DevOps Kata

# Introduction to Automated Unit Testing and Architectural mapping with Visual Studio Enterprise 2015

Last updated: 10/16/2016



1. **TABLE OF CONTENT**

[Introduction to Automated Unit Testing and Architectural mapping with Visual Studio Enterprise 2015 1](#_Toc464460924)

[Overview 3](#_Toc464460925)

[Prerequisites 3](#_Toc464460926)

[Exercises 3](#_Toc464460927)

[Exercise 1: Introduction to Modern Debugging 5](#_Toc464460928)

[Task 1: Explore the code with code maps 5](#_Toc464460929)

[Task 2: Using Intellitest 12](#_Toc464460930)

## Overview

Unit testing is essential to a DevOps lifecycle. Unit tests give developers and testers a quick way to look for logic errors in the methods of classes in Visual C#, Visual Basic, and Visual C++ projects. You can check that your code is working as expected by creating and running unit tests. It’s called unit testing because you break down the functionality of your program into discrete testable behaviors that you can test as individual units. Visual Studio Test Explorer provides a flexible and efficient way to run your unit tests and view their results in Visual Studio. Visual Studio installs the Microsoft unit testing frameworks for managed and native code. Use a unit testing framework to create unit tests, run them, and report the results of these tests. Rerun unit tests when you make changes to test that your code is still working correctly. When you use Visual Studio Enterprise, you can run tests automatically after every build.

IntelliTest explores your .NET code to generate test data and a suite of unit tests. For every statement in the code, a test input is generated that will execute that statement. A case analysis is performed for every conditional branch in the code. For example, if statements, assertions, and all operations that can throw exceptions are analyzed. This analysis is used to generate test data for a parameterized unit test for each of your methods, creating unit tests with high code coverage.

Code maps can help you avoid getting lost in large code bases, unfamiliar code, or legacy code. For example, when you’re debugging, you might have to look at code across many files and projects. Use code maps to navigate around pieces of code and understand the relationships between them. That way, you don't have to keep track of this code in your head, or draw a separate diagram. So, when your work is interrupted, code maps help refresh your memory about the code you're working on.

When used together code maps and intelligent can help you understand code and create unit tests to defend against unexpected issues.

### Prerequisites

1. In order to complete this exercise, you will need Visual Studio 2015 Enterprise Edition and the source code that you can find at the following link: <https://github.com/jamestramel/taxation>. Download the zip file and extract the contents.

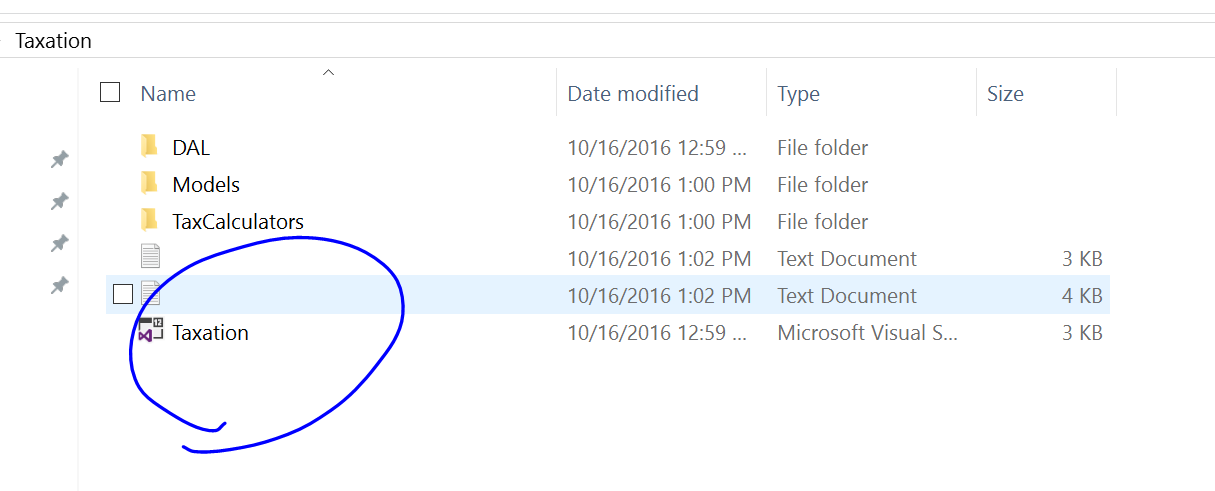
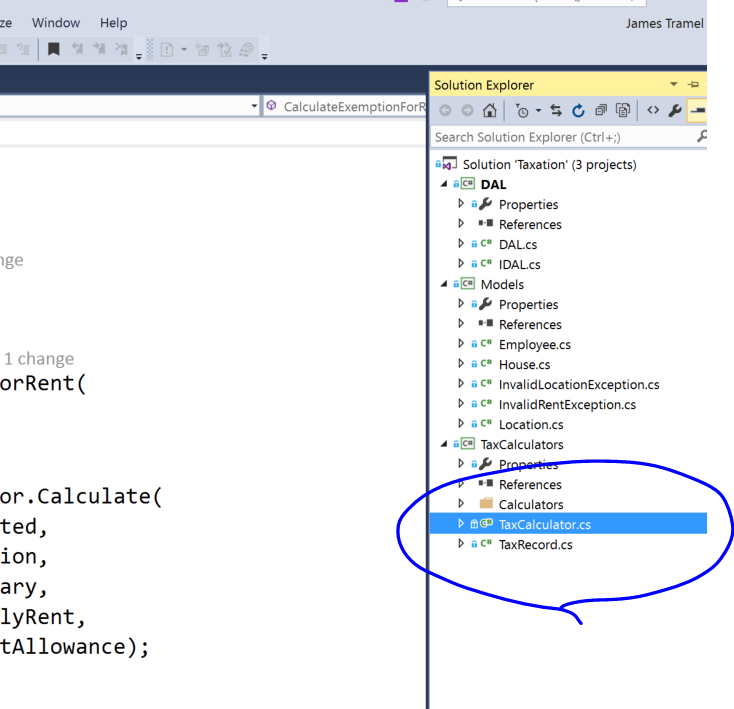
### Exercises

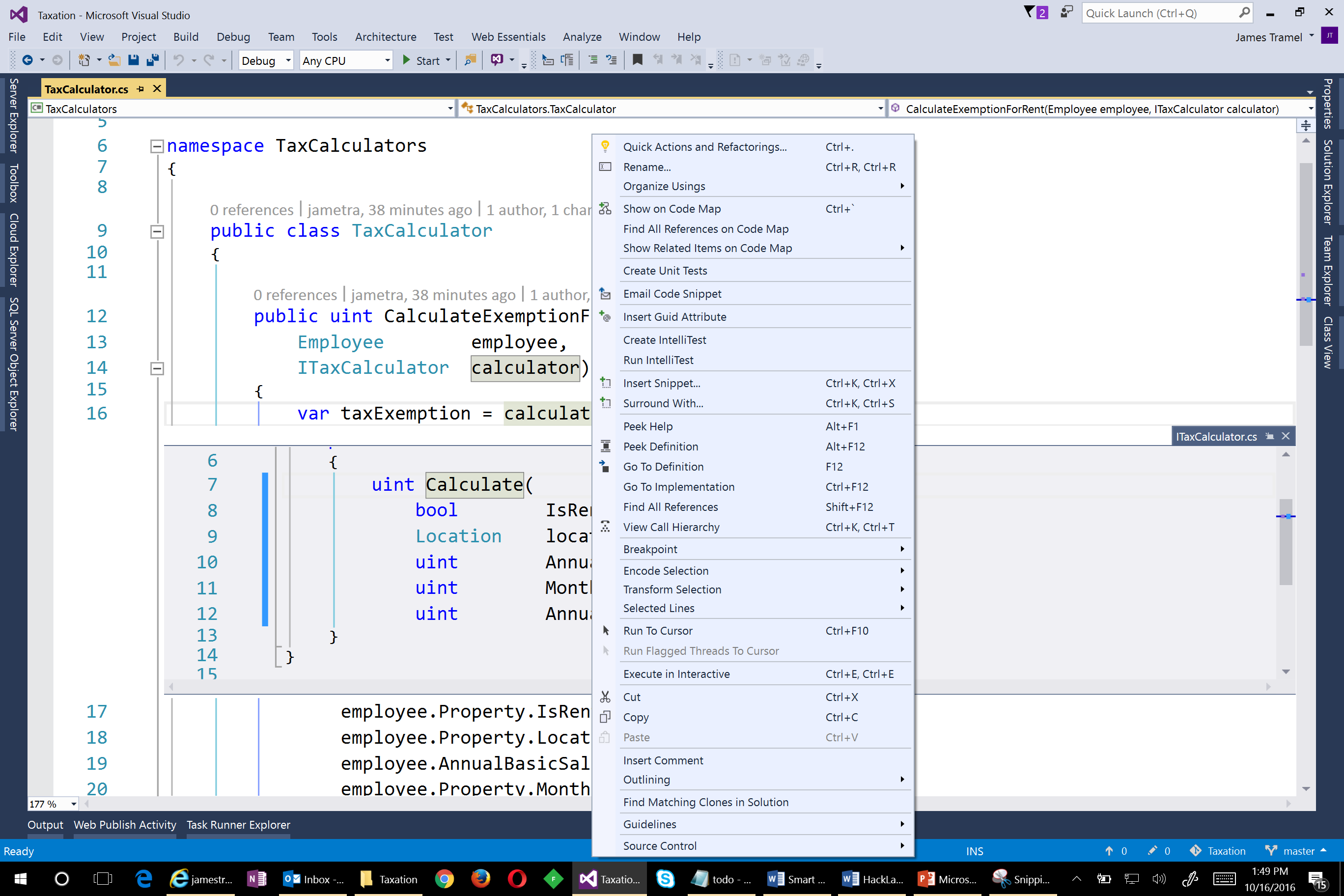
* 1. This hands-on lab includes the following exercises:
  2. Explore the code with code maps
  3. Use intellitest to generate unit tests
  4. Detail how you would resolve this issue, and send your fix to the instructor
  5. Estimated time to complete this exersize: **15 minutes**.

## Exercise 1: Introduction to Modern Debugging

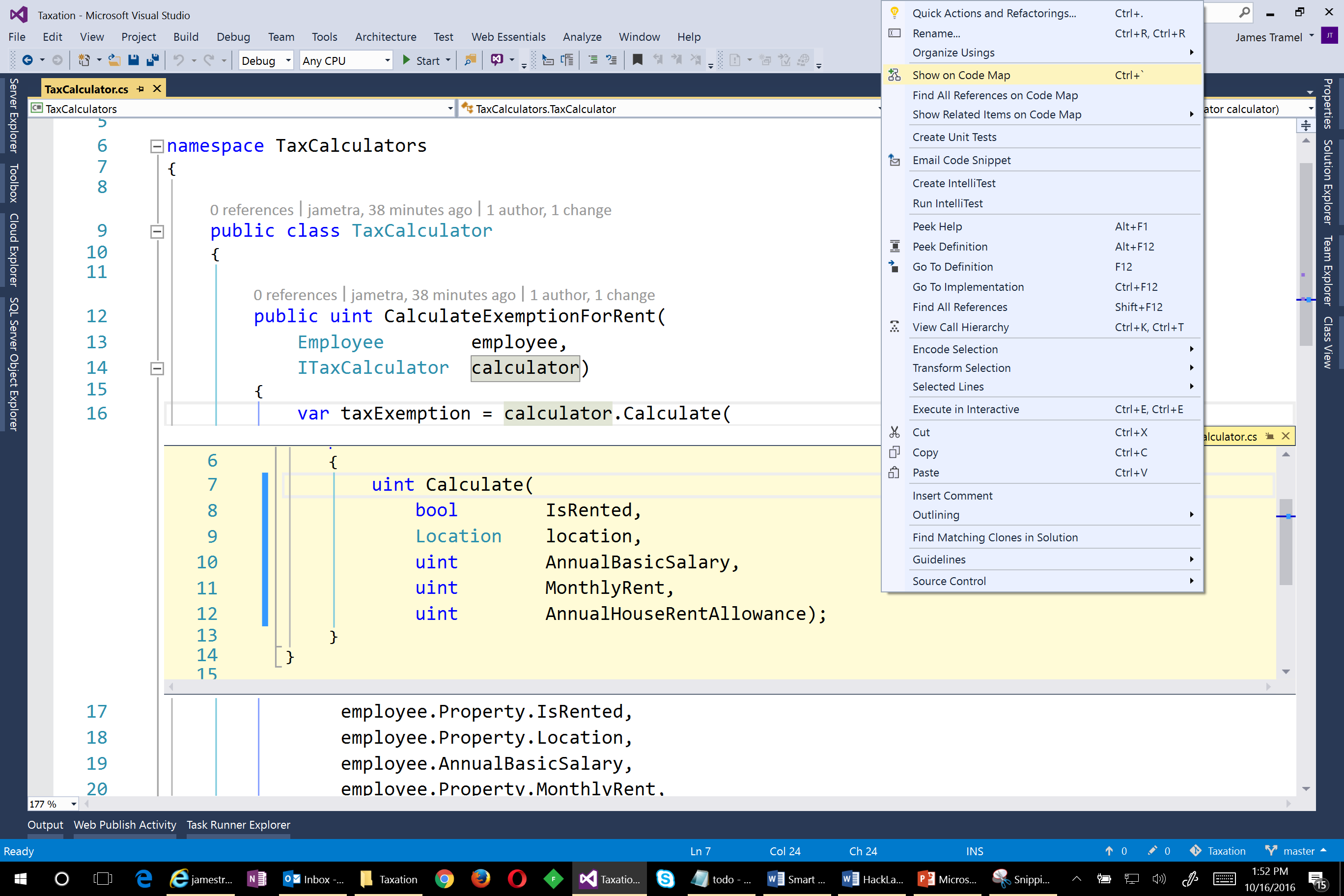
* 1. In today’s world of shorter release cycles, unit testing plays a bigger role than ever! As you modify code, you need to make sure it works as expected and that your change didn’t inadvertently cause other problems. Unfortunately, unit testing can be both a difficult and time-consuming upfront cost. Visual Studio’s IntelliTest is intended to make authoring unit tests both easier and faster.
  2. For this demonstration, we have some code for a small financial services business application.

#### Task 1: Explore the code with code maps

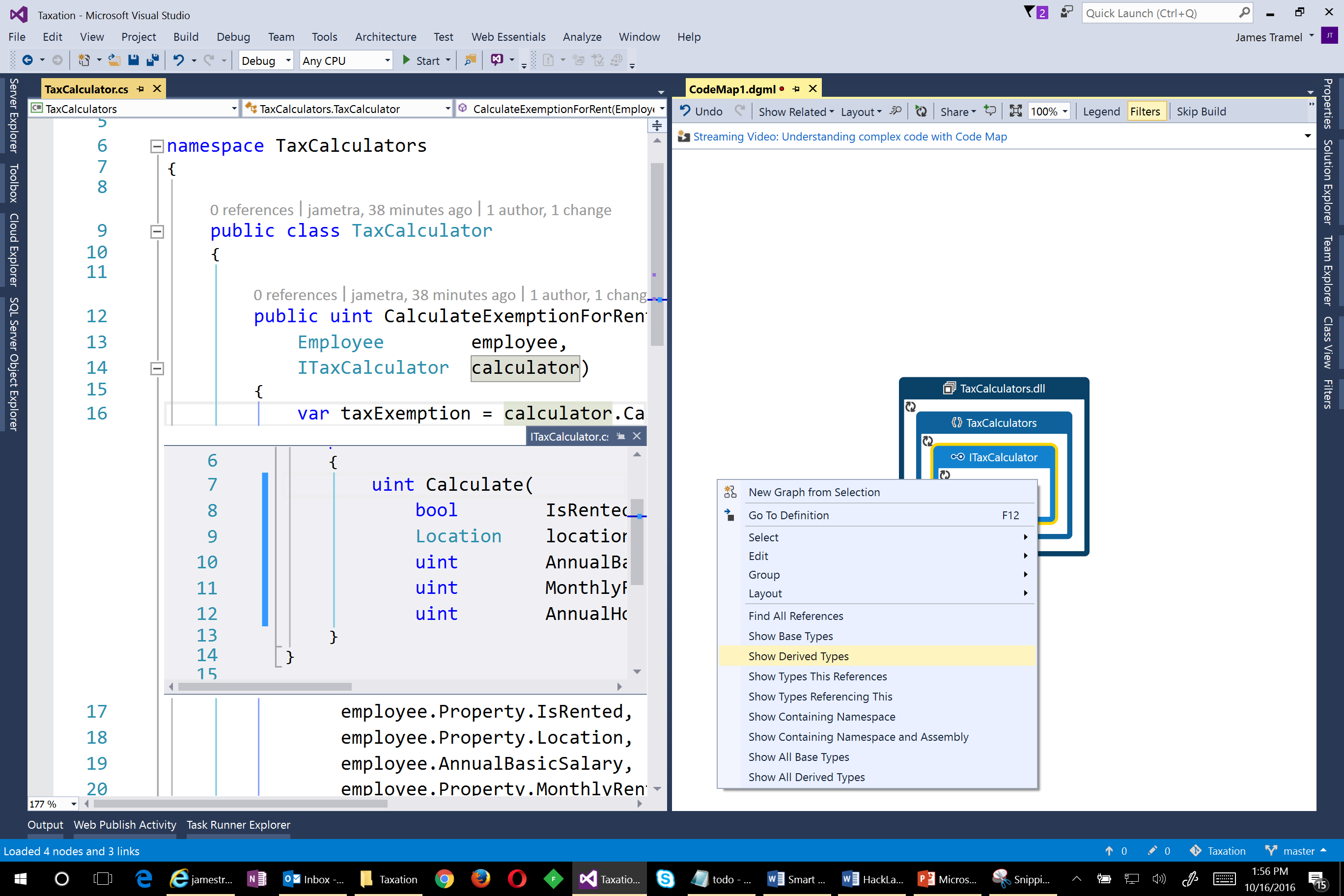
* 1. Open an **Explorer** window and then open **Taxation.sln** in Visual Studio 2015 Enterprise from the source code provided.
     1. 
  2. In **Solution Explorer**, expand the **TaxCalculators** project and double-click the **TaxCalculator.cs** file  
     
  3. This class’s CalculateExemptionForRent method takes both an Employee parameter and a concrete implementation of the iITaxCalculator interface. To test this method, we’ll need to handle the creation of a valid Employee object and we’ll need to first choose, and then instantiate, at least one concrete implementation of the ITaxCalculator method.

Let’s get a better sense of the Calculate method to see how complex the logic will be to test. Right-click the **calculator.Calculate** method call, and select **Peek Definition**.

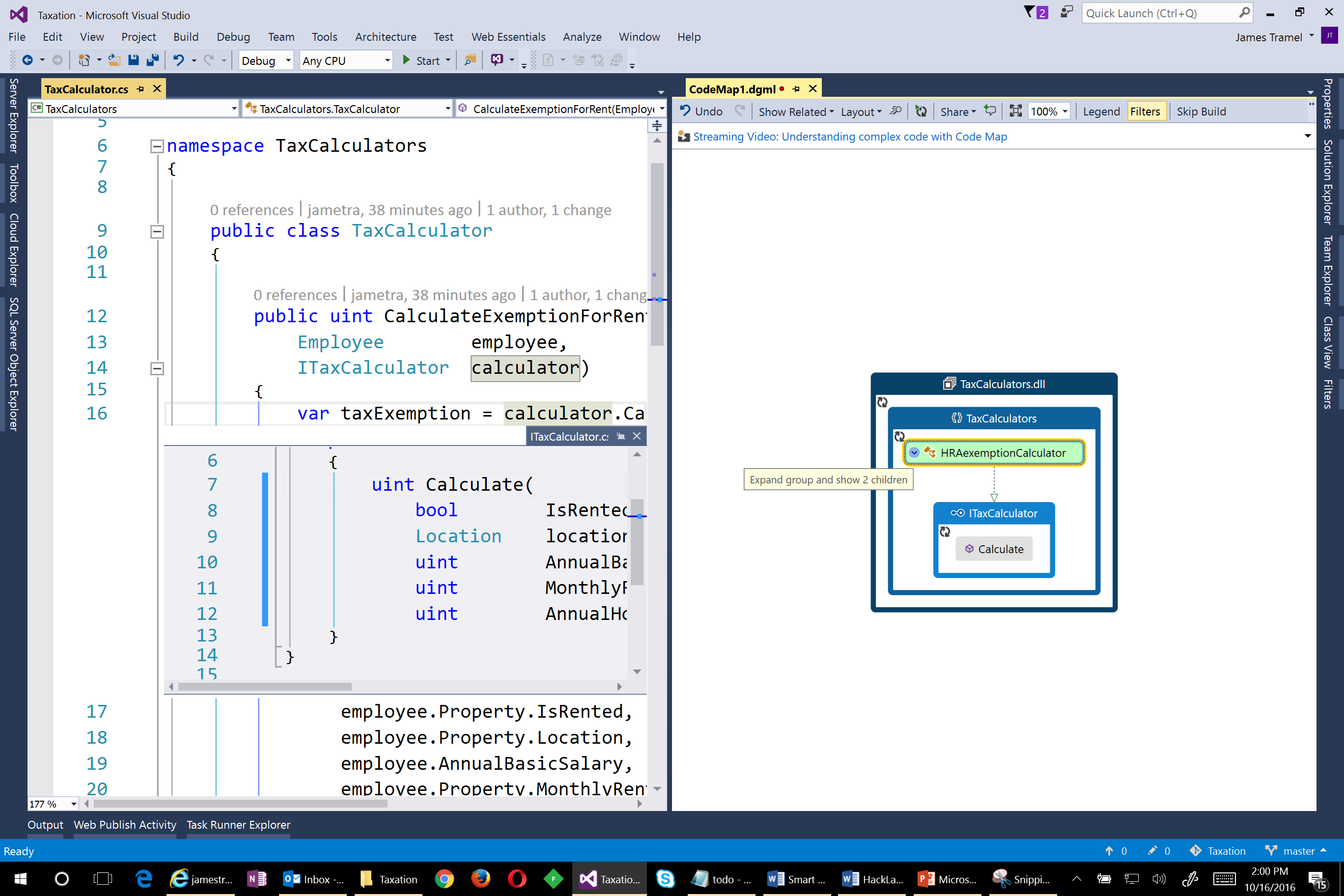


* 1. Peek only lets us see the interface and we want to see the definition of the Calculate method. In the Peek window, right-click **ITaxCalculator,** and select **Show on Code Map**. 

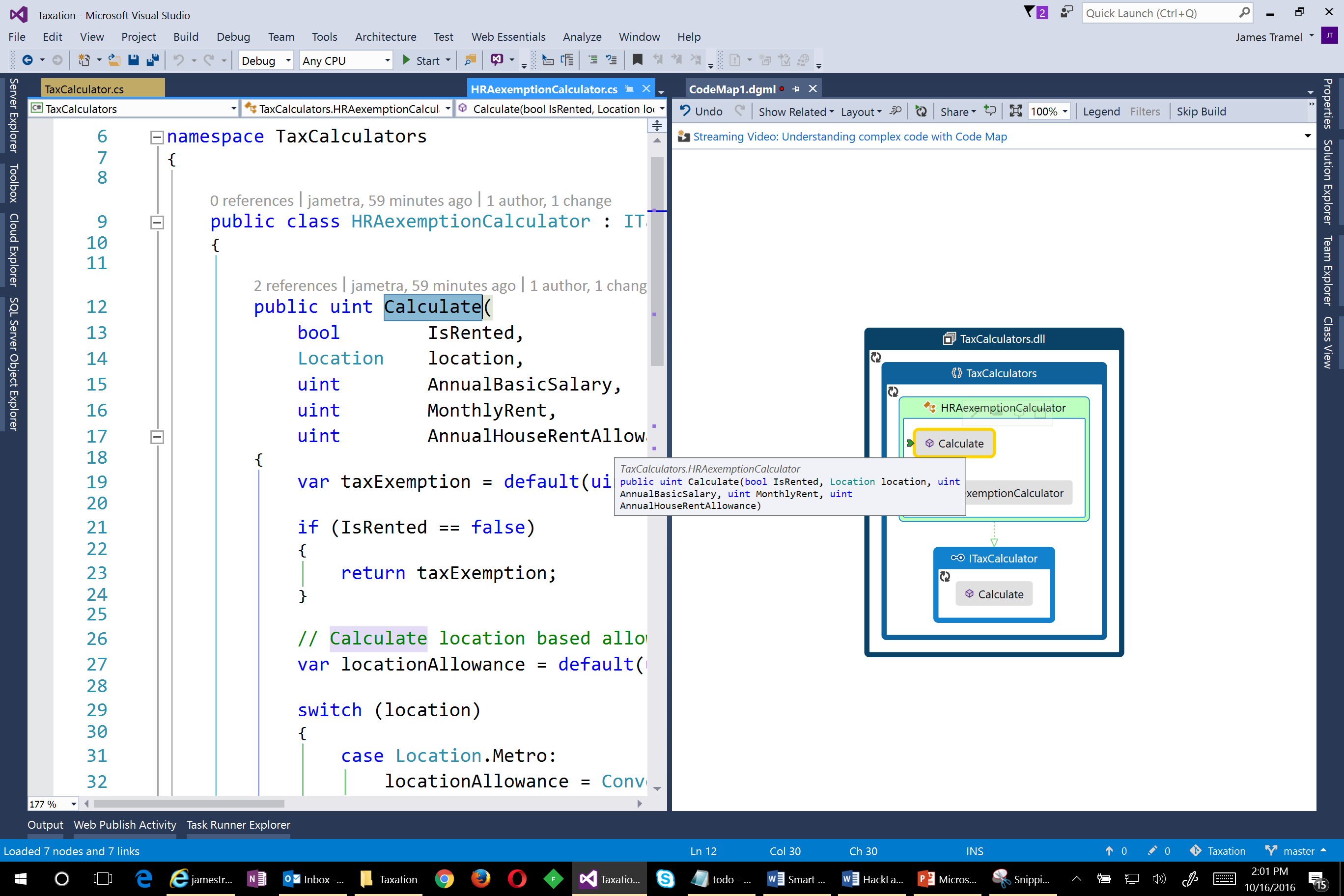


* 1. Code Map is a useful tool that helps us explore our solution. In this case, let’s find all the classes that derive from our interface. Code In the Code Map window, right-click the ITaxCalculator node, and select Show Derived Types. 



* 1. There’s just one type deriving from our interface. Let’s see how this type implements the Calculate method. Expand the **HRAExemptionCalculator** node. 



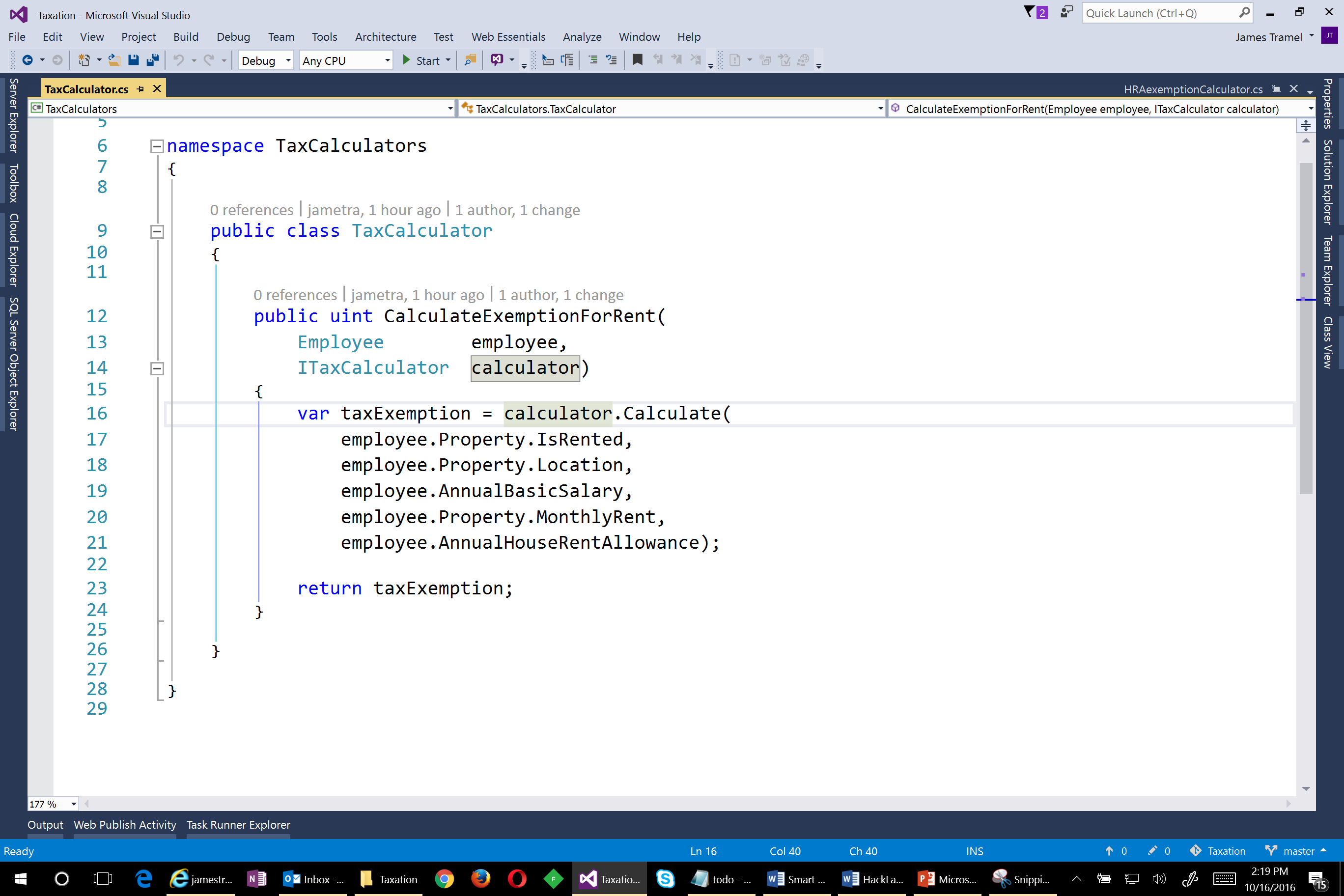
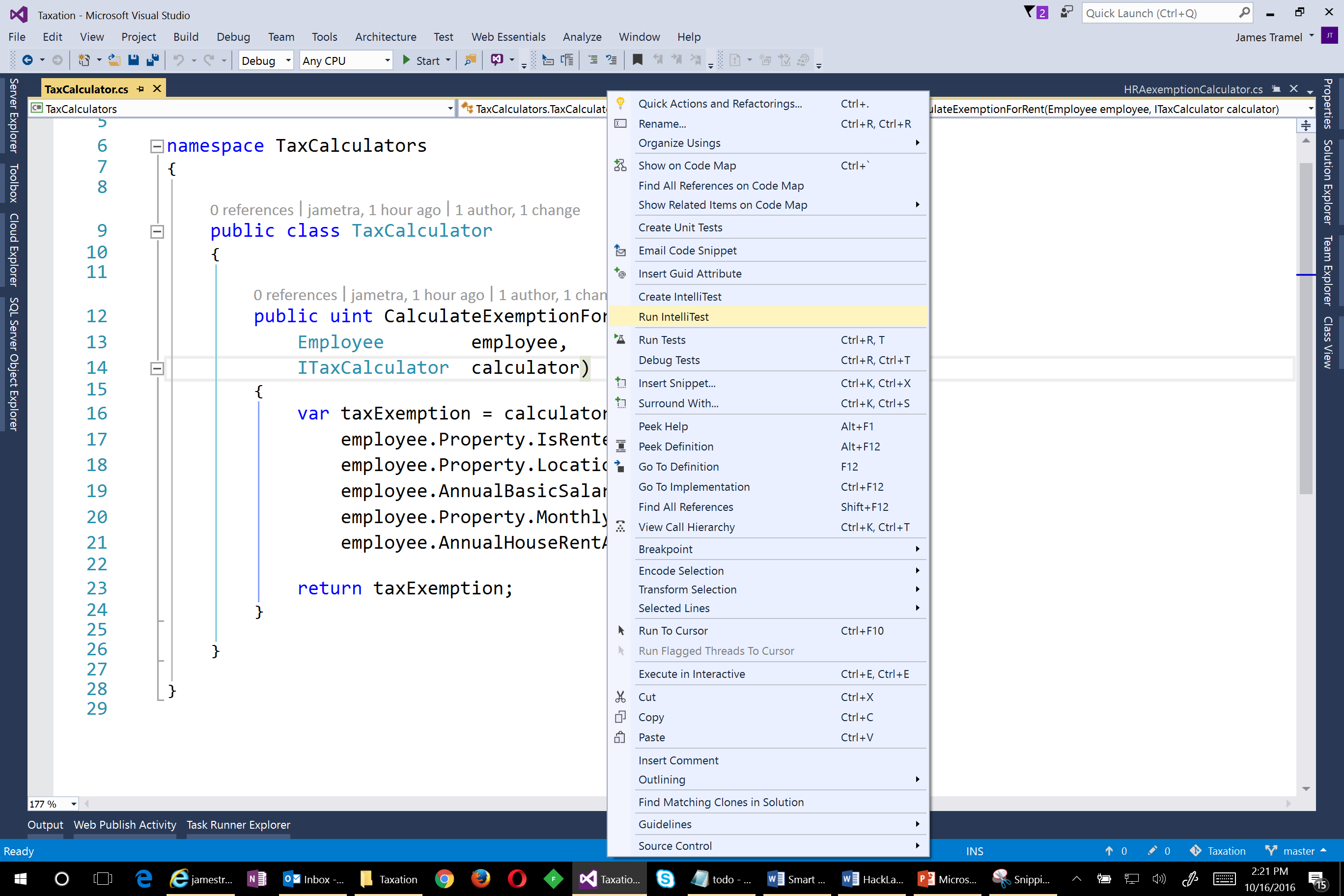
* 1. Double-click the **Calculate** method. Close the **CodeMap1.dgml** tab. When prompted to save changes to the **CodeMap1.dgml** file, click the **No** button. 

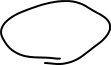


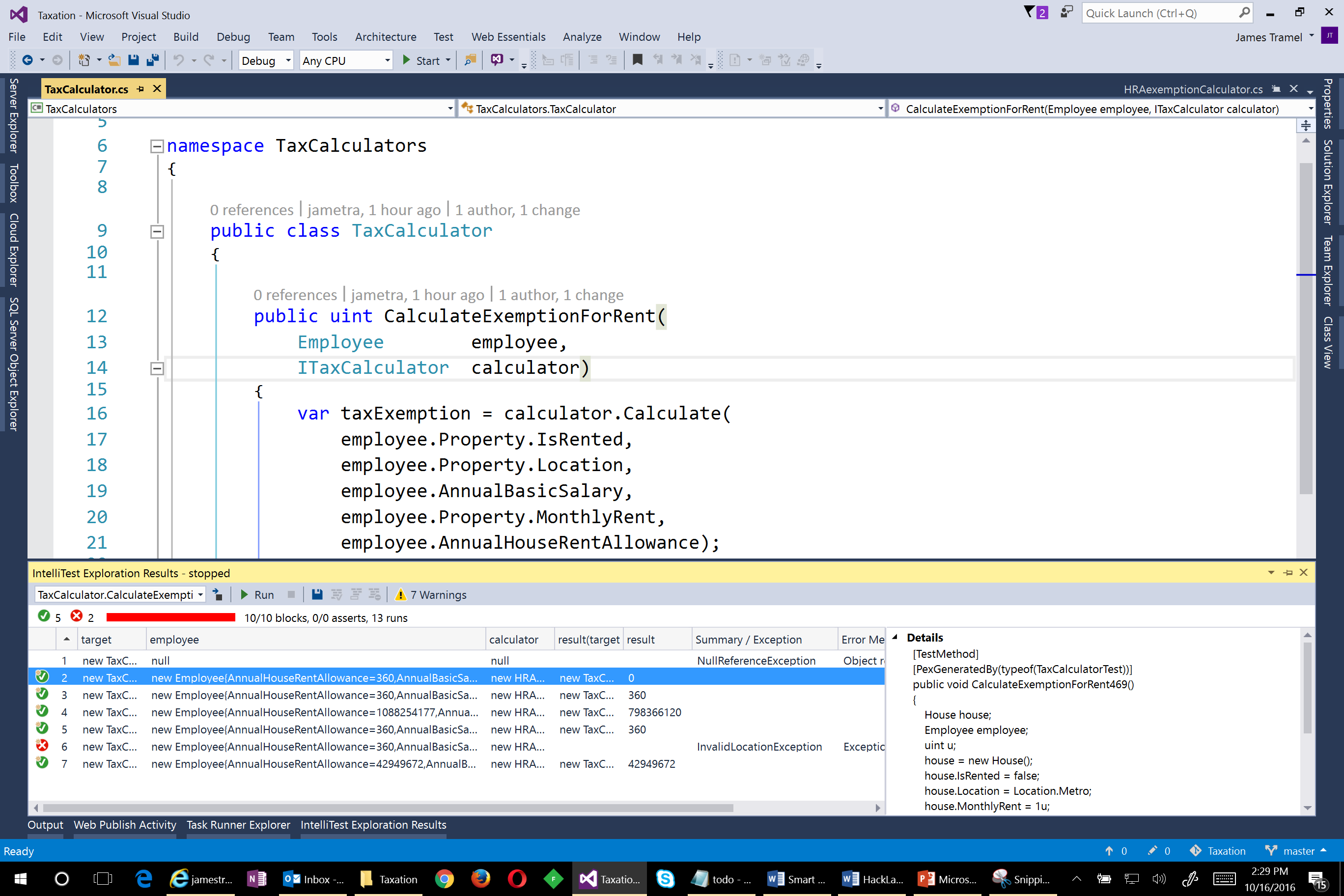
* 1. We already know this class is a concrete implementation of the ITaxCalculator interface, which only requires its derived classes to implement the Calculate method. Each of these concrete calculators could be used in different circumstances and may contain complex logic like you see here. To hit 100% code coverage, you’d have to write tests that consider each of these logic branches.

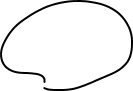
(We aren’t saying that code coverage is the ideal metric. You can hit 100% code coverage and still not have good tests, but if you have a set of good tests, you will likely achieve 100% code coverage.

#### Task 2: Using Intellitest

* 1. In Solution Explorer, double-click the TaxCalculator.cs file to get back to it. 
  2. Right-click within CalculateExemptionForRent method and select Run IntelliTest (can be done from anywhere within the method). 



* 1. The IntelliTest Exploration Results pane appears. Just glancing at this, you can see we’ve got five passing and two failing tests. It also indicates 10 of 10 code blocks are covered by these tests. Before we go too much further, let’s talk about what’s really been generated at this point. Our first test passes in a null Employee and a null ITaxCalculator. Not surprisingly, this results in a NullReferenceException somewhere in our code – right where the Calculate method is called. If we expand out the employee column, we can see the remaining tests pass in Employee objects with different property values that will exercise our Calculate method’s logic branches. A couple of these tests are failing and IntelliTest can help us understand and resolve these issues too. Each of these rows is a unit test. Select a row to explore the details of a unit test. In the **IntelliTest Exploration Results pane**, expand the Employee column and select the second test. To the right, you can see the unit test used here. Various properties are set on an Employee object and an instance of the HRAExemptionCalculator is created. Both are passed to the method under test and a couple of basic Asserts are added.



* 1. You can see IntelliTest generated some code for each of these tests, but this code is not persisted anywhere. This means we are limited in what further customization we can make to the unit test and we can’t yet use it as part of an automated build. This is easily resolved by selecting the desired tests and clicking the Save button (in the intellitest results pane) to create and save a unit test project within your solution. 