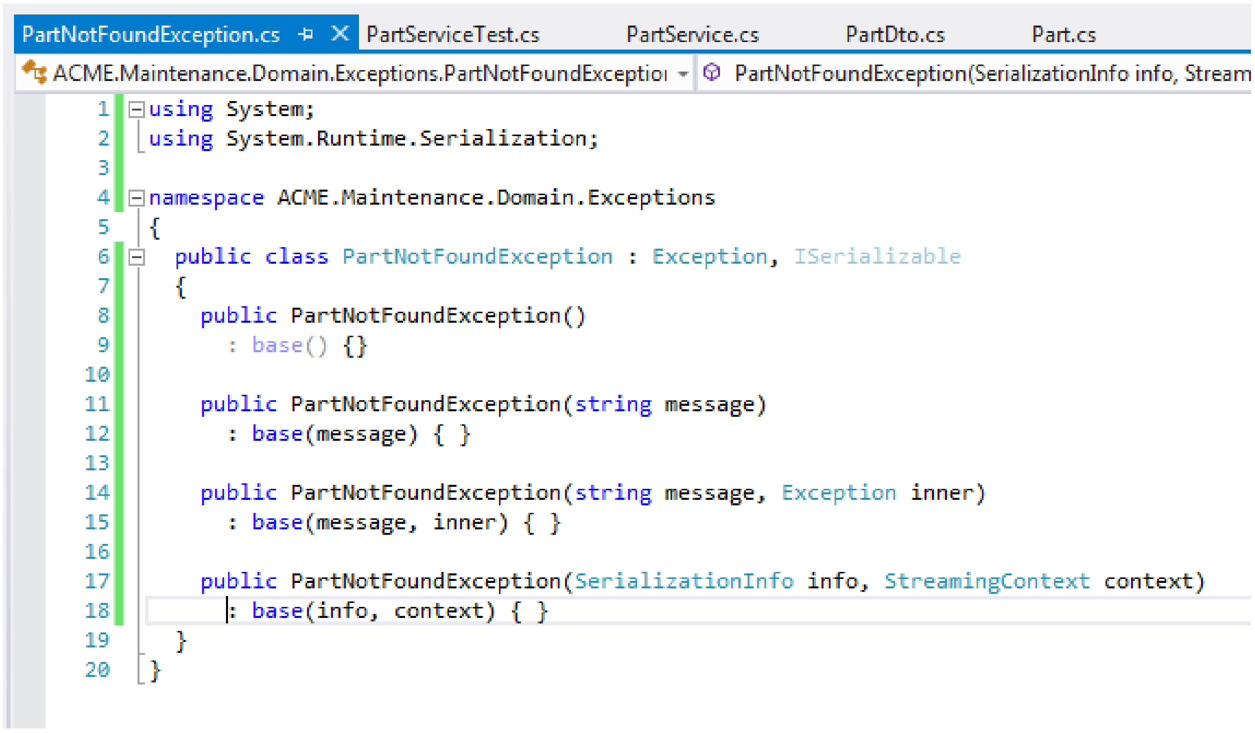
(2) While I'm here, I refactor the name of the private member that holds a reference to the fake repository. I use Visual Studio's refactor helper (the little red-dash under the last character of the variable name) to choose to propogate that change to the rest of the code file. This will insure that the rename happens everywhere I had used the old name.

15. Add the PartNotFoundException.cs to the Domain Layer project's Exceptions folder



16. Implement the PartNotFoundException class

We've seen the proper way to implement a custom exception a couple of times already, so this should not be new to you.


PartNotFoundException.cs
PartServiceTest.cs
PartService.cs
PartDto.cs
Part.cs
ACME.Maintenance.Domain.Exceptions.PartNotFoundExceptio
PartNotFoundException(SerializationInfo info, Stream

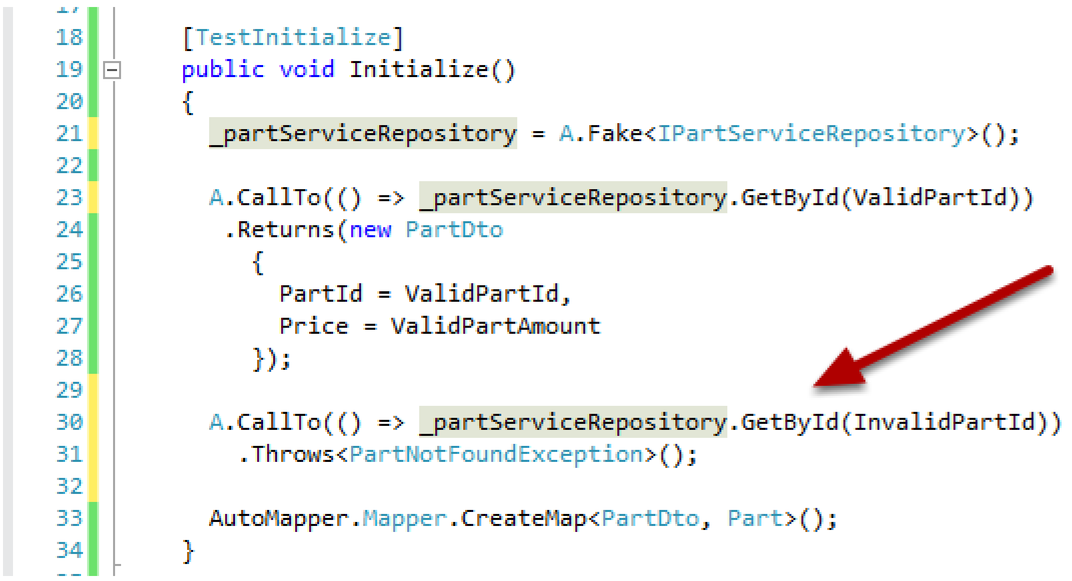
```
using System;
using System.Runtime.Serialization;

namespace ACME.Maintenance.Domain.Exceptions
{
  public class PartNotFoundException : Exception, ISerializable
  {
    public PartNotFoundException()
      : base() {}

    public PartNotFoundException(string message)
      : base(message) { }

    public PartNotFoundException(string message, Exception inner)
      : base(message, inner) { }

    public PartNotFoundException(SerializationInfo info, StreamingContext context)
      : base(info, context) { }
  }
}
```

Get this code:

https://gist.github.com/LearnVisualStudio/82dc6e11252db2800f61

17. Back in the PartServiceTest.cs, add the FakeItEasy expectation for the invalid PartId

We need to tell FakeItEasy that when a caller asks the fake IPartServiceRepository to GetById and passes an InvalidPartId, it should respond by throwing our new custom exception, PartNotFoundException.



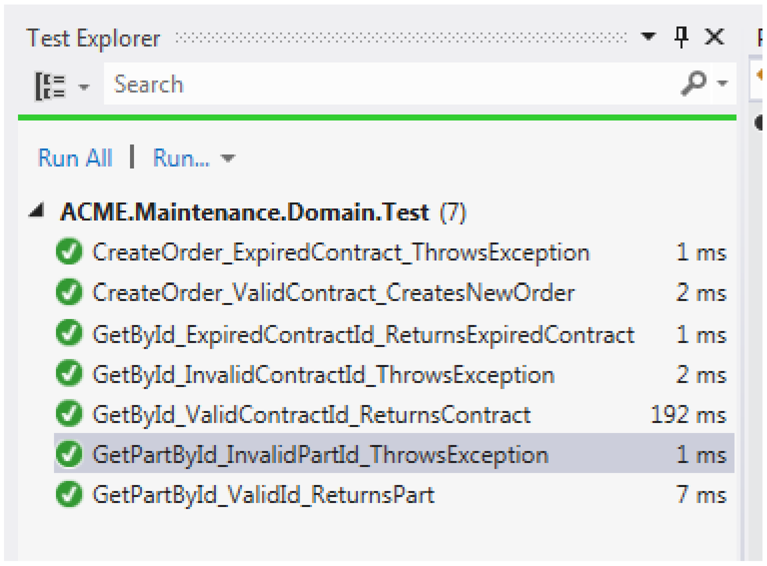
```
    [TestInitialize]
    public void Initialize()
    {
      _partServiceRepository = A.Fake<IPartServiceRepository>();

      A.CallTo(() => _partServiceRepository.GetById(ValidPartId))
       .Returns(new PartDto
         {
           PartId = ValidPartId,
           Price = ValidPartAmount
         });

      A.CallTo(() => _partServiceRepository.GetById(InvalidPartId))
        .Throws<PartNotFoundException>();

      AutoMapper.Mapper.CreateMap<PartDto, Part>();
    }
```

Get this code:

https://gist.github.com/LearnVisualStudio/f4600f7fd5ce98afdc0d

Does it work?

18. Run All Unit Tests



```
ACME.Maintenance.Domain.Test (7)
  CreateOrder_ExpiredContract_ThrowsException          1 ms
  CreateOrder_ValidContract_CreatesNewOrder            2 ms
  GetById_ExpiredContractId_ReturnsExpiredContract     1 ms
  GetById_InvalidContractId_ThrowsException            2 ms
  GetById_ValidContractId_ReturnsContract            192 ms
  GetPartById_InvalidPartId_ThrowsException            1 ms
  GetPartById_ValidId_ReturnsPart                      7 ms
```

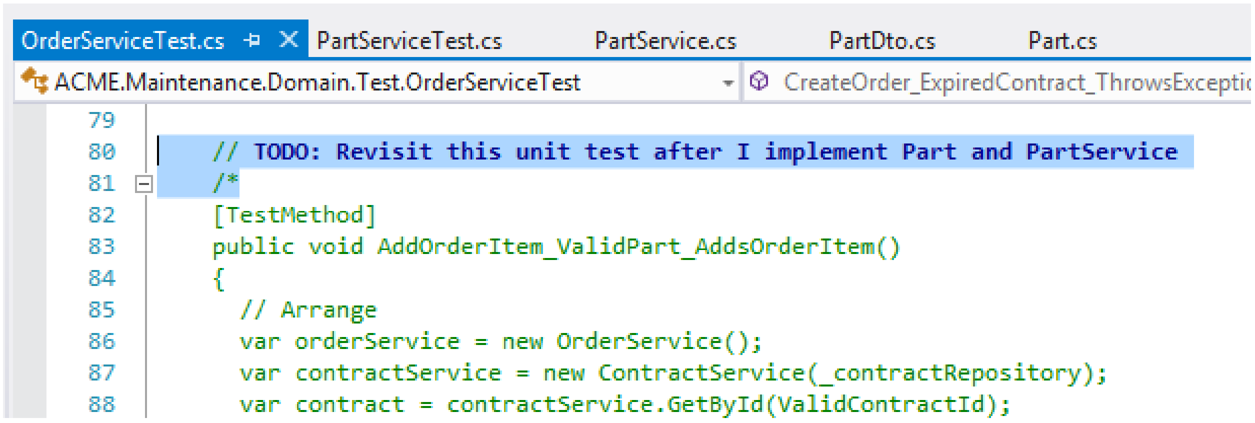
Yes!

Lesson 19 - Adding Order Items to the Order - Part 2

Now that we have a solid direction for the scenario where we create an OrderItem and add it to the Order, I think it's time to revisit that unit test. You'll recall a couple of lessons ago, we commented it temporarily so that we could flesh out the PartService and Part classes.

1. Uncomment out the AddOrderItem Unit Test

We're ready to tackle our original unit test, so remove the TODO as well as the /* and */ multi-line comment symbols.

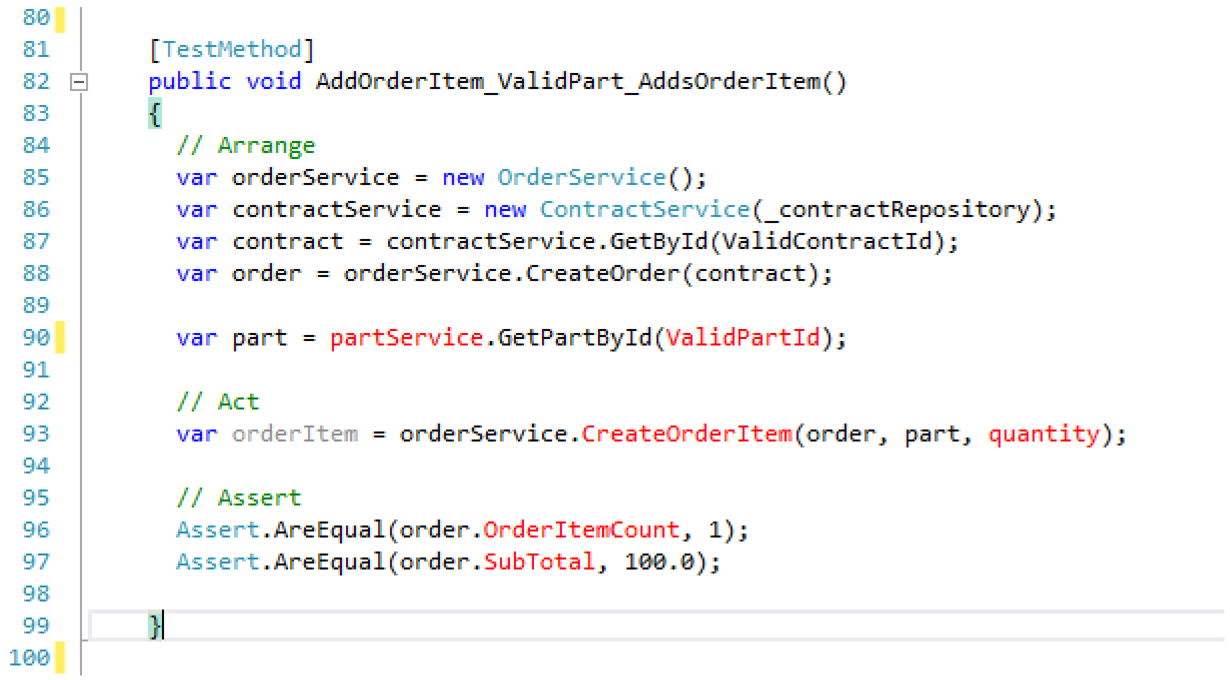


OrderServiceTest.cs
PartServiceTest.cs
PartService.cs
PartDto.cs
Part.cs
ACME.Maintenance.Domain.Test.OrderServiceTest
CreateOrder_ExpiredContract_ThrowsExceptic

```
    // TODO: Revisit this unit test after I implement Part and PartService
    /*
    [TestMethod]
    public void AddOrderItem_ValidPart_AddsOrderItem()
    {
      // Arrange
      var orderService = new OrderService();
      var contractService = new ContractService(_contractRepository);
      var contract = contractService.GetById(ValidContractId);
```

2. Review the state of the AddOrderItem Unit Test

At this point, there's a lot broken here, and it may not even represent our best intentions for the design of the classes involved, specifically, the OrderService, Order and OrderItem classes.



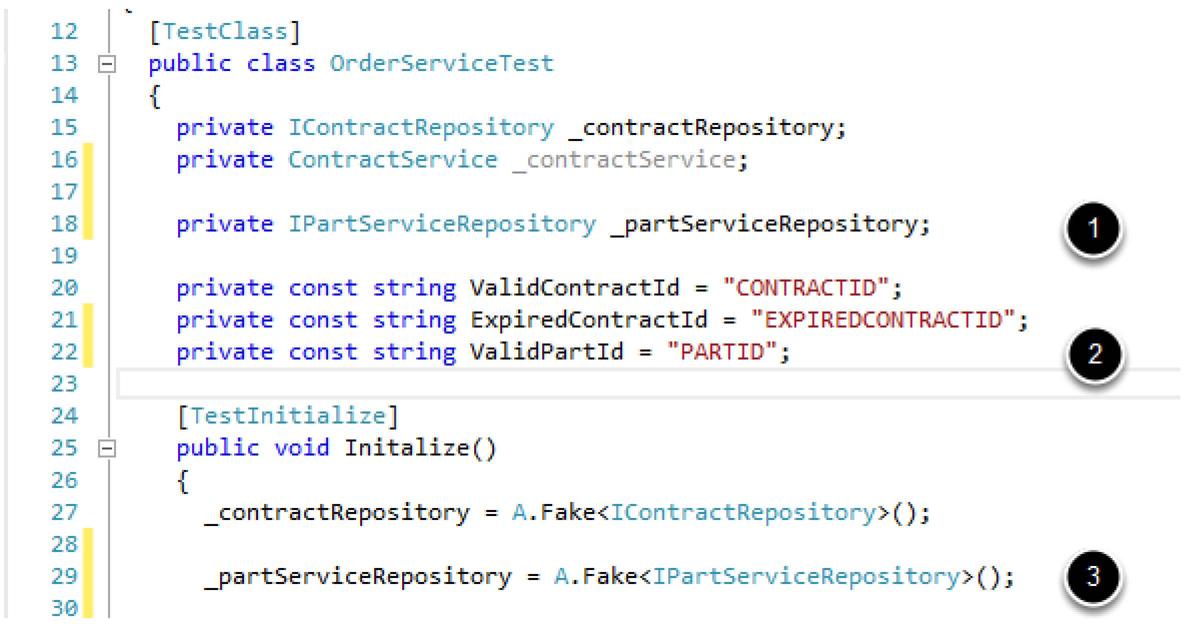
```
80
81        [TestMethod]
82        public void AddOrderItem_ValidPart_AddsOrderItem()
83        {
84          // Arrange
85          var orderService = new OrderService();
86          var contractService = new ContractService(_contractRepository);
87          var contract = contractService.GetById(ValidContractId);
88          var order = orderService.CreateOrder(contract);
89
90          var part = partService.GetPartById(ValidPartId);
91
92          // Act
93          var orderItem = orderService.CreateOrderItem(order, part, quantity);
94
95          // Assert
96          Assert.AreEqual(order.OrderItemCount, 1);
97          Assert.AreEqual(order.SubTotal, 100.0);
98
99        }
100
```

First, I'll address line 90 where I'm using the PartService before even creating an instance of it. This was because several lessons ago, there was too much to think about all at one time. Now that we've gotten clarity on the PartService and Part classes, we can revisit this.

Furthermore, in line 90 I intended to use a constant called ValidPartId, however I never implemented it.

3. Refactor the OrderServiceTest.cs and the AddOrderItem Unit Test

I'll make some incremental improvements to the unit test. I'll start by adding the fake IPartServiceRepository reference, and a constant for the ValidPartId.



```
[TestClass]
public class OrderServiceTest
{
  private IContractRepository _contractRepository;
  private ContractService _contractService;

  private IPartServiceRepository _partServiceRepository;

  private const string ValidContractId = "CONTRACTID";
  private const string ExpiredContractId = "EXPIREDCONTRACTID";
  private const string ValidPartId = "PARTID";

  [TestInitialize]
  public void Initalize()
  {
    _contractRepository = A.Fake<IContractRepository>();

    _partServiceRepository = A.Fake<IPartServiceRepository>();

```

(1) I add a private member to hold our fake IPartServiceRepository

(2) I add a constant for ValidPartId

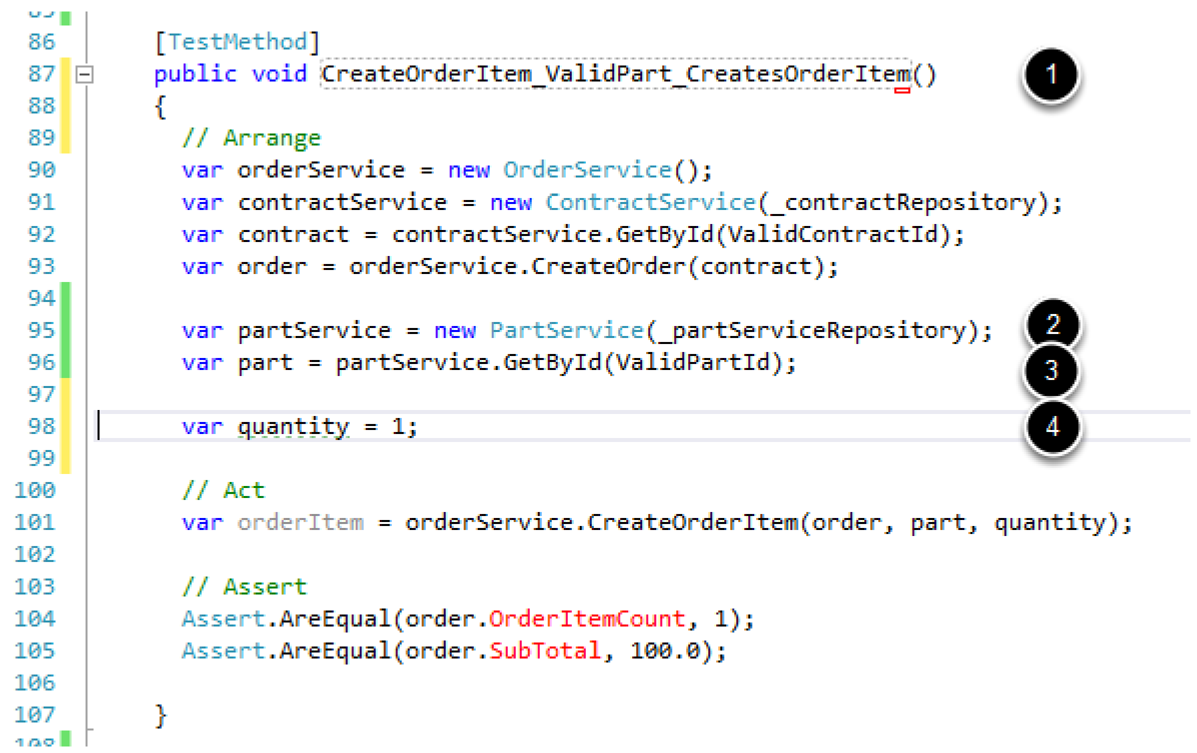
(3) I create the fake IPartServiceRepository instance and save it in the new private member I just created, _partServiceRepository

Now, I'll revisit the unit test. With some of the setup out of the way, I can create a new instance of the PartService.

But first, something catches my eye ... the name of the unit test is suspect.

AddOrderItem_ValidPart_AddsOrderItem()

I realize this name no longer matches what I'm trying to do in the unit test. What I want to do is CREATE AN ORDER ITEM, not necessarily ADD IT TO AN ORDER. Yet. So, I'll refactor this method a little ...


```
    [TestMethod]
    public void CreateOrderItem_ValidPart_CreatesOrderItem()
    {
      // Arrange
      var orderService = new OrderService();
      var contractService = new ContractService(_contractRepository);
      var contract = contractService.GetById(ValidContractId);
      var order = orderService.CreateOrder(contract);

      var partService = new PartService(_partServiceRepository);
      var part = partService.GetById(ValidPartId);

      var quantity = 1;

      // Act
      var orderItem = orderService.CreateOrderItem(order, part, quantity);

      // Assert
      Assert.AreEqual(order.OrderItemCount, 1);
      Assert.AreEqual(order.SubTotal, 100.0);

    }
```

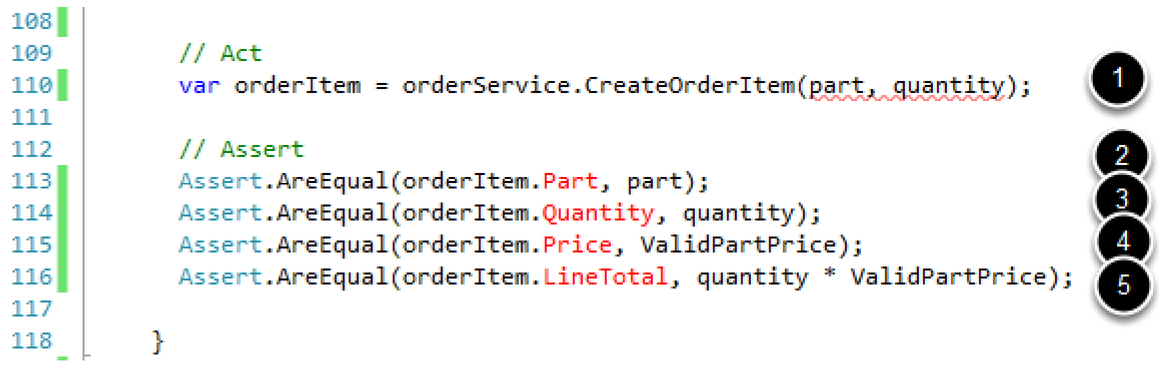
(1) I refactor the unit test's name to something that matches the intent of the unit test. I merely want to create a new instance of OrderItem. I'll add it to an Order some other time.

(2) I create a new instance of PartService passing in the fake IPartServiceRepository

(3) I realize the name of the method should be GetById, and make the change here

(4) I create a variable for the "magic number" of 1. I probbably should make this a constant, but I'll leave it for now.

Now, time to turn my attention to the Act and Assert parts of the unit test...



```
            // Act
            var orderItem = orderService.CreateOrderItem(part, quantity);

            // Assert
            Assert.AreEqual(orderItem.Part, part);
            Assert.AreEqual(orderItem.Quantity, quantity);
            Assert.AreEqual(orderItem.Price, ValidPartPrice);
            Assert.AreEqual(orderItem.LineTotal, quantity * ValidPartPrice);

        }
```

(1) Again, the intent is to CREATE AN ORDER ITEM. At this point, that will have nothing to do with Order. So, I remove the first input parameter, the order, from the CreateOrderItem() method.

(2) I realize that I'm asserting all the wrong things -- all related to Order. Now, I refactor the unit test to focus on testing the new instance of OrderItem by testing that the Part is correct ...

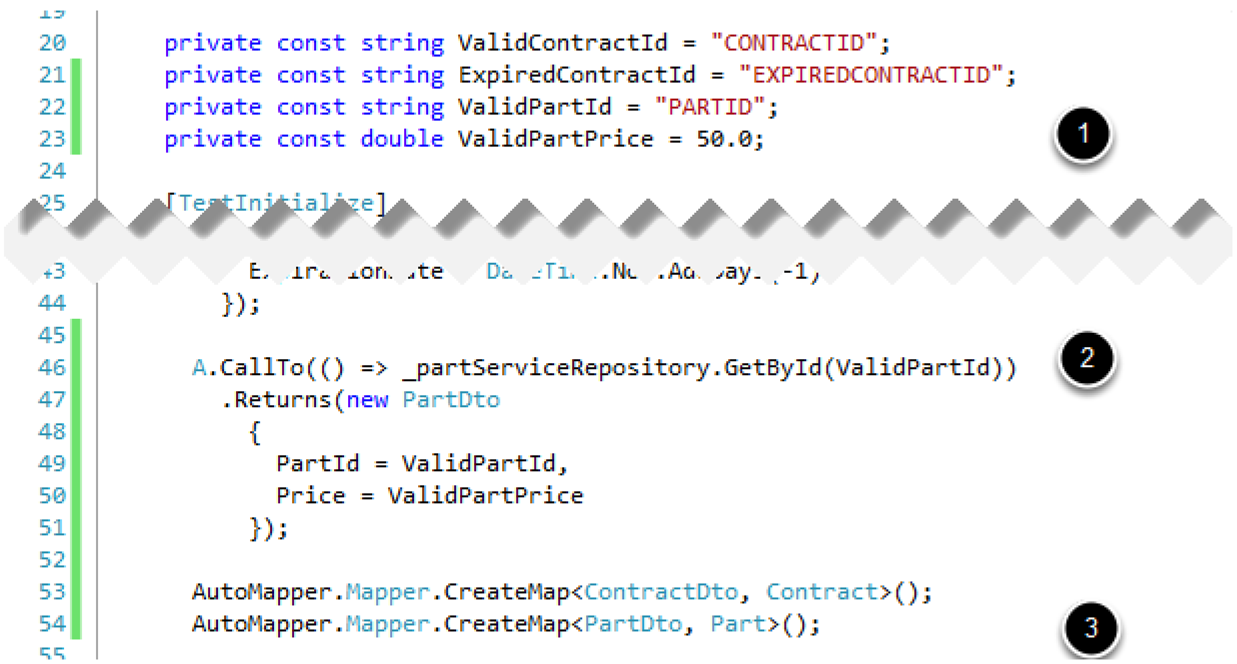
(3) the Quantity property is correct ...

(4) the Price property is correct, and ...

(5) the LineTotal property is correct

I have a moment of angst about the LineTotal property. Should I create a property just to store this value when it can be easily calculated? I always am conflicted about it. In the end, I decide to leave it. It could come in handy and make processing easier somewhere down the road.

I need to make a few final tweaks to the top of the OrderServiceTest.cs file before I can move on...



```
    private const string ValidContractId = "CONTRACTID";
    private const string ExpiredContractId = "EXPIREDCONTRACTID";
    private const string ValidPartId = "PARTID";
    private const double ValidPartPrice = 50.0;

    [TestInitialize]

          E  ira ion  te    Da  Ti  .Nc  .Ac  ay  -1,
        });

      A.CallTo(() => _partServiceRepository.GetById(ValidPartId))
        .Returns(new PartDto
          {
            PartId = ValidPartId,
            Price = ValidPartPrice
          });

      AutoMapper.Mapper.CreateMap<ContractDto, Contract>();
      AutoMapper.Mapper.CreateMap<PartDto, Part>();
```

(1) I add a constant for the ValidPartPrice

(2) I create an expectation for the fake PartServiceRepository ... when GetById() receives a ValidPartId, it should return a PartDto that is rigged with specific settings

(3) I create an AutoMapper mapping between PartDto and Part

That should just about do it. A lot of updates, but now we're focusing on just the creation of OrderItem and not overstepping into working on the Order. We'll save that for another lesson.

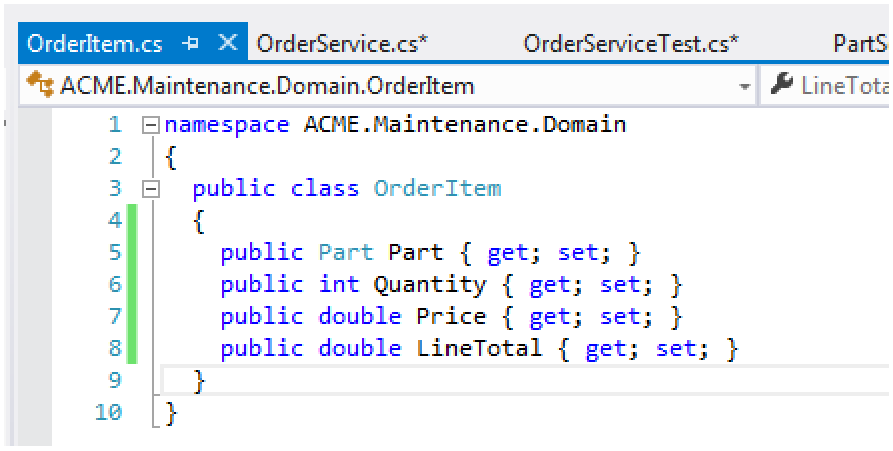
Get this code:

https://gist.github.com/LearnVisualStudio/a9cefb3b006af608ac7d

Now, we need to implement the properties in the OrderItem class and implement the OrderService.CreateOrderItem() method.

4. Implement Properties in the OrderItem class

Based on the unit test design, I add the four properties to the OrderItem.



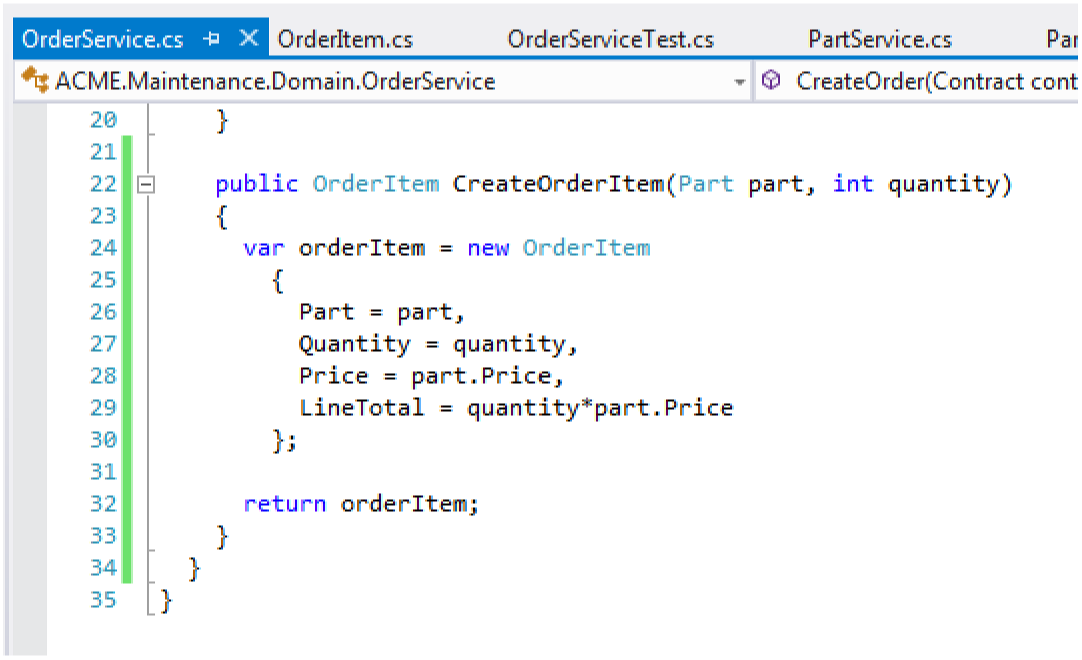
```
namespace ACME.Maintenance.Domain
{
  public class OrderItem
  {
    public Part Part { get; set; }
    public int Quantity { get; set; }
    public double Price { get; set; }
    public double LineTotal { get; set; }
  }
}
```

Get this code:

https://gist.github.com/LearnVisualStudio/a1d5a2f68b8aa332c638

5. Implement the OrderService class' CreateOrderItem() method

In OrderService.CreateOrderItem(), I'll create a new instance of OrderItem and initialize its properties.

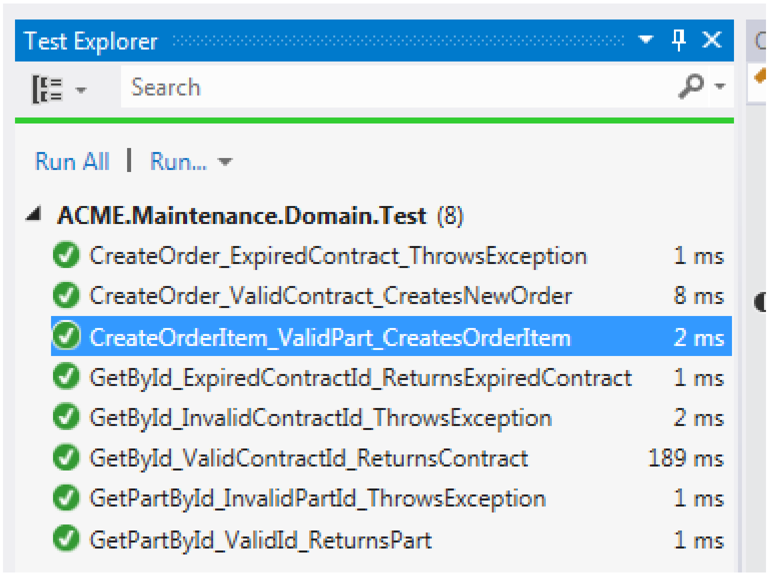

OrderService.cs
OrderItem.cs
OrderServiceTest.cs
PartService.cs
ACME.Maintenance.Domain.OrderService
CreateOrder(Contract cont

```
    }

    public OrderItem CreateOrderItem(Part part, int quantity)
    {
      var orderItem = new OrderItem
        {
          Part = part,
          Quantity = quantity,
          Price = part.Price,
          LineTotal = quantity*part.Price
        };

      return orderItem;
    }
  }
}
```

Get this code:

https://gist.github.com/LearnVisualStudio/d79ff3618abf7e76f852

6. Run All Unit Tests



```
ACME.Maintenance.Domain.Test (8)
  CreateOrder_ExpiredContract_ThrowsException        1 ms
  CreateOrder_ValidContract_CreatesNewOrder          8 ms
  CreateOrderItem_ValidPart_CreatesOrderItem         2 ms
  GetById_ExpiredContractId_ReturnsExpiredContract   1 ms
  GetById_InvalidContractId_ThrowsException          2 ms
  GetById_ValidContractId_ReturnsContract          189 ms
  GetPartById_InvalidPartId_ThrowsException          1 ms
  GetPartById_ValidId_ReturnsPart                    1 ms
```

Success!

I try to think of a case where I cannot create an OrderItem. At this point, I can't think of an exceptional case. We've covered the possibility that the user submits an incorrect PartId in another use case (and lesson).

While staring at this for a few moments, I do ask myself whether the OrderService is the right place to implement the CreateOrderItem() method. Should it be delegated the responsibility of creating OrderItems? Or should that responsibility belong to another class? Perhaps even, Order? That's a tricky question to answer. I don't know the answer to it just yet, but I decide to keep an eye on it and see if I gain some clarity as I move forward.

I think we're good to move on to talk about Order.

Lesson 20 - Adding Order Items to the Order - Part 3

Now that I can create an Order and I can create OrderItems, I want to associate those OrderItems with the Order.

I decide to delegate that responsibility to the Order itself. It should know which items belongs to it.

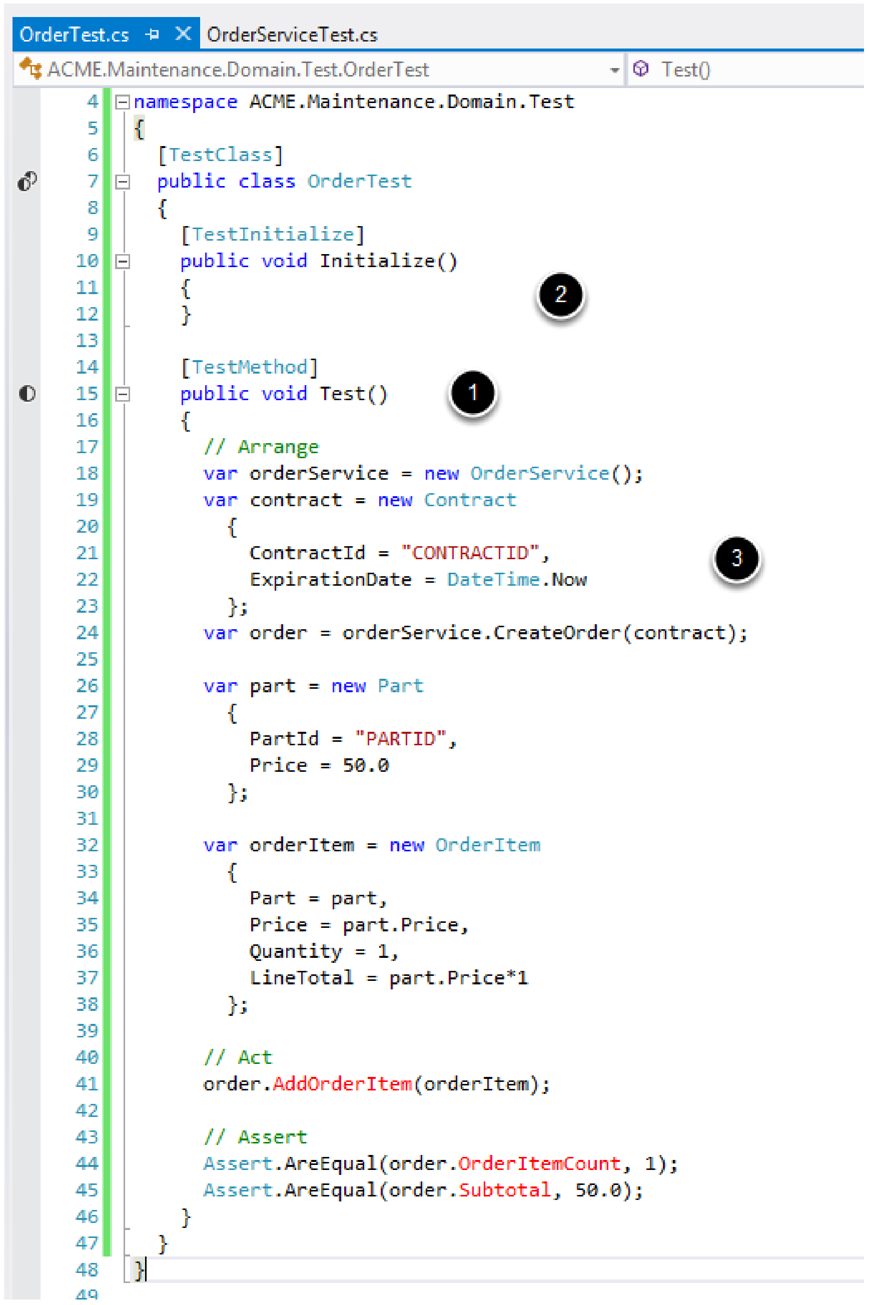
I have lingering concerns about the Order class, however. I'm concerned about how I will expose a generic list of OrderItems but not allow the caller to add items to the Order outside of my AddOrderItem() method, which will handle other tasks like tallying order items, a subtotal, etc.

I decide to put that concern on hold for now and simply get to the point where I can add OrderItems to the Order.

1. Add OrderTest.cs to the unit test project, create unit test

I first add a new OrderTest.cs to the Unit Test project in my solution.

Then, I begin implementing the new unit test to add order items. Here's what I've created so far:



```
namespace ACME.Maintenance.Domain.Test
{
  [TestClass]
  public class OrderTest
  {
    [TestInitialize]
    public void Initialize()
    {
    }

    [TestMethod]
    public void Test()
    {
      // Arrange
      var orderService = new OrderService();
      var contract = new Contract
        {
          ContractId = "CONTRACTID",
          ExpirationDate = DateTime.Now
        };
      var order = orderService.CreateOrder(contract);

      var part = new Part
        {
          PartId = "PARTID",
          Price = 50.0
        };

      var orderItem = new OrderItem
        {
          Part = part,
          Price = part.Price,
          Quantity = 1,
          LineTotal = part.Price*1
        };

      // Act
      order.AddOrderItem(orderItem);

      // Assert
      Assert.AreEqual(order.OrderItemCount, 1);
      Assert.AreEqual(order.Subtotal, 50.0);
    }
  }
}
```

I have a moment of tired-head and I can't come up with the right name for the unit test, so I merely name it Test(). I anticipate needing to do a lot of setup, and create an Initialize() method. I think I'm tired and just using a cookie-cutter approach rather than really thinking whether I need it or not (which, as it turns out, I do not need it).

Next, I want to create an Order and OrderItem. To create an Order, I'll need an OrderService and a Contract. To create a Contract, I'll need a ContractService and a (fake) ContractServiceRepository. To create an OrderItem, I'll need a Part, which in turns needs a PartService, which in turn needs a (fake) PartServiceRepository.

I'm not in the mood to create all of those. So, I cheat. I just create a hard-coded Contract, Part and OrderItem. I feel bad about this because I foresee a potential problem with this in the future. What if I change the process of instantiation in any of these three classes in the future? I've completely circumvented the roles of the __Service classes. This is not good.

But I'm tired of writing all that Initialization() code ... creating fakes, etc. ... every time I want to run unit tests that somewhat inter-depend on each other.

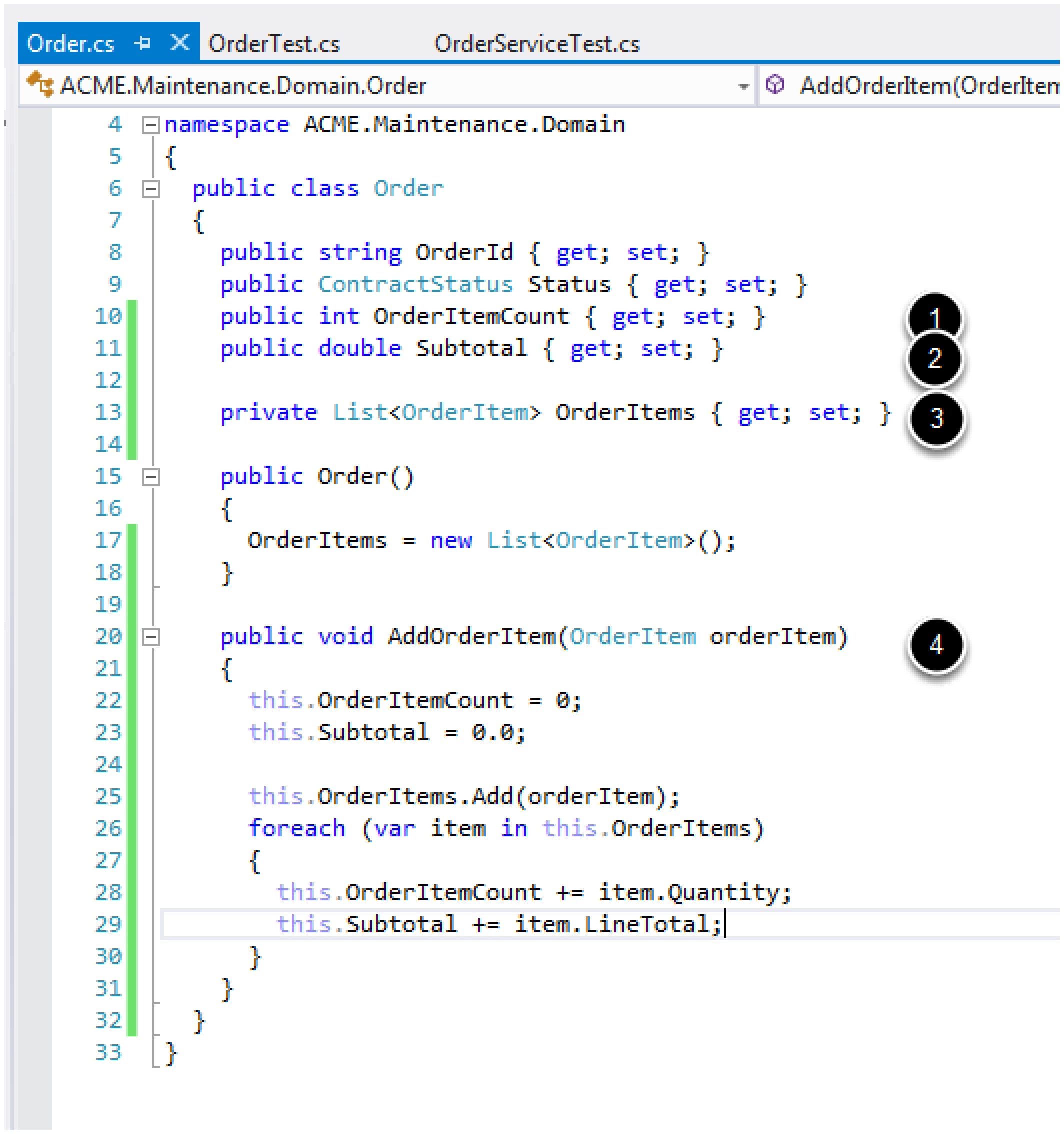
I consider for a moment the advice in the book Growing Object-Oriented Software, Guided by Tests by Freeman & Price: if a test is hard to write, it might be an indication that there's something wrong about the system under test. I think about that for a moment, but discount it. The test is not HARD to write per se, it's just laborious. It's laborious because I've chosen to split up my unit tests into files, each file corresponding with the class it is testing. If I were to lump all unit tests into the same class, the single Initialize() method would suffice to supply the fakes, expectations, etc.

Perhaps I could take all of those expectations and refactor them into a set of helper methods in a helper class. Maybe later.

For now, I decide to proceed on with the "lazy man's" approach to creating the Contract, Part and OrderItem instances.

2. Implement the AddOrderItem() method, as well as other properties of the Order class

The unit test drives the addition of a few new properties and the AddOrderItem() method.



```
namespace ACME.Maintenance.Domain
{
  public class Order
  {
    public string OrderId { get; set; }
    public ContractStatus Status { get; set; }
    public int OrderItemCount { get; set; }
    public double Subtotal { get; set; }

    private List<OrderItem> OrderItems { get; set; }

    public Order()
    {
      OrderItems = new List<OrderItem>();
    }

    public void AddOrderItem(OrderItem orderItem)
    {
      this.OrderItemCount = 0;
      this.Subtotal = 0.0;

      this.OrderItems.Add(orderItem);
      foreach (var item in this.OrderItems)
      {
        this.OrderItemCount += item.Quantity;
        this.Subtotal += item.LineTotal;
      }
    }
  }
}
```

(1) I add an OrderItemCount property

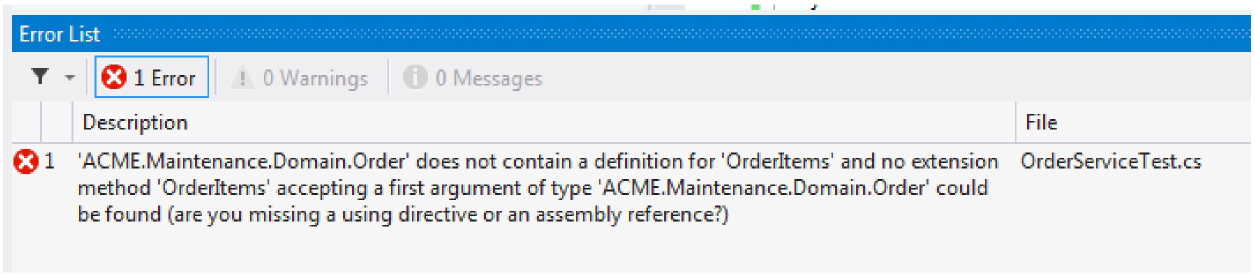
(2) I add a Subtotal property

(3) I refactor the OrderItems property to be private. This probably wasn't necessary, since I would test this in another unit test. When I made this change, I had grandiose plans for the AddOrderItem() method. In other words, it made sense at the time, but in retrospect, I failed in allowing the unit tests to COMPLETELY drive the development process. Old habits die hard.

(4) I implement the AddOrderItem() method. This is mostly straight-forward code.

I decide that I've done all I can do for now, time to run the unit tests.

3. Run All Unit Tests reveals a problem



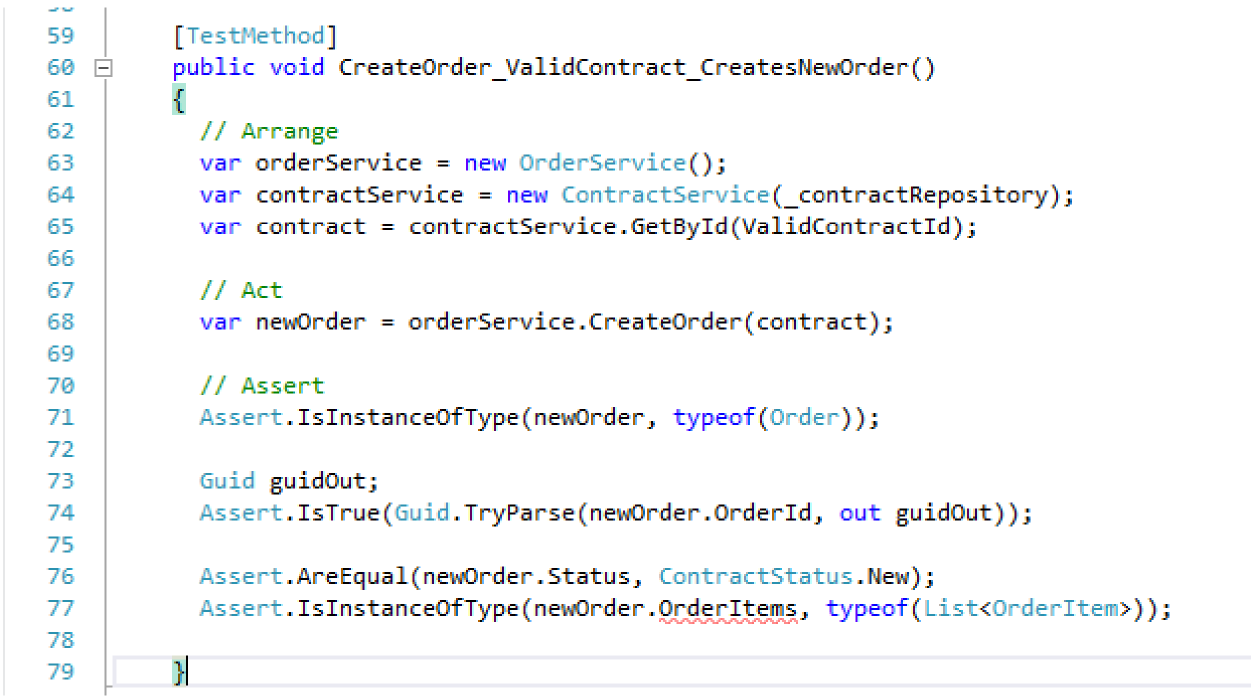
Error List

1 Error | 0 Warnings | 0 Messages

	Description	File
1	'ACME.Maintenance.Domain.Order' does not contain a definition for 'OrderItems' and no extension method 'OrderItems' accepting a first argument of type 'ACME.Maintenance.Domain.Order' could be found (are you missing a using directive or an assembly reference?)	OrderServiceTest.cs

I click Run All on the Test Explorer, but the solution doesn't compile. Instead, I see an error has sprung up in the OrderServiceTest.cs. Clearly, something I changed in the Order class broke a unit test.

I double-click on the error in the Error List window and it shows me the problem in line 78 ...



```
59  [TestMethod]
60  public void CreateOrder_ValidContract_CreatesNewOrder()
61  {
62    // Arrange
63    var orderService = new OrderService();
64    var contractService = new ContractService(_contractRepository);
65    var contract = contractService.GetById(ValidContractId);
66
67    // Act
68    var newOrder = orderService.CreateOrder(contract);
69
70    // Assert
71    Assert.IsInstanceOfType(newOrder, typeof(Order));
72
73    Guid guidOut;
74    Assert.IsTrue(Guid.TryParse(newOrder.OrderId, out guidOut));
75
76    Assert.AreEqual(newOrder.Status, ContractStatus.New);
77    Assert.IsInstanceOfType(newOrder.OrderItems, typeof(List<OrderItem>));
78
79  }
```

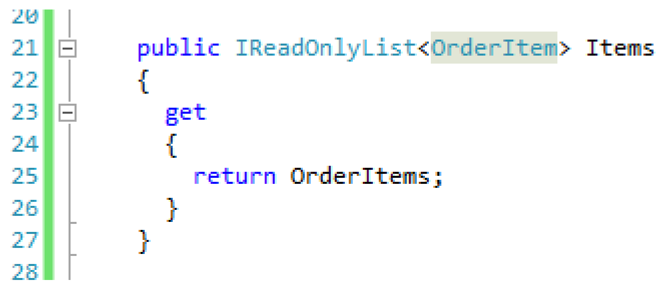
... in line 78, I am looking to verify that OrderItems is of type List<OrderItem>. By making it private in the Order class, I've broken this unit test.

Herein lies the value of the unit tests. I can see what I broke, and since I'm running the unit tests often, I can easily narrow down the source of the problem. In this case, the source of the problem was clear ... by messing around with the visibility of the OrderItems property of Order, I now have to decide what needs to change.

After a few moments of contemplation, I realize BOTH my implementation AND the unit test needs to change.

4. Add an Items property

I need to keep the List<OrderItems> private for all the reasons I mentioned before. However, I can expose the MEMBERS of the List<OrderItem> without giving the caller complete access by creating a readonly property called Items.



```
public IReadOnlyList<OrderItem> Items
{
  get
  {
    return OrderItems;
  }
}
```

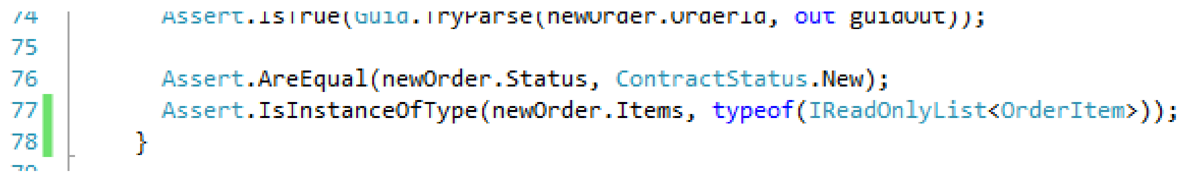
I use the new IReadOnlyList<T> added in the .NET Framework 4.5 to address these situations. I merely implement a get for the property and return the List<OrderItem>. However, the CALLER is now constrained to treat the List<OrderItem> as a READONLY list. Perfect!

Get this code:

https://gist.github.com/LearnVisualStudio/ea8d6101a6c163de640d

5. Refactor the CreateOrder_ValidContract_CreatesNewOrder() method to check the new return type

Now, I want to ensure that Items (not OrderItems, which continues to be private) is of type IReadOnlyList<OrderItem>.



```
    Assert.IsTrue(Guid.TryParse(newOrder.OrderId, out guidOut));

    Assert.AreEqual(newOrder.Status, ContractStatus.New);
    Assert.IsInstanceOfType(newOrder.Items, typeof(IReadOnlyList<OrderItem>));
}
```

Get this code:

https://gist.github.com/LearnVisualStudio/ed14dcb3b75580ca9d1c

6. Refactor the name of the OrderTest.cs unit test

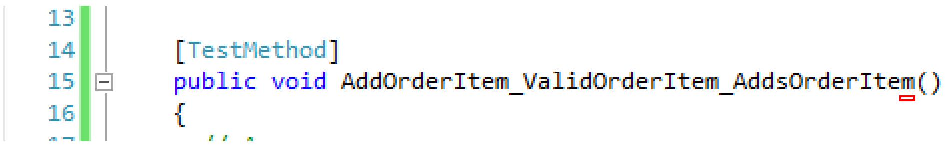
I change the name of the unit test from ...

Test()

... To...

AddOrderItem_ValidOrderItme_AddsOrderItem()

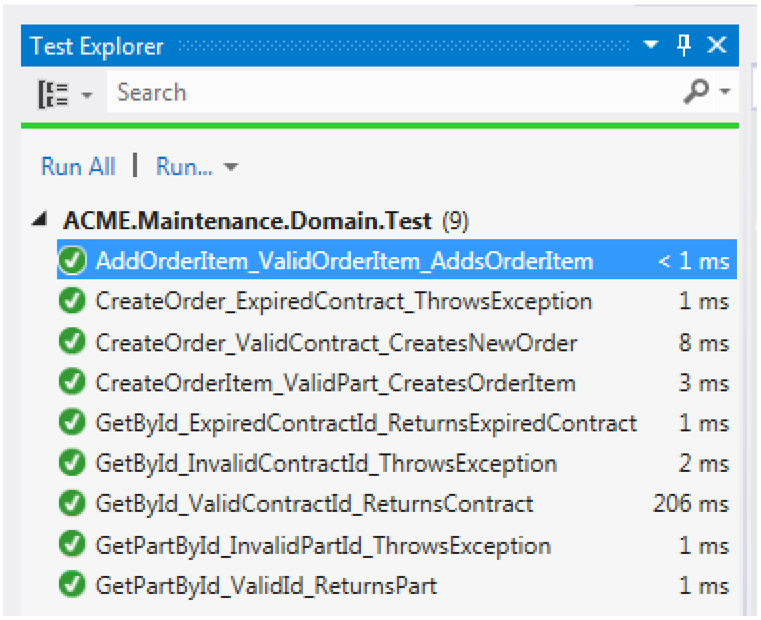
... which is much more descriptive of what's happening in this unit test.


```
    [TestMethod]
    public void AddOrderItem_ValidOrderItem_AddsOrderItem()
    {
```

Get this code:

https://gist.github.com/LearnVisualStudio/a5629866750b88f6075a

7. Run All Unit Tests



```
ACME.Maintenance.Domain.Test (9)
  AddOrderItem_ValidOrderItem_AddsOrderItem              < 1 ms
  CreateOrder_ExpiredContract_ThrowsException              1 ms
  CreateOrder_ValidContract_CreatesNewOrder                8 ms
  CreateOrderItem_ValidPart_CreatesOrderItem               3 ms
  GetById_ExpiredContractId_ReturnsExpiredContract         1 ms
  GetById_InvalidContractId_ThrowsException                2 ms
  GetById_ValidContractId_ReturnsContract                206 ms
  GetPartById_InvalidPartId_ThrowsException                1 ms
  GetPartById_ValidId_ReturnsPart                          1 ms
```

And it all works, even the newest unit test.

There's really only one more scenario I need to enable in my use case, and so we'll begin to wrap up this series in the next lesson.

Lesson 21 - Getting a List of Order Items

The final aspect of the user story we want to cover is being able to list each OrderItem, the quantity, the part name, the price, the line total as well as the Order's subtotal. As a result, the focus will be on the Items property added in the previous lesson (along with properties on the Order itself).

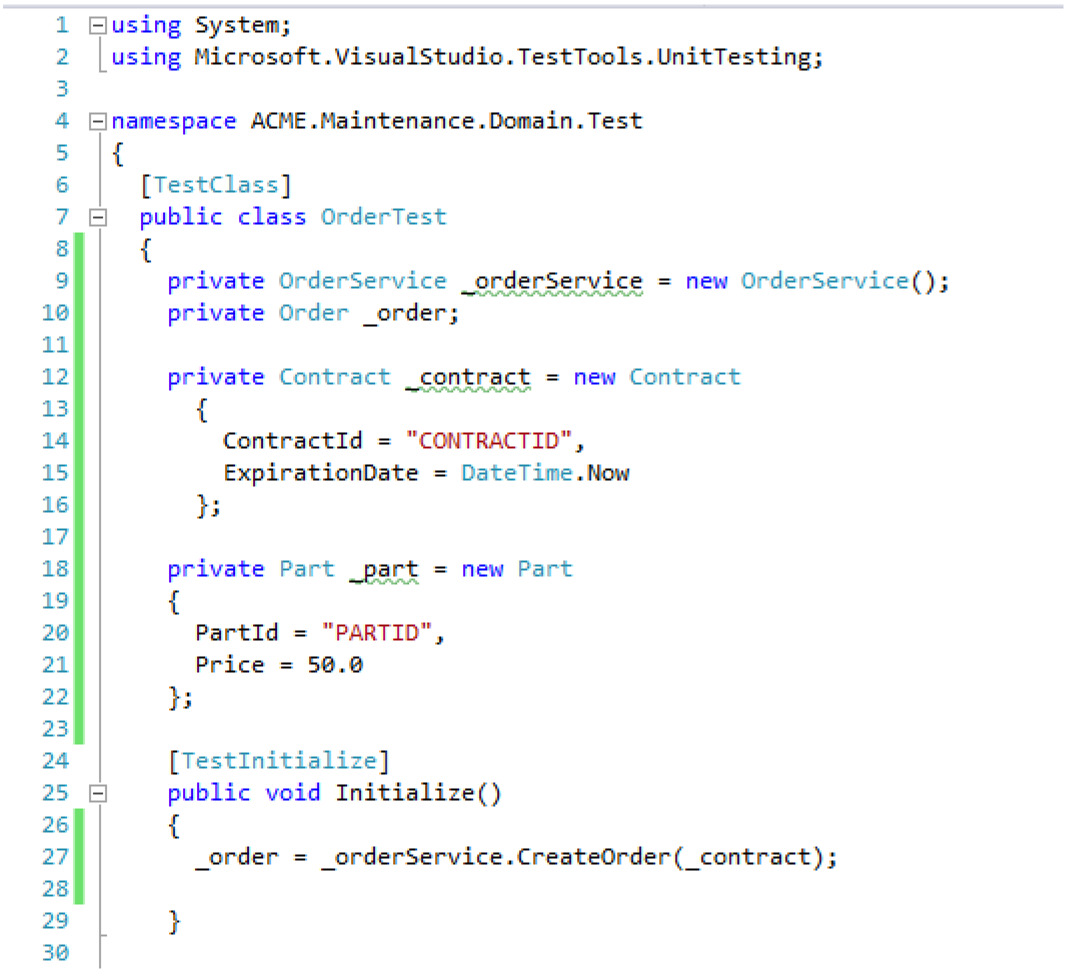
This should be relatively easy. In fact, we probably have this scenario already covered -- we know from the previous lesson that Order now has an IReadOnlyList<OrderItem> property called Items. All that's left to do is add a number of OrderItems and make sure everything adds up.

We could probably test the exceptional case as well -- ensuring that nothing "blows up" when there are no items in the Item's collection, or the other Order properties we'll be using for this scenario.

The first order of business is to extract out the common arrange code from the previous unit test and put it at the top of the class or in the Initialize() method where it can be shared by both the old and our new unit test.

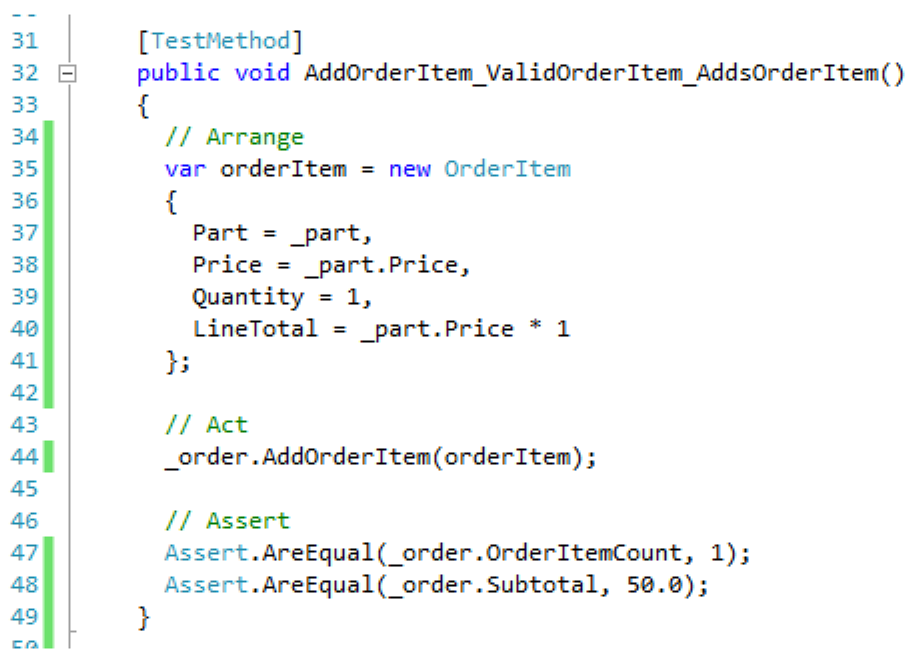
1. Refactor commonalities into the Initialize() method

I strip out most of the arrange functionality from the AddOrderItem_ValidOrderItem_AddsOrderItem() method and either create private member variables (renaming as I go along) or in the Initialize() method.



```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ACME.Maintenance.Domain.Test
{
  [TestClass]
  public class OrderTest
  {
    private OrderService _orderService = new OrderService();
    private Order _order;

    private Contract _contract = new Contract
      {
        ContractId = "CONTRACTID",
        ExpirationDate = DateTime.Now
      };

    private Part _part = new Part
    {
      PartId = "PARTID",
      Price = 50.0
    };

    [TestInitialize]
    public void Initialize()
    {
      _order = _orderService.CreateOrder(_contract);

    }

```

As you can see now, this leaves the old unit test much more concise than before:



```
[TestMethod]
public void AddOrderItem_ValidOrderItem_AddsOrderItem()
{
  // Arrange
  var orderItem = new OrderItem
  {
    Part = _part,
    Price = _part.Price,
    Quantity = 1,
    LineTotal = _part.Price * 1
  };

  // Act
  _order.AddOrderItem(orderItem);

  // Assert
  Assert.AreEqual(_order.OrderItemCount, 1);
  Assert.AreEqual(_order.Subtotal, 50.0);
}
```

2. Create the new unit test to test for enumeration

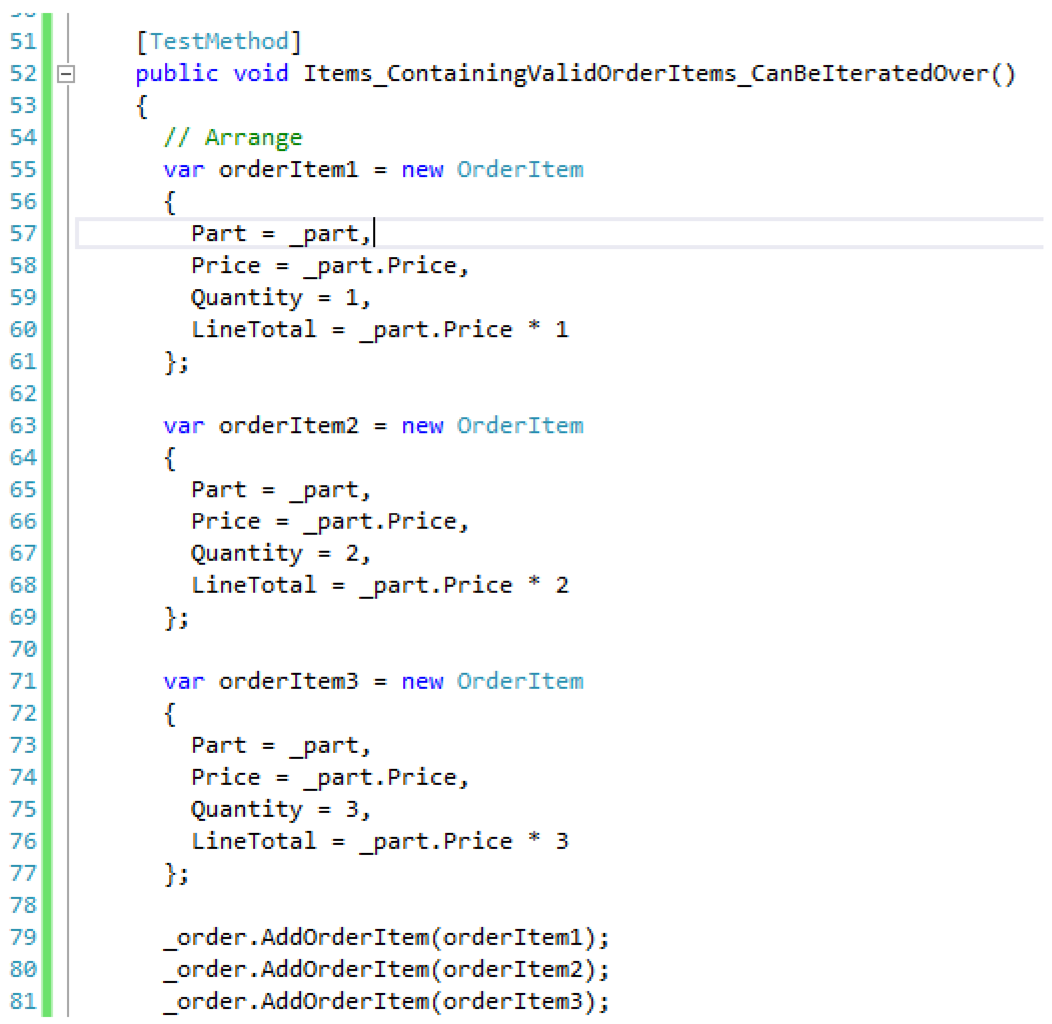
Next, I begin working on the unit test at the heart of this lesson. There's two things different about this unit test:

(A) I'm testing a property, not a method. I want to make sure I can do what I need to do with the Order.Items property in the future, and that any future changes will not affect it. Remember, the Items property is read only, and it is of type IReadOnlyList<OrderItem>, meaning it should have many of the features of List<T>, without the ability to add or remove items from it.

(B) I don't anticipate that this will force me to ADD any code. I'm not completely sure as I set out ... I THINK I have everything I need to properly test the Items property.

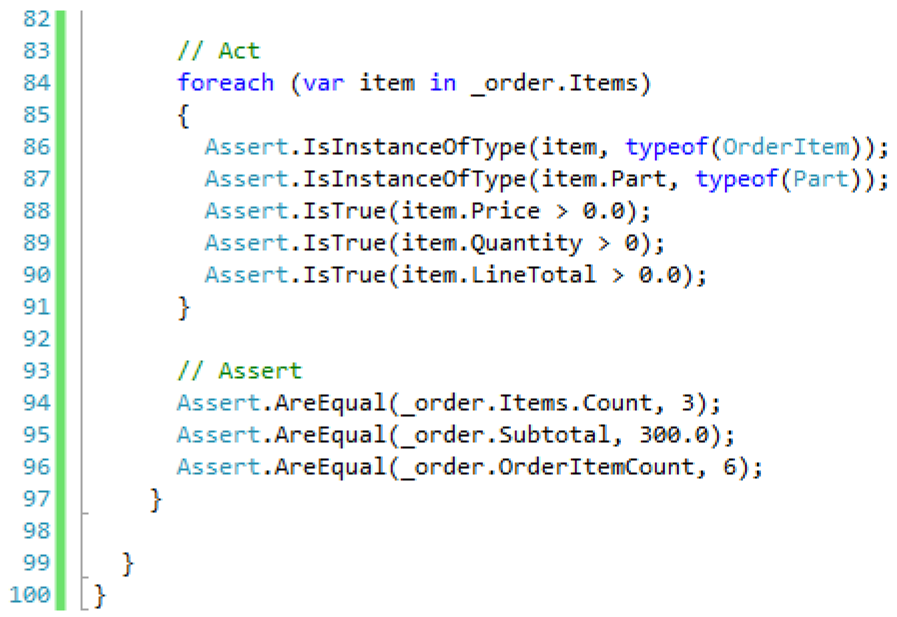
I ponder how best to test the Items property. I decide to add a few OrderItems to an Order, then iterate through them and check each of the items and the properties of the items. Adittedly, it feels a bit paranoid to do this. However I'm not sure how else to test this functionality to ensure that it will work for my user story.

So, I create three order items and add them to an order:



```
    [TestMethod]
    public void Items_ContainingValidOrderItems_CanBeIteratedOver()
    {
      // Arrange
      var orderItem1 = new OrderItem
      {
        Part = _part,
        Price = _part.Price,
        Quantity = 1,
        LineTotal = _part.Price * 1
      };

      var orderItem2 = new OrderItem
      {
        Part = _part,
        Price = _part.Price,
        Quantity = 2,
        LineTotal = _part.Price * 2
      };

      var orderItem3 = new OrderItem
      {
        Part = _part,
        Price = _part.Price,
        Quantity = 3,
        LineTotal = _part.Price * 3
      };

      _order.AddOrderItem(orderItem1);
      _order.AddOrderItem(orderItem2);
      _order.AddOrderItem(orderItem3);
```

Now for the hard part ... I foreach over the Items property:



```
82
83        // Act
84        foreach (var item in _order.Items)
85        {
86          Assert.IsInstanceOfType(item, typeof(OrderItem));
87          Assert.IsInstanceOfType(item.Part, typeof(Part));
88          Assert.IsTrue(item.Price > 0.0);
89          Assert.IsTrue(item.Quantity > 0);
90          Assert.IsTrue(item.LineTotal > 0.0);
91        }
92
93        // Assert
94        Assert.AreEqual(_order.Items.Count, 3);
95        Assert.AreEqual(_order.Subtotal, 300.0);
96        Assert.AreEqual(_order.OrderItemCount, 6);
97      }
98
99    }
100 }
```

Inside of the foreach, lines 86 through 90 I check each instance of OrderItem (the "item" object) and ensure types and values. I merely check to make sure that the values are non-zero. I suppose I could attempt to make sure each item in the list is the correct value, but that would require that I rely on the ordinal position of each OrderItem in the Items collection, and I don't want to do that. This will suffice.

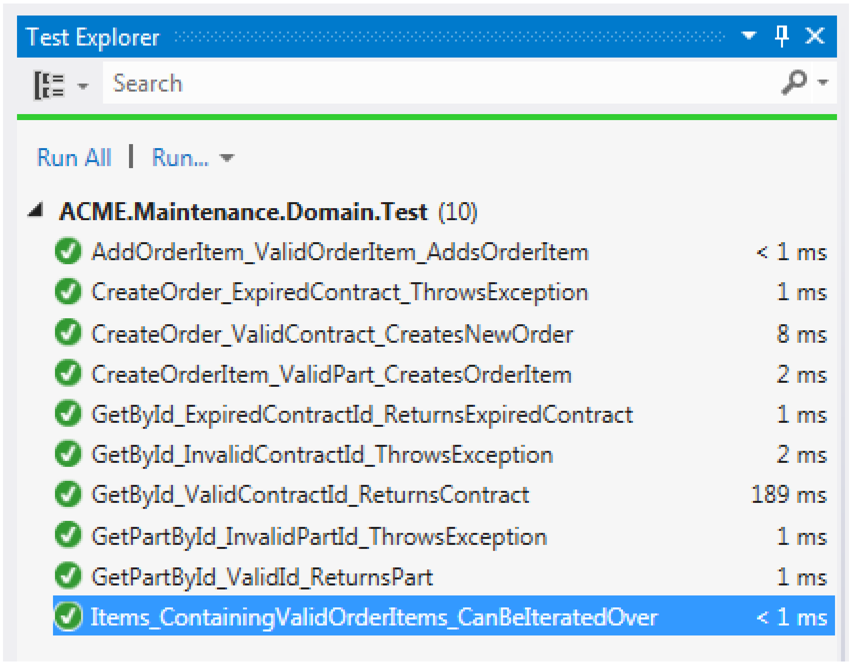
Finally, in lines 94 through 96, I ensure that the Item.Count is 3, and the sum of quantities and prices match expectations as well. I feel bad about all the "magic numbers" here. I probably should refactor those out and use constants as well that express the meaning of those values. I decide to forego that.

Get this code:

https://gist.github.com/LearnVisualStudio/a601e938cd25a192132a

A lot of coding in the unit test, and it doesn't seem like anything new was needed in the production code. Will it work?

3. Run All Unit Tests



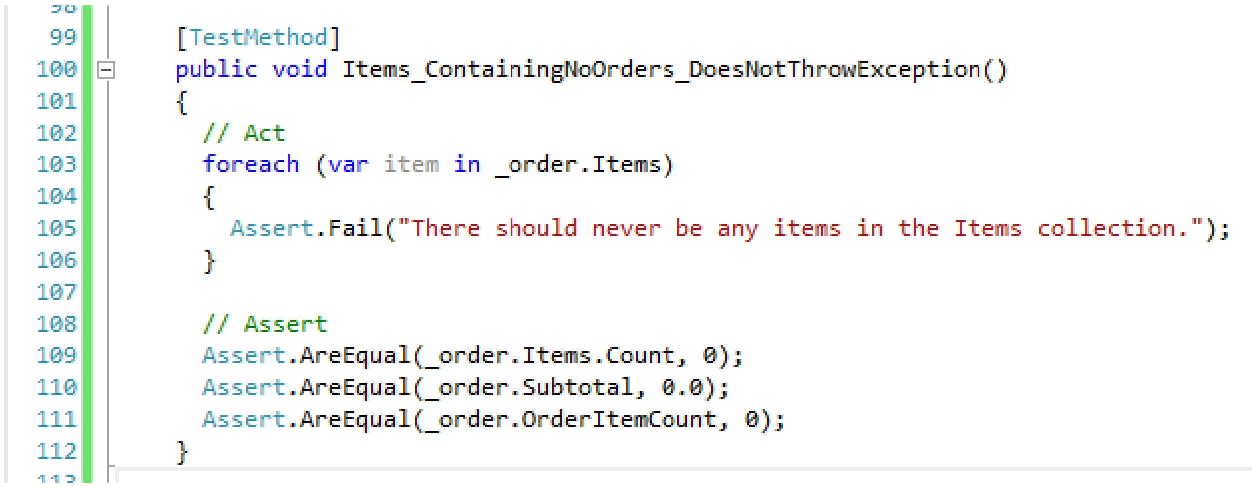
```
ACME.Maintenance.Domain.Test (10)
AddOrderItem_ValidOrderItem_AddsOrderItem                    < 1 ms
CreateOrder_ExpiredContract_ThrowsException                    1 ms
CreateOrder_ValidContract_CreatesNewOrder                      8 ms
CreateOrderItem_ValidPart_CreatesOrderItem                     2 ms
GetById_ExpiredContractId_ReturnsExpiredContract               1 ms
GetById_InvalidContractId_ThrowsException                      2 ms
GetById_ValidContractId_ReturnsContract                      189 ms
GetPartById_InvalidPartId_ThrowsException                      1 ms
GetPartById_ValidId_ReturnsPart                                1 ms
Items_ContainingValidOrderItems_CanBeIteratedOver            < 1 ms
```

Yes, it works.

My final case (described at the beginning of this lesson) is to ensure that we can reference the Items property PRIOR to adding any items. While I don't anticipate needing to do that, I would consider that an "edge case" ... bugs find their ways into our code around the boundaries and edges, and testing a aggregate with zero items seems like a good measure of caution to exercise.

4. Create Unit Test to ensure empty Items property is indeed empty

In my mind, this is a simplified version of the previous unit test. Here's what I come up with:



```
 99        [TestMethod]
100        public void Items_ContainingNoOrders_DoesNotThrowException()
101        {
102          // Act
103          foreach (var item in _order.Items)
104          {
105            Assert.Fail("There should never be any items in the Items collection.");
106          }
107
108          // Assert
109          Assert.AreEqual(_order.Items.Count, 0);
110          Assert.AreEqual(_order.Subtotal, 0.0);
111          Assert.AreEqual(_order.OrderItemCount, 0);
112        }
```

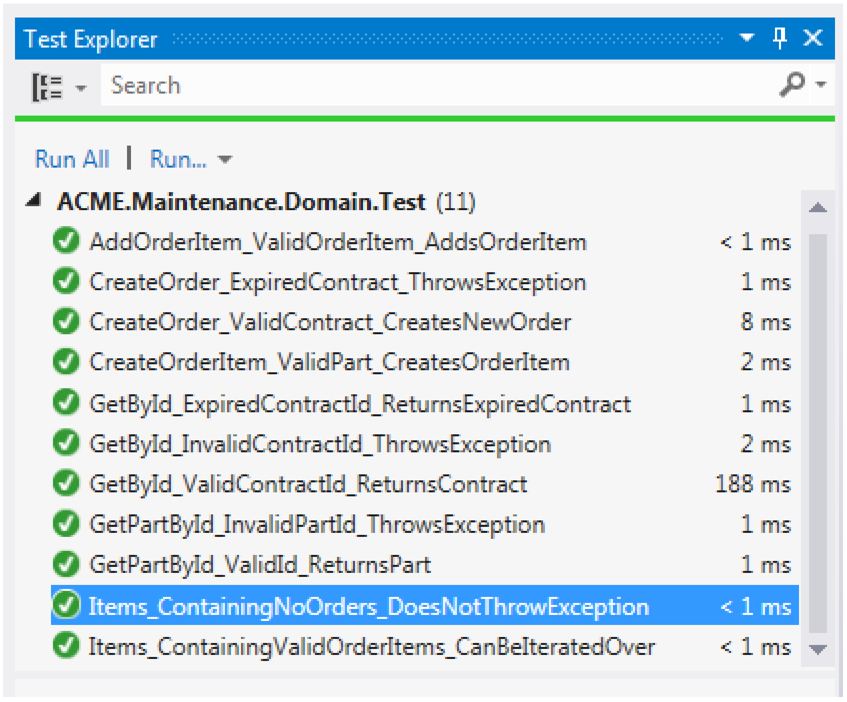
The key idea is lines 103 through 106 where I call Assert.Fail() ... in other words, if I make it to that line of code, that means there's something in the Items collection prior to calling AddOrderItem(). Again, I might be way too paranoid. However, since IReadOnlyList<T> is a new feature in the .NET Framework and I've not become as familiar with it as I would like, I feel the need to build some hedges around it. If nothing else, this gives me confidence to better understand it going forward.

Lines 109 through 111 finish up the Assertions, checking to make sure that (a) everything is zero, and (b) merely by referencing it, it doesn't throw an exception.

Get this code:

https://gist.github.com/LearnVisualStudio/0746e8271056250c08f9

5. Run All Unit Tests



```
ACME.Maintenance.Domain.Test (11)
AddOrderItem_ValidOrderItem_AddsOrderItem              < 1 ms
CreateOrder_ExpiredContract_ThrowsException              1 ms
CreateOrder_ValidContract_CreatesNewOrder                8 ms
CreateOrderItem_ValidPart_CreatesOrderItem               2 ms
GetById_ExpiredContractId_ReturnsExpiredContract         1 ms
GetById_InvalidContractId_ThrowsException                2 ms
GetById_ValidContractId_ReturnsContract                188 ms
GetPartById_InvalidPartId_ThrowsException                1 ms
GetPartById_ValidId_ReturnsPart                          1 ms
Items_ContainingNoOrders_DoesNotThrowException         < 1 ms
Items_ContainingValidOrderItems_CanBeIteratedOver      < 1 ms
```

And it works!

I'm satisfied with what I have so far. Admittedly, I only have created 11 unit tests, but this was a small (and easy) user story, and we were only focusing on the domain layer. Our architectural spike is going well. I'm not claiming it's perfect, and we may discover that we've forgotten something! If that's the case, we'll come back and revisit the Domain Layer project and unit tests.

To recap, the point of this series was to employ those ideas we talked about in the Application Architecture Fundamentals series. I laid out a scenario and we used that to analyze requirements, identify candidate architectures and classes for our domain layer.

I then decided to build an architectural spike, and chose to allow the Domain and Unit Tests to guide the design of the software. Along the way, hopefully you saw the thought process, the angst, the use of Microsoft's Unit Test Framework and Visual Studio, and perhaps even picked up a few new C# keywords. It wasn't always smooth sailing ... there was a lot of angst and second guessing, some regret, some struggle, some messy-ness and disorganization at times, but in the end I'm satisfied (enough, for now) with what we've accomplished in a relatively short amount of time.

There's plenty more to work on, but we'll pick it back up in the next series. See you there.