Hands-On Lab

Developing Robust and Reliable Applications with Microsoft Visual Studio 2005 Team System

Visual Studio Team System

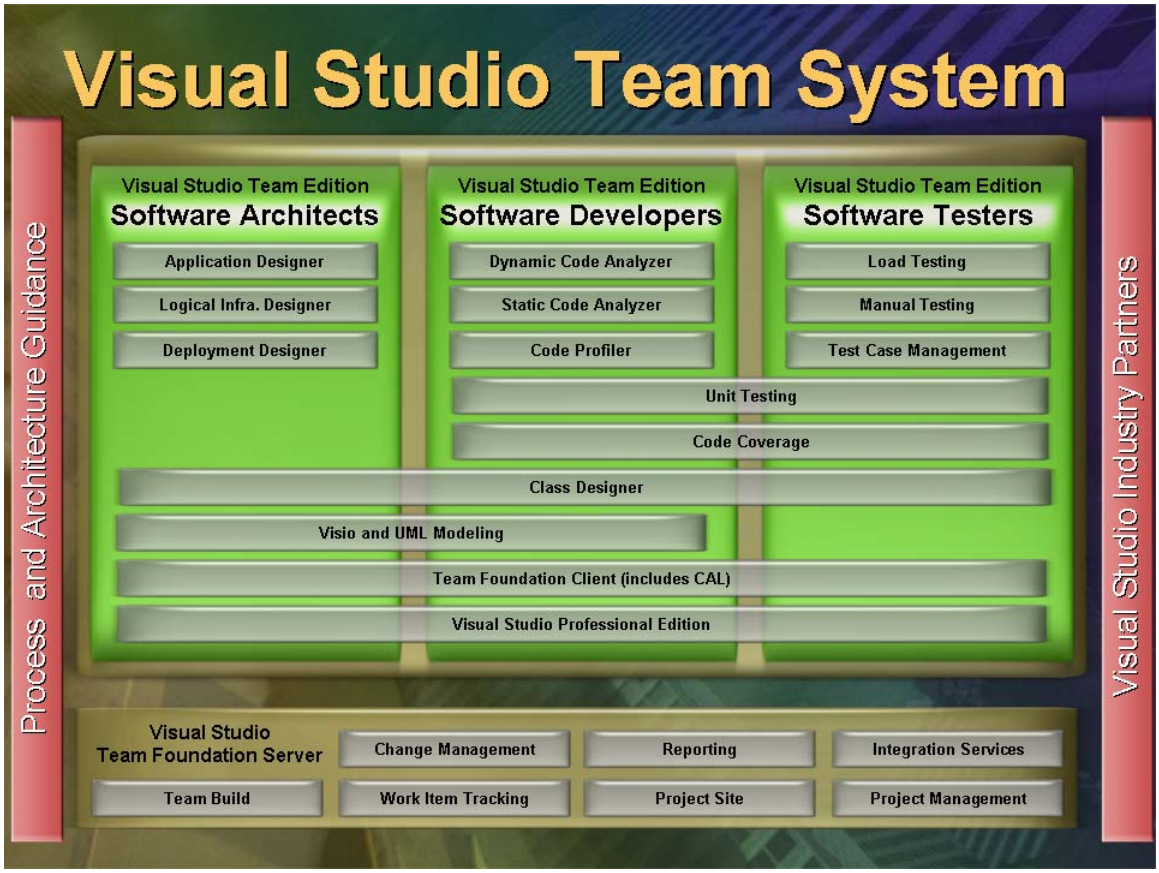
To help IT teams become top performers, Microsoft® needs to do more than provide developer tools…we need to provide development tools.

To that end, we are expanding the Visual Studio product line to include new tools for a variety of roles in the IT lifecycle.

Microsoft's announcements around Team System include…

Visual Studio® Team System, an extensible lifecycle tools platform that significantly expands the Visual Studio product line and helps software teams collaborate to reduce the complexity of delivering modern service-oriented solutions **Microsoft's offerings now include** a comprehensive set of proven process frameworks, best practices, prescriptive architecture guidance, and integrated lifecycle tools that enable IT organizations to successfully deliver custom solutions on the Microsoft Windows® platform

The following diagram shows all aspects of Visual Studio 2005 Team System; all of these components are part of this lab.



Numerous industry partners have agreed to extend our lifecycle tools:

GSIs: Accenture, Avanade, Capgemini, EDS, Fujitsu, Unisys

ISVs: Amberpoint, Avicode, Borland, Compuware, Serena Telelogic

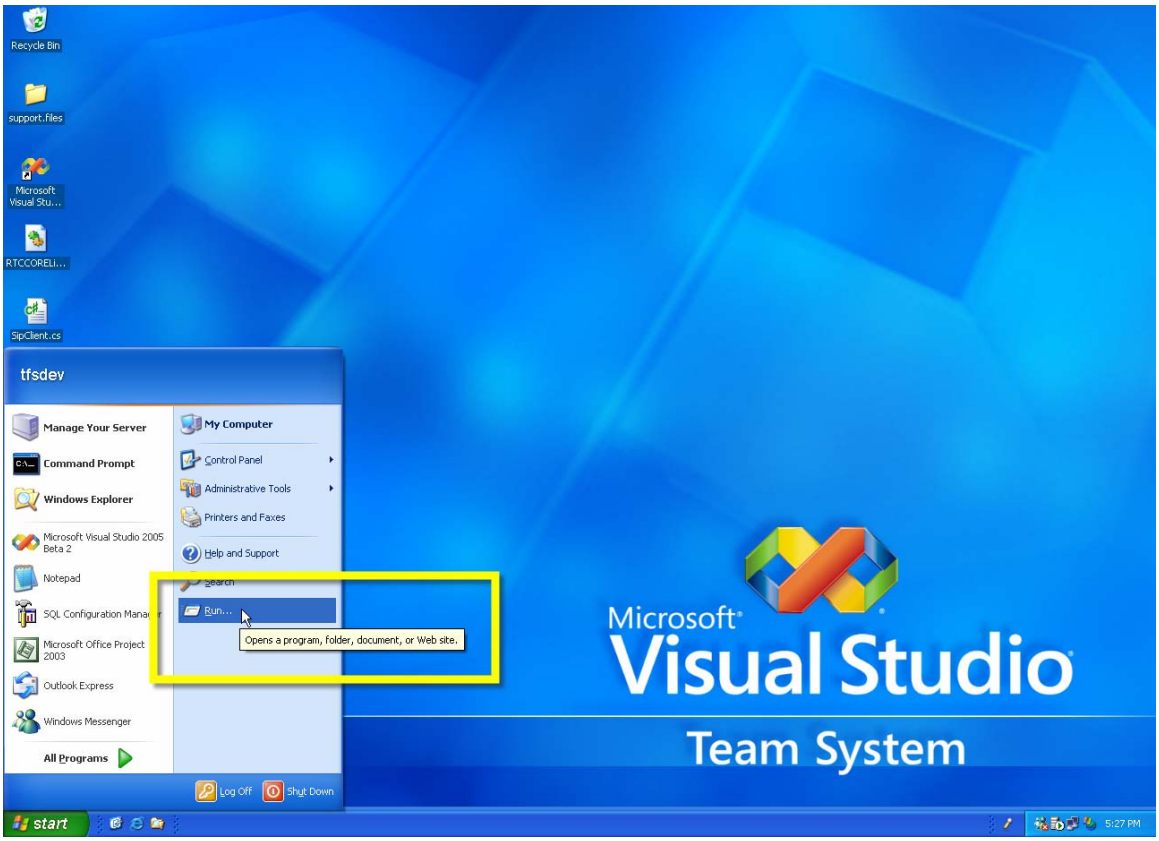
With **Visual Studio Team System**, organizations can:

- **Reduce the complexity** of delivering modern service-oriented solutions that are designed for operations
- **Facilitate collaboration** among all members of a software team (including architects, developers, testers, and operations managers), speeding development time and ensuring the predictability and reliability of the development process
- **Customize and extend** the Team System with their own internal tools and process frameworks or choose from over 450 supplemental products from over 190 partners

Before you begin

Visual Studio Team System makes heavy use of XML Web Services; these web services are hosted in Internet Information Services (IIS). Sometimes on a VPC, IIS can be lead into a corrupt state, it is best to restart this service before your lab.

Actions	From the **Start** menu, choose the **Run** option as shown

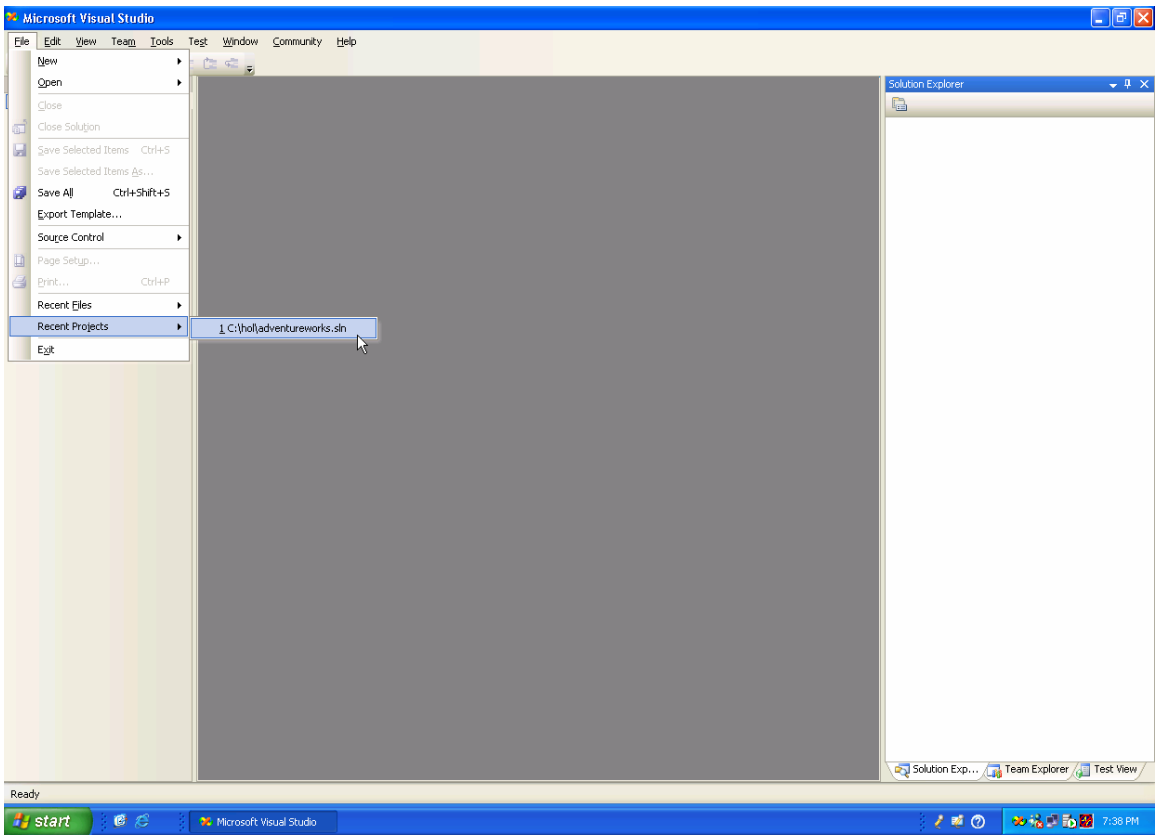


Actions	Type **iisreset** and press **OK**

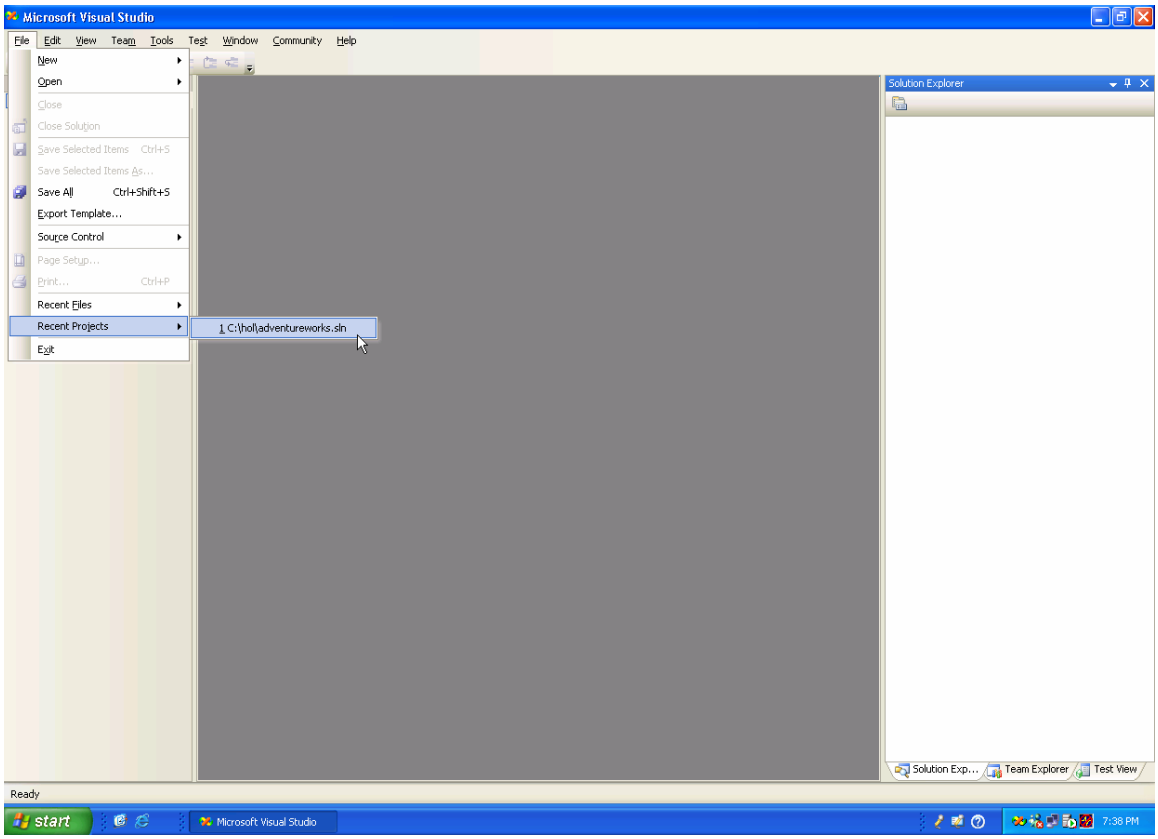
Now we are ready to start Visual Studio 2005 Team System; all aspects of Visual Studio 2005 Team System are installed on this VPC.

Actions	Double-click the **Microsoft Visual Studio 2005 Beta 2** desktop short cut to start **Visual Studio 2005 Team System**

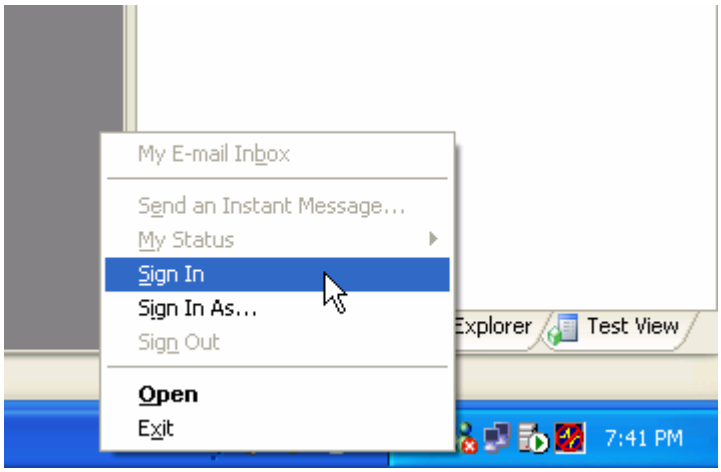
Actions	Open the **AdventureWorks** solution from the **File -> Recent Projects** menu option



Actions	Open the **AdventureWorks** solution from the **File -> Recent Projects** menu option



Actions	Right-click on the **Messenger** icon on the **Windows Task Bar** Choose **Sign In**; it will take a few moments to sign-in



What we will do in this lab

Adventure Works is an online retailer of sporting goods; in order to stay ahead of the competition, Adventure Works is striving to make sure their customer's get the most interactive and friendly experience possible.

One aspect of this effort is to send a personalized IM message to each customer that makes a purchase.

The goal of this lab will be to plan, design, implement and test this feature using Visual Studio 2005 Team System.

Format of this lab

This lab contains the following exercises:

Doing all of these exercises will likely take longer than 60 minutes, so please feel free to pick and choose the ones that interest you.

These exercises can be done sequentially or individually in any order. Instructions in each section will indicate if any prerequisite sections are required. Generally speaking, most of these sections can stand alone.

The following conventions are used in this lab.

Text in this format is for instructions and background information related to a lab or exercise.

In each exercise, each step has an associated screen shot. These steps are formatted as follows:

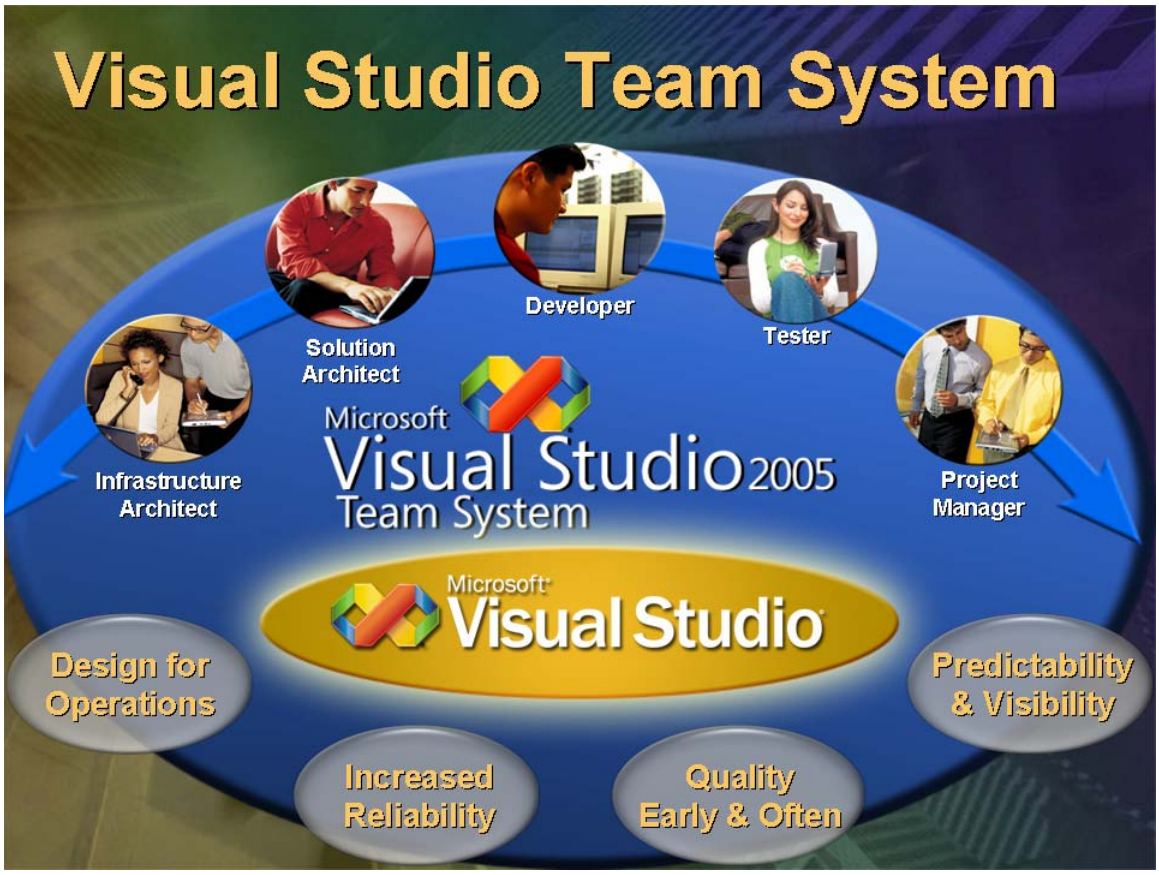
Actions	*The tasks that you have to perform in this step are listed here; there can be one or more tasks listed*
A screen shot of the most significant action is shown here.	

Good luck, and have fun with Visual Studio 2005 Team System!

Exercise - Syncing up with your Team

Visual Studio Team System was designed to integrate and bring productivity to all f the disciplines in a typical software lifecycle.

In this lab, we will concentrate mainly on the Developer and Test roles.

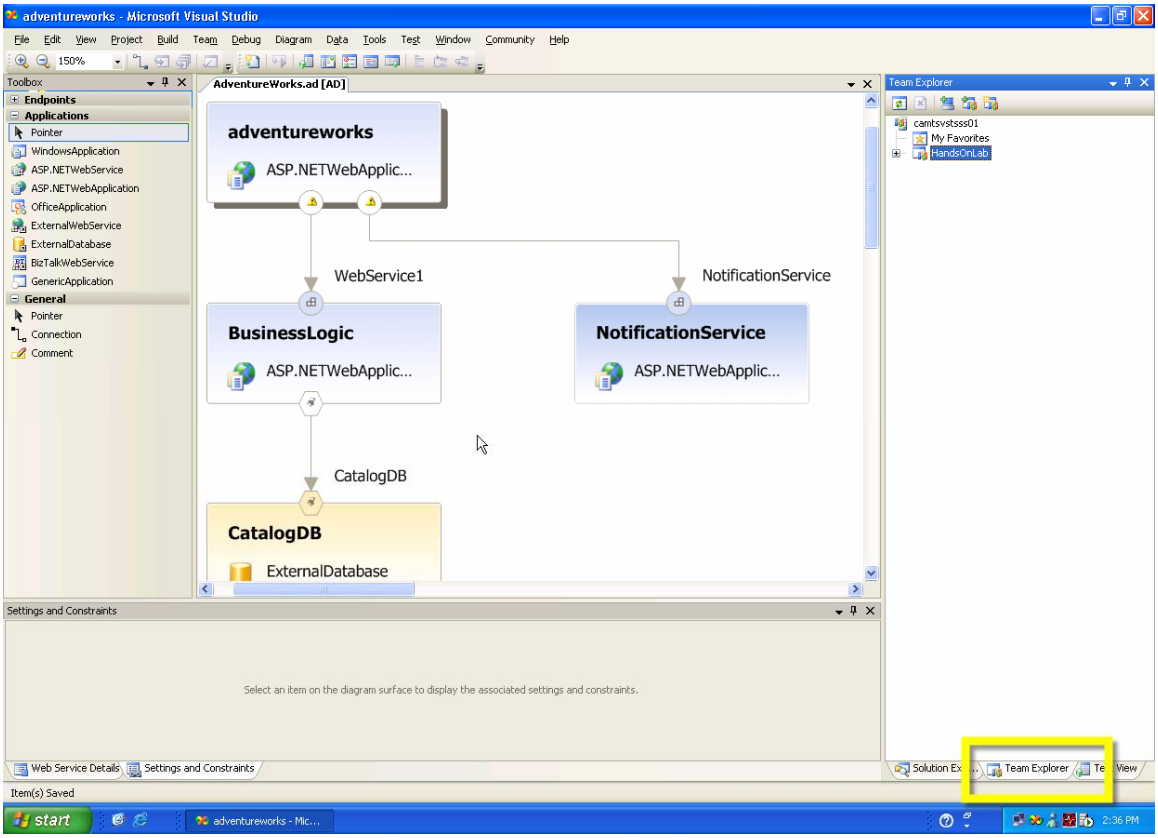


As a developer or tester, one of the best ways to ensure that you are writing robust and reliable software is to make sure you are on the same page with the rest of your team.

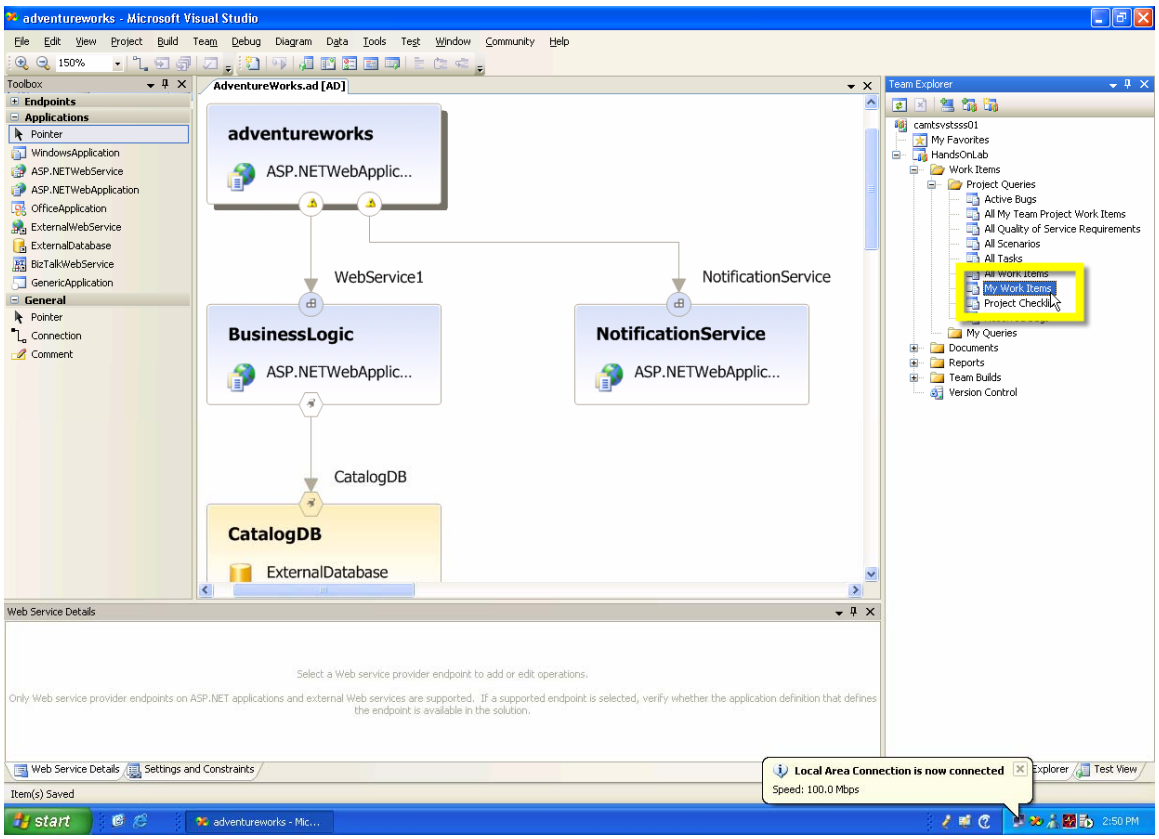
In this short section, we will demonstrate how a developer can use Visual Studio Team System to view a list of work items that are assigned to them.

Work items are very pervasive in Visual Studio Team System – see HOL189 for an example of how a project manager might create a set of work items.

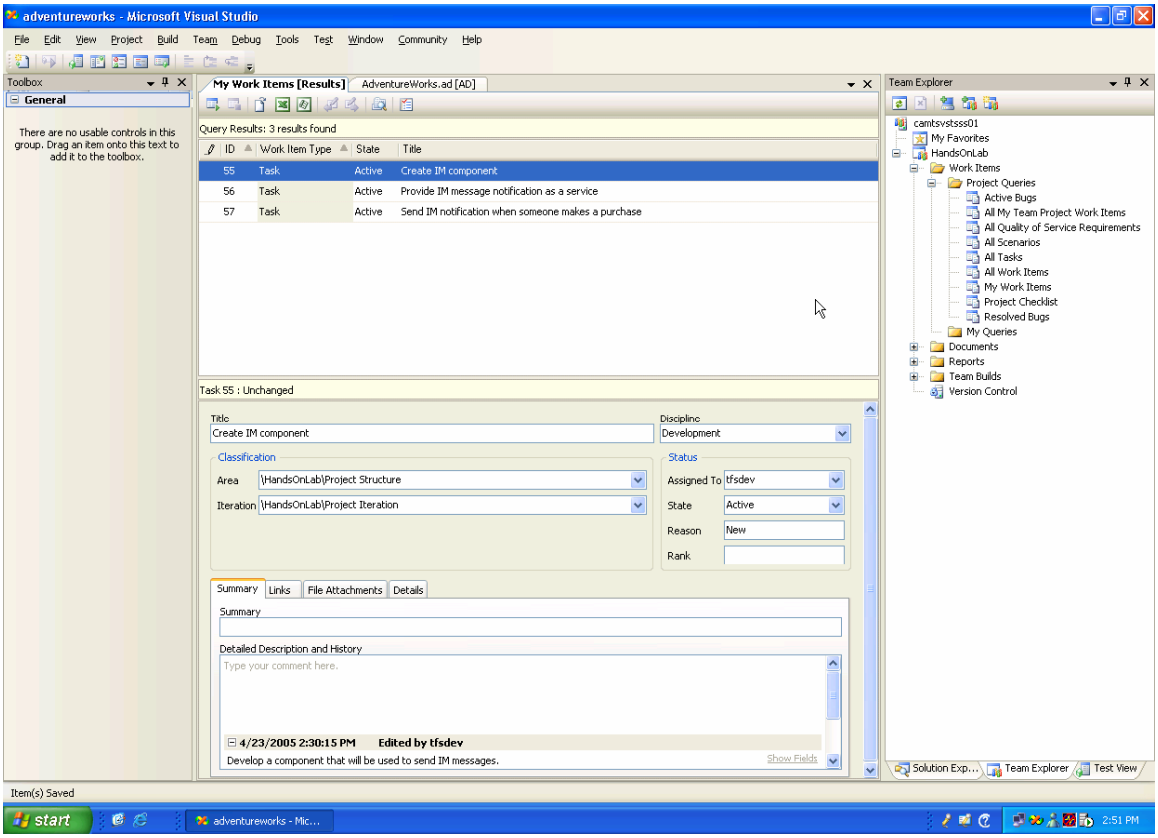
Actions	Click on the **Team Explorer** tab



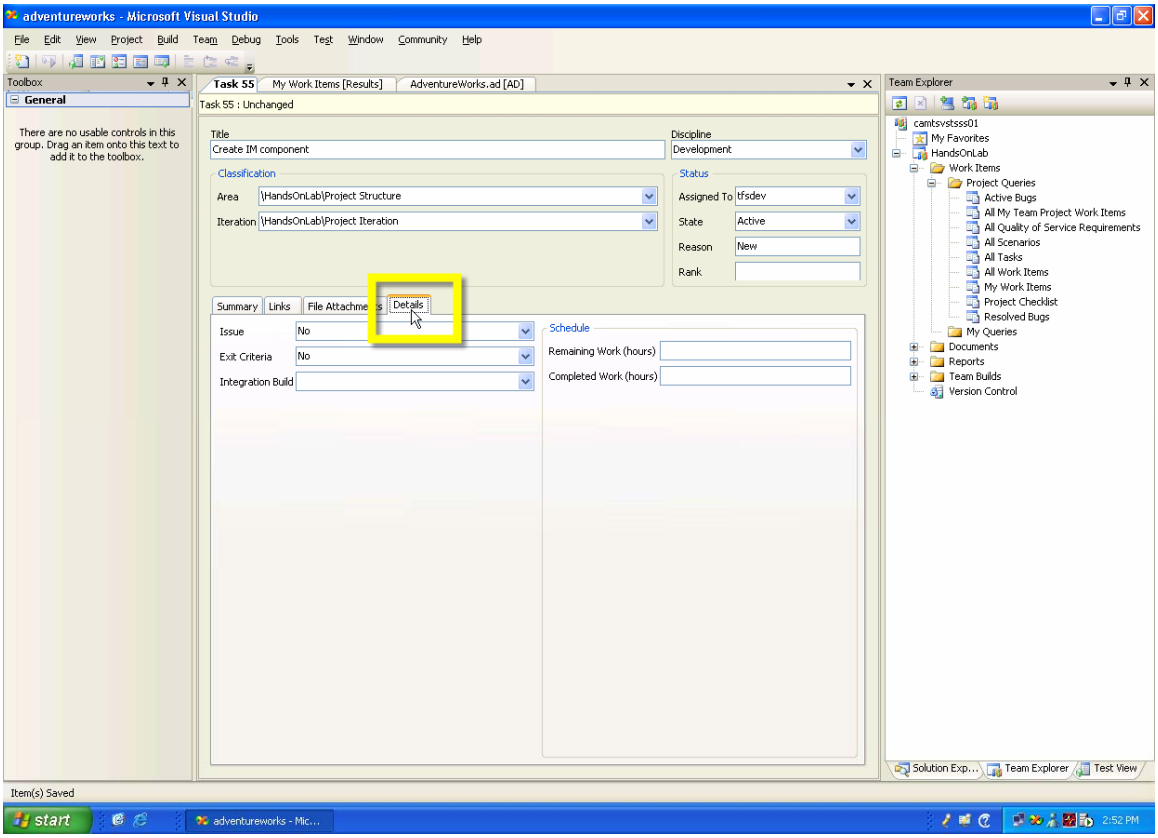
Actions	Double-click on **HandsOnLab -> Work Items -> Project Queries -> My Work Items**



Actions	Double-click on the **Create IM Component** work item

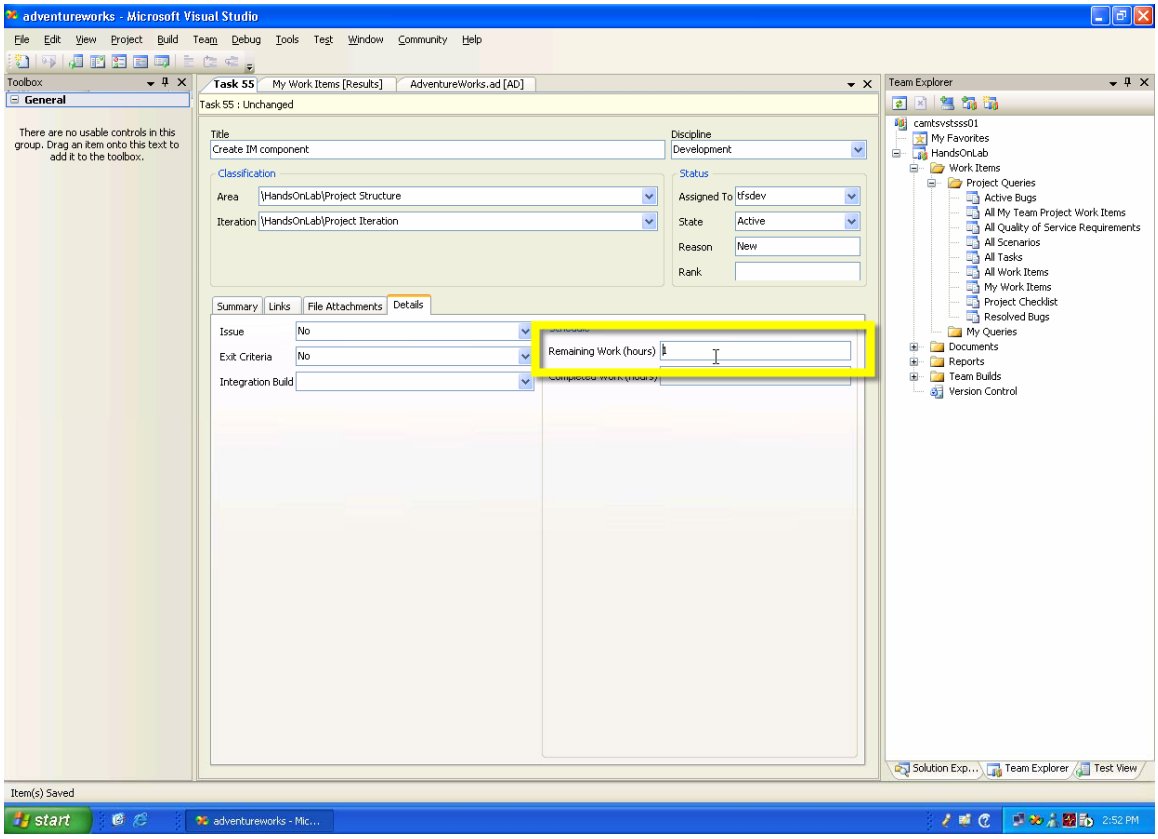


Actions	Click on the **Details** tab

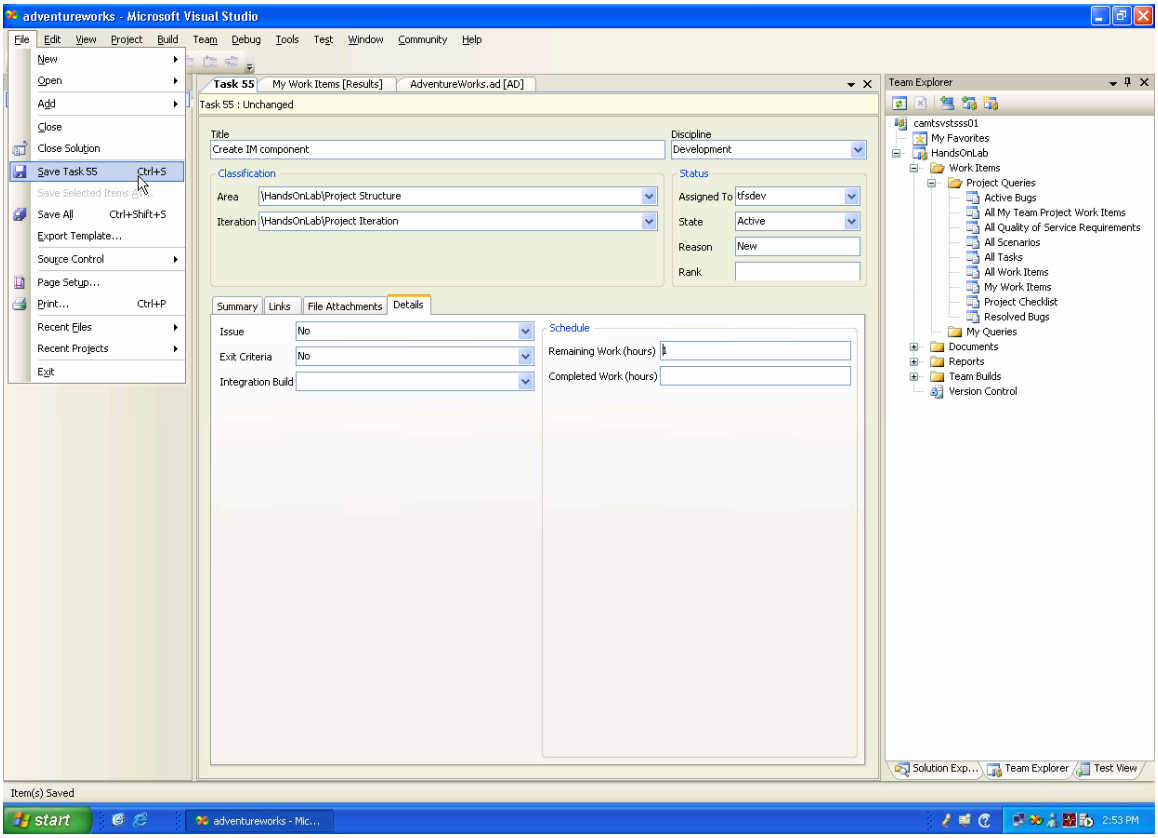


Remaining Work allows us to estimate how much effort is left in this task – the unit of this data point depends on the organization you are working in.

Actions	Set the **Remaining Work** value to 1



Actions	Save this **task** using the **File -> Save Task 55** menu option



We have assigned a work item to ourselves, so we are all set to start our development work.

By assigning this task to ourselves, the rest of our team knows that we are working on this task, so no effort is wasted.

In the next section, we will use an interesting approach to development called Test Driven Development.

Exercise - Test Driven Development

We will house our core IM-sending functionality into a component. We expect this component to be used by many other groups in the company, so it is crucial that we come up with a good design.

To help us come up with an intuitive design, let's forget for a moment about writing the component. Instead, let's think about how a user would *want* to use our component to send a message.

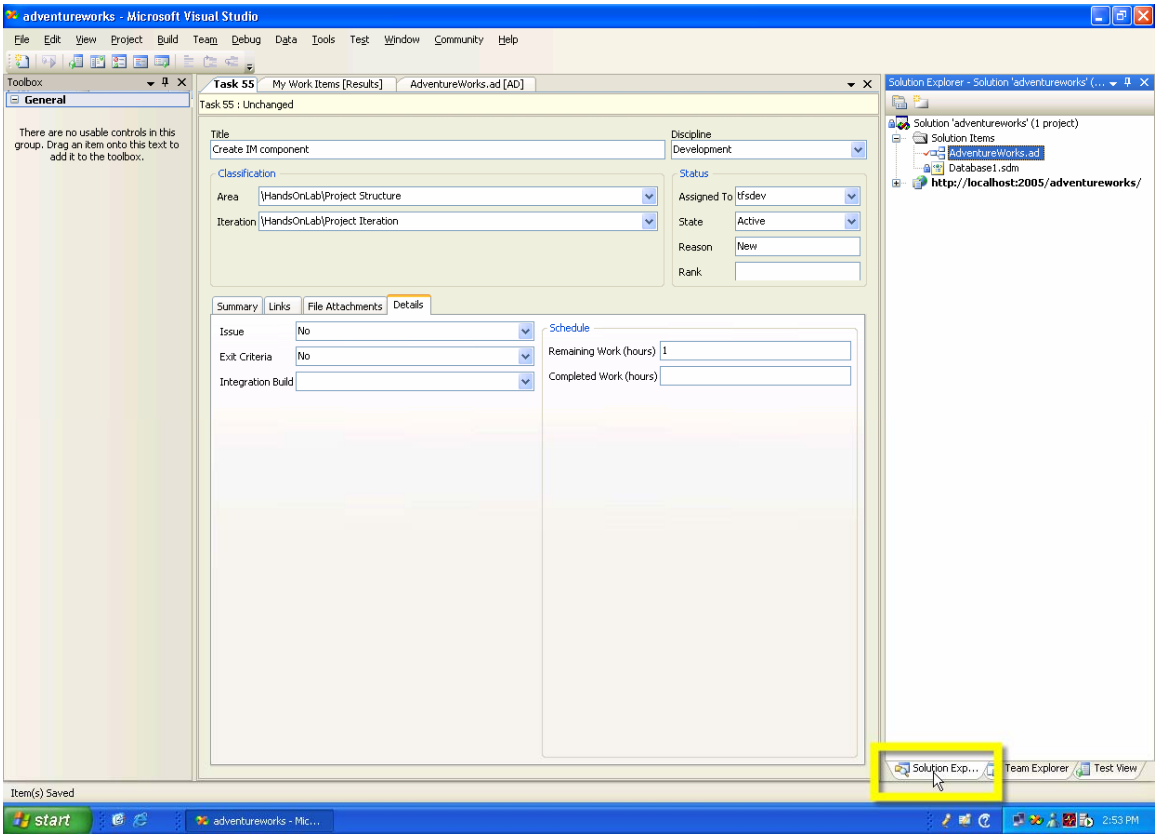
This approach is commonly known as 'Test Driven Development'.

Visual Studio 2005 Team System supports many different types of development; everything from very structure approaches like 'Test Driven Development', to traditional, free-form development. Visual Studio Team System supports all of these approaches equally well.

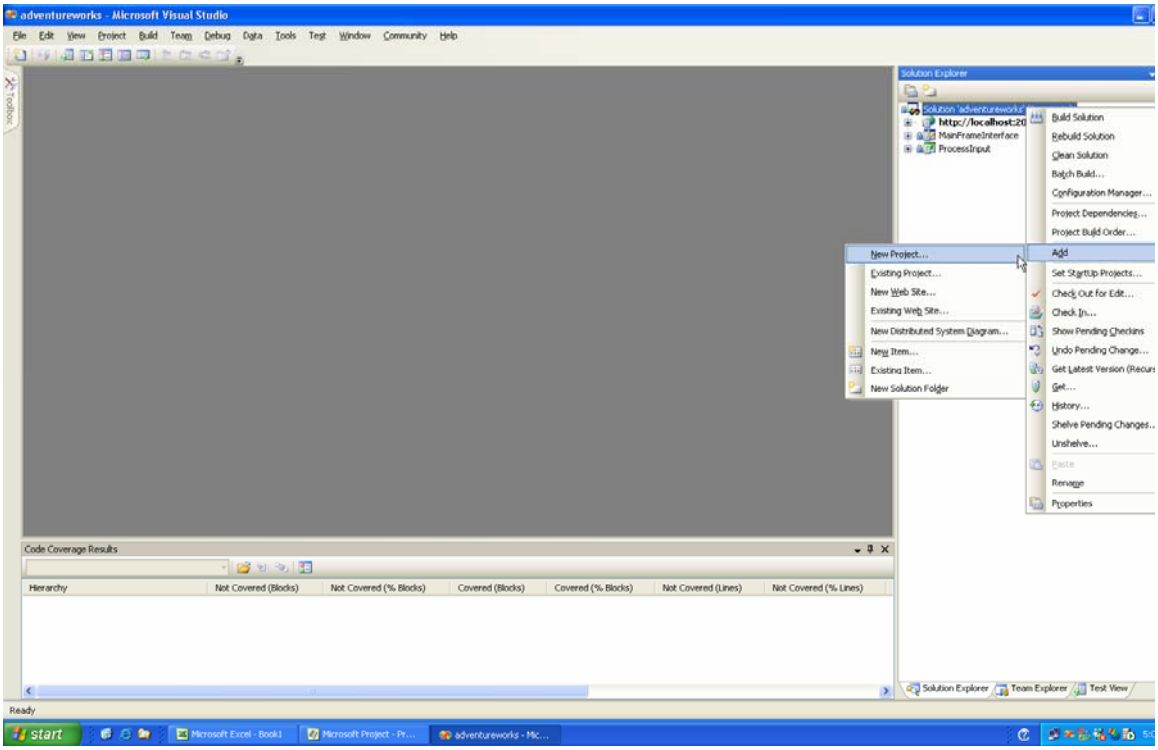
This section of our lab is meant to be a very gentle introduction to 'Test Driven Development'; die-hard followers of this approach will recognize this right away, please be patient.

Our first step is to create the test for the code that we want to write. Visual Studio Team System introduces a dedicated project type to house tests.

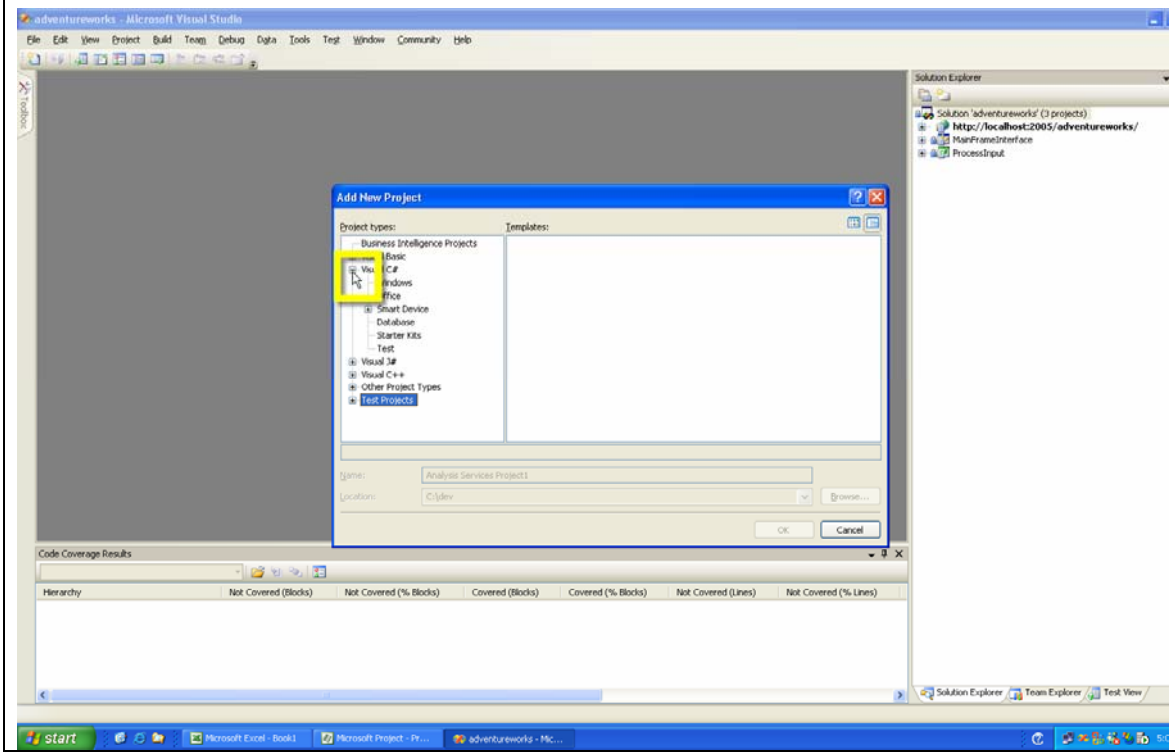
Actions	
	Click on the **Solution Explorer** tab



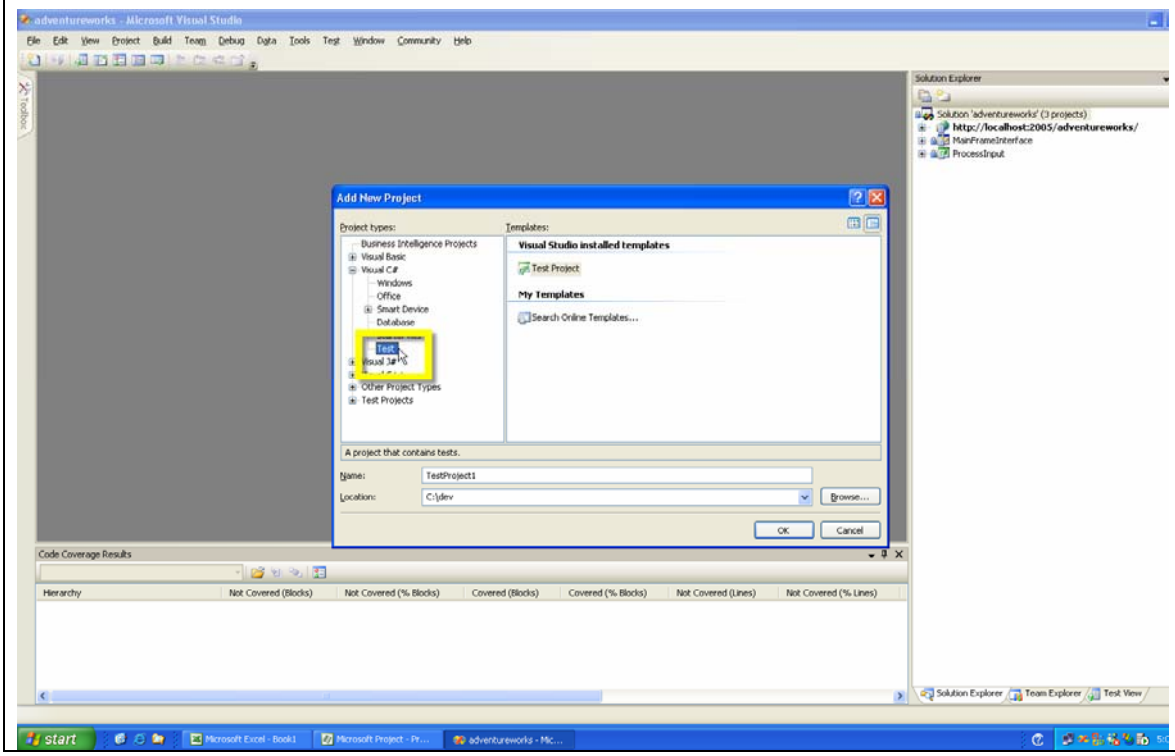
Actions	Select the **AdventureWorks** solution node and right-click choose the **Add -> New Project** menu option



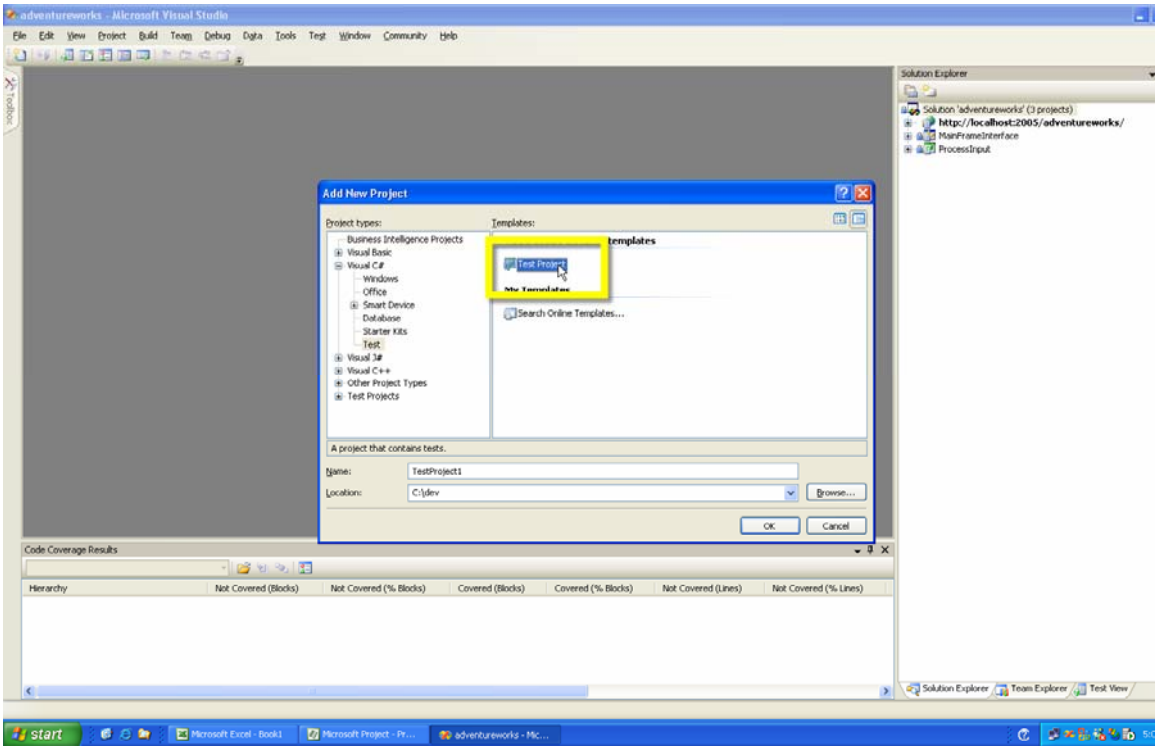
Actions	Expand the **Visual C#** node



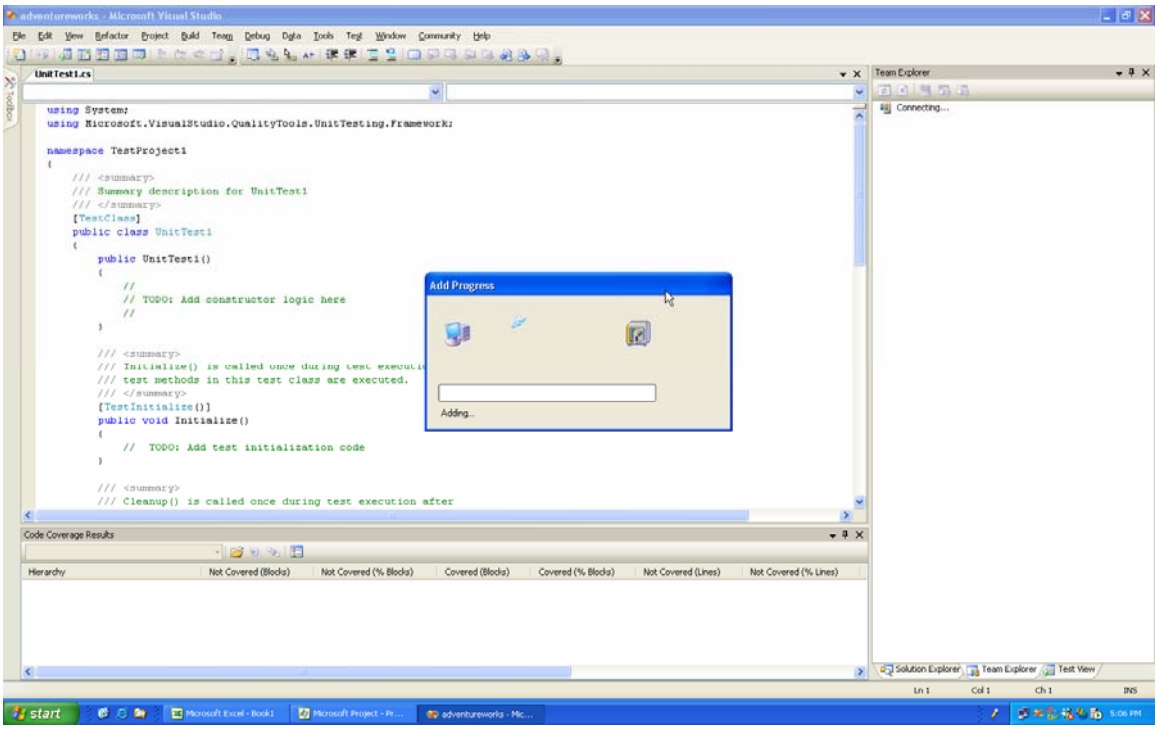
Actions	Select the **Test** node



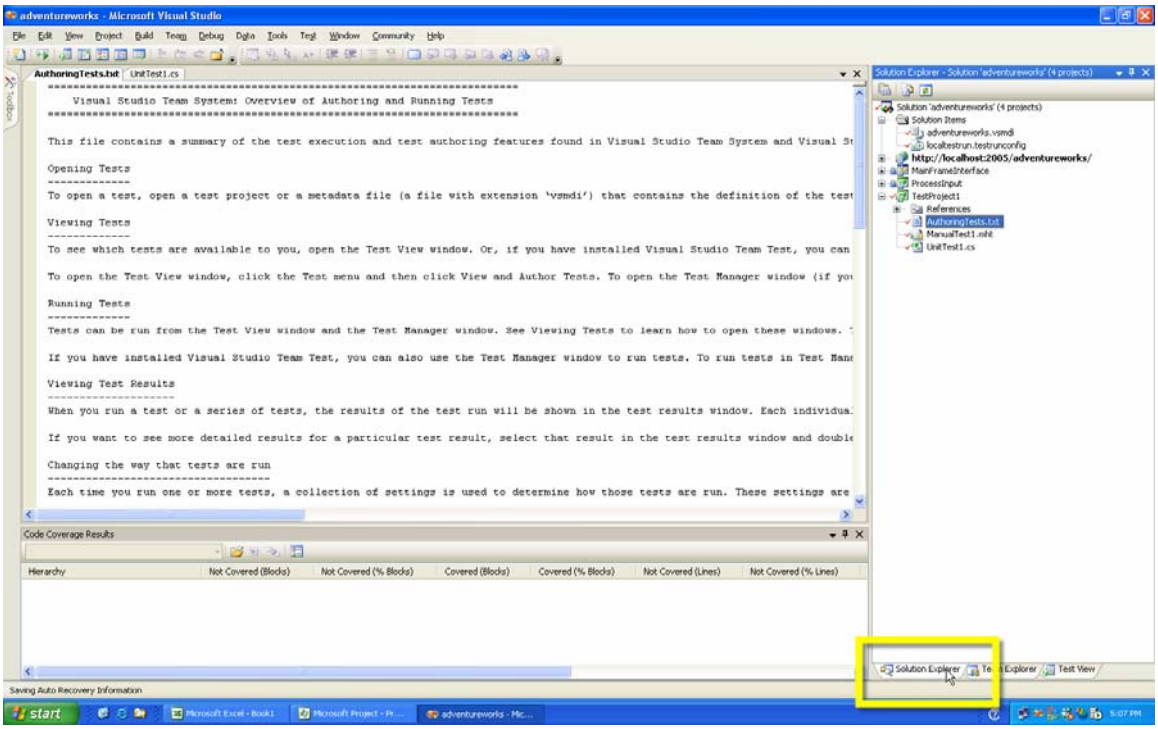
Actions	Select the **Test Project** node Press the **OK** button



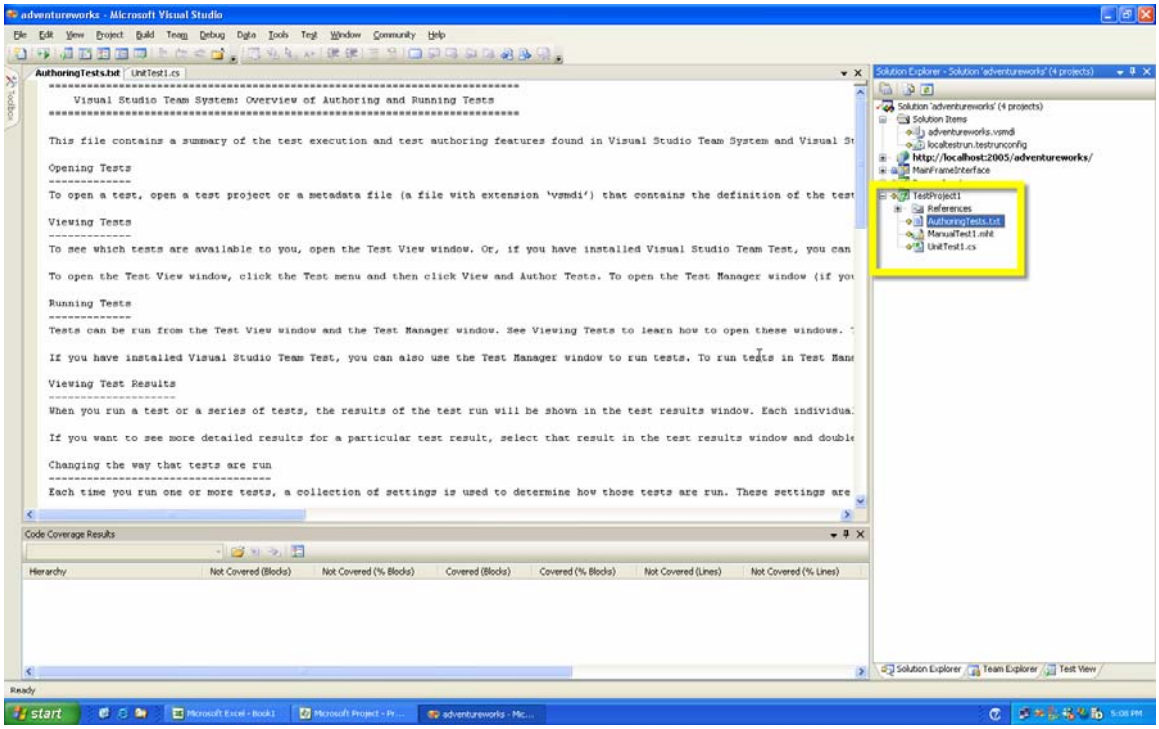
Actions	Watch as your **Test Project** is added to **Source Code Control**



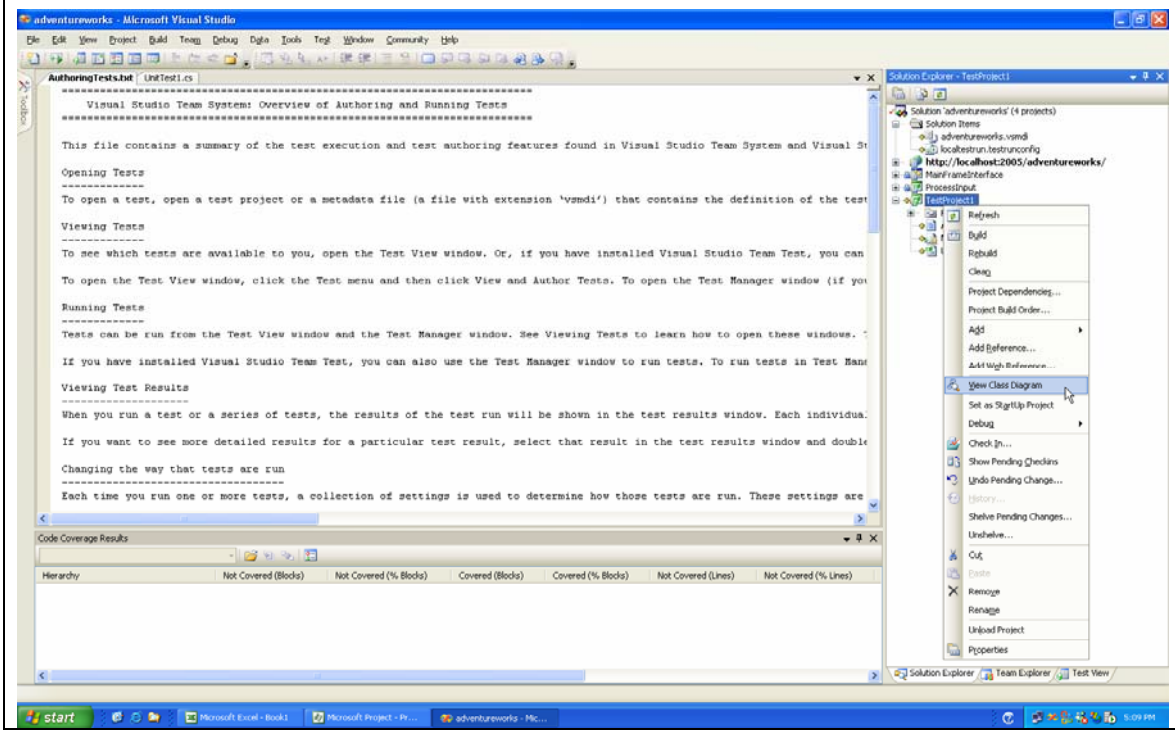
Actions	Now that our test project has been created, let's look at the test code that was generated for us. Select the **Solution Explorer** tab



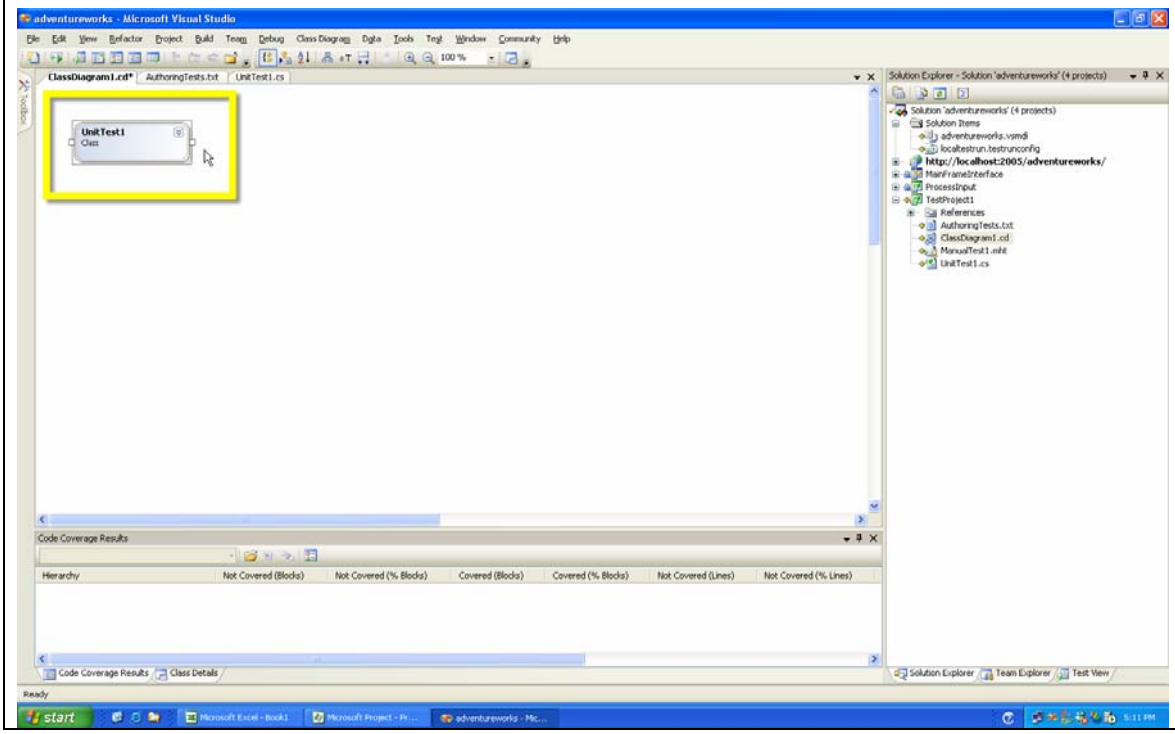
Actions	The files shown are part of your new **Test Project**



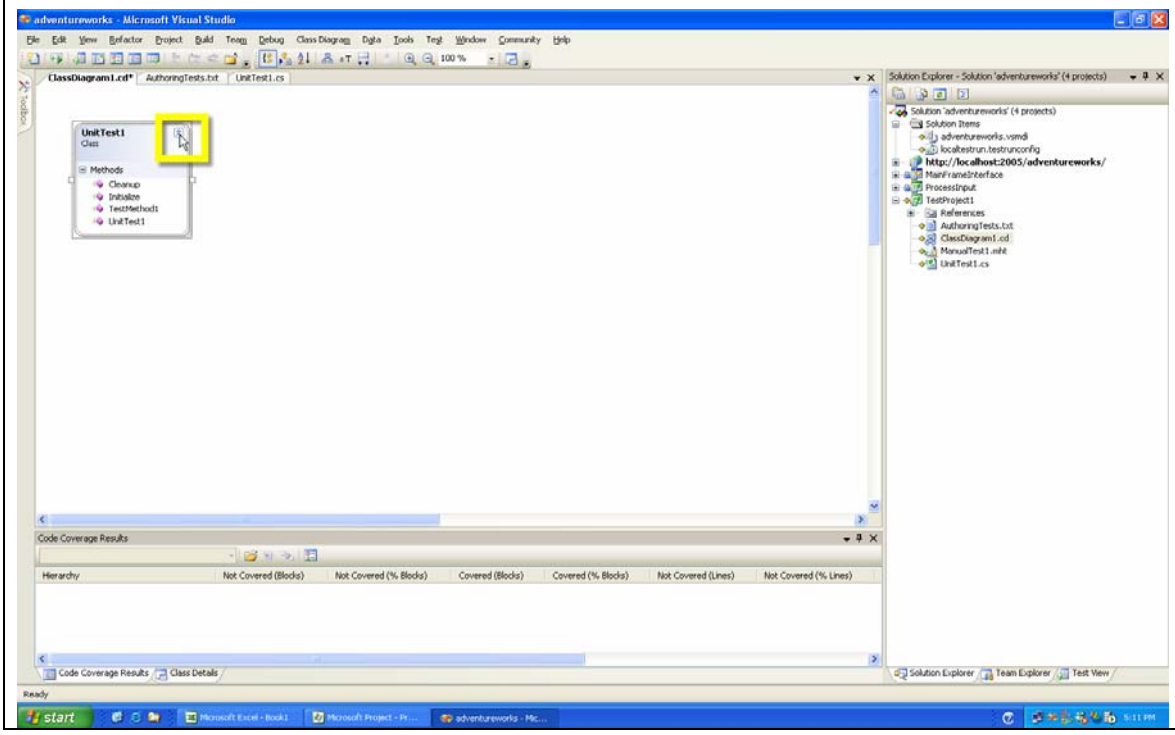
Actions	Select the **TestProject1** node and choose the **View Class Designer** menu option



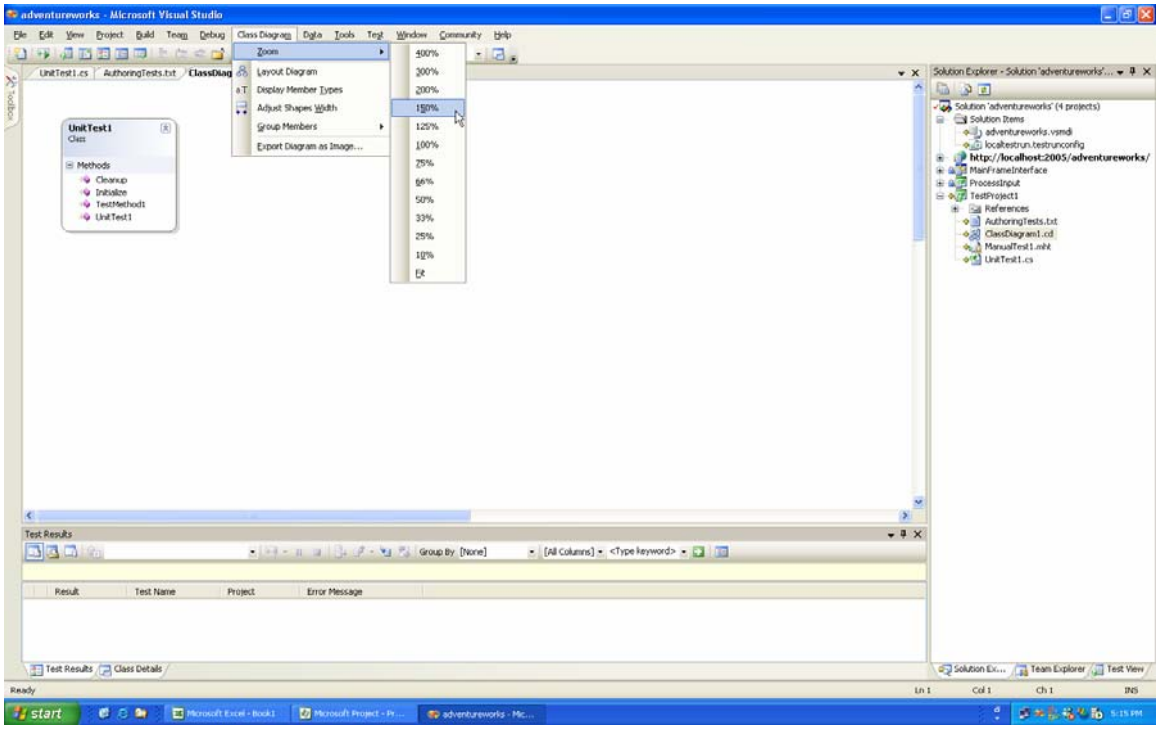
Actions	Your **UnitTest1** class should be visualized



Actions	The Class Designer allows me to visually design my class. Click on the **chevron** to show the methods in **UnitTest1**

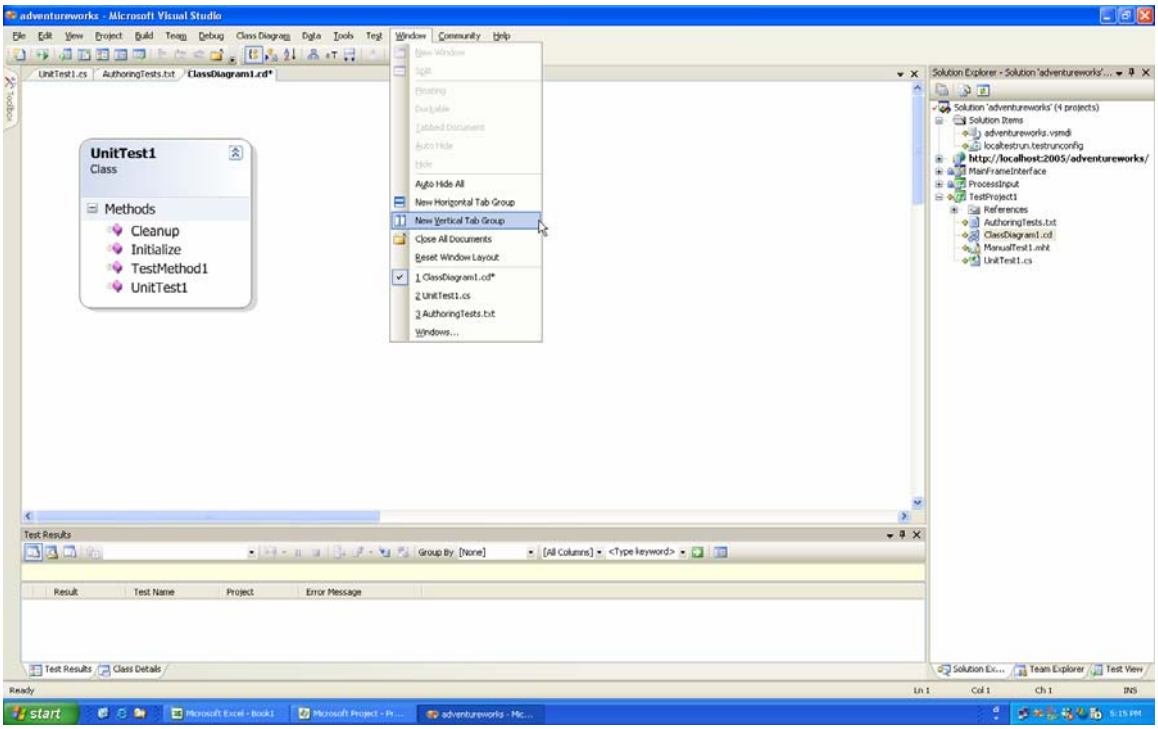


Actions	Set the **Zoom** to **150%**

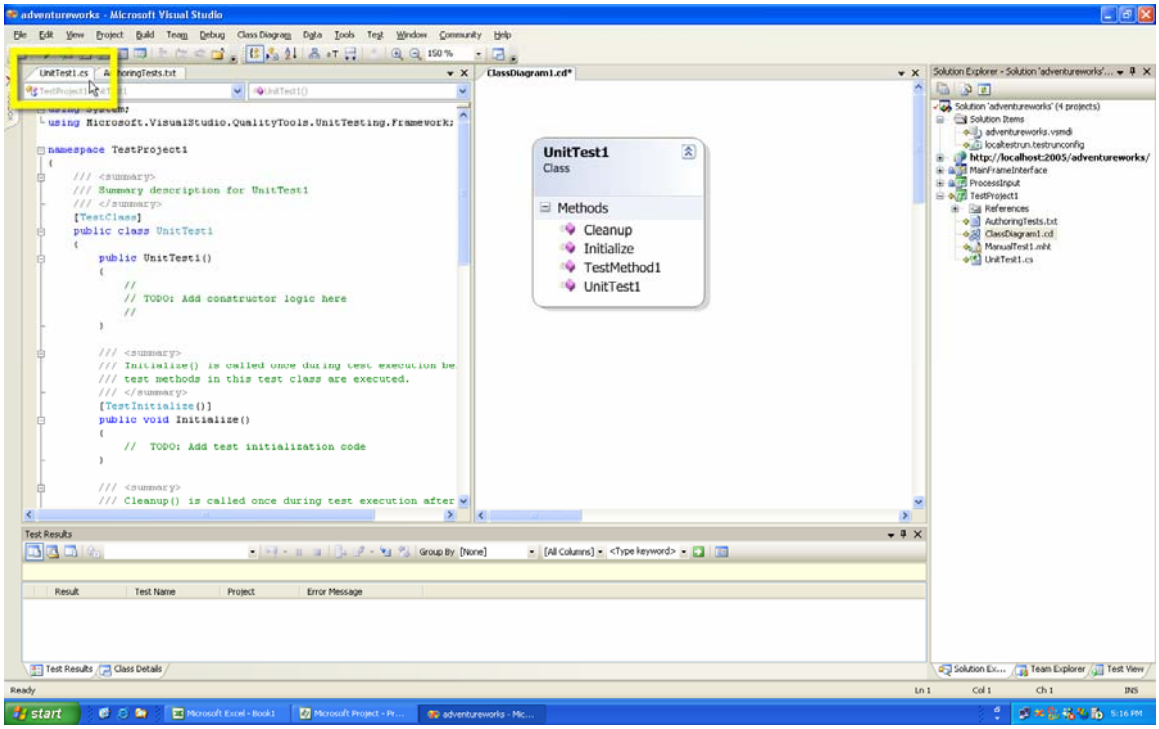


Any change I make in the Class Designer is automatically reflected in my source code. Let's look at the Class Designer and the source code it represents side-by-side.

Actions	Choose the **Window -> New Vertical Tab Group** menu option



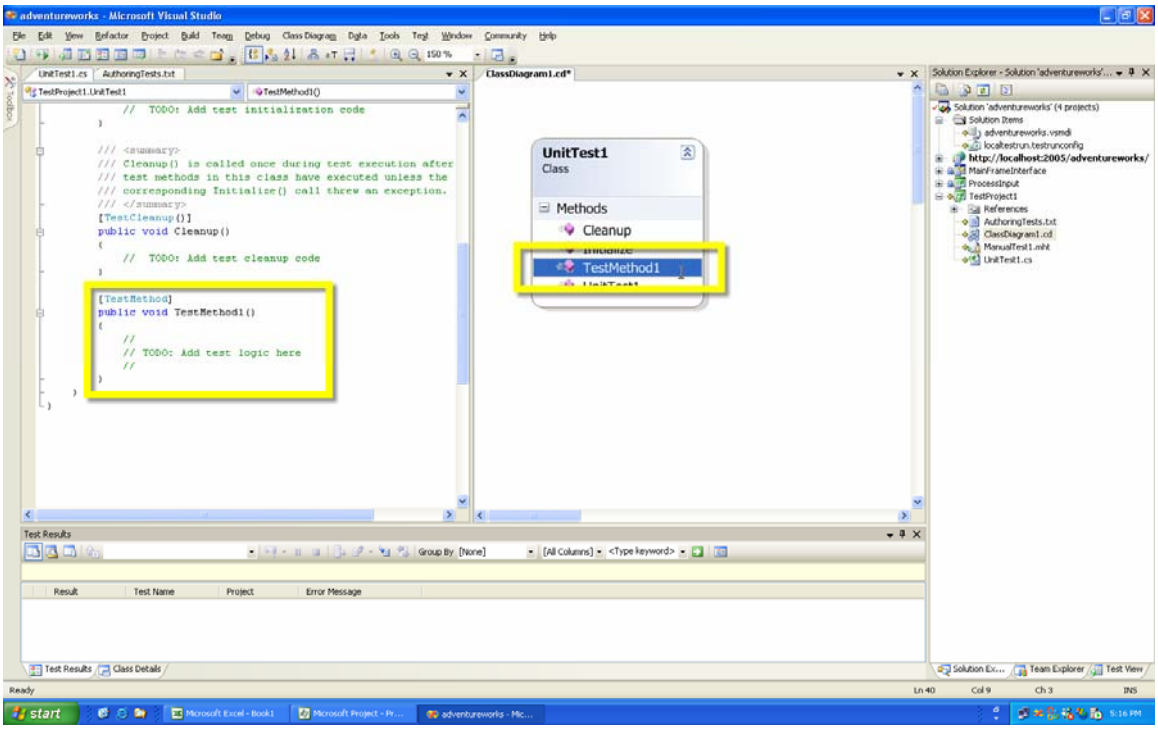
Actions	Click on the **UnitTest1.cs** tab



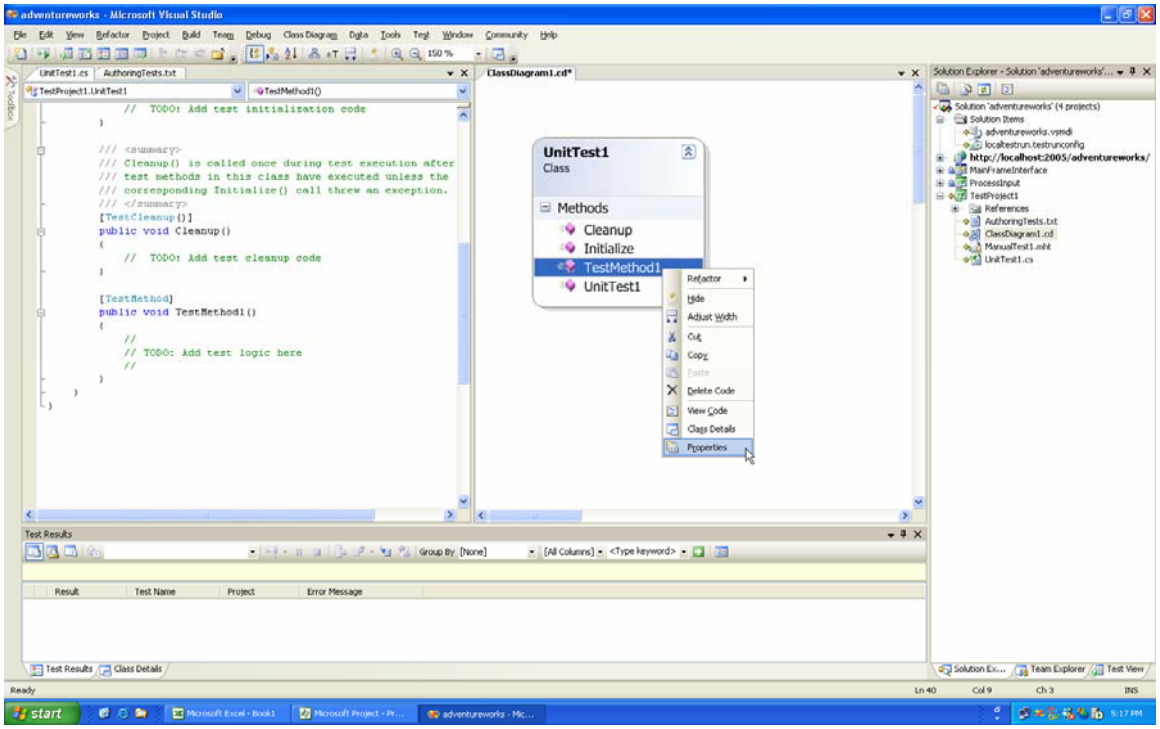
Now I have both a visual representation of my class in the Class Designer, as well as its source code. The Class Designer is really just another view onto the source code, so any change I make is automatically reflected in my source code.

The following is an example

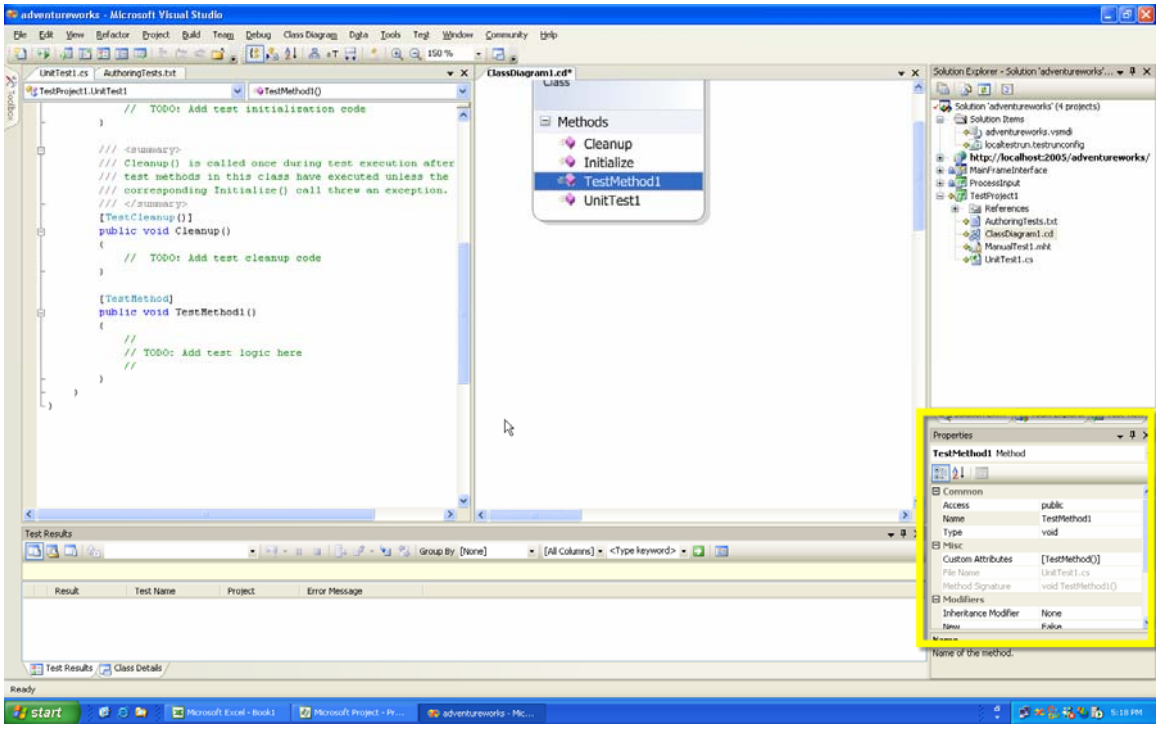
Actions	Select the **TestMethod1** method on the **UnitTest1** class diagram



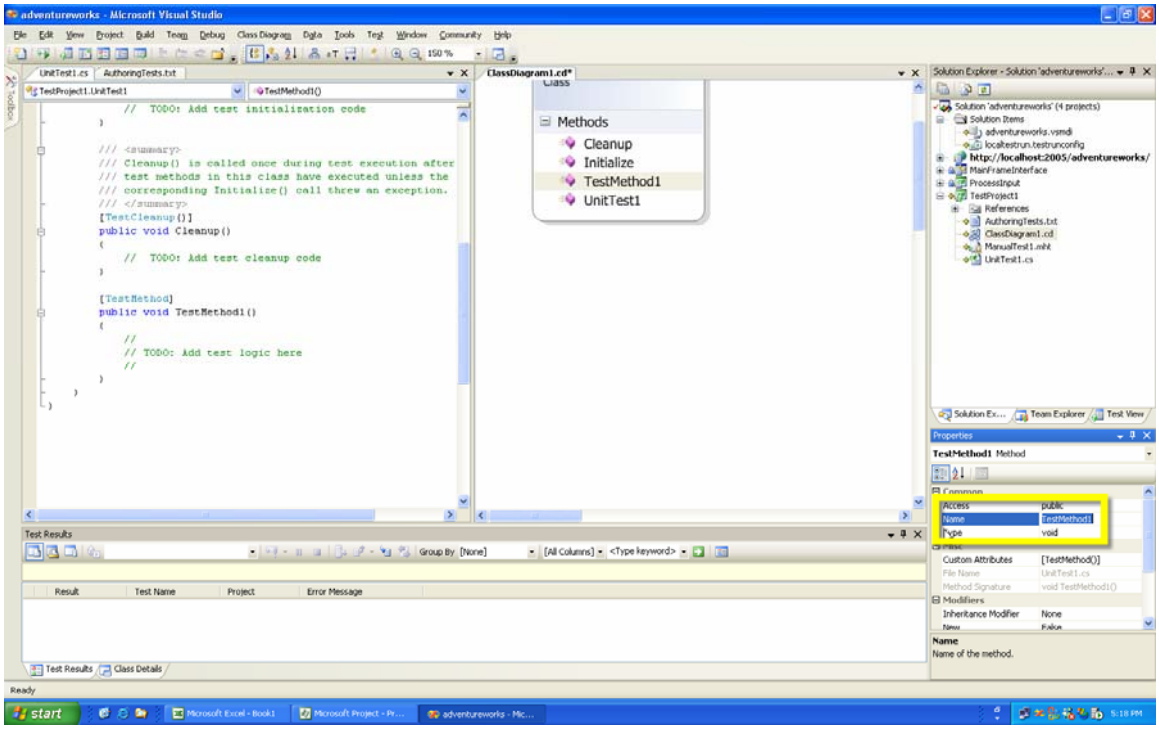
Actions	With the **TestMethod1** method selected, right-click and choose **Properties**



Actions	The **Property Grid** will be visible, as shown

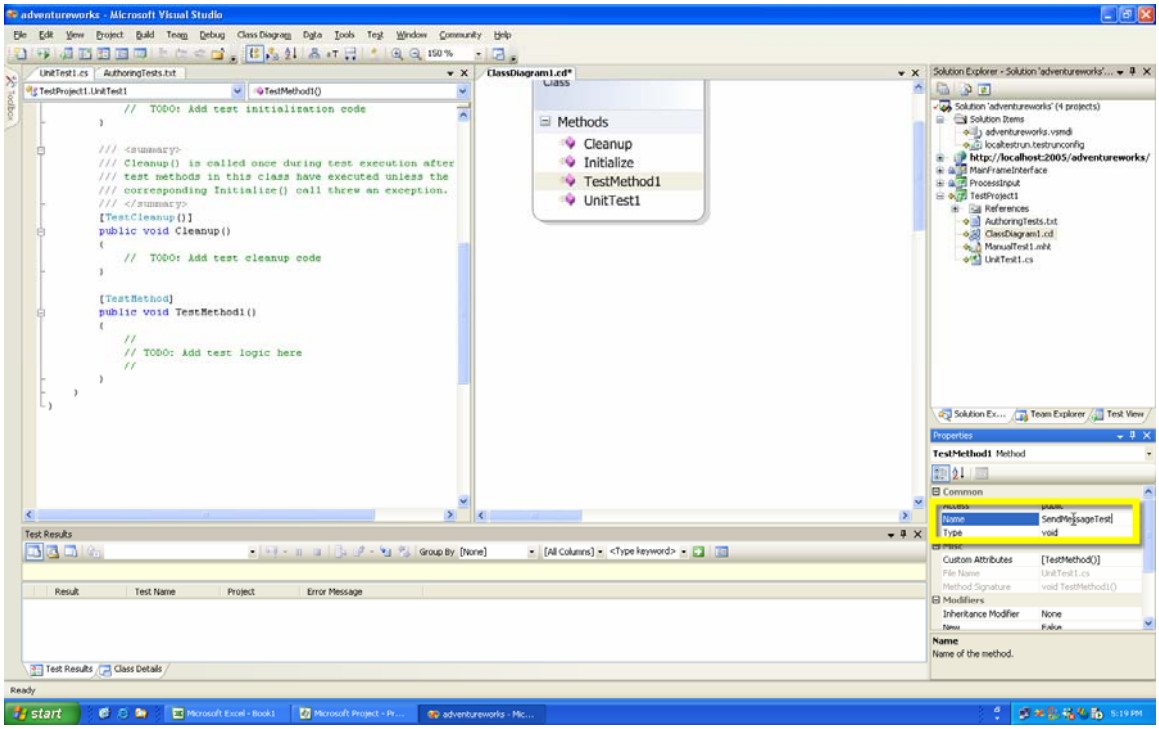


Actions	Select the **Name** property



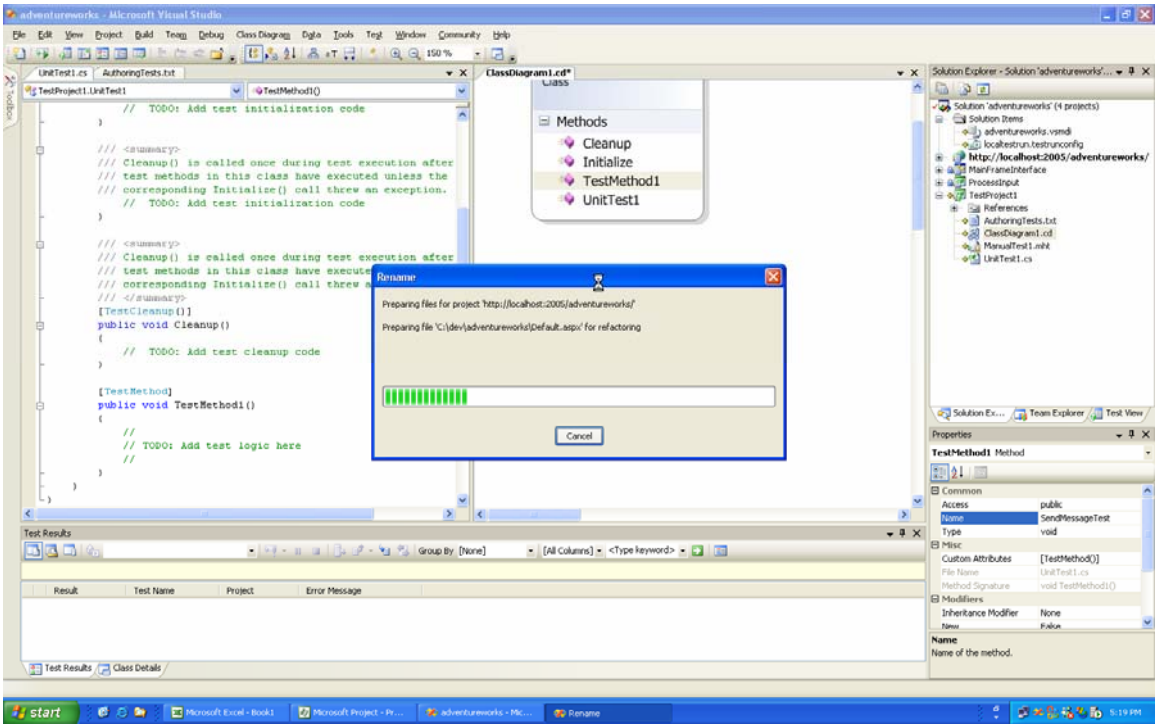
When we change the name of our method in the Class Designer, the source code is automatically updated.

Actions	Change this property to **SendMessageTest**

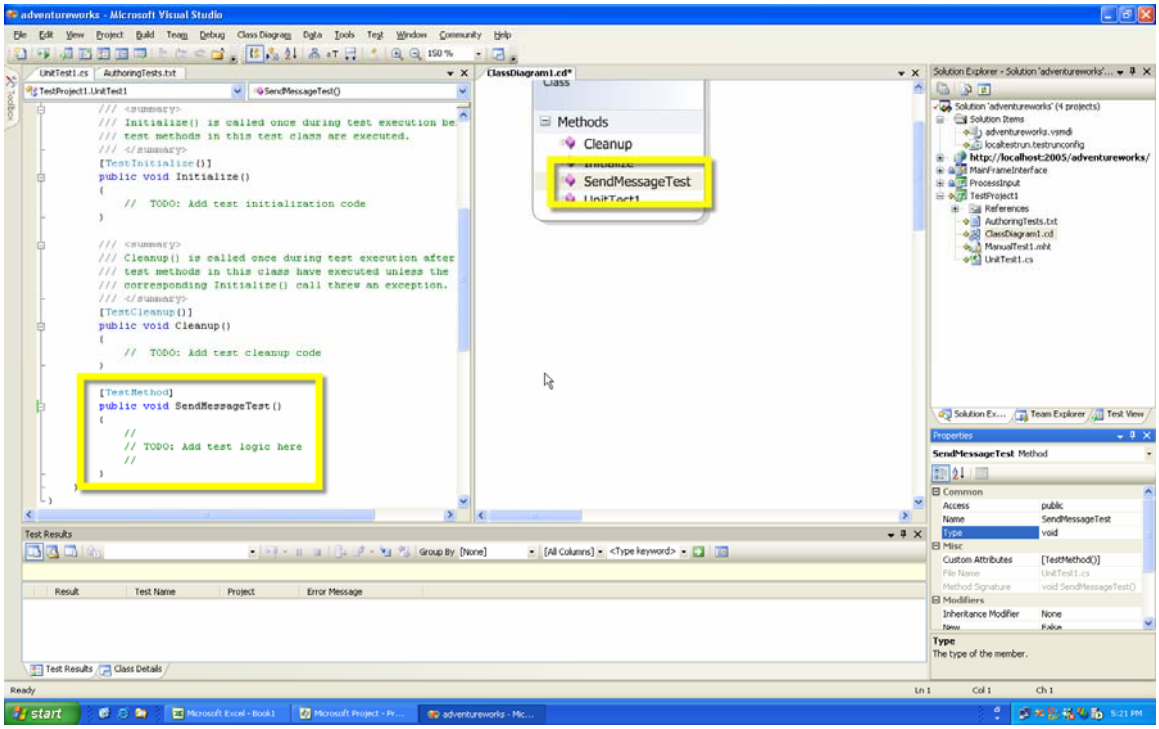


Any reference I made to this class by its old name will be updated; this combination of visual class design and refactoring let's me quickly get up to speed on existing code bases and get productive right away.

Actions	**Visual Studio 2005 Team System** will refactor all the references to this method to reflect the new name

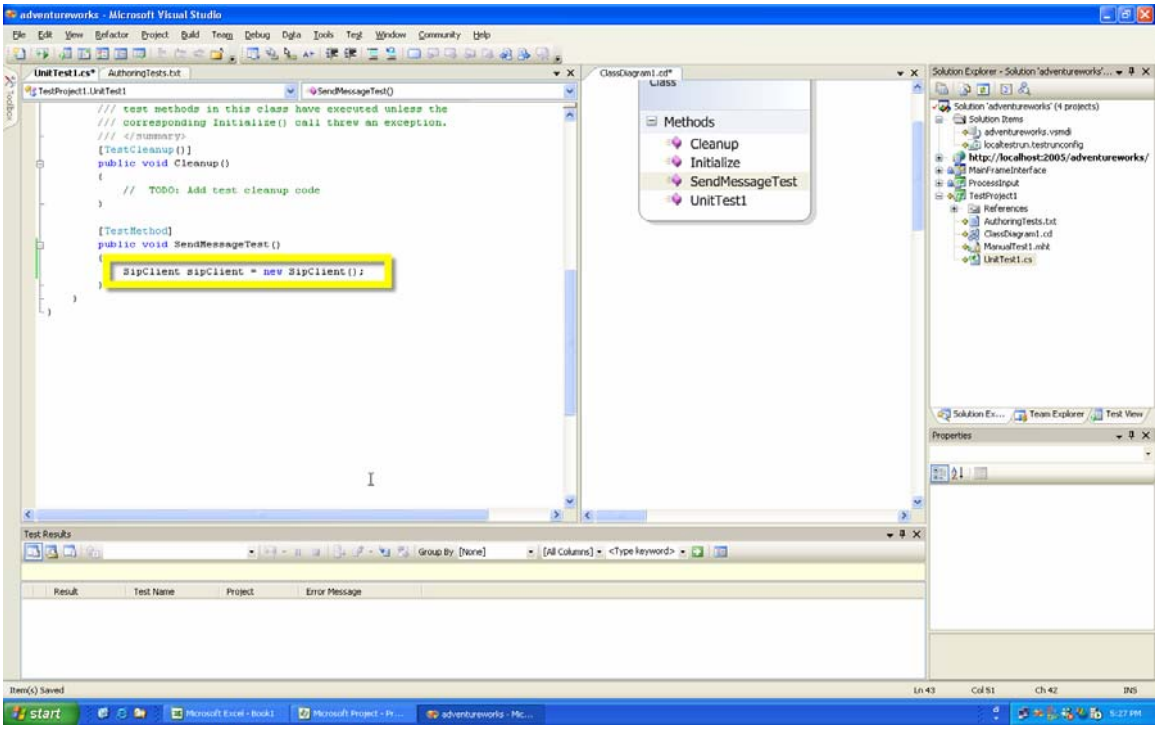


Actions	Notice the changes in the **Class Designer** and **source code**



The first step is Test Driven Development (TDD) is to write a basic unit test that will test the code that we *intend* to write. In this case, we will write the code to instantiate a class called *SipClient*. This class will encapsulate the functionality that we need to send IM messages.

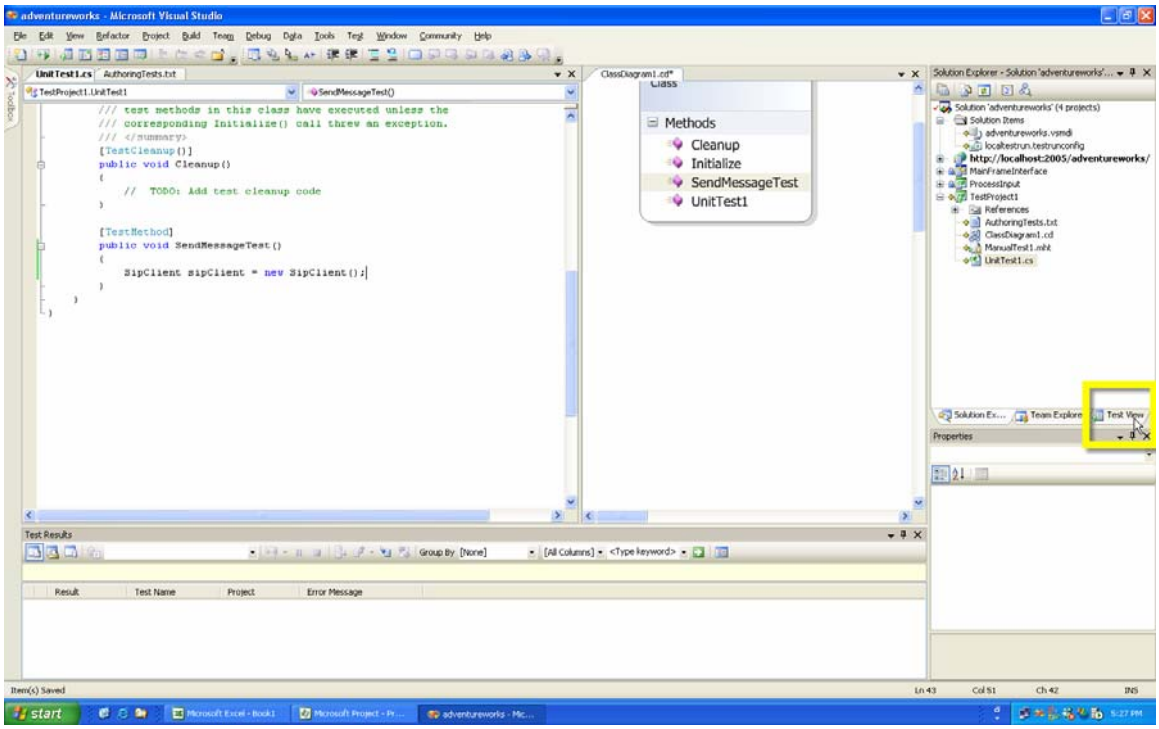
Actions	Insert the following lines of code into **UnitTest1.cs**: `Notification.SipClient sipClient = new Notification.SipClient()`



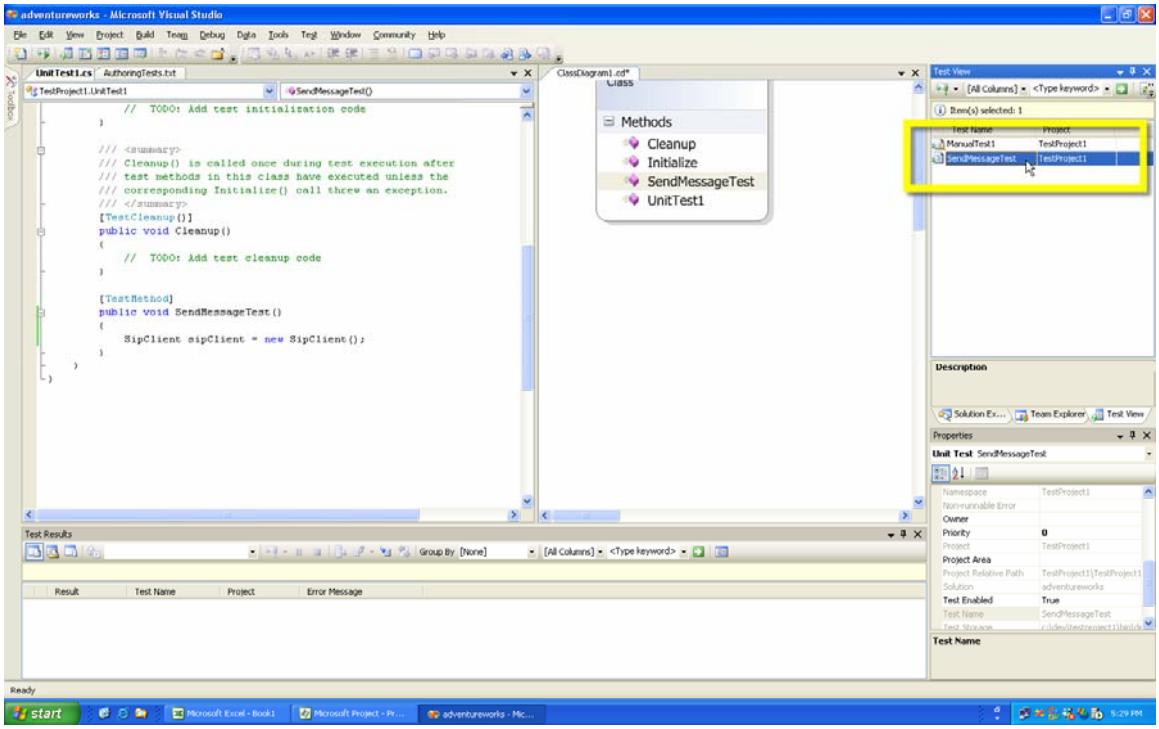
Now we are going to run our basic unit test – we know that this test won't compile because our *SipClient* class does not exist yet.

One of the tenets behind TDD is to write your test, have it fail, and then write just enough code to allow it to pass. As you iterate over this cycle, you start to build the functionality behind your class.

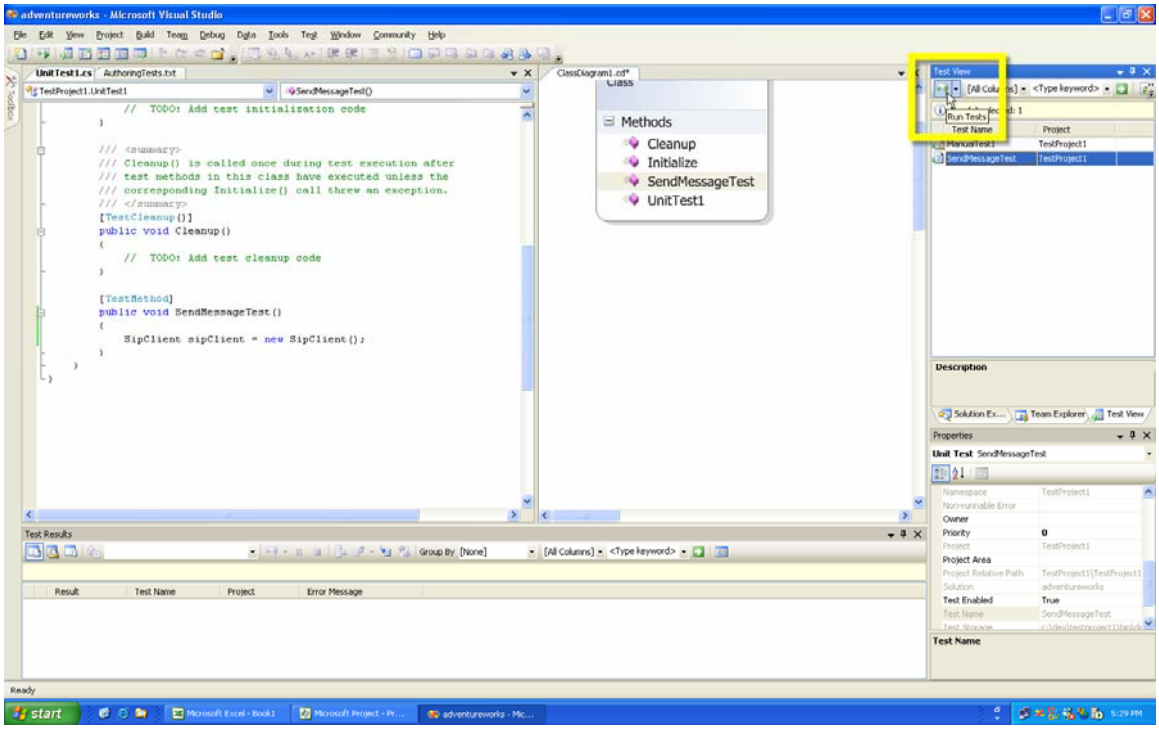
Actions	Click on the **Test View** as shown



Actions	Select the **SendMessageTest** unit test

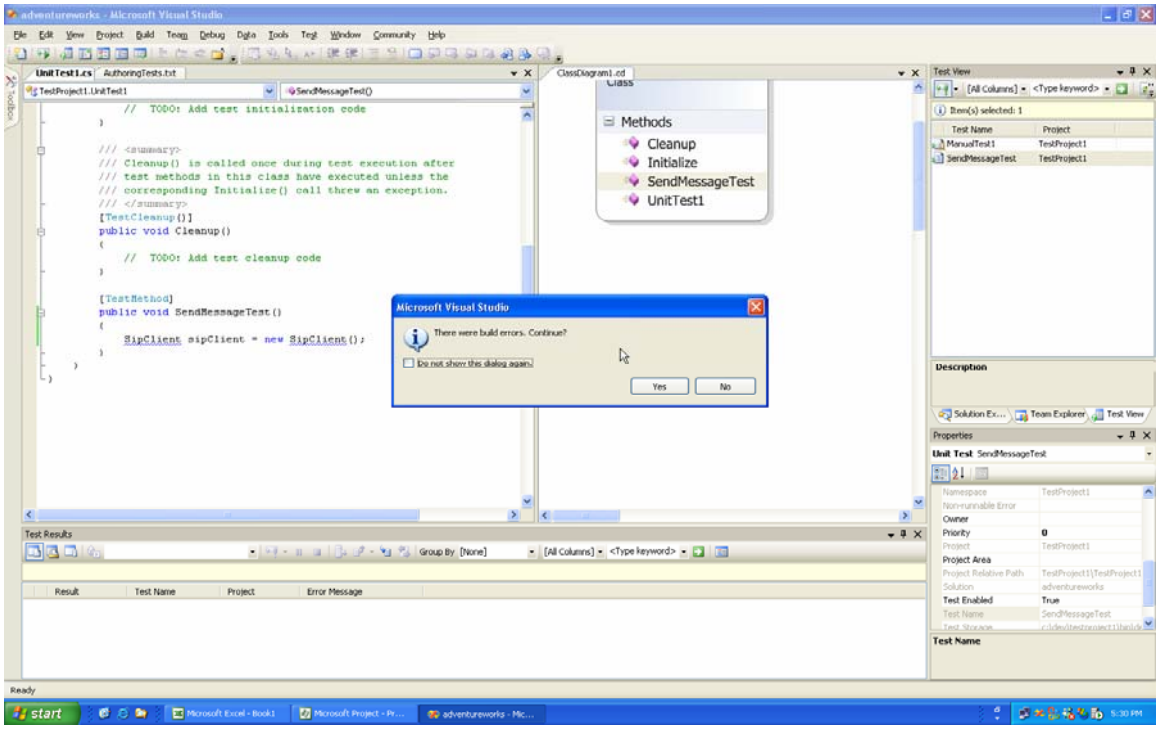


Actions	Press the **Run Tests** tool bar button as shown

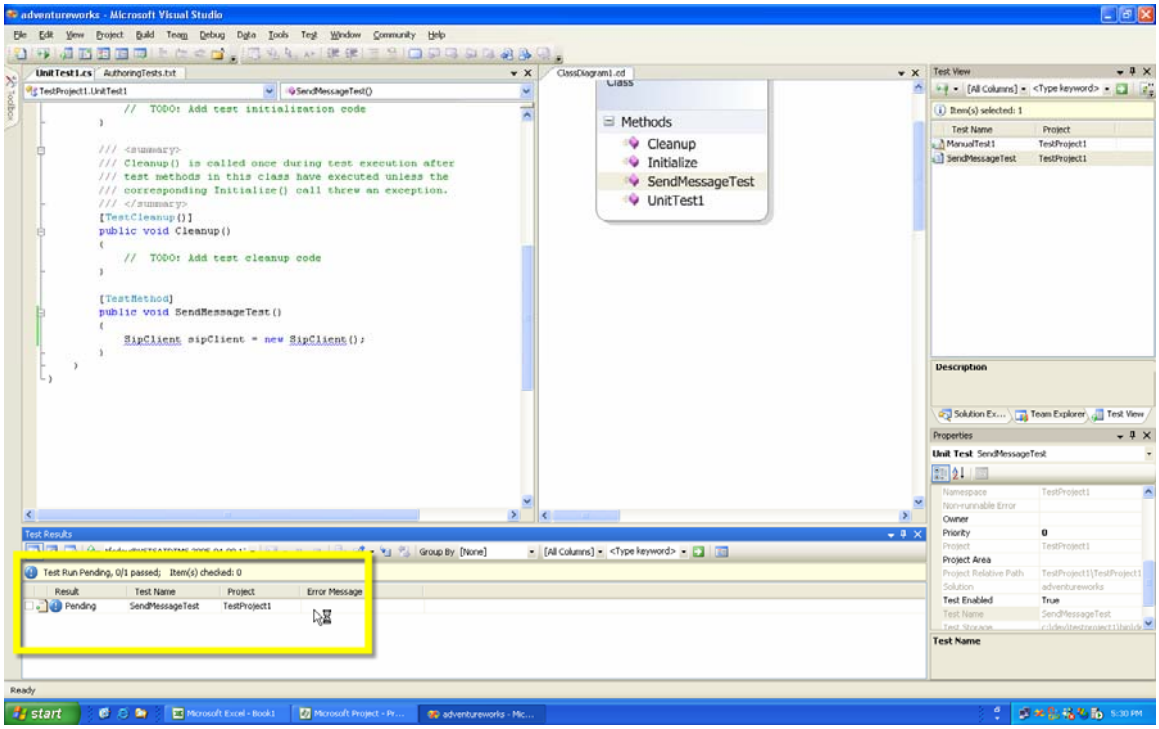


Our test will fail as expected since it does not compile – we have not written our *SipClient* class yet.

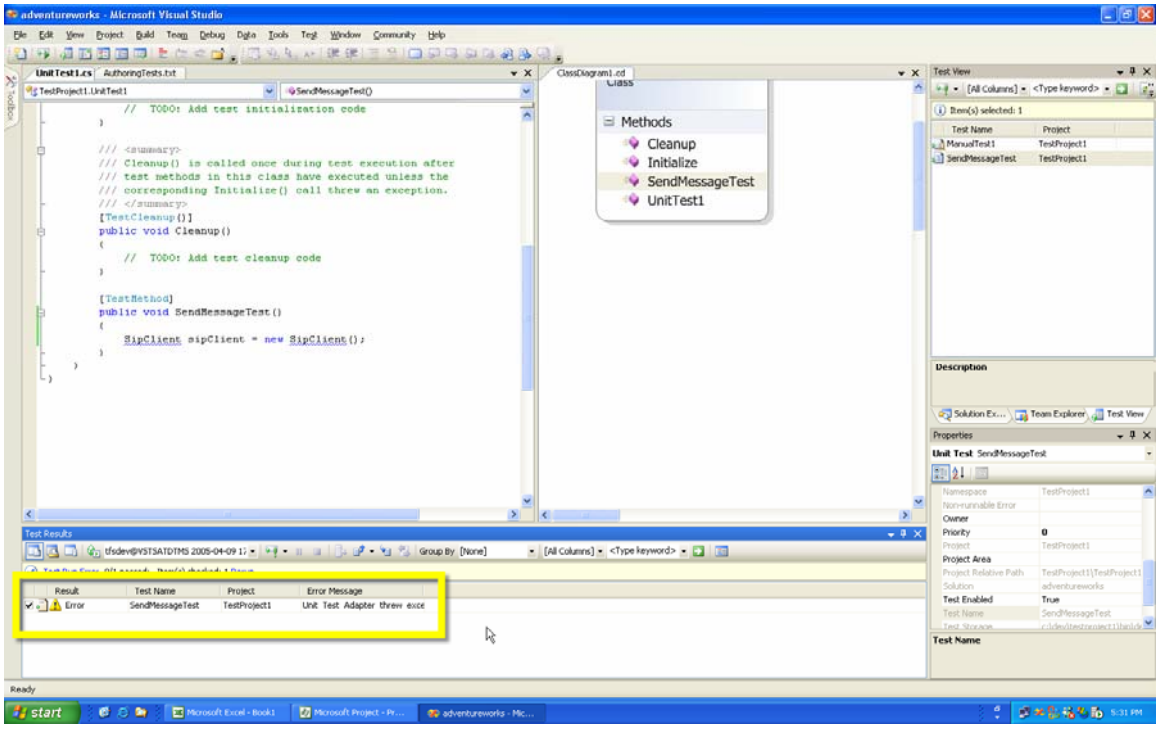
Actions	Press the **No** button in this dialog



Actions	Watch as your test progresses

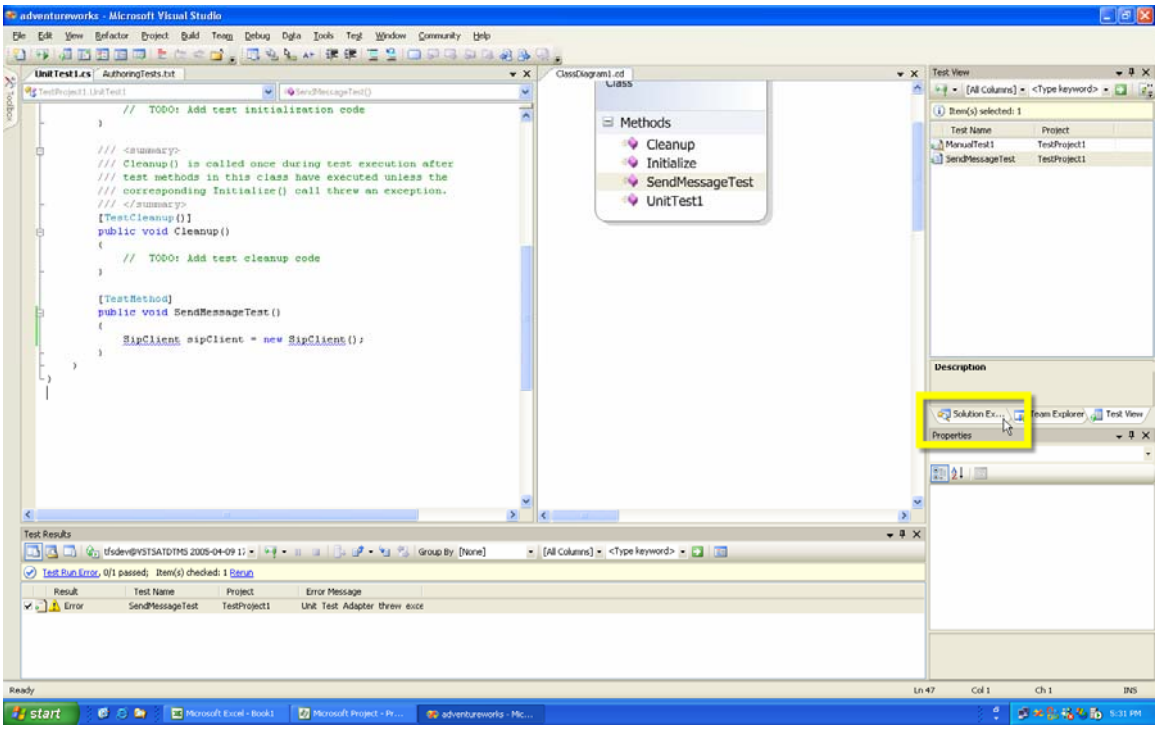


Actions	It will fail as expected

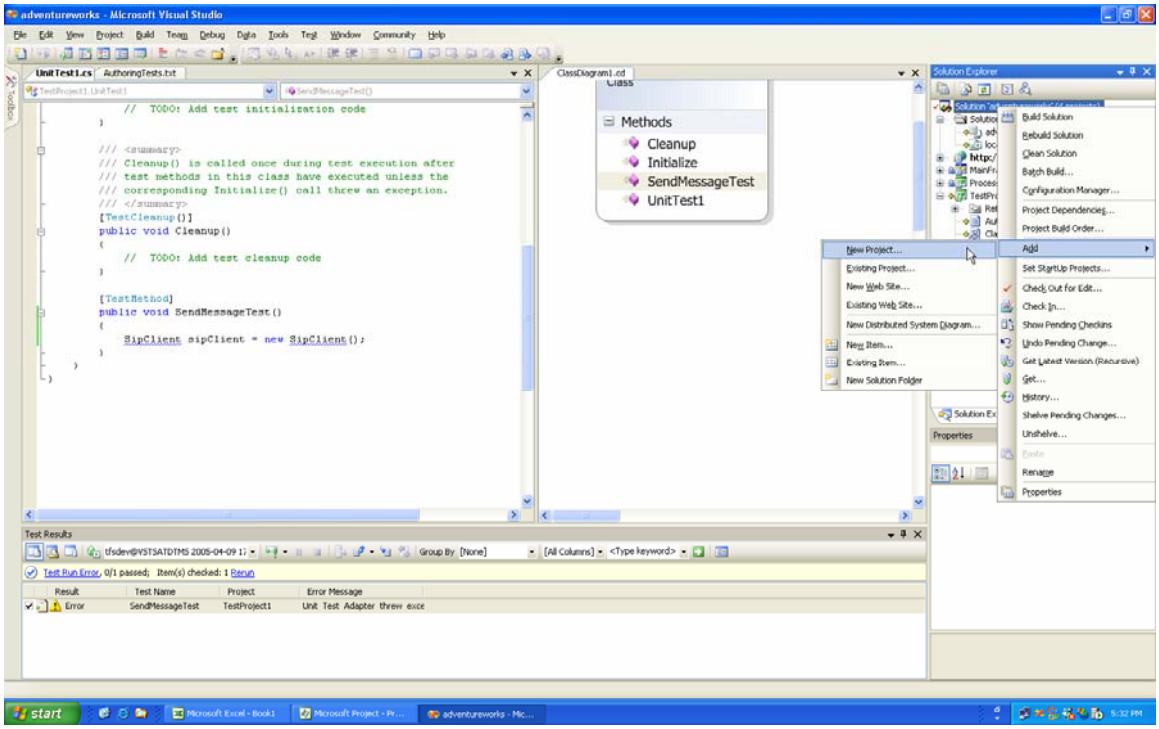


Now we are going to write just enough code to allow our unit test to pass. That means creating the new project that will house our SipClient class.

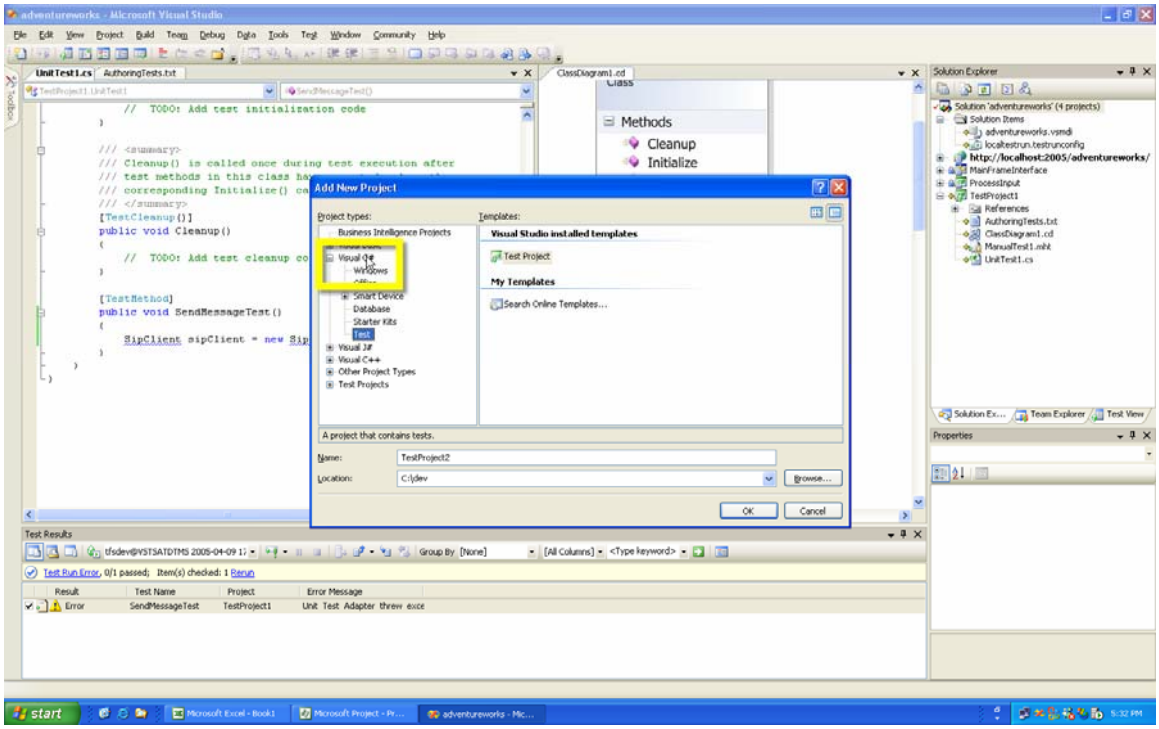
Actions	Click on the **Solution Explorer** tab



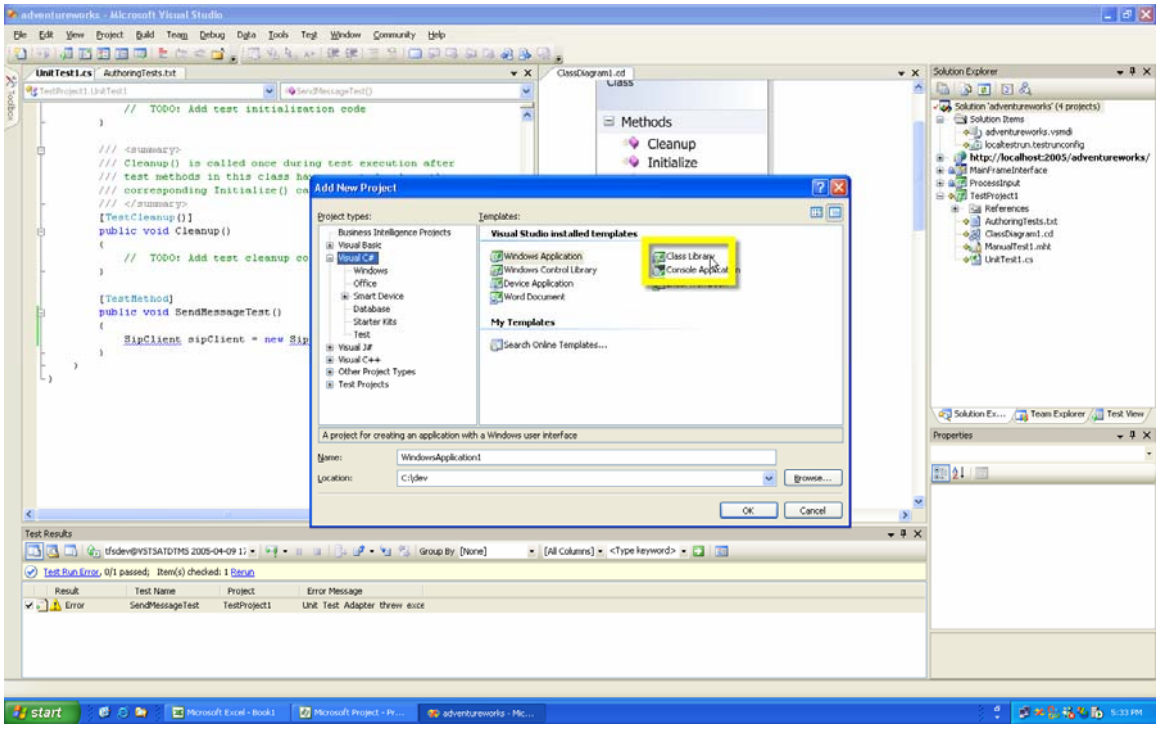
Actions	Right-click on the **Solution** node and choose **Add -> New Project**



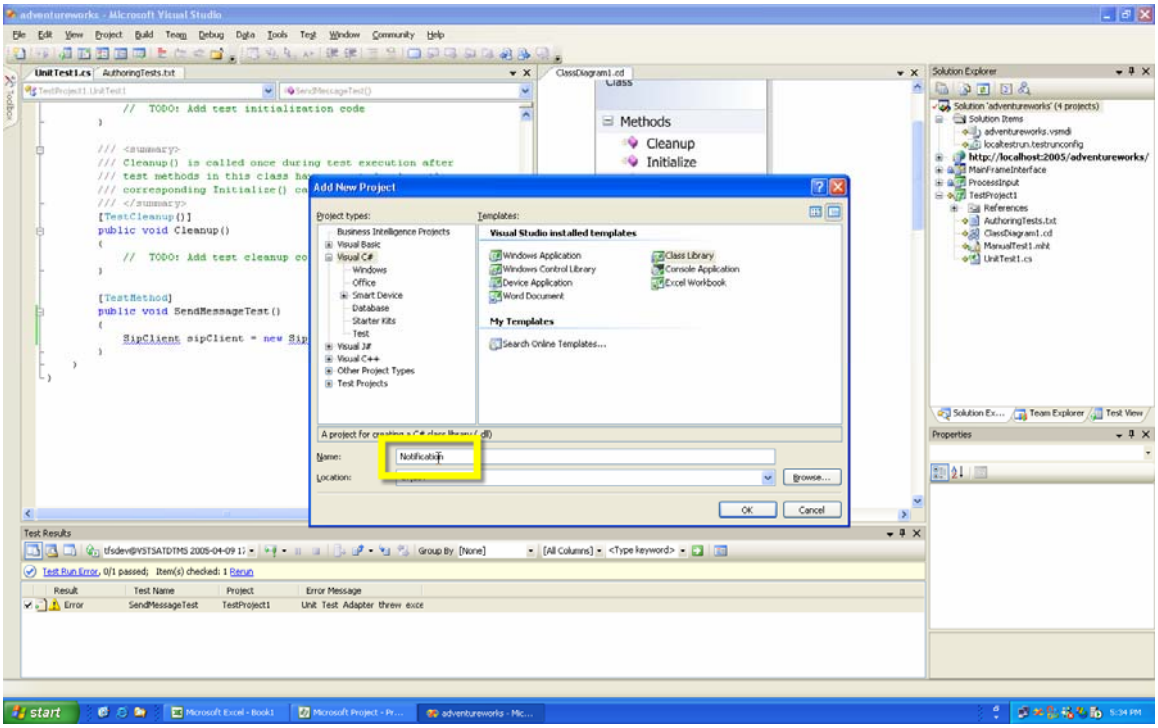
Actions	Select the **Visual C# node**



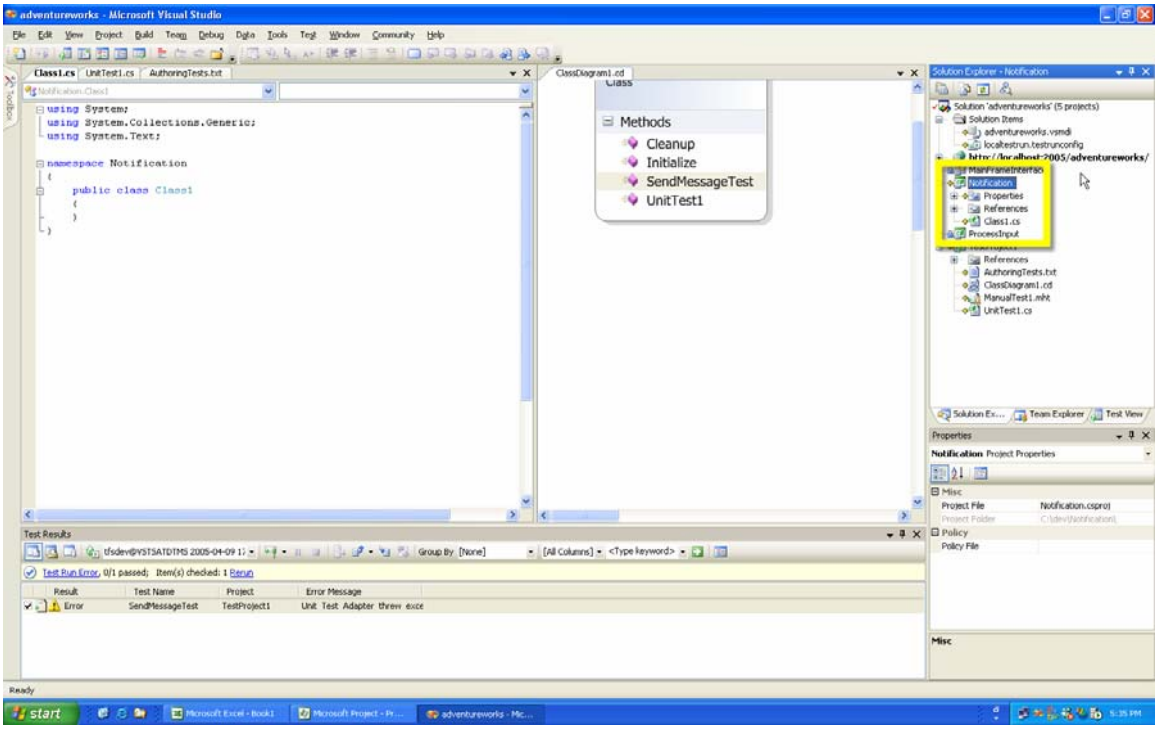
Actions	Select the **Class Library** project type



Actions	Name your new project **Notification** Press the **OK** button



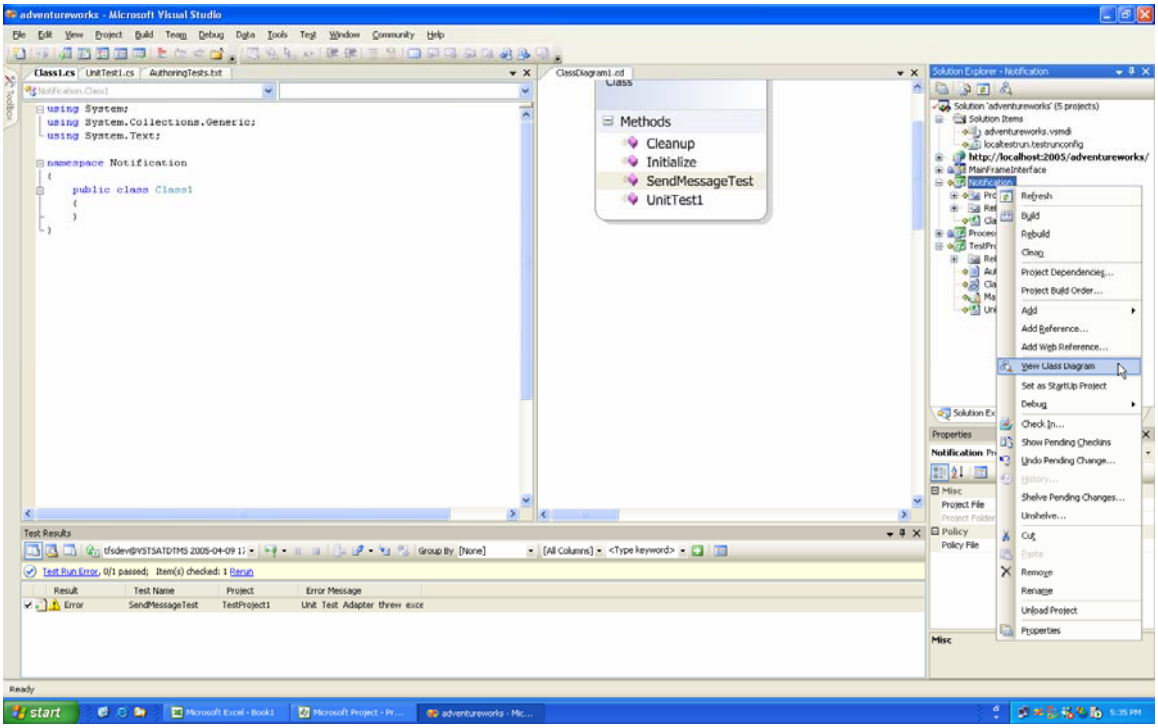
Actions	Notice the **Notification** project that has been added to your solution



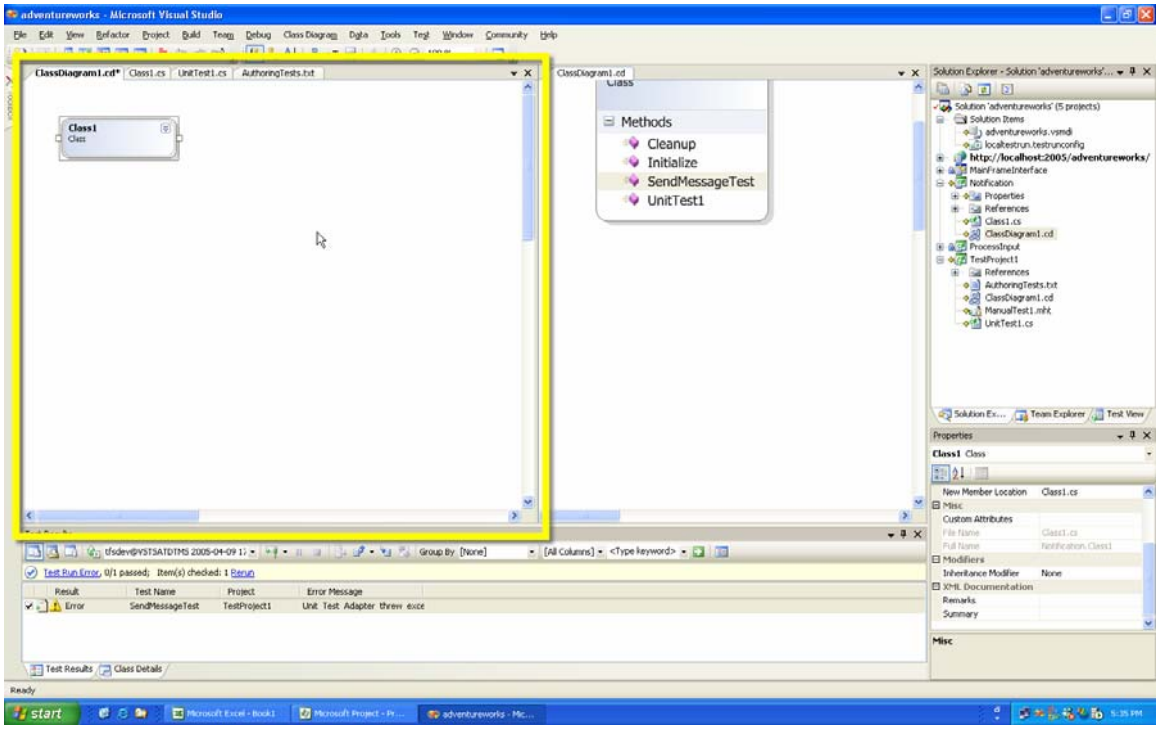
We will use the Class Designer again to create our SipClient class. In this case, we will just rename the default class that was created for us to SipClient.

Once our class is created, we will add a reference from our new project to our test project. This will allow our unit test to pass. This will be our first iteration of the TDD approach.

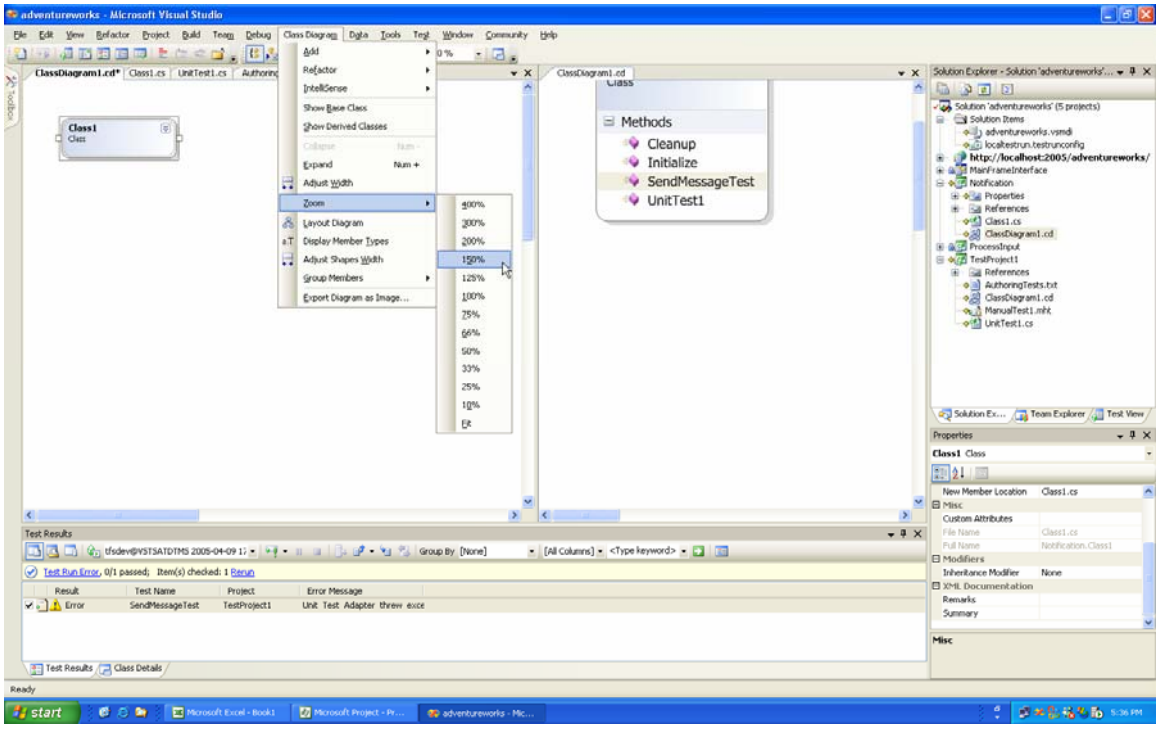
Actions	Right-click on this project node and choose the **View Class Diagram** menu option



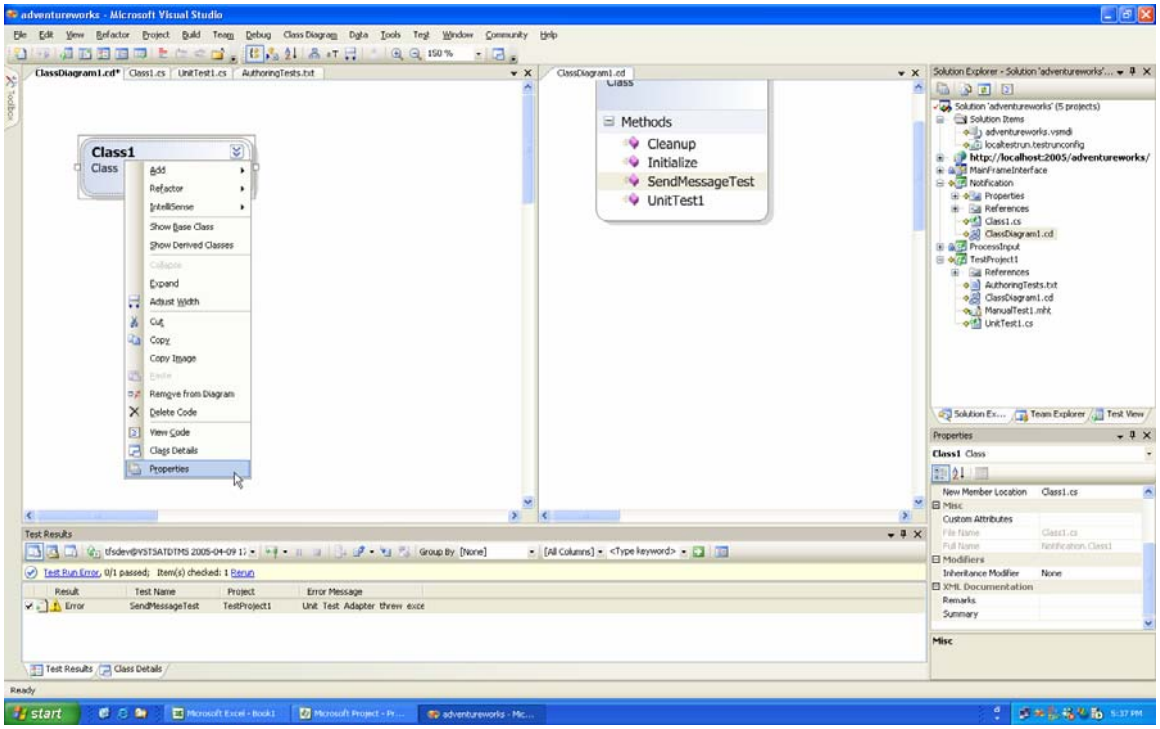
Actions	Your new **Class Diagram** should be side-by-side with your previous one



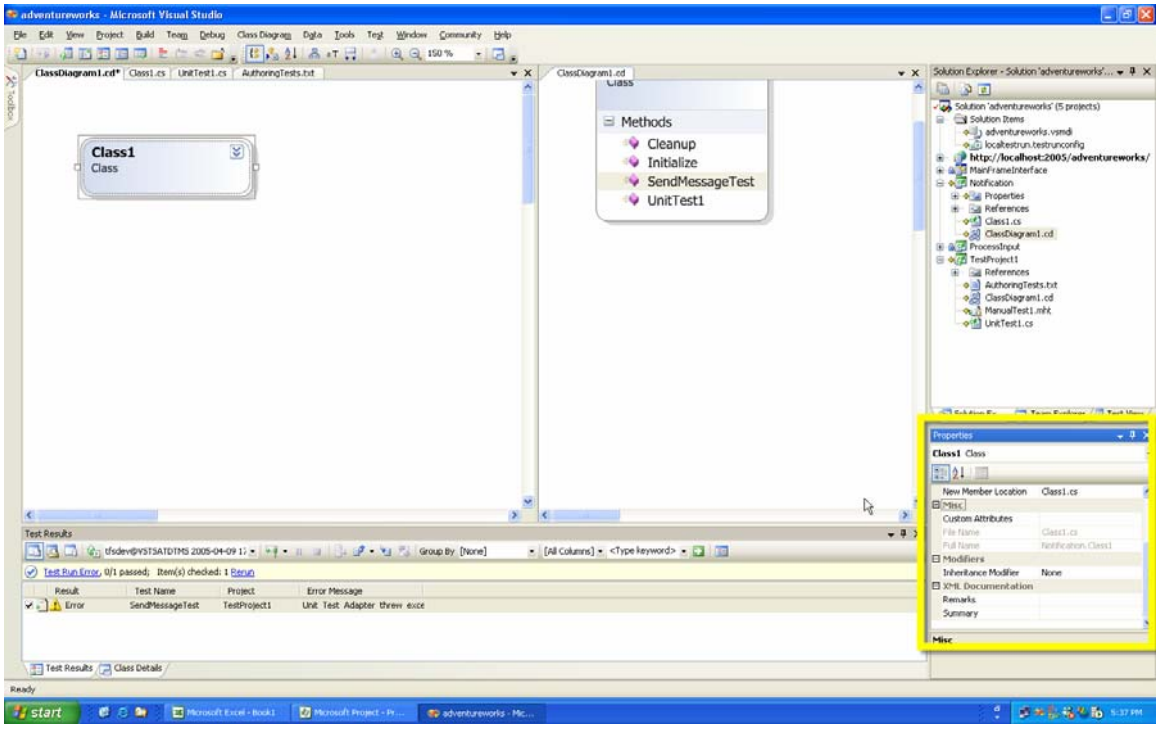
Actions	Set the **Zoom** to **150%**



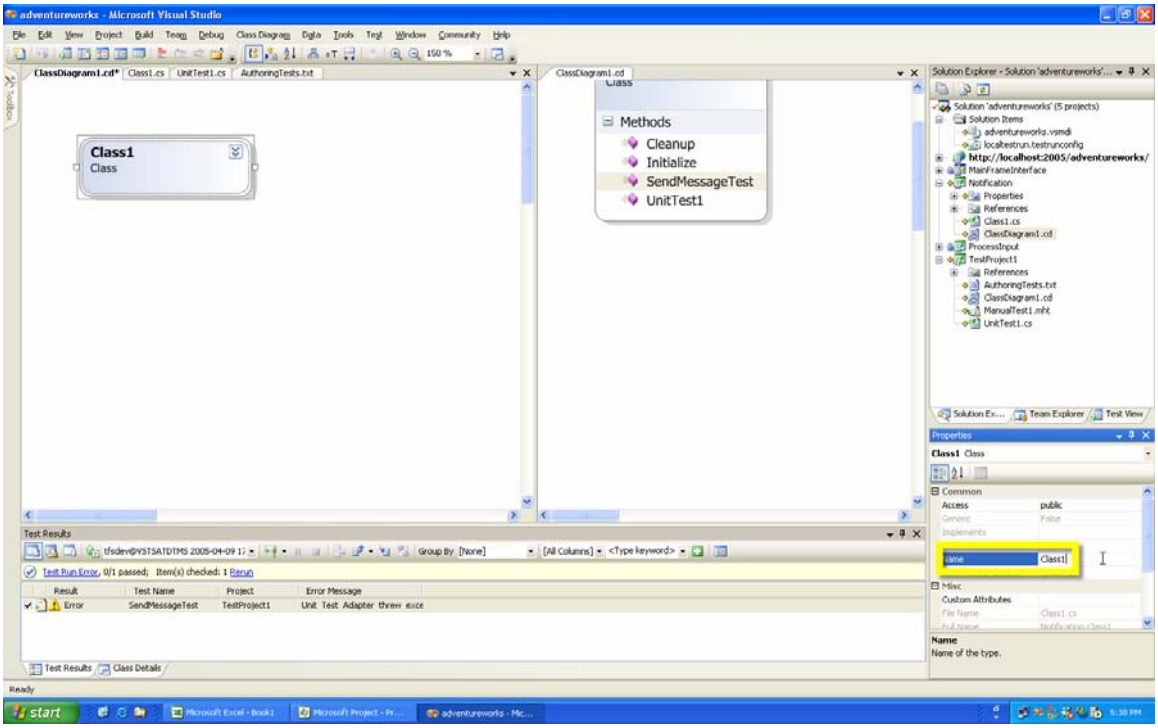
Actions	Right-click on **Class1** and choose **Properties**



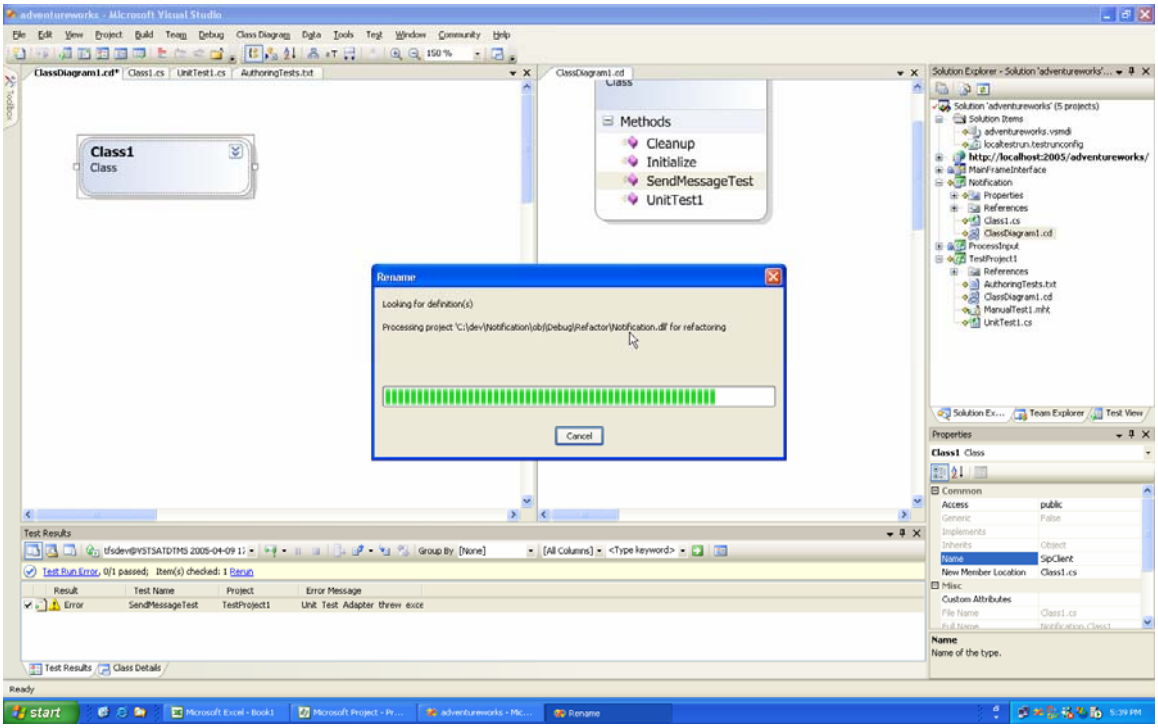
Actions	The Property Grid will be visible, as shown



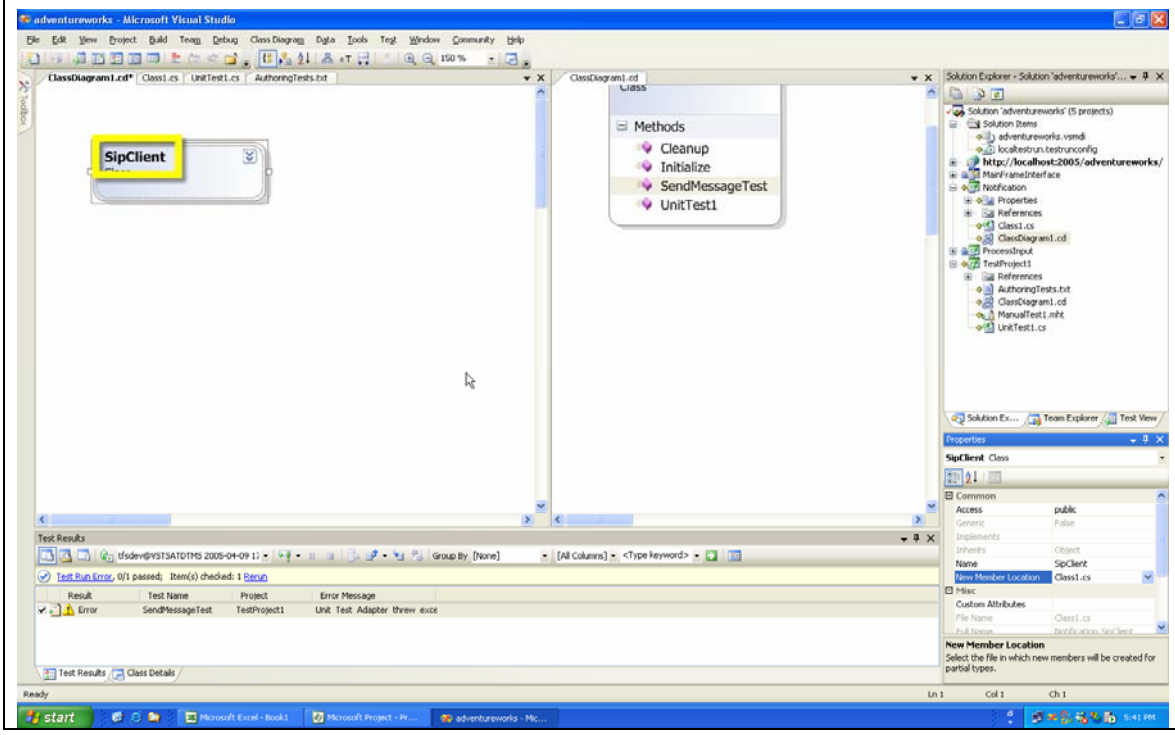
Actions	Scroll down to the **Name** property and select it Change the name to **SipClient**



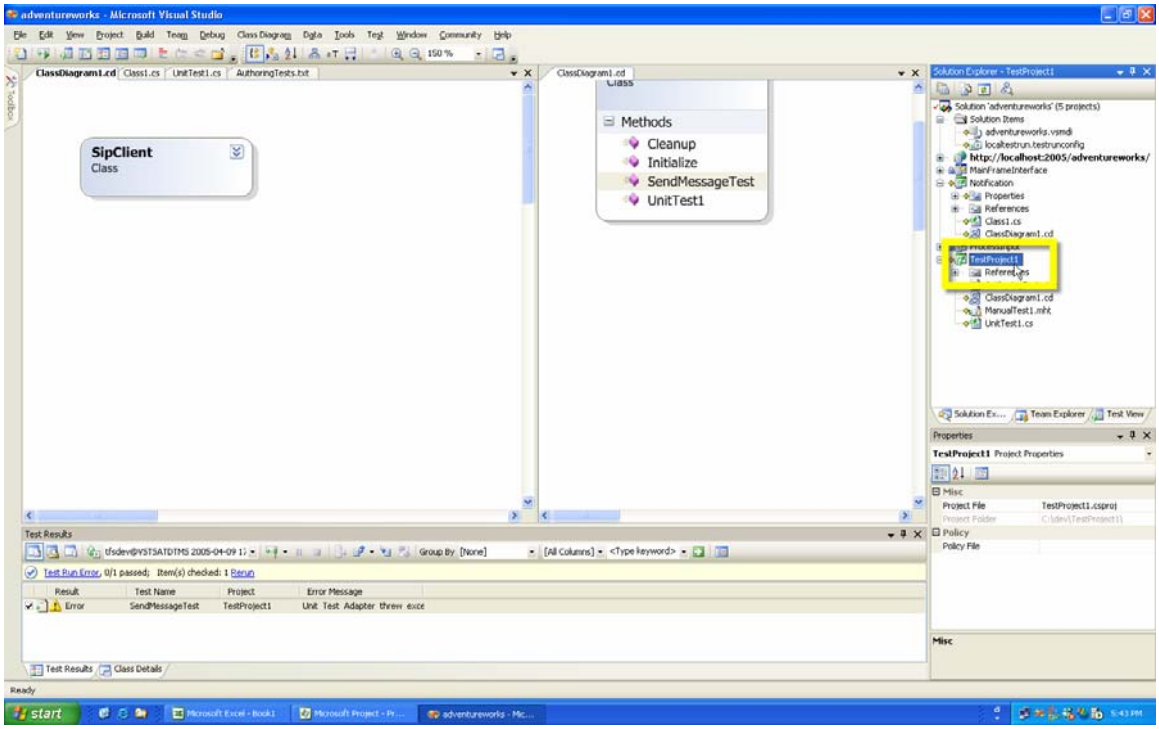
Actions	Visual Studio 2005 Team System will refactor all the references to this method to reflect the new name



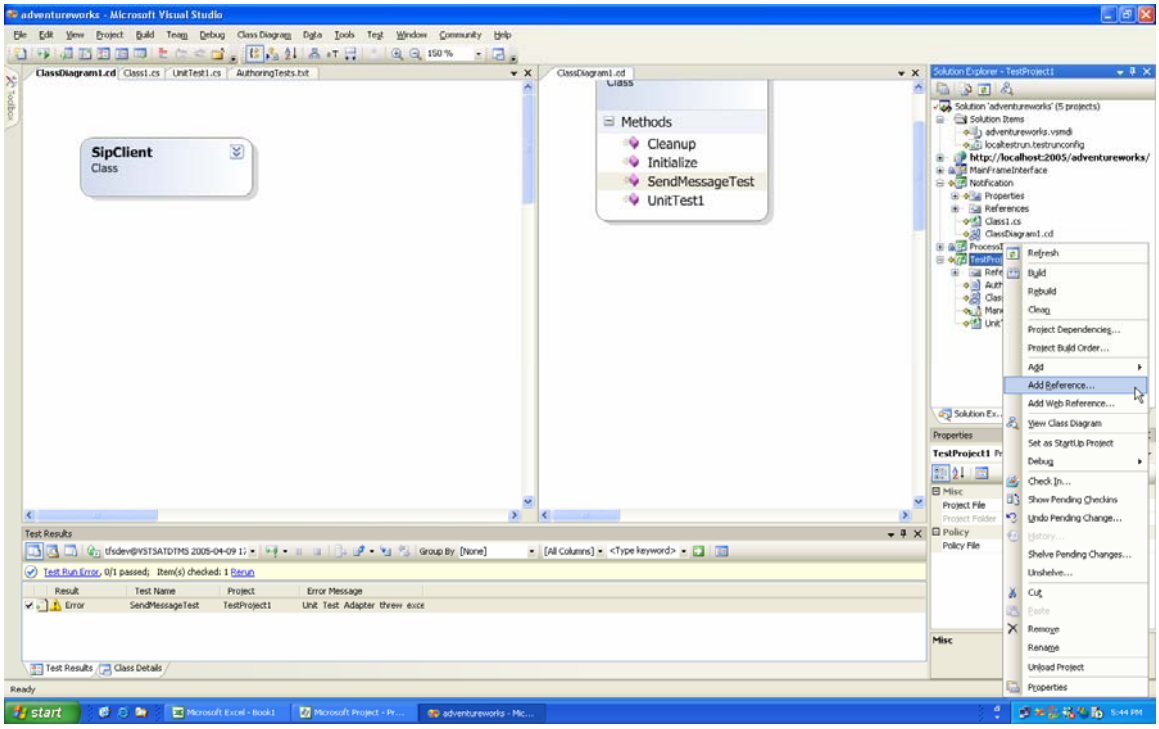
Actions	Notice the change in your **Class Designer**



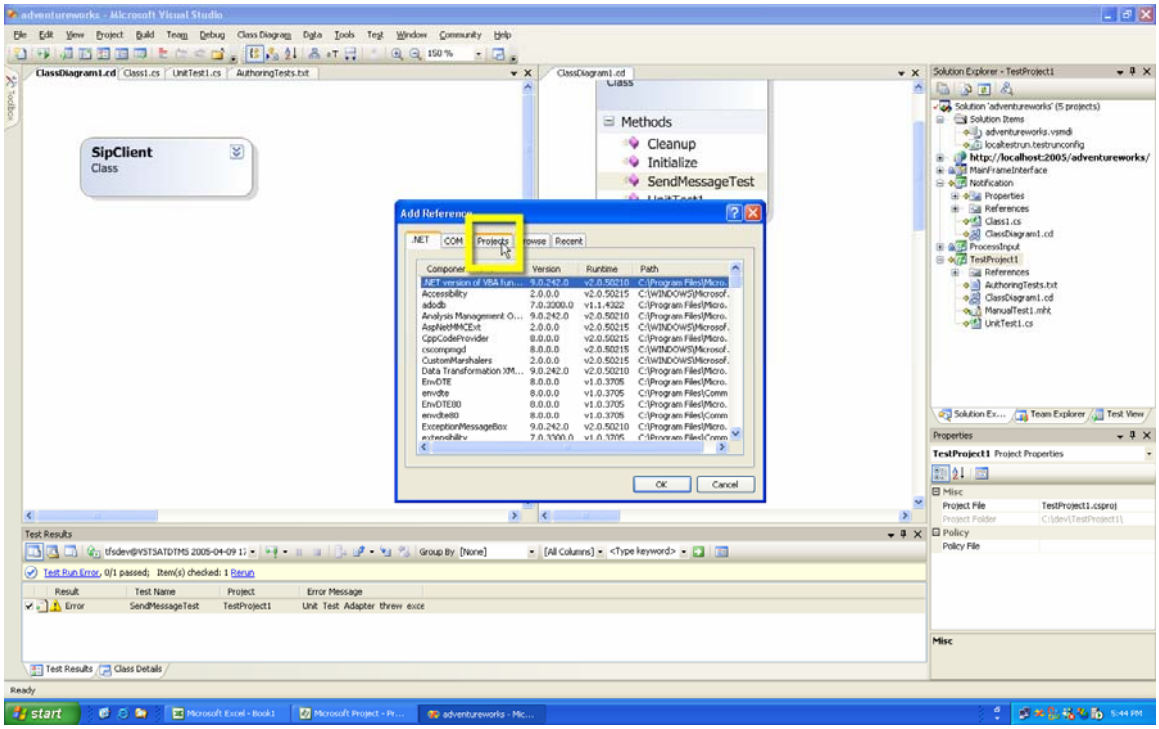
Actions	Select the **TestProject1** node



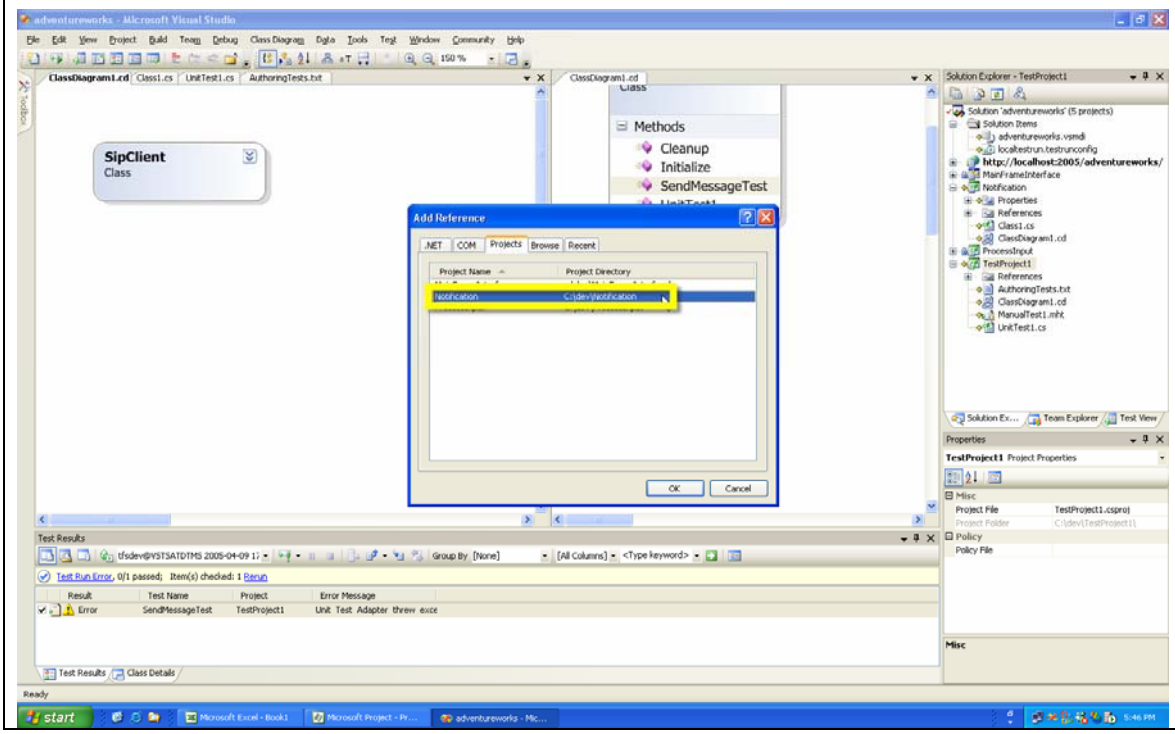
Actions	Right-click and choose **Add Reference**



Actions	Click on the **Projects** tab

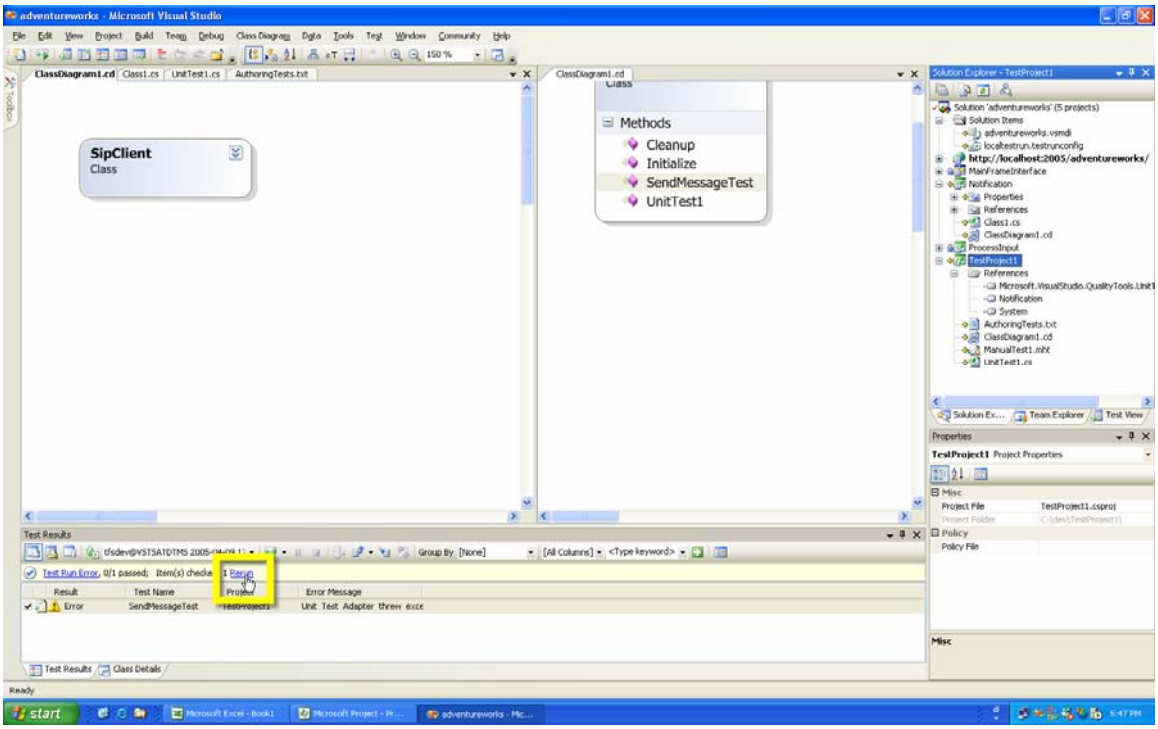


Actions	Choose the **Notification** project Press the **OK** button

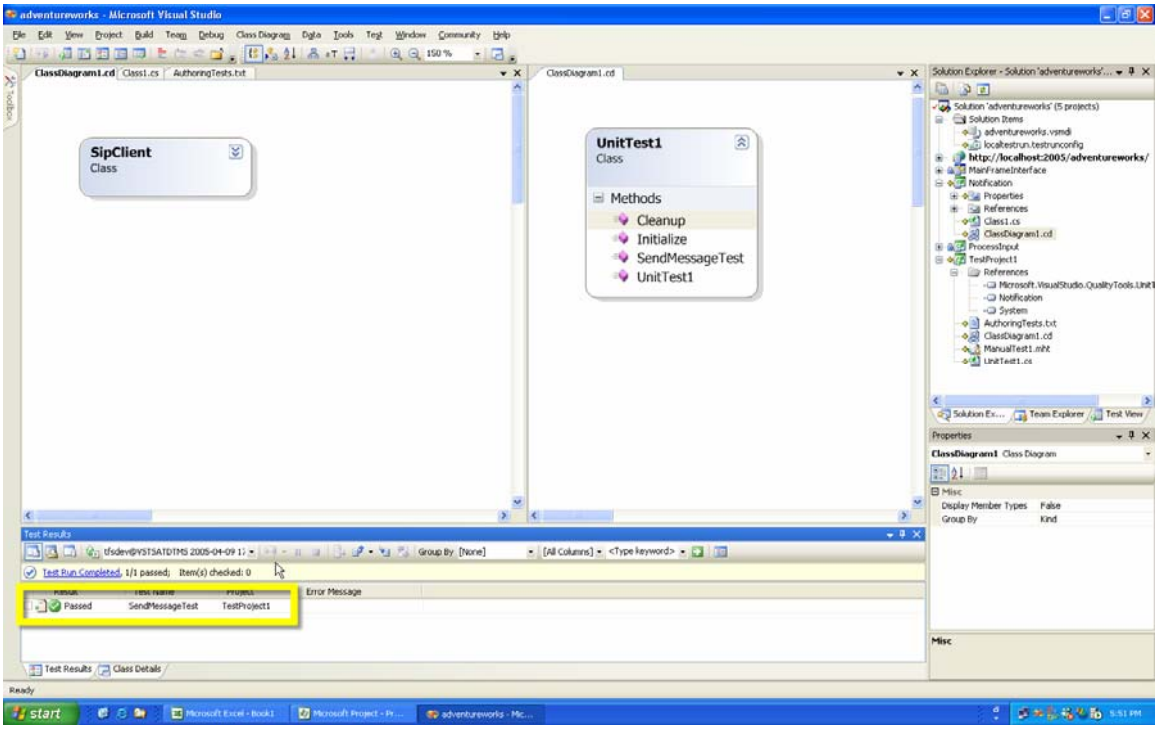


We've completed our first iteration of TDD – we have written just enough code to allow our unit test to pass. Let's rerun our test again to ensure that it does indeed pass.

Actions	Press the **Rerun** link in the **Test Results** window



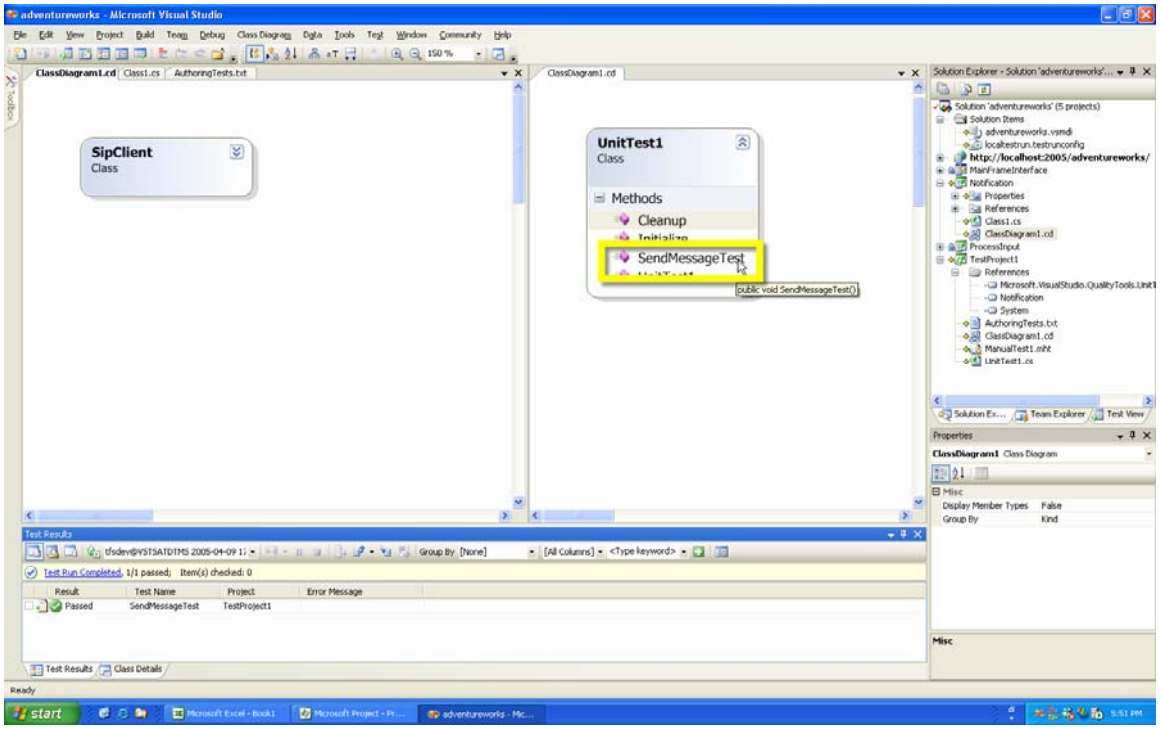
Actions	This time the test should **pass**



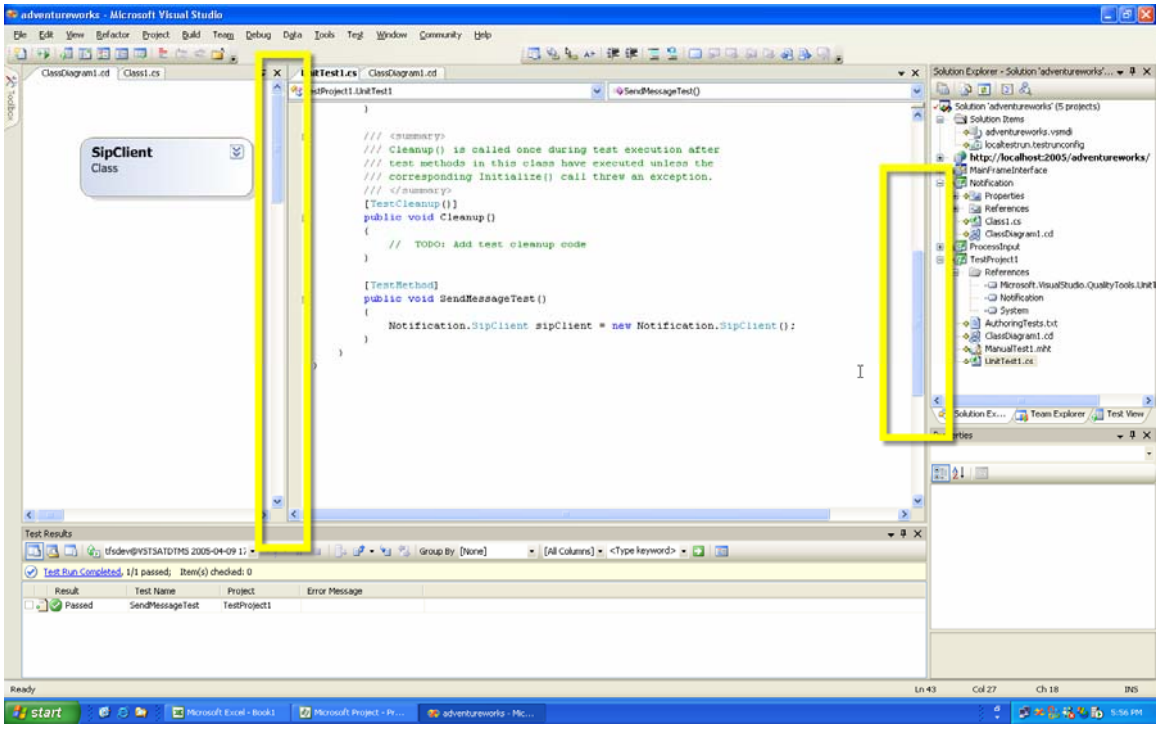
Our test passes now, but it doesn't really measure quality at this point. We are just instantiating an empty class.

Now, we'll go into the code behind our SendMessageTest and make it more strenuous.

Actions	Double-click on the **SendMessageTest** method in your **Class Diagram**



Actions	Use the vertical splitter bar to give more room to the **UnitTest1.cs** file



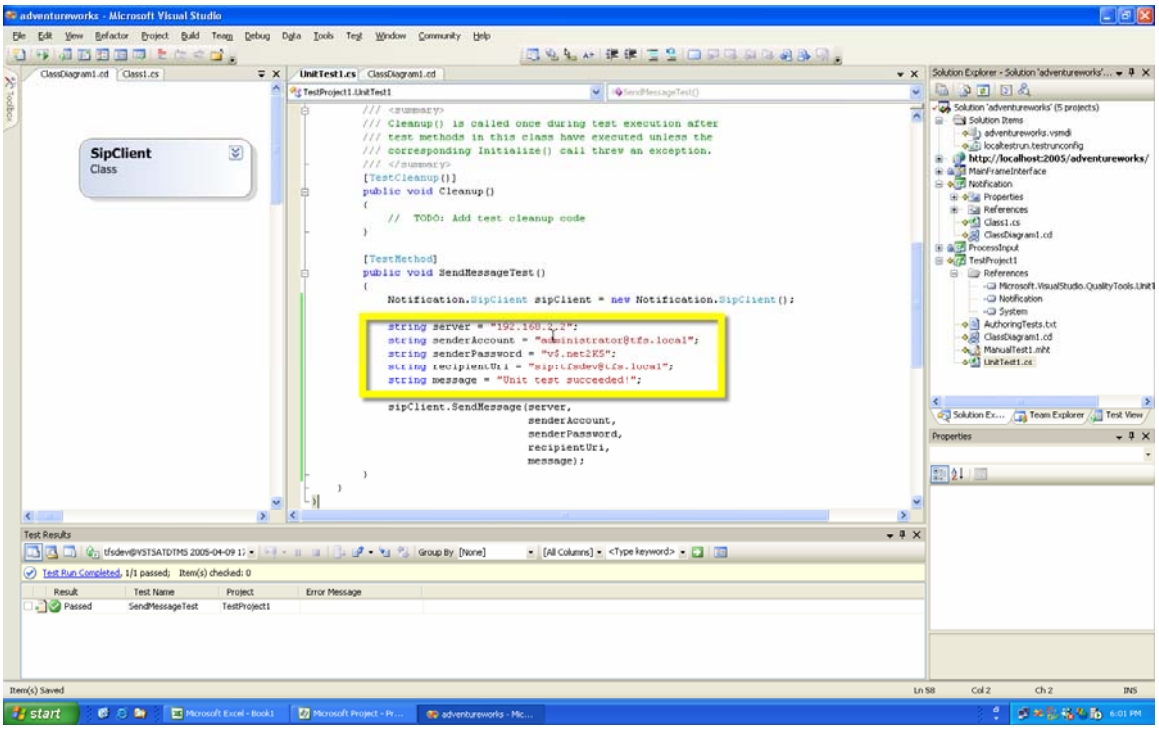
This is where a TDD approach really pays off. Instead of thinking about how we write a component to send IM messages, we can think about how we would want to use such a component.

This is a very subtle change in the way we approach software development, but it can yield surprisingly well-designed code.

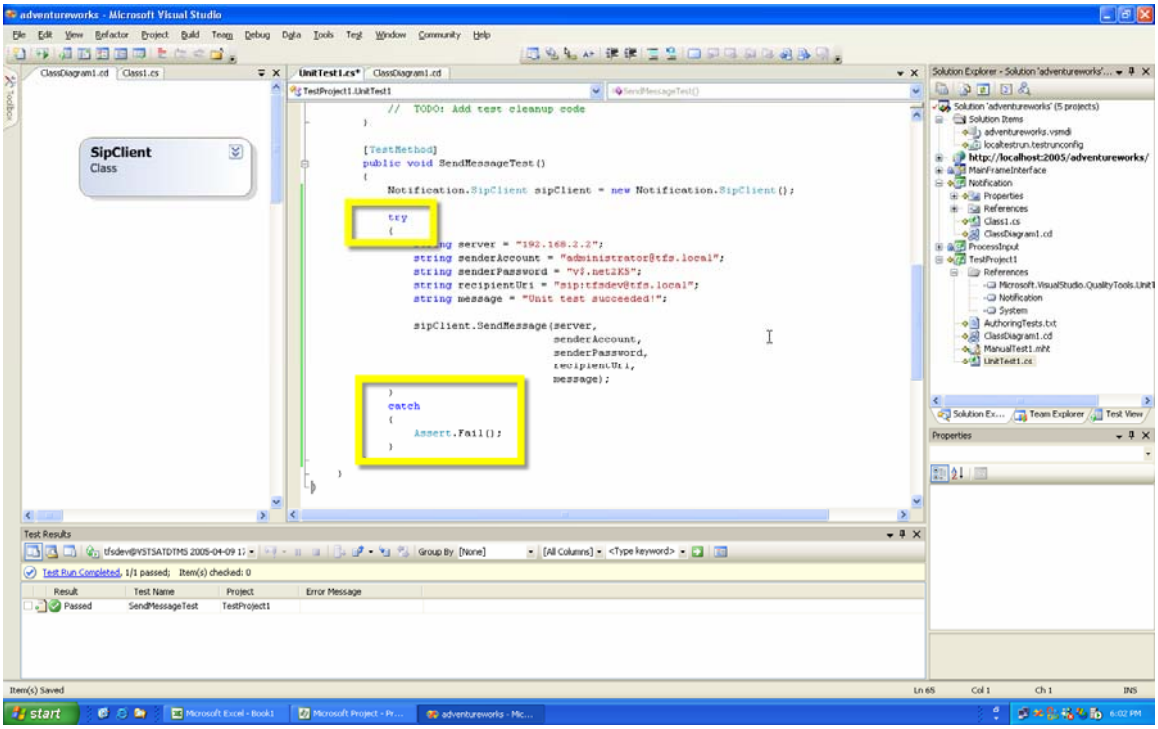
The code that we will write in our unit test is for the method that we want to call to send our IM message. We want to be able to specify:

- The address of the IM server we want to use
- The account of the sender
- The password of the sender
- Who the message is intended for
- What the message is

Actions	In the **SendMessageTest** method, insert the following code:
	
	`string server = "192.168.2.100";`
	`string senderAccount = "administrator@tfs.local";`
	`string senderPassword = "P@ssw0rd";`
	`string recipientUri = "sip:tfsdev@tfs.local";`
	`string message = "Unit test succeeded!";`
	`target.SendMessage(server,`
	`senderAccount,`
	`senderPassword,`
	`recipientUri,`
	`message);`
	
	`Assert.AreEqual(true, true);`

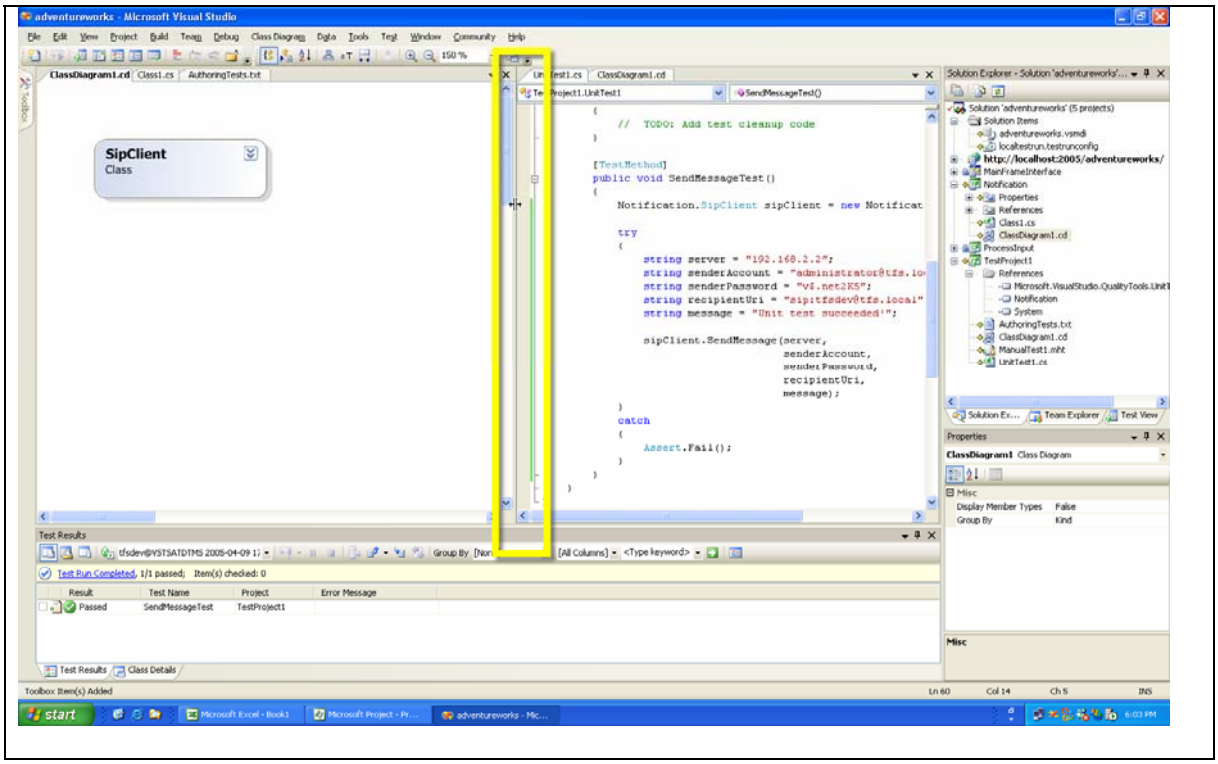


Actions	Wrap your code with **try/catch** statements

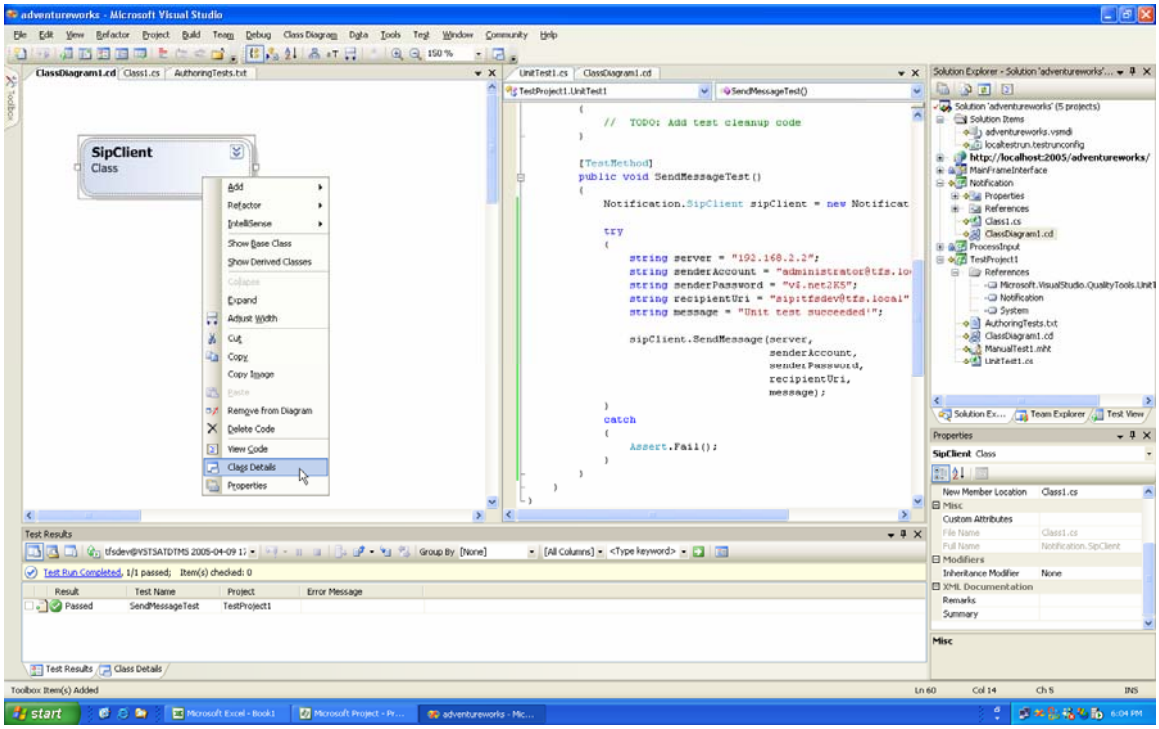


Our test is complete at this point – we have the code to send an IM message, and if there is an exception thrown, we will fail the test. Now it's time to actually write the code we need.

Actions	Use the vertical splitter bar again to give both sides the same amount of room as shown

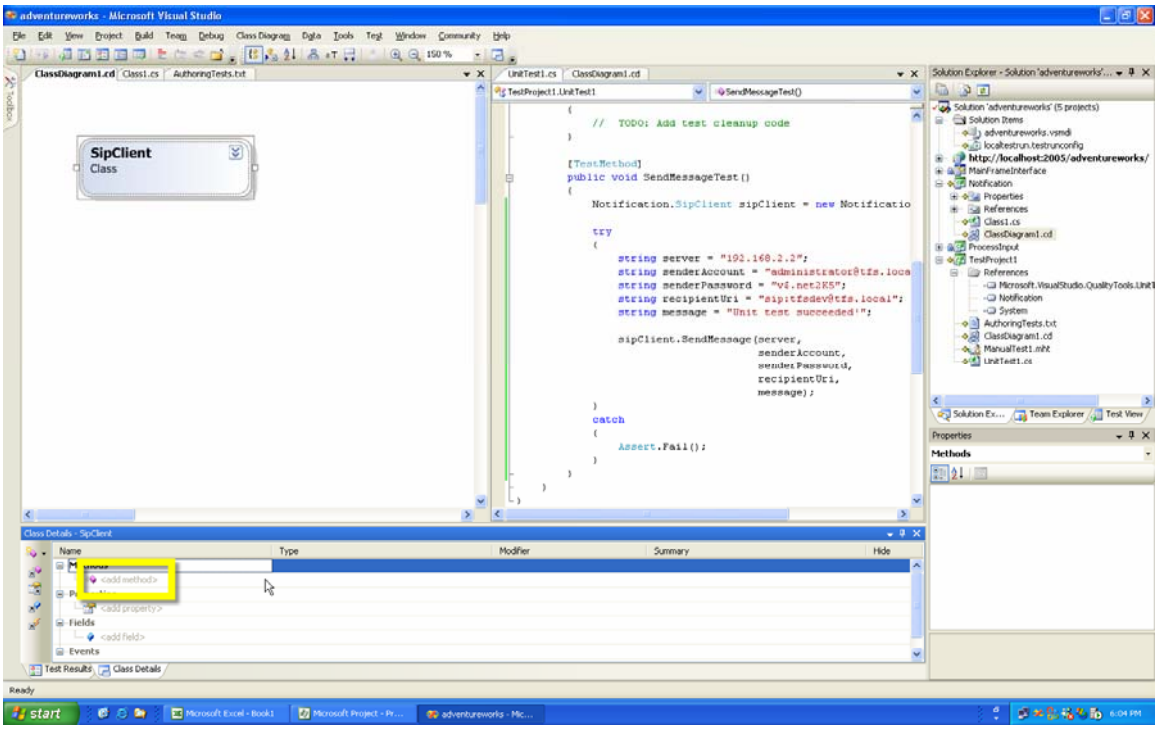

adventureworks - Microsoft Visual Studio
SipClient
Class
Test Results
Test Run Completed. 1/1 passed; Item(s) checked: 0
Result
Test Name
Project
Error Message
Passed
SendMessageTest
TestProject1
Solution Explorer - Solution 'adventureworks'...
Properties
Misc
start

Actions	Right-click on the **SipClient** class and choose **Class Details**

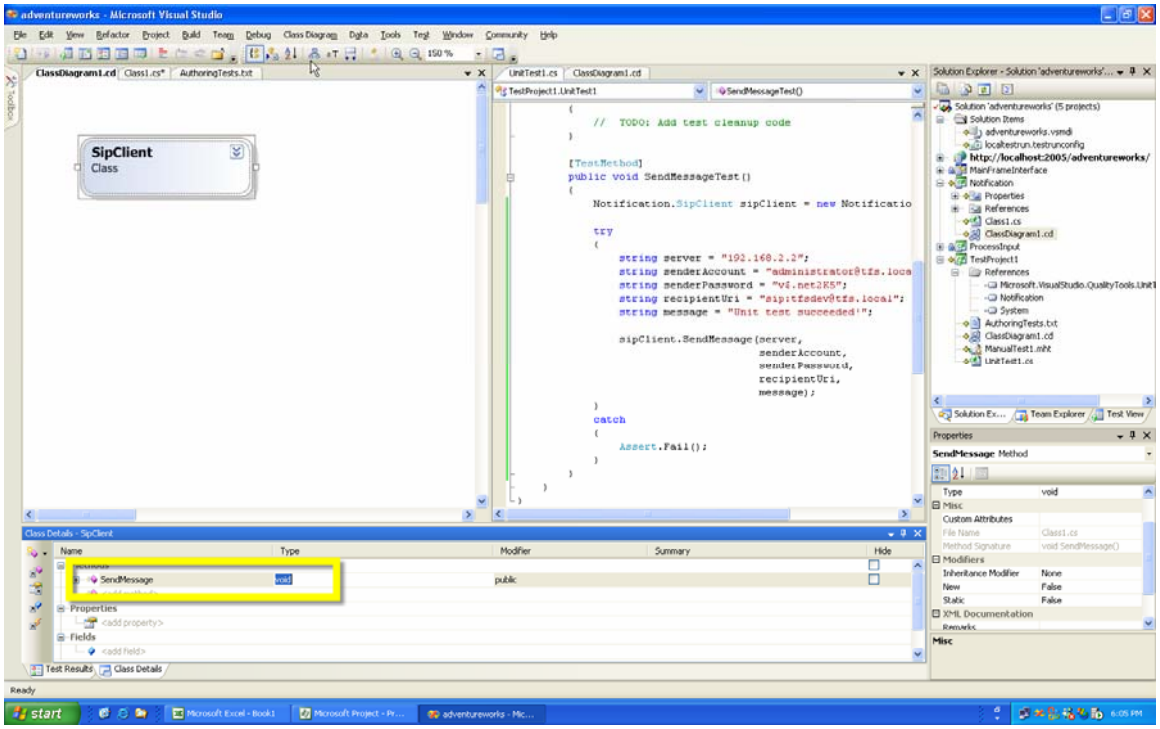


Following TDD, we will write just enough code to allow our test to pass. So, we will use Class Designer and create a method with a signature that matches our unit test.

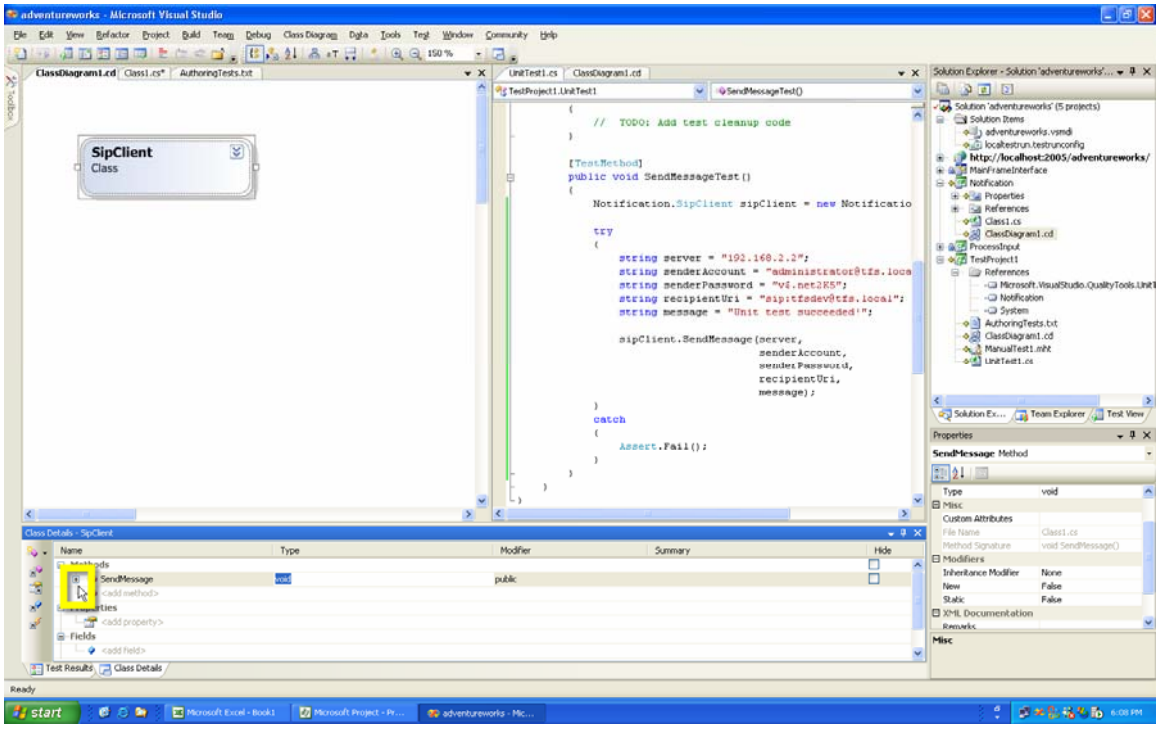
Actions	Click on the **<add method>** text



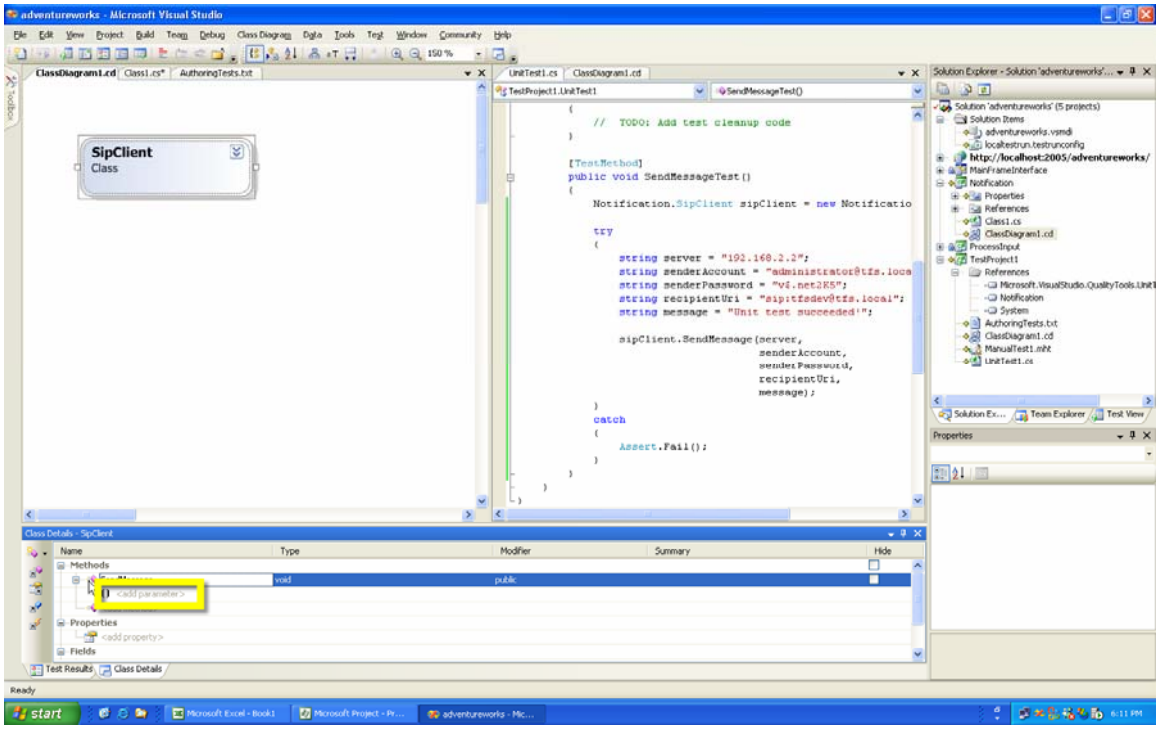
Actions	Type **SendMessage** and Press the **ENTER** key



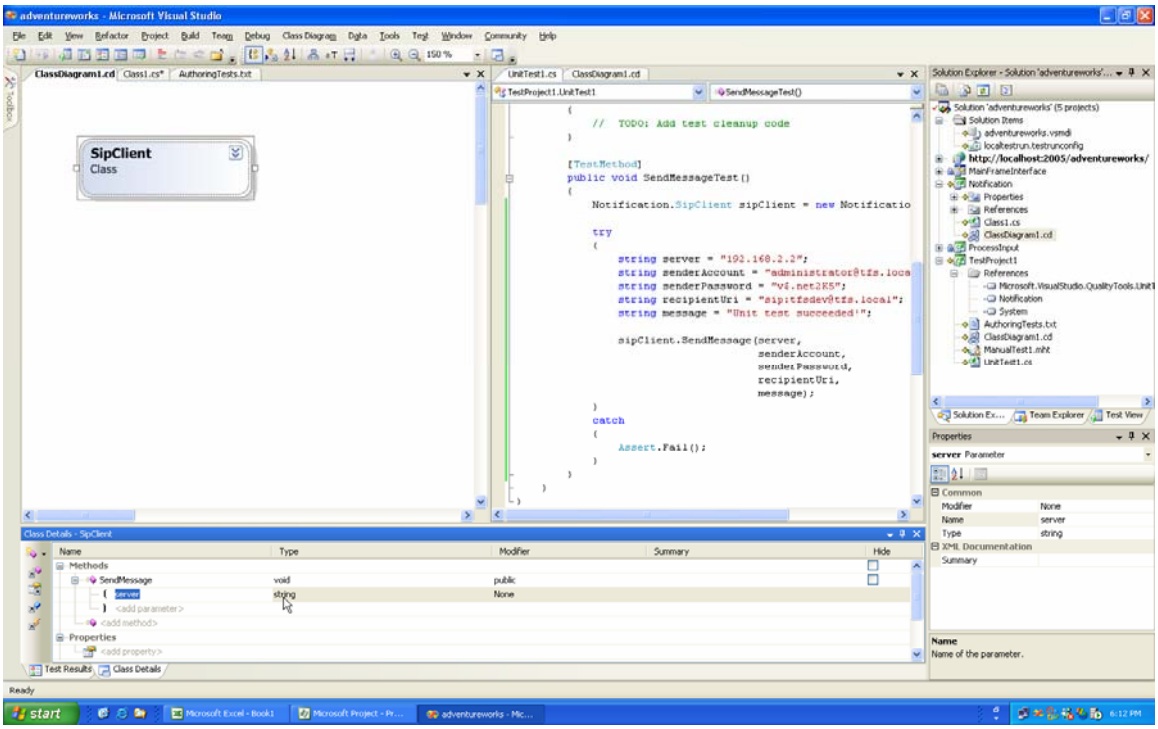
Actions	Expand the **SendMessage** node



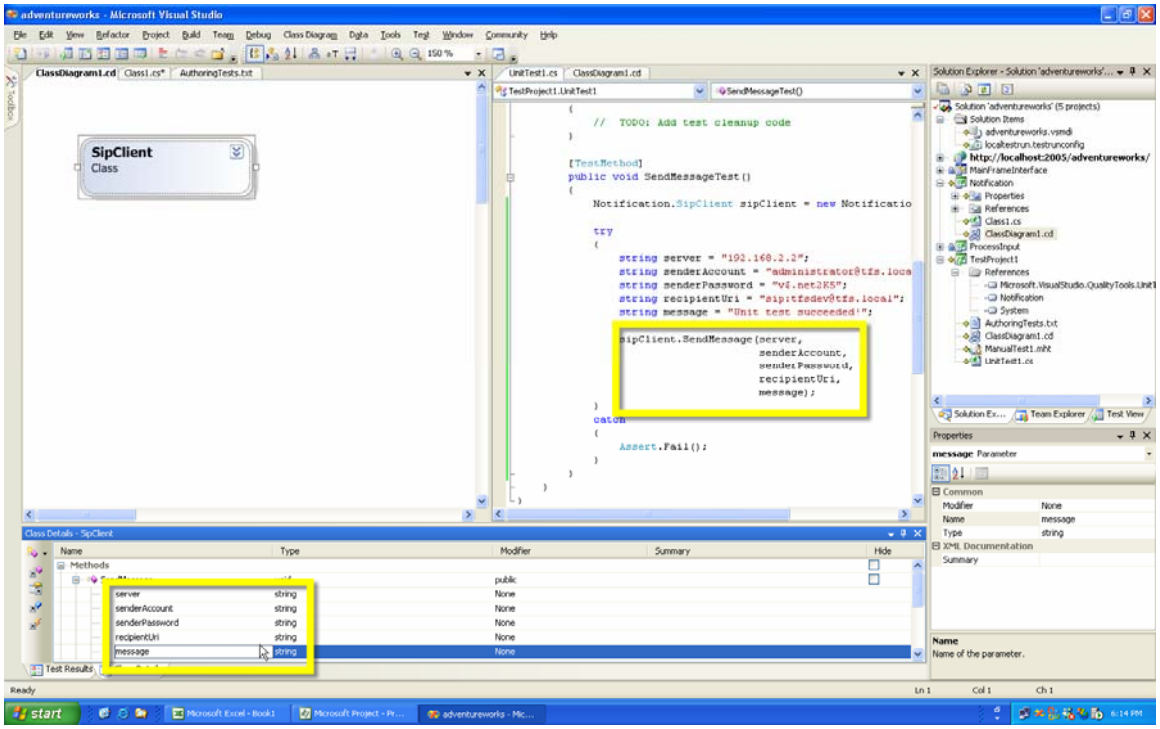
Actions	Click on the **<add parameter>** text



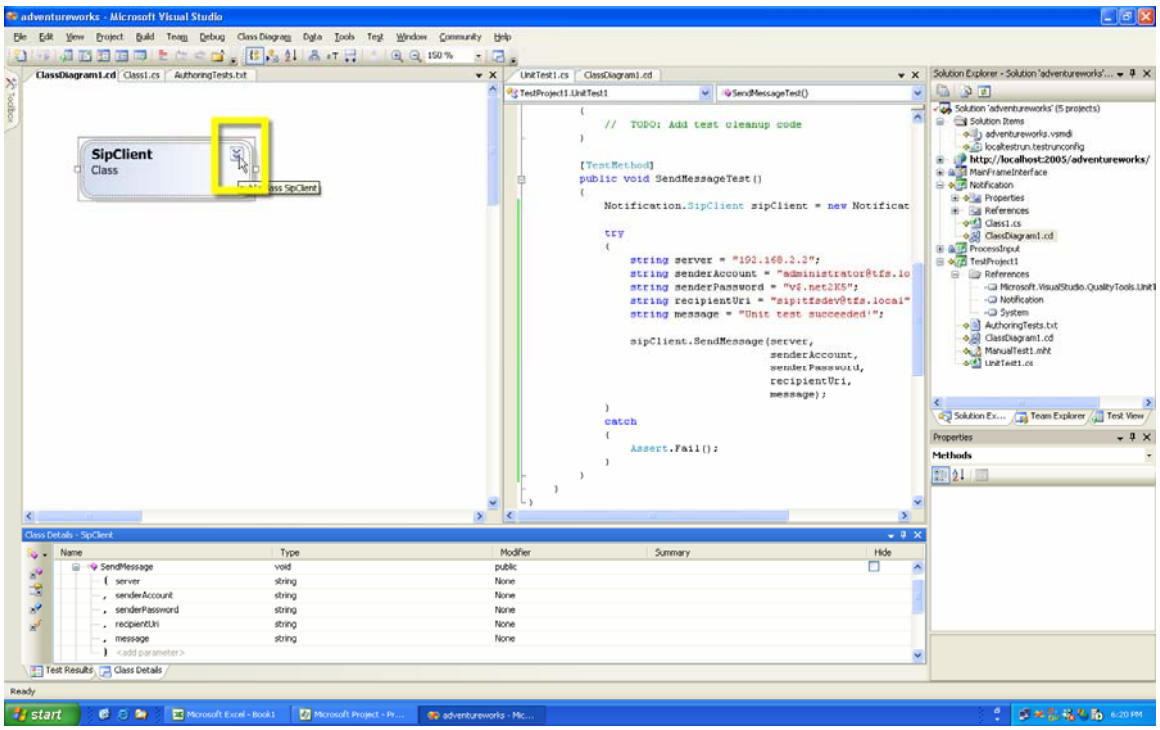
Actions	Type **server** and Press the **ENTER** key Type **senderAccount** and Press the **ENTER** key Type **senderPassword** and Press the **ENTER** key Type **recipientUri** and Press the **ENTER** key Type **message** and Press the **ENTER** key



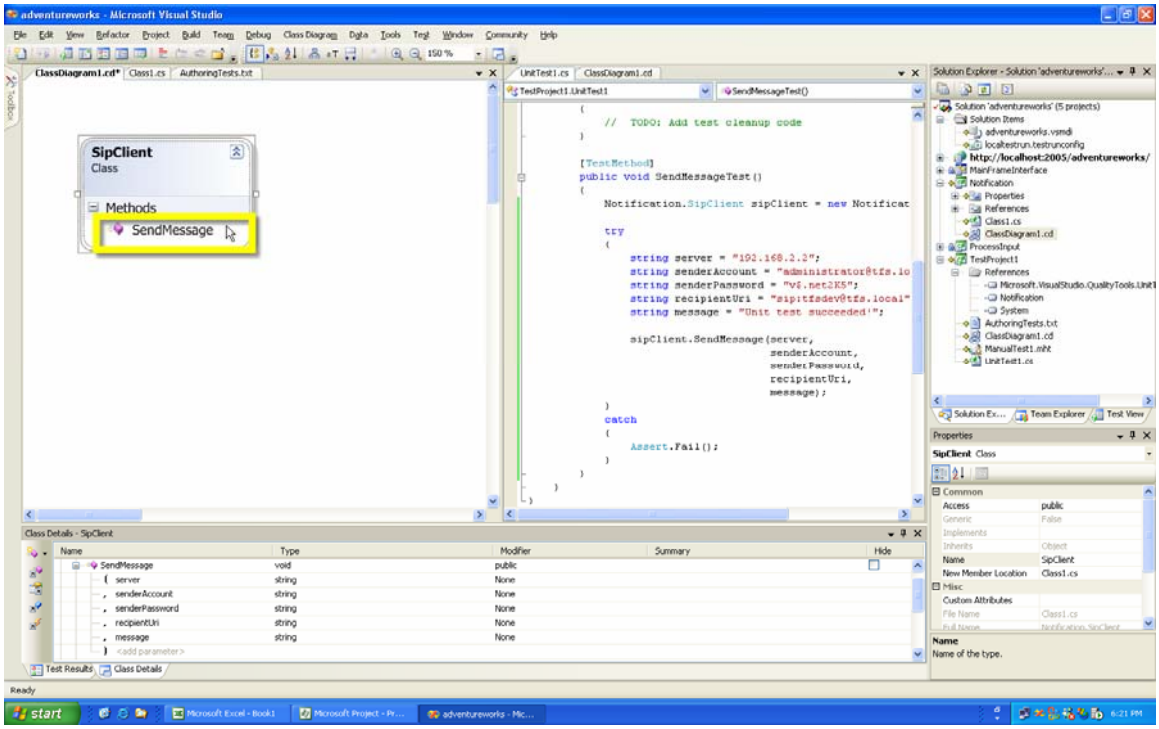
Actions	Your class diagram and code should look as shown



Actions	Click on the **chevron** for your **SipClient** class diagram as shown



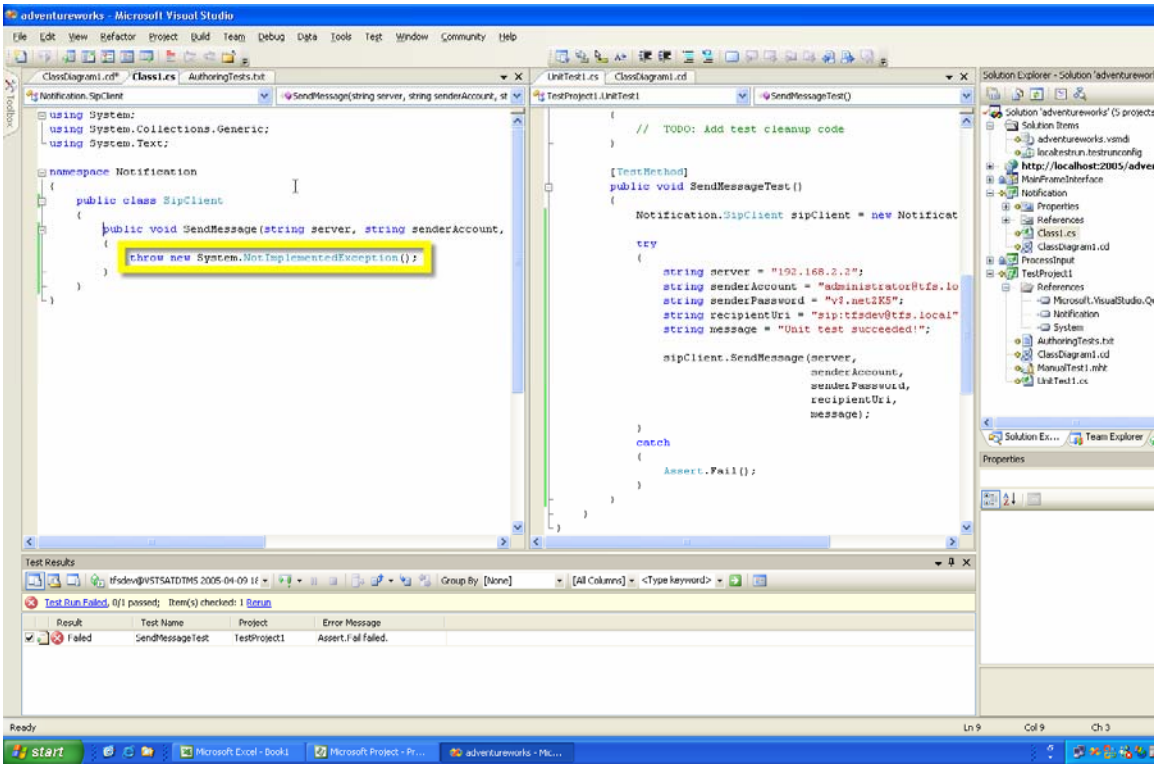
Actions	Double-click on the **SendMessage** method



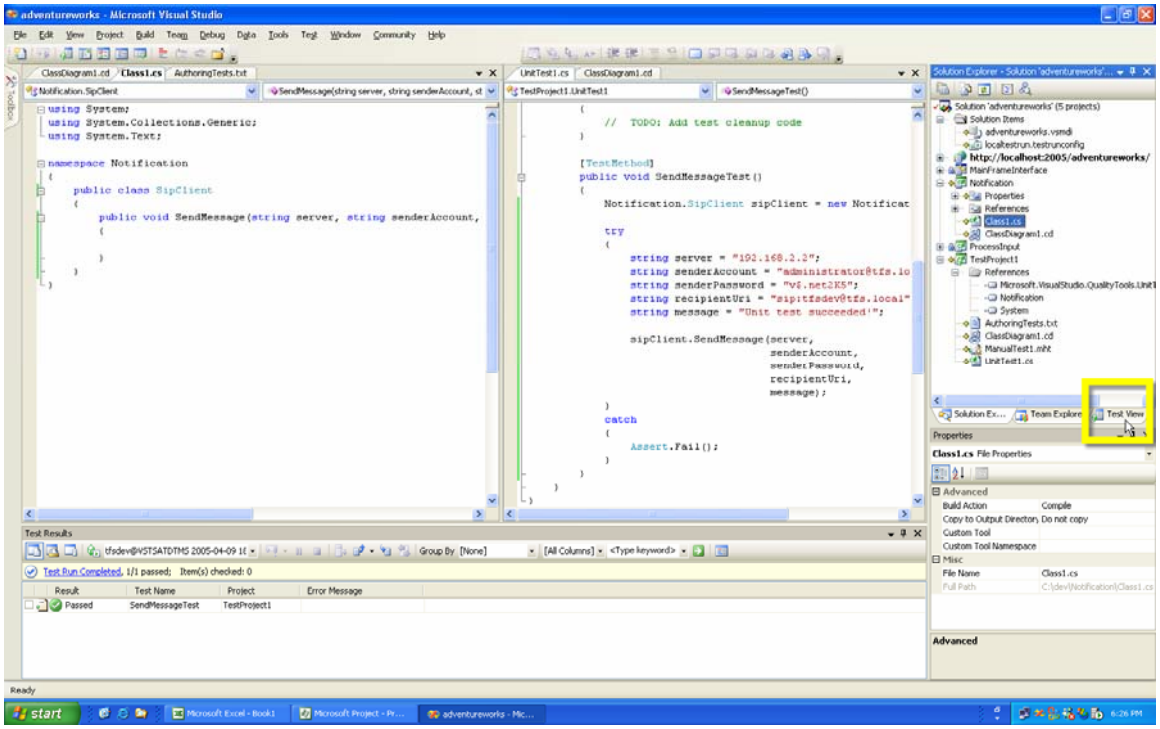
When the Class Designer generates a method for us, it puts in a 'not implemented' exception. We'll remove this statement.

Now let's run our test again.

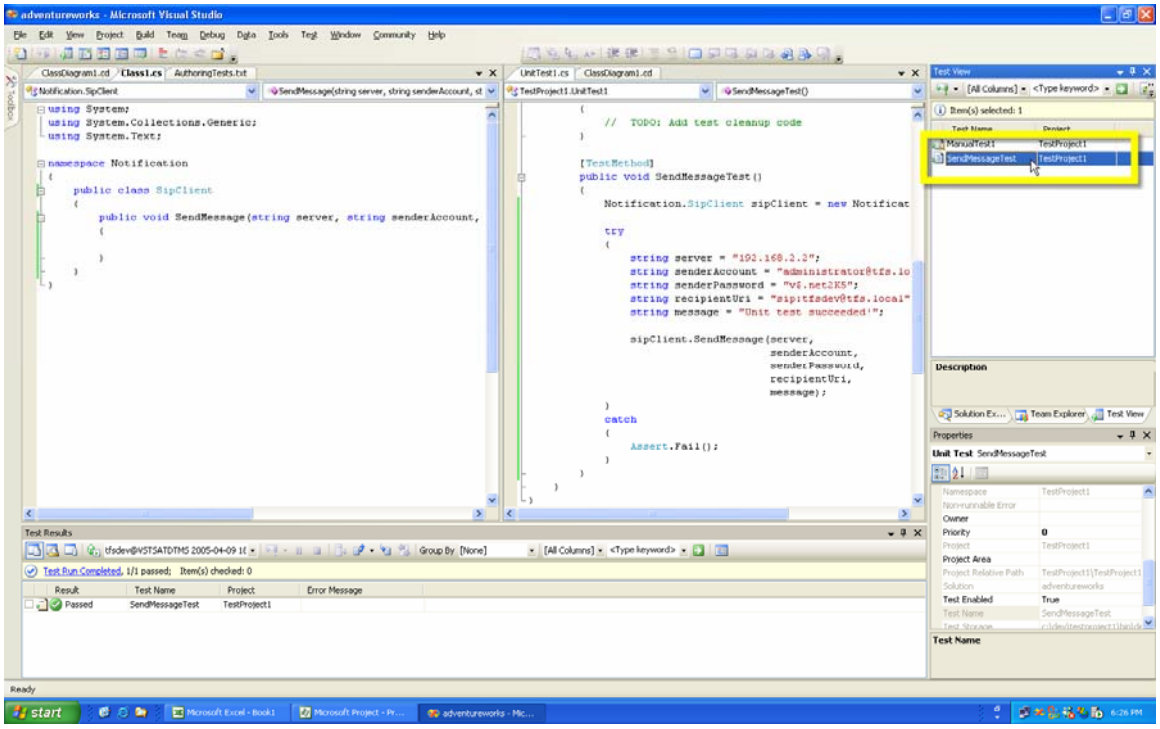
Actions	Remove the line of code as shown



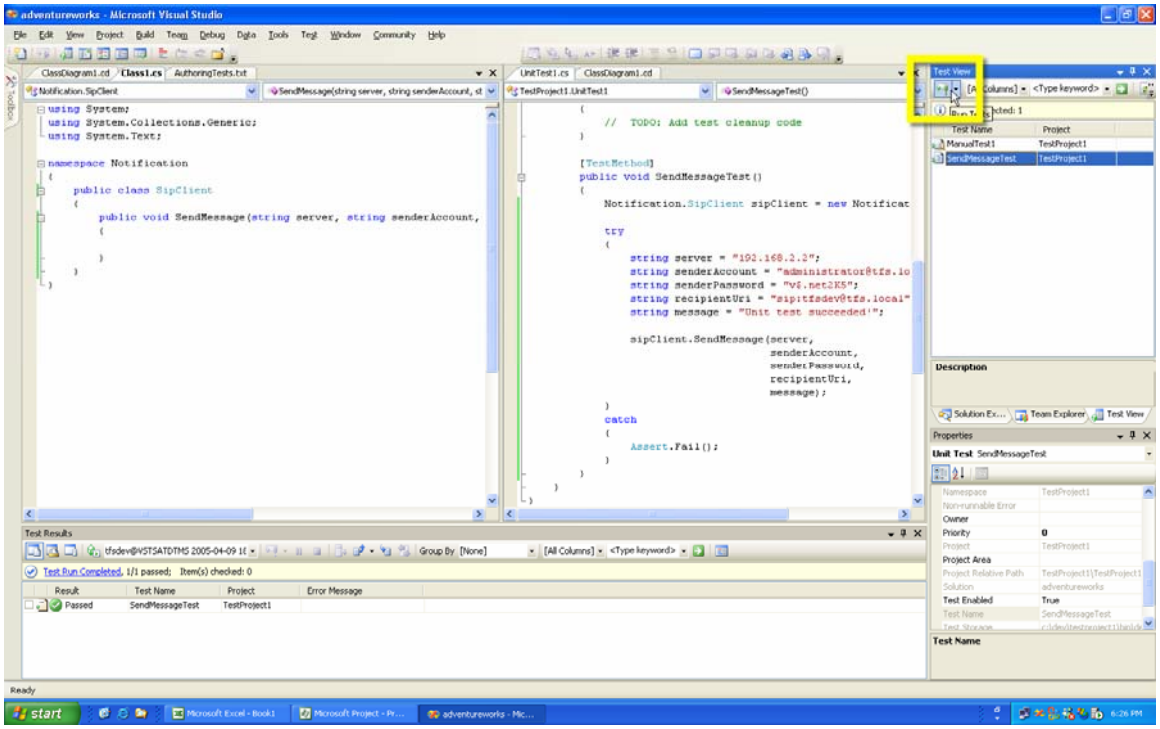
Actions	Click on the **Test View** tab



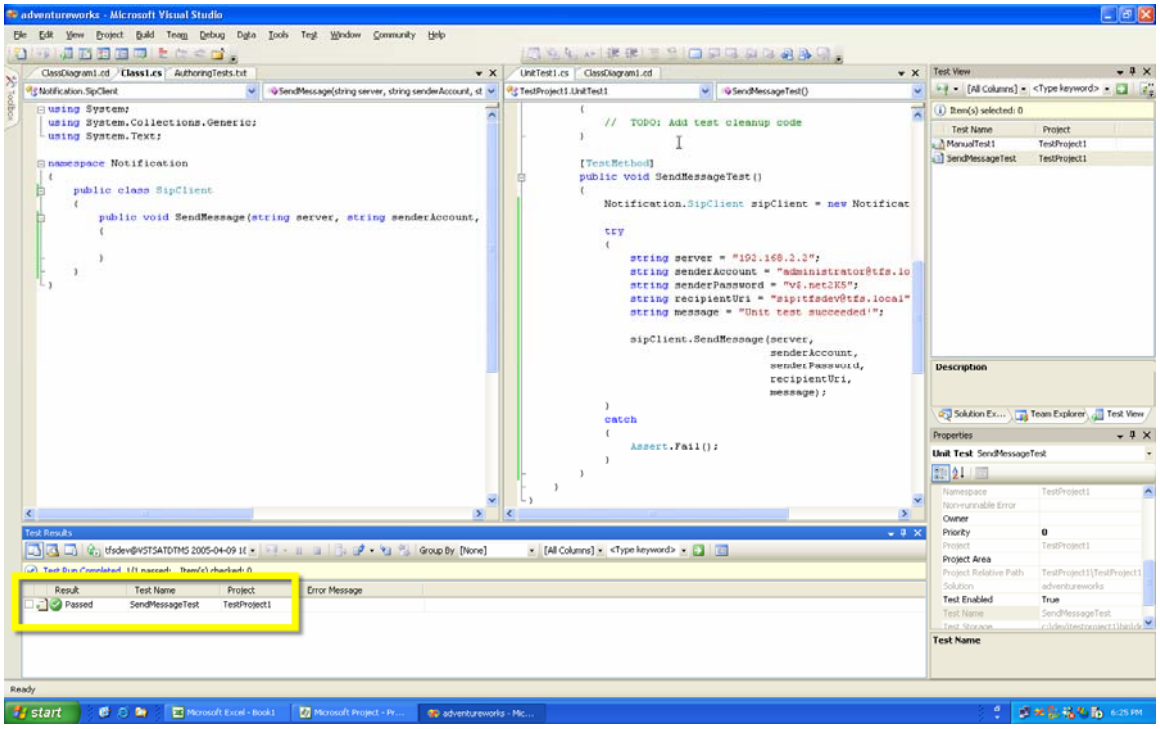
Actions	Select the **SendMessageTest** item



Actions	Press the **Run Tests** tool bar button



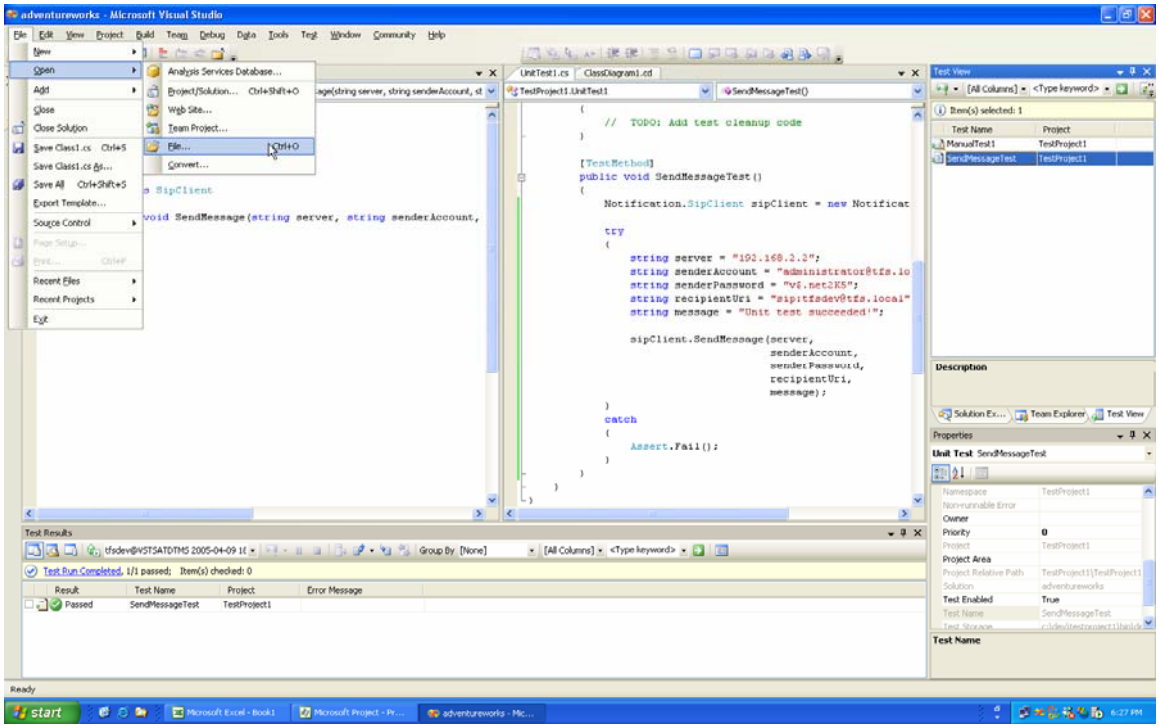
Actions	Your test should pass again



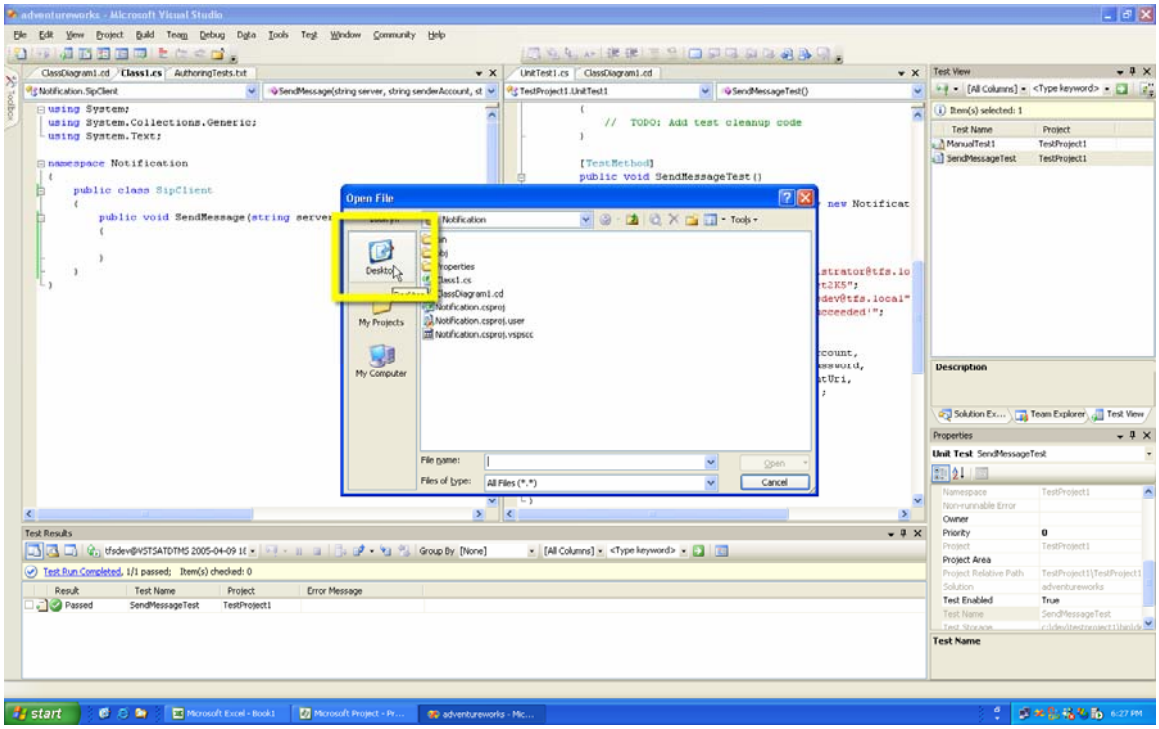
This is our last iteration of TDD – we will write the actual code behind our SendMessage method.

For the purpose of this hands on lab, this code has already been written for us. We will open the existing file (SipClient.cs) and cut-and-paste its code into our class.

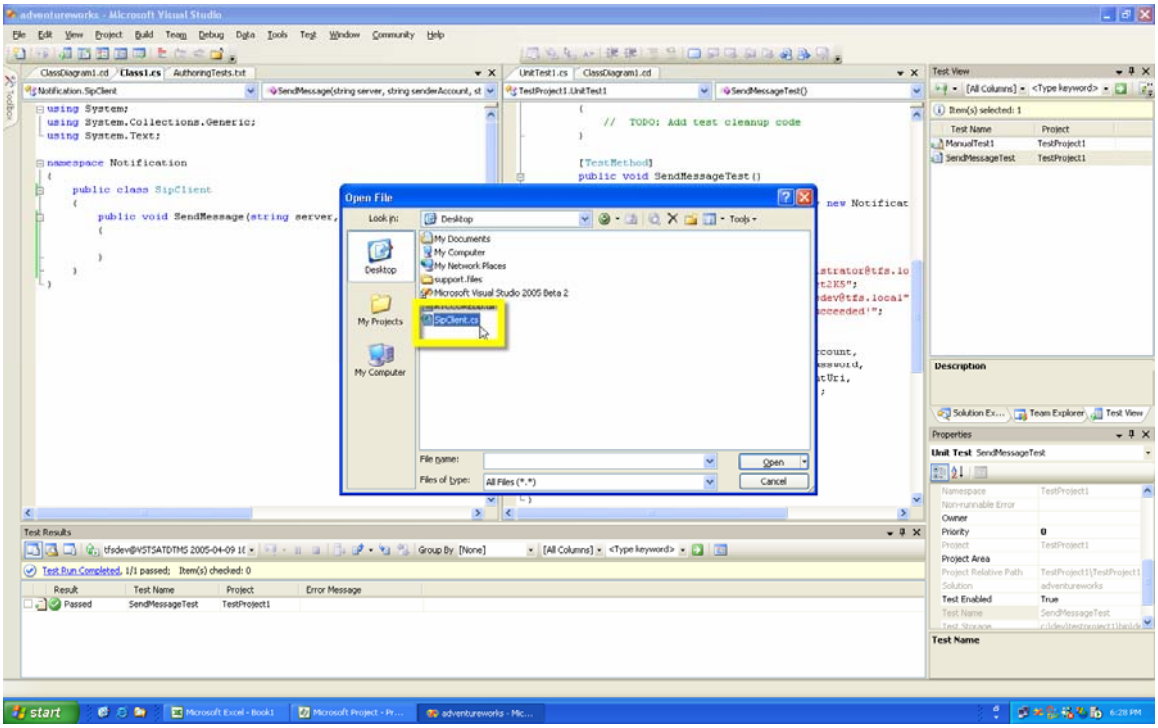
Actions	Choose the **File -> Open -> File** menu option



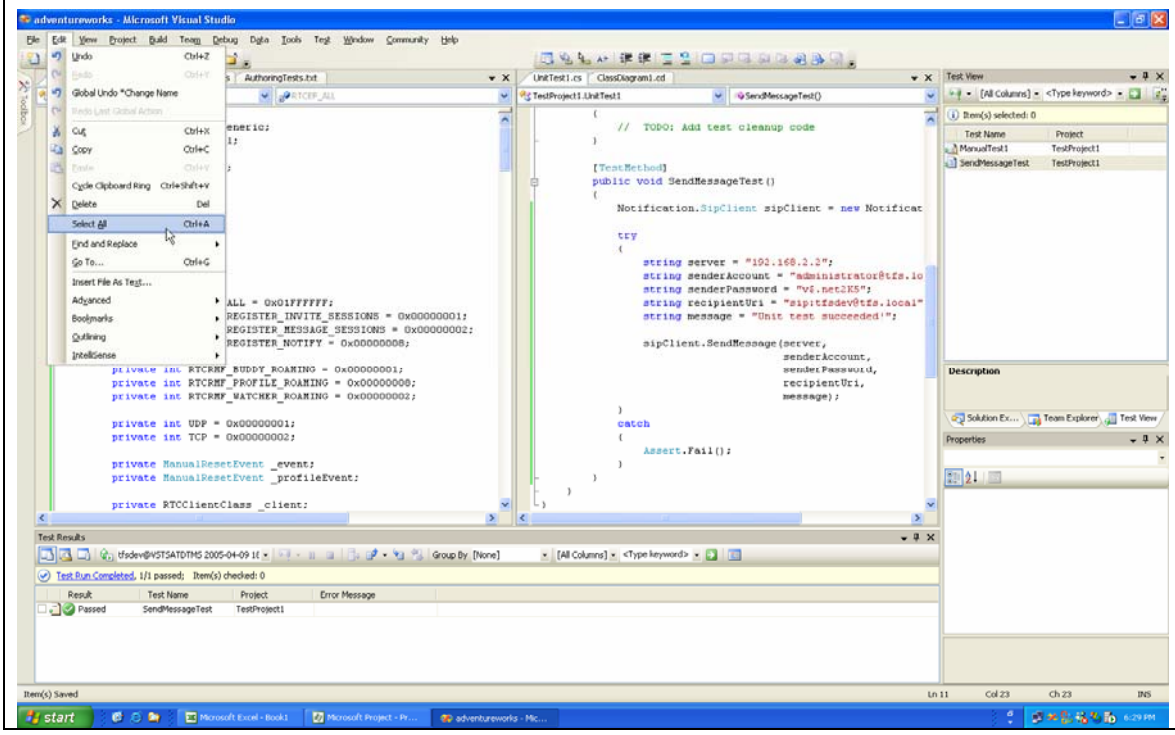
Actions	Press the **Desktop** button



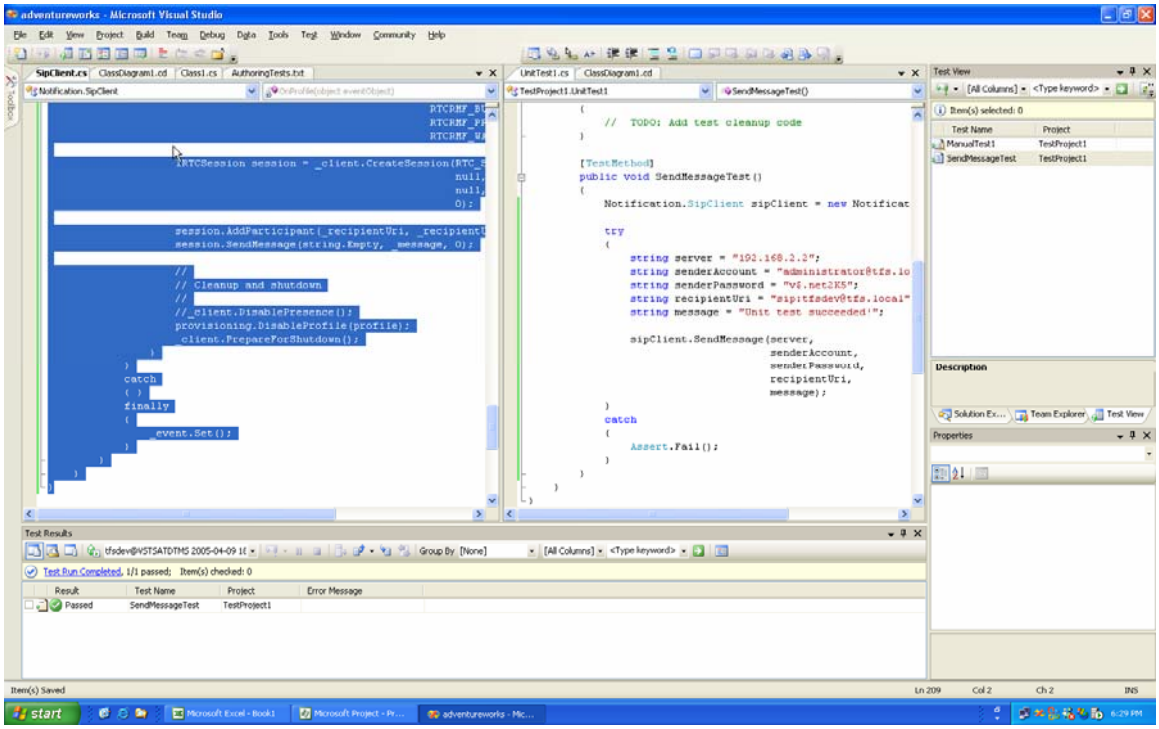
Actions	Choose the **SipClient.cs** file Press the **Open** button



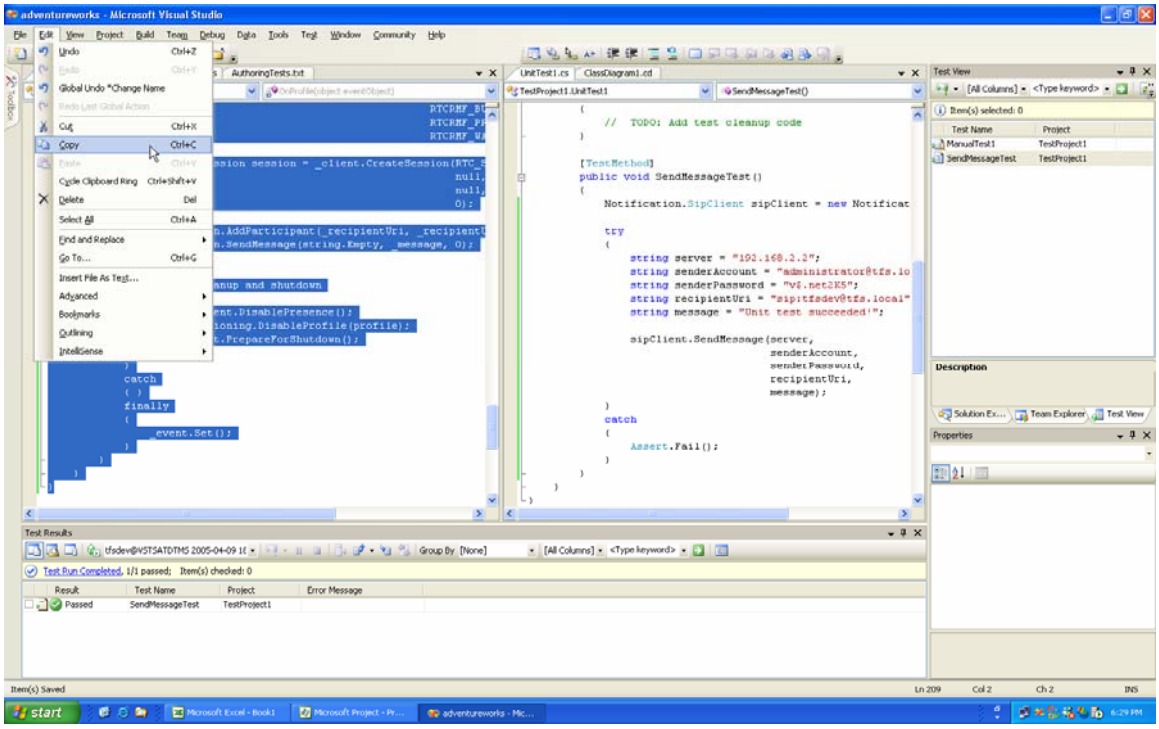
Actions	When **SipClient.cs** opens, choose the **Edit -> Select All** menu option



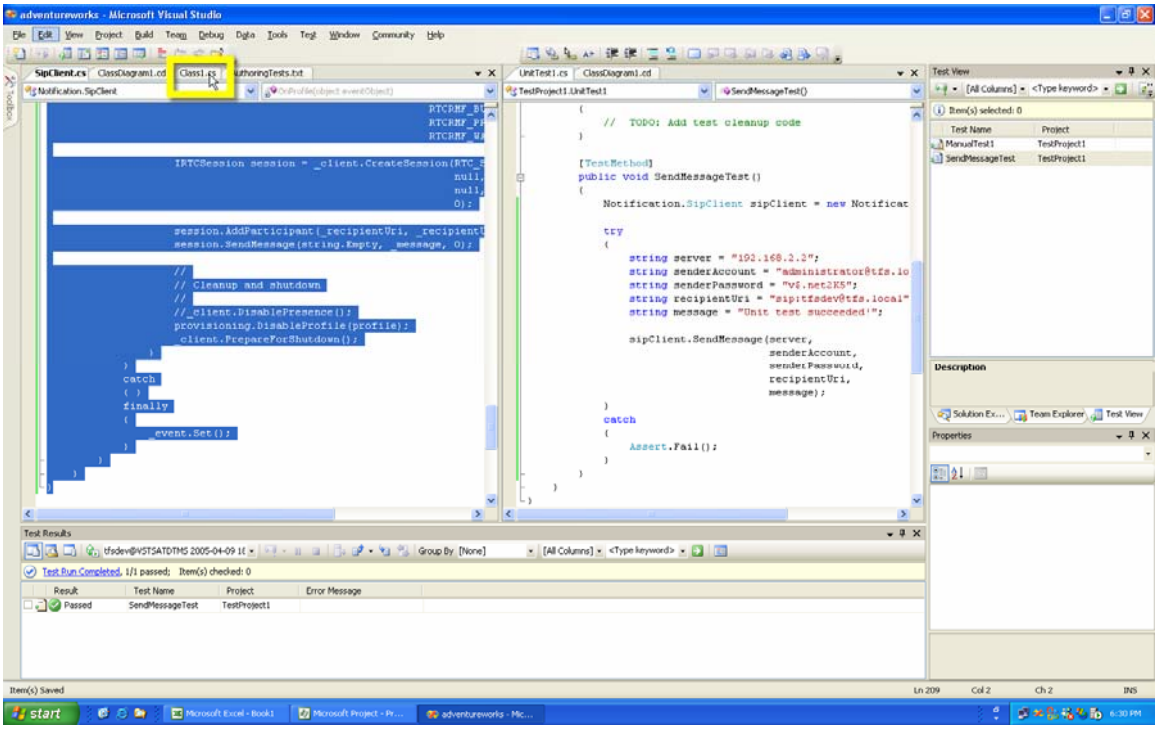
Actions	All of the text in **SipClient.cs** should be selected



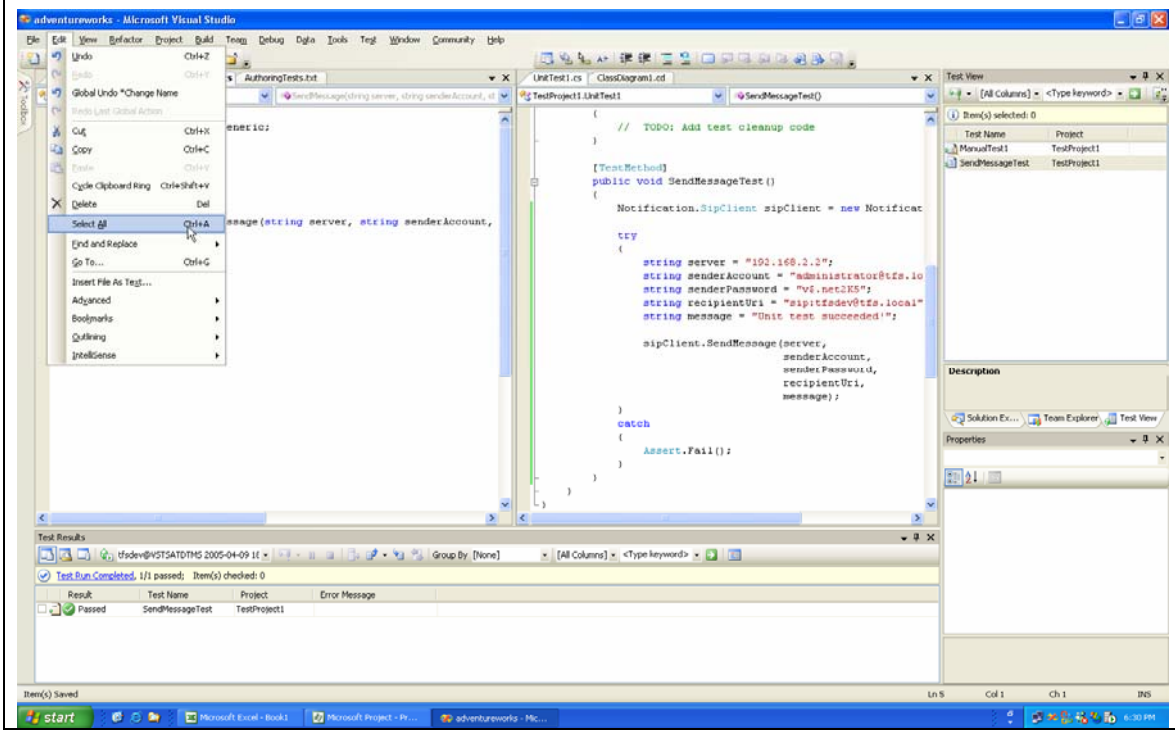
Actions	Choose the **Edit -> Copy** menu option



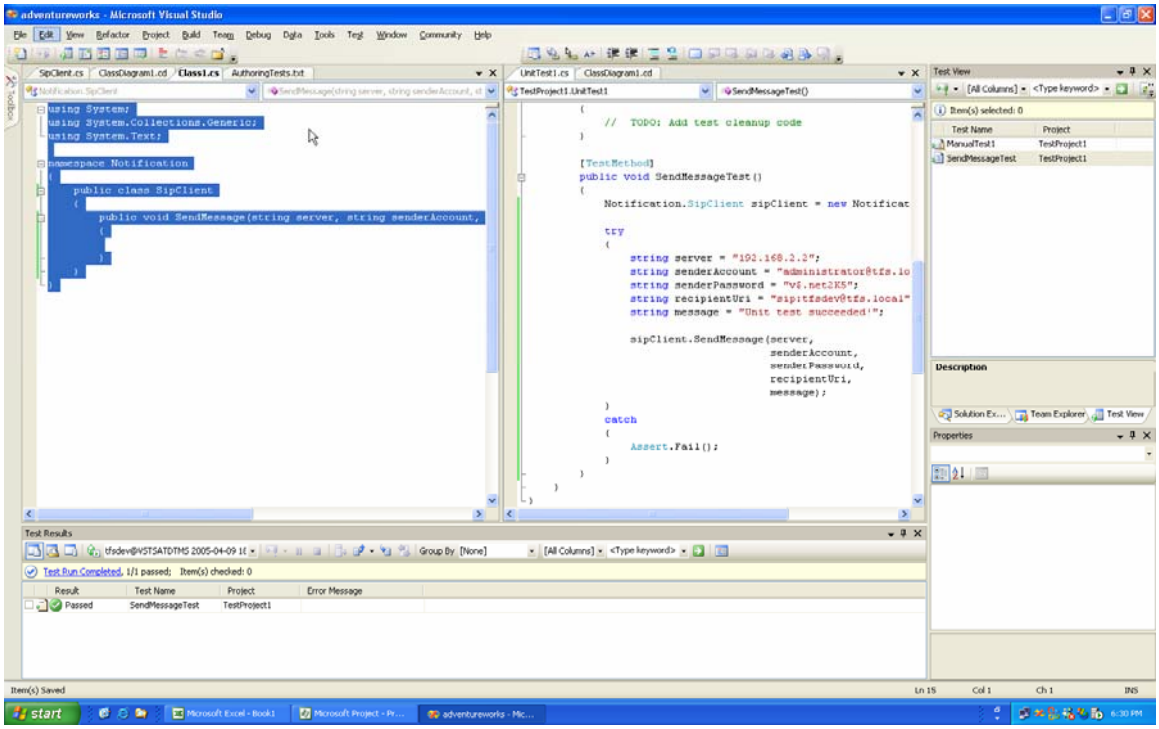
Actions	Select the **Class1.cs** file as shown



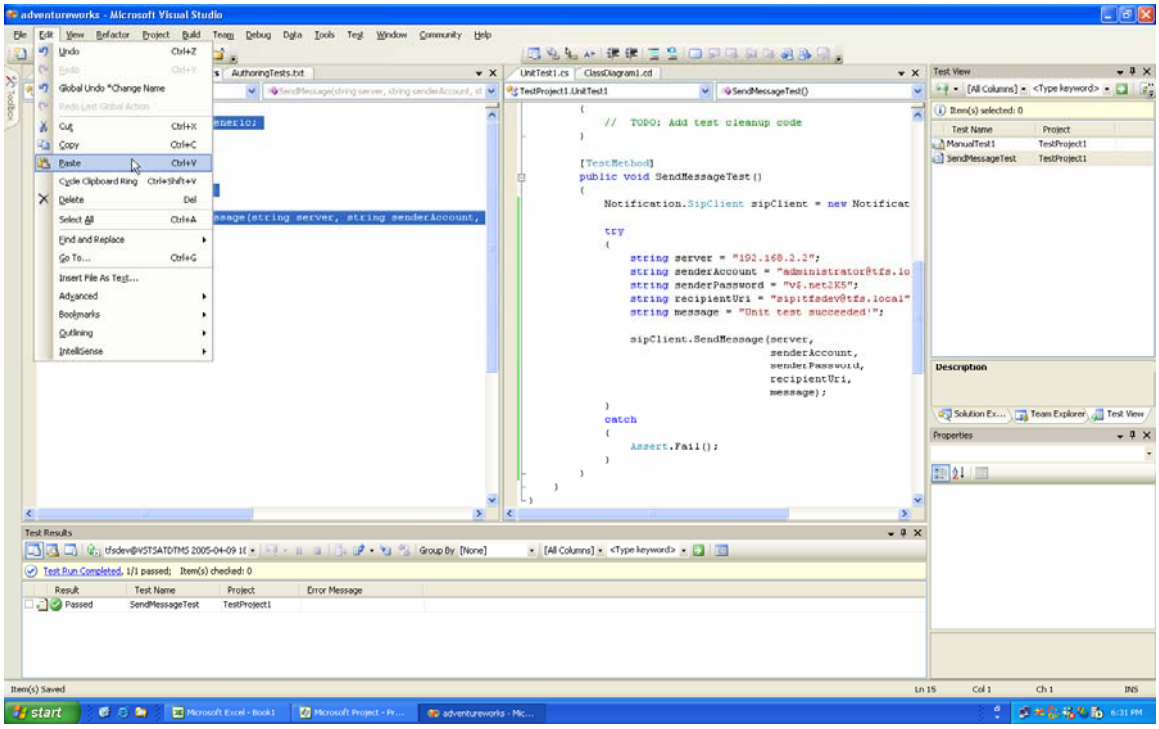
Actions	Choose the **Edit -> Select All** menu option



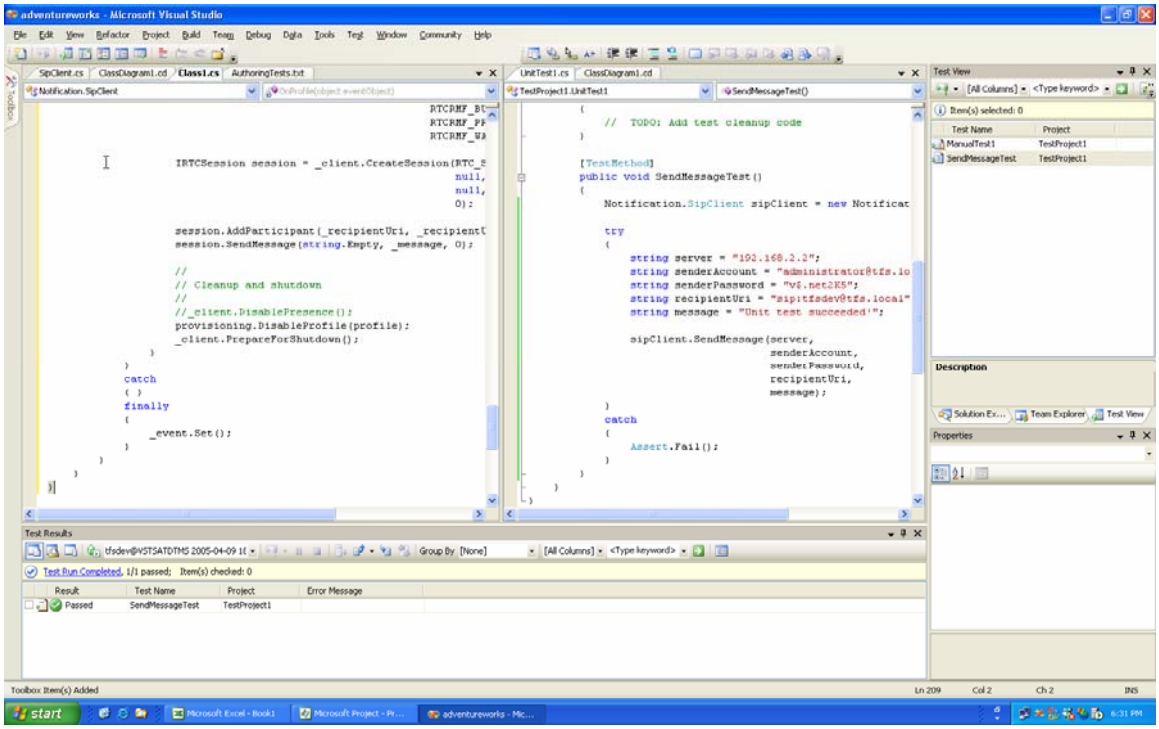
Actions	All of the text in **Class1.cs** should be selected



Actions	Choose the **Edit -> Paste** menu option

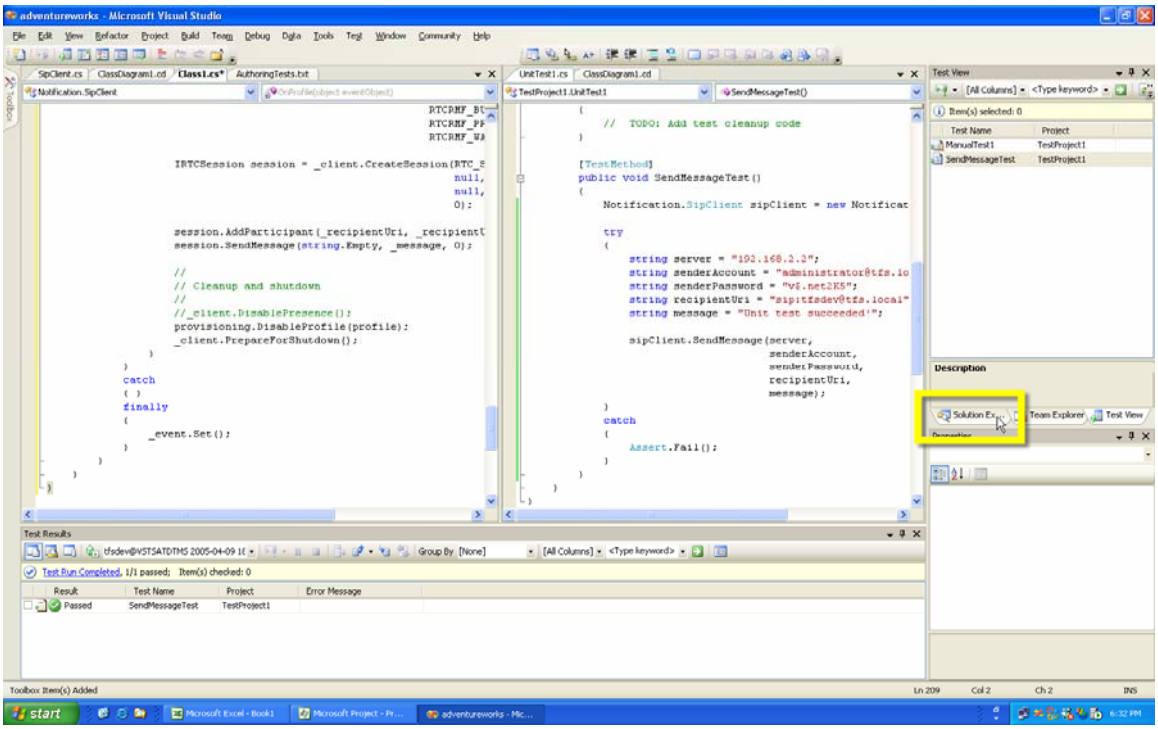


Actions	This should paste real code into **Class1.cs**

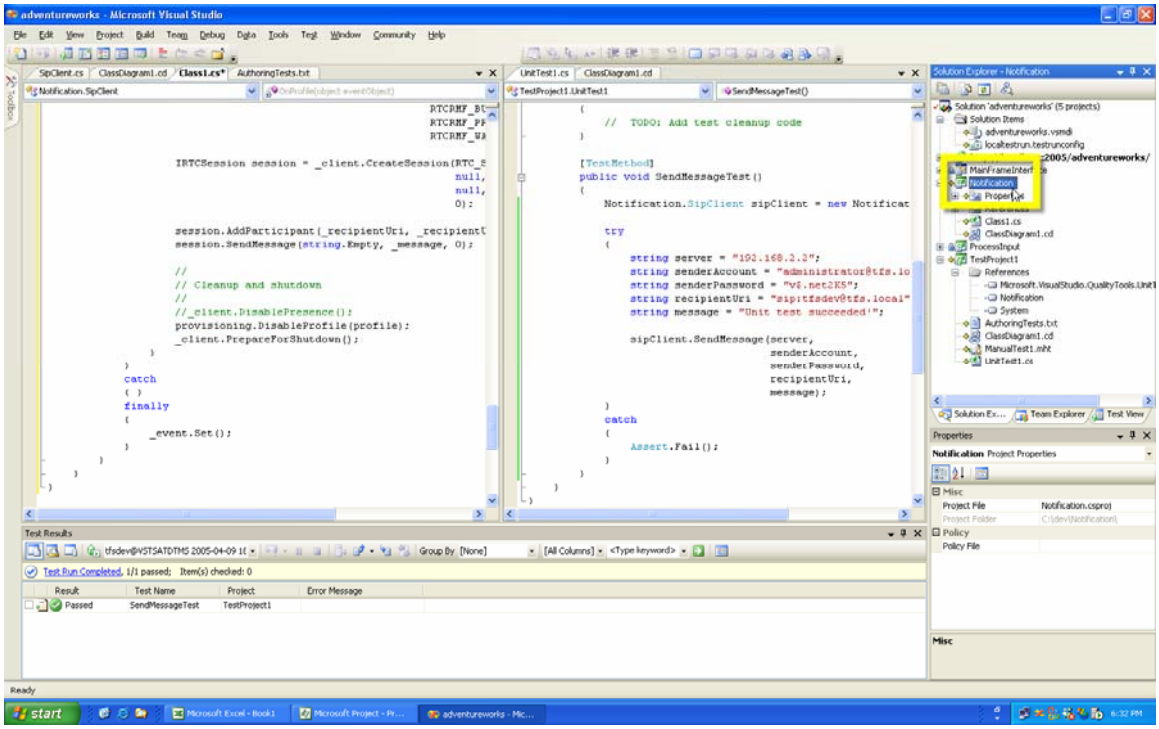


All of our real IM sending code is written – now we need to add a reference to an external library. This library provides the infrastructure that we need to communicate with an IM server.

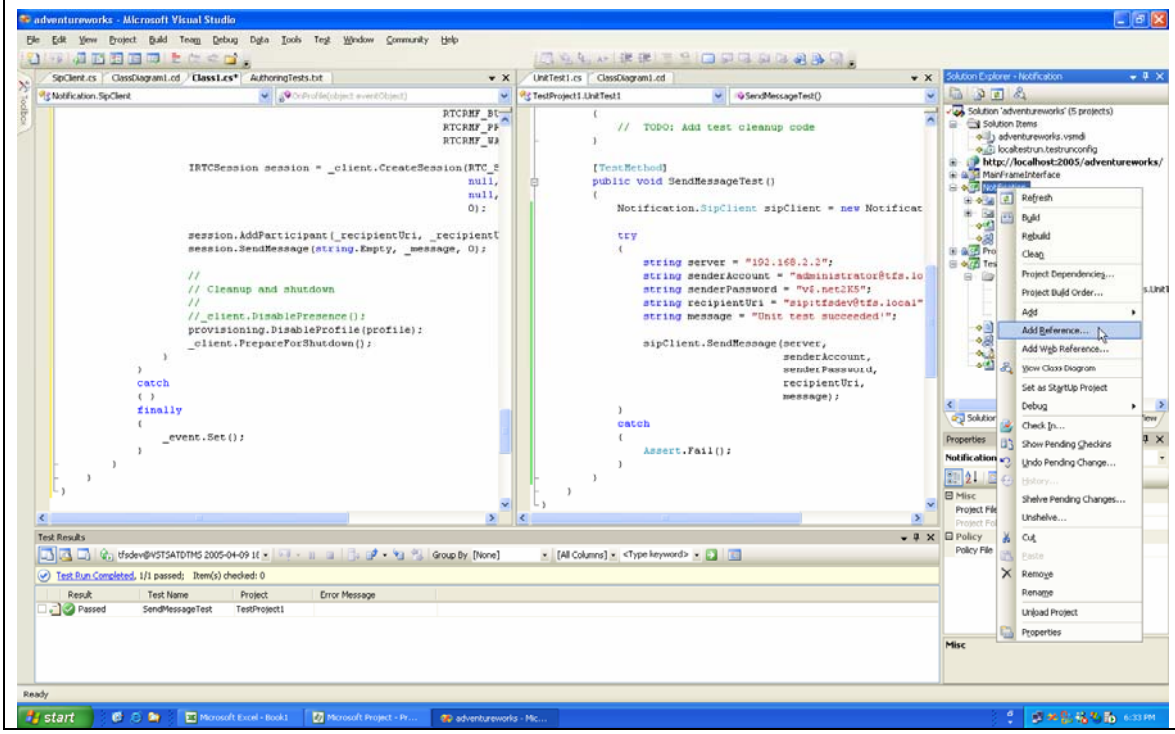
Actions	Click on the **Solution Explorer** tab



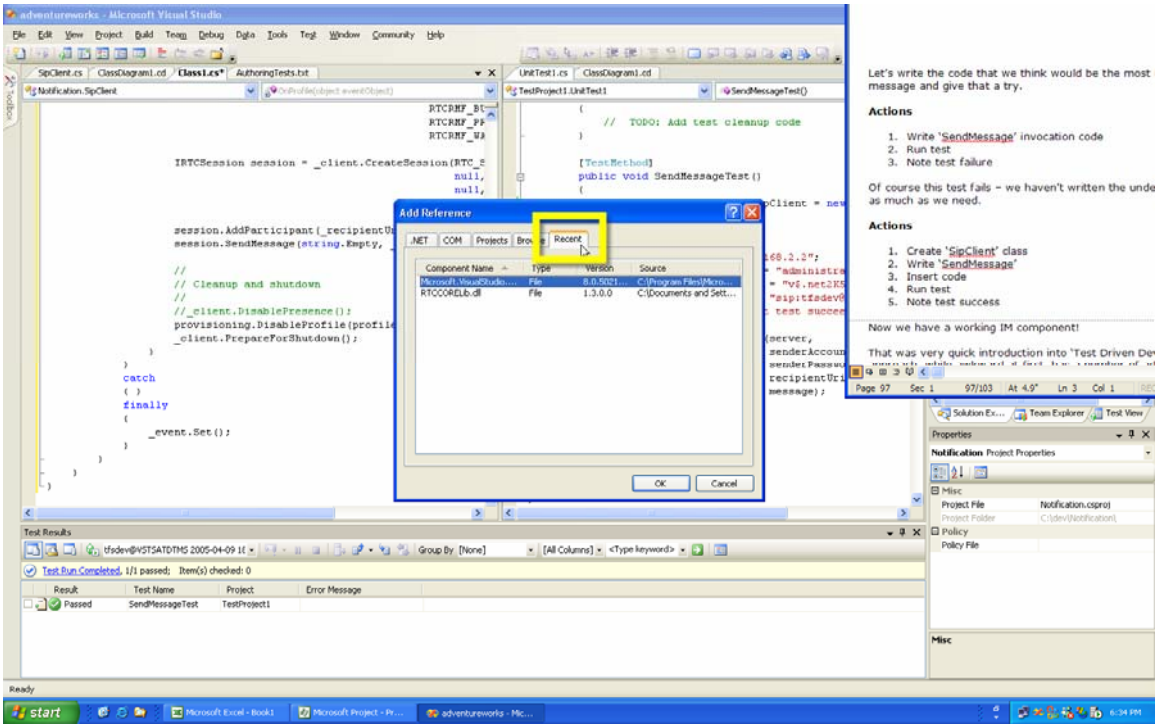
Actions	Select the **Notification** node



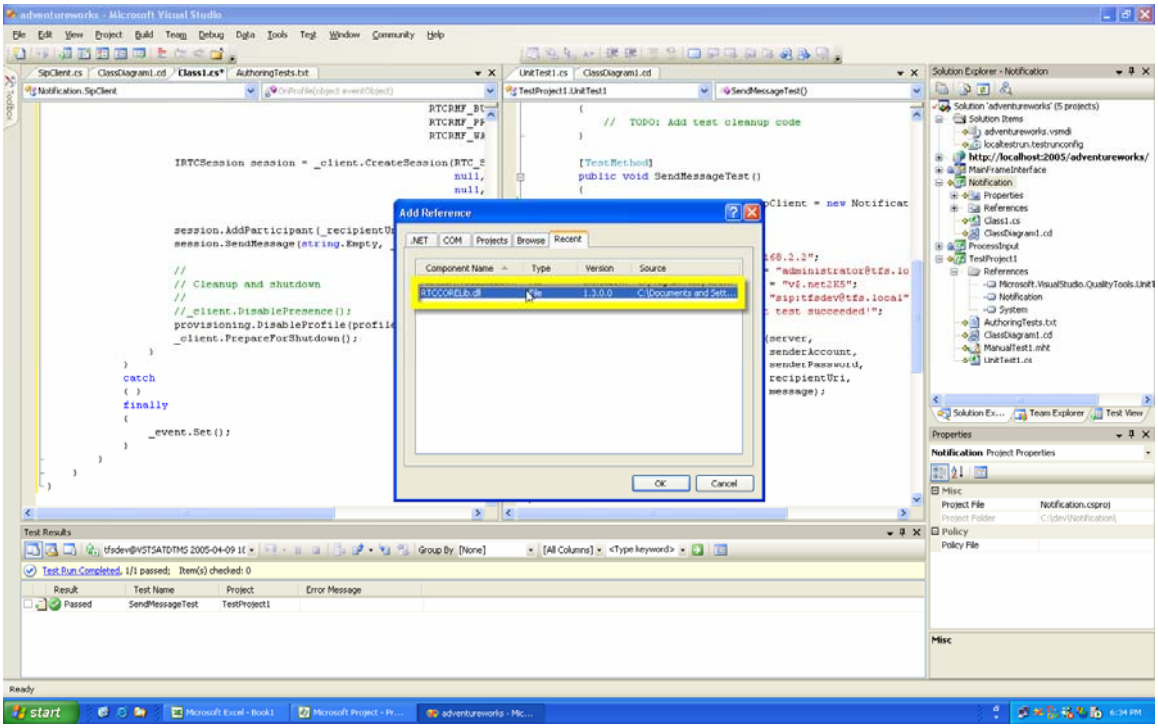
Actions	Right-click and choose **Add Reference**



Actions	Click on the **Recent** tab

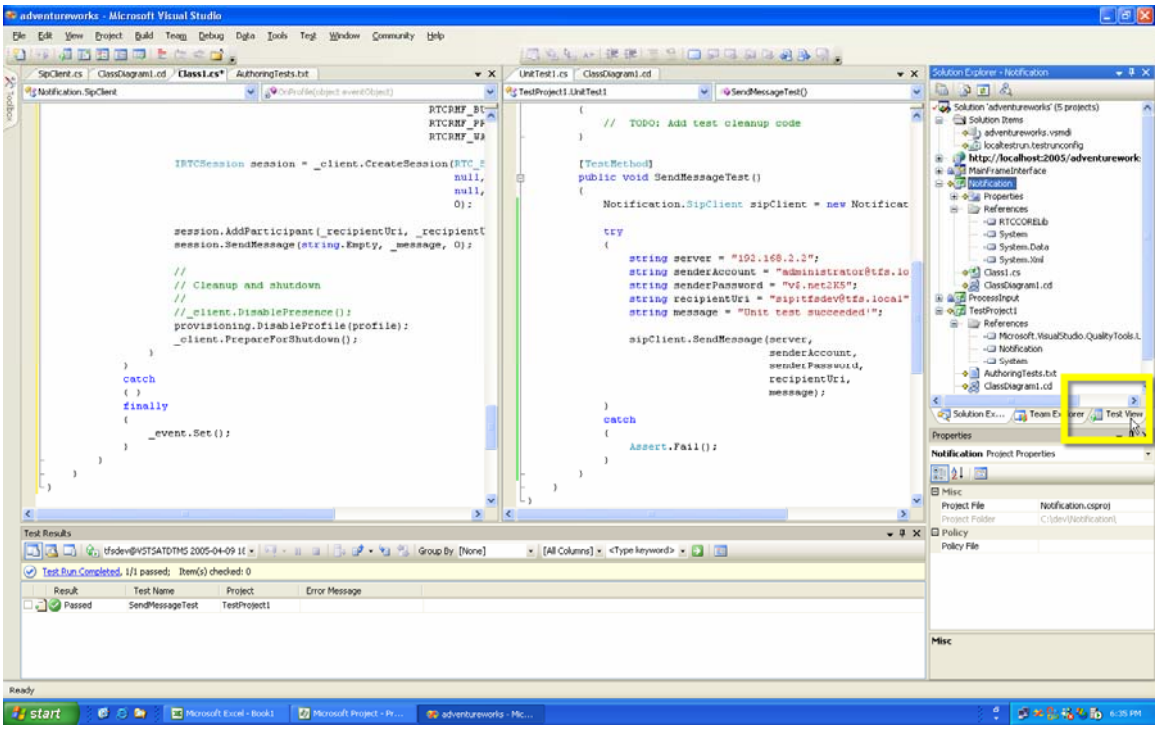


Actions	Select the **RTCCORElib.dll** item Press the **OK** button

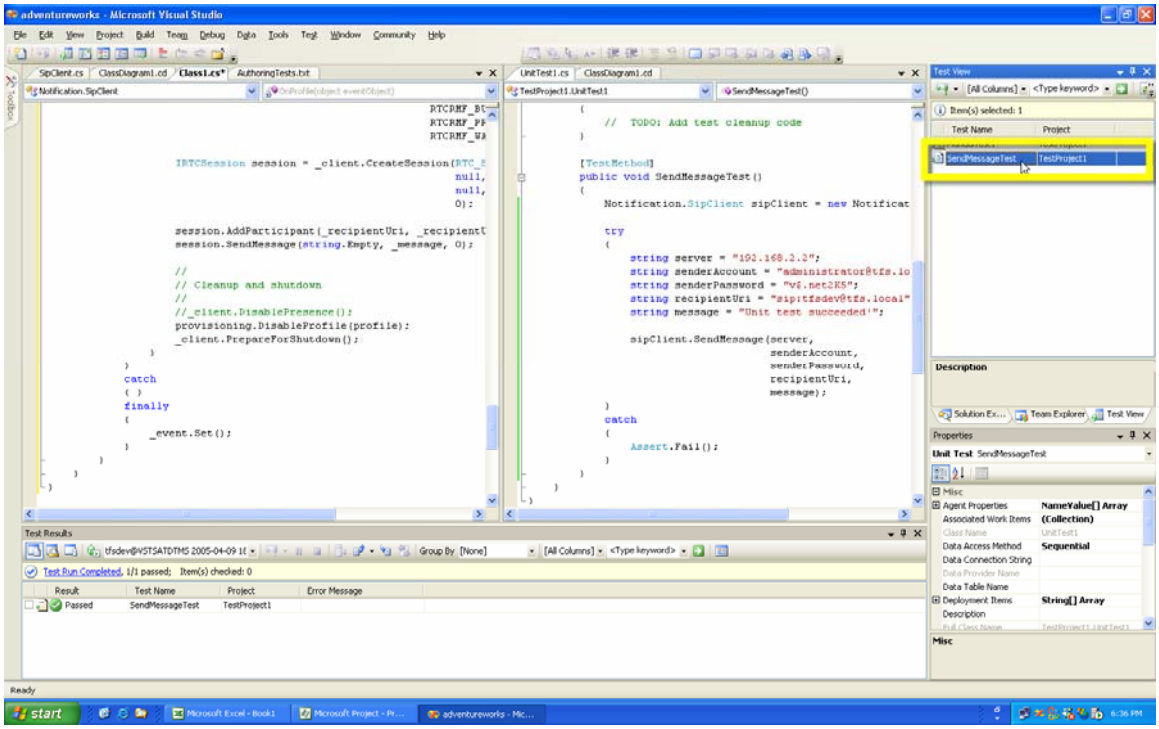


We have a real IM sending component written now; when we run our unit test this time, we are actually going to execute it.

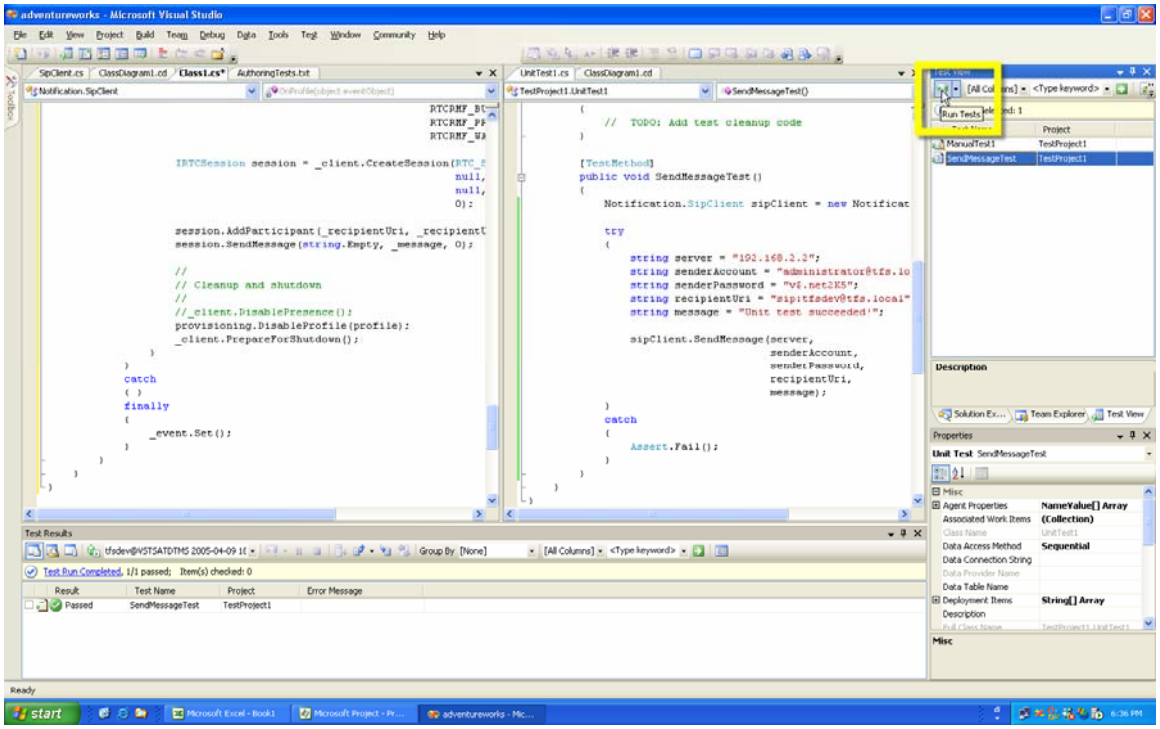
Actions	Click on the **Test View** tab



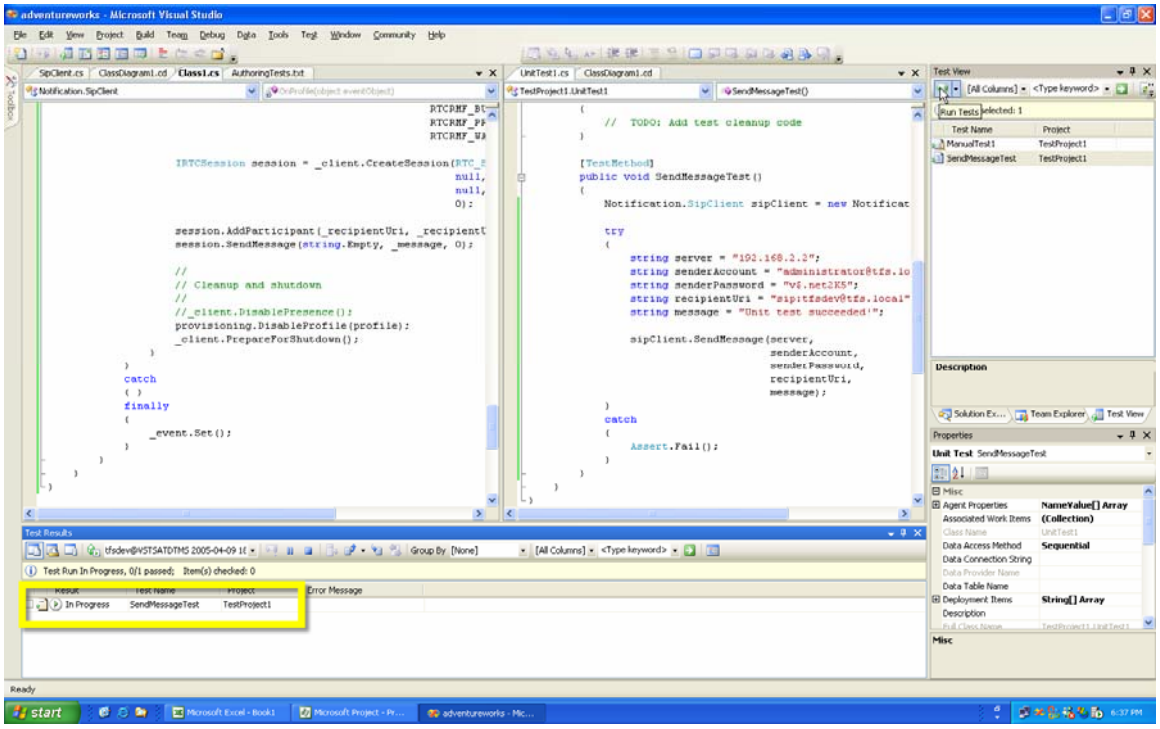
Actions	Select the **SendMessageTest** item



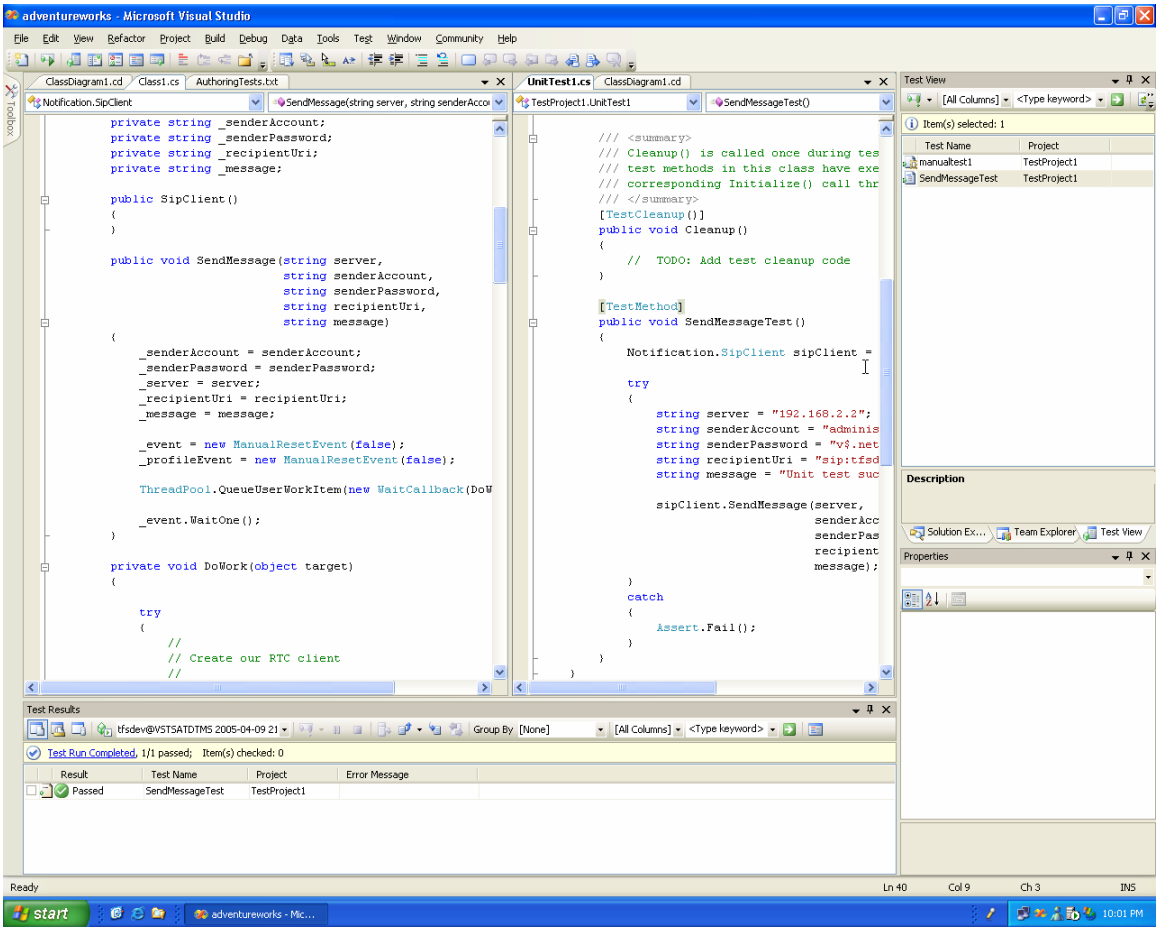
Actions	Press the **Run Tests** tool bar button



Actions	Watch as your tests run



Actions	Your test should pass and an IM message should be sent!



Now we have a working IM component!

That was very quick introduction into 'Test Driven Development' (TDD). This approach, while awkward at first, has a number of advantages.

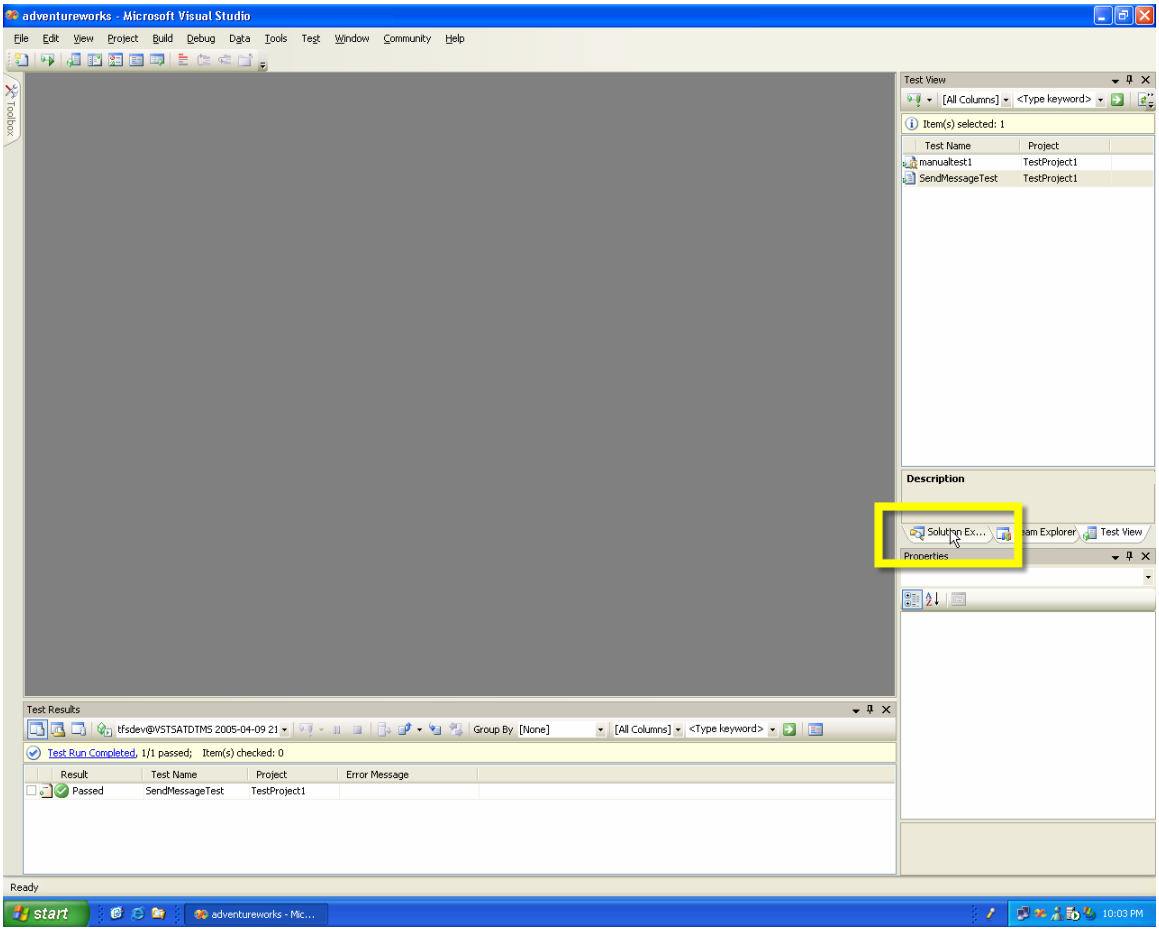
Writing code from the perspective of the end-user almost always results in an intuitive design. This code that you write can easily become your unit test. A more thorough interpretation of TDD involves more iteration – you write only the code needed to pass your test each time. This approach almost always results in a high degree of code coverage.

Exercise Adding our work to Source Control

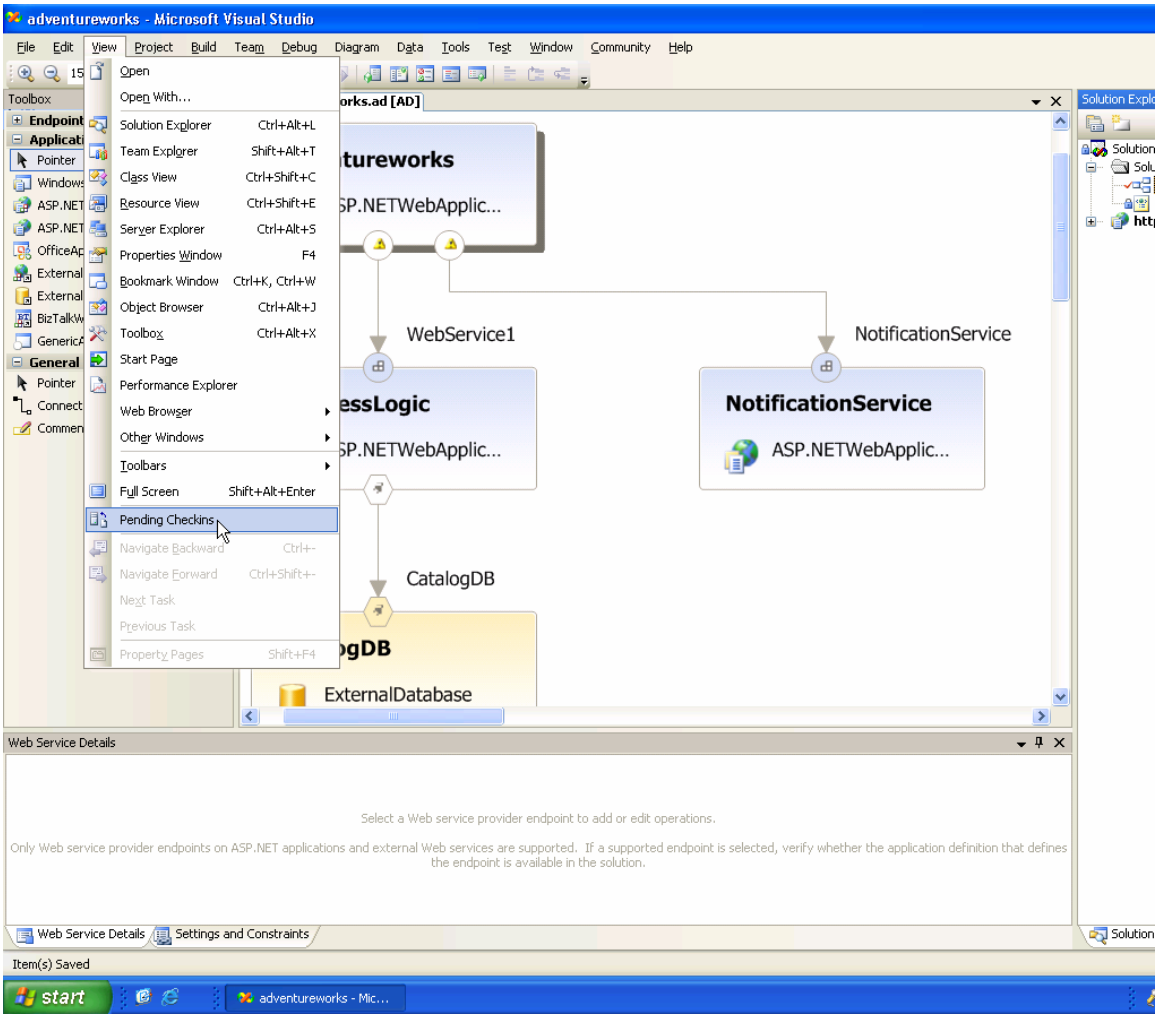
With our component written, it's time to safe guard our work and add it to source control. Visual Studio Team System's source control is built from scratch to be enterprise-ready.

Source control is a tightly integrated part of Visual Studio Team System; let's check in our changes and see this integration in action.

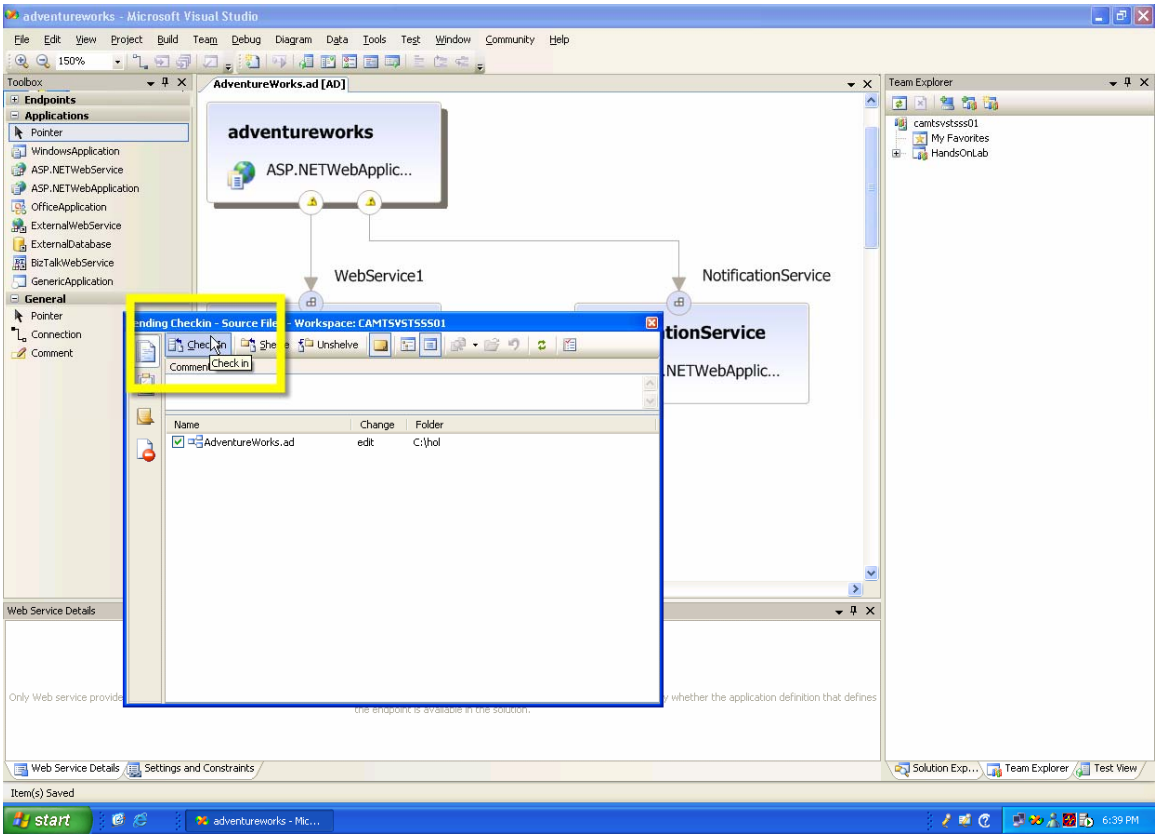
Actions	Click on the **Solution Explorer** tab



Actions	Choose **View -> Pending Checkins** menu option



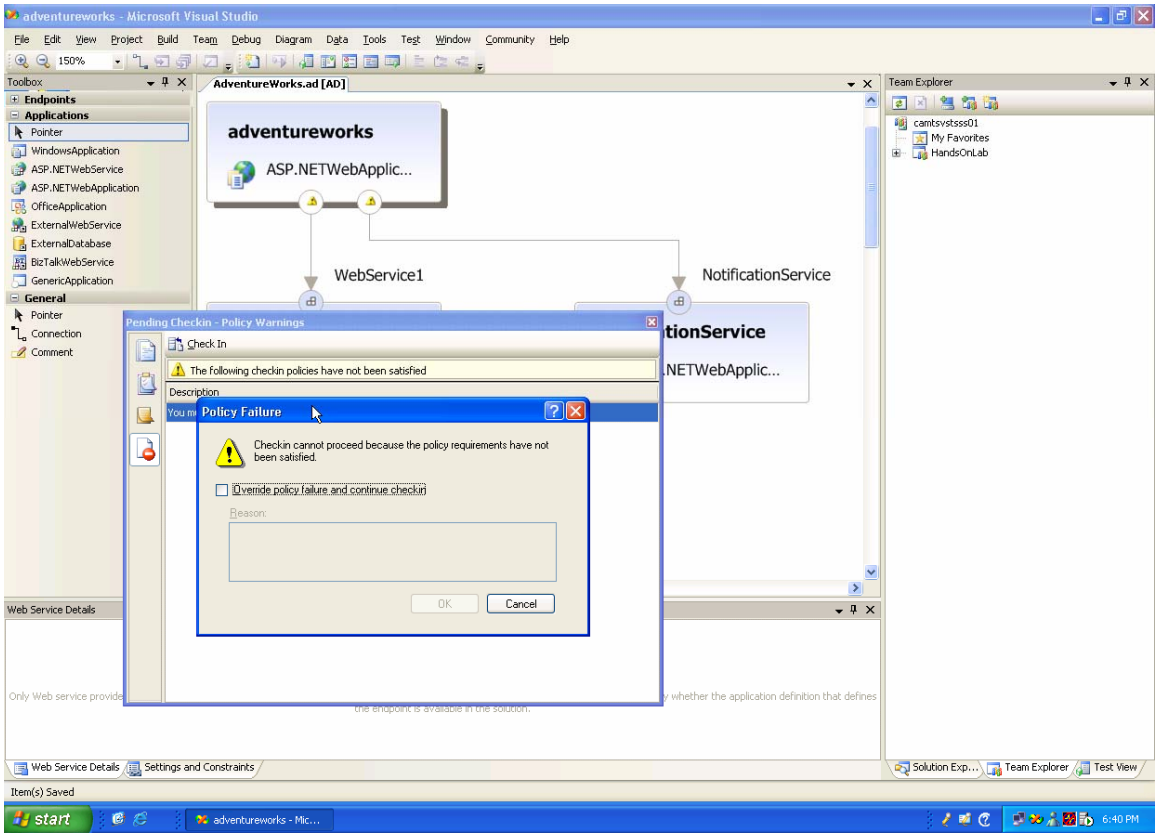
Actions	Press the **Check In** button



Visual Studio Team System's source control is tightly integrated with check-in policy. We can specify that certain conditions must be true before we are allowed to check code in.

Check in policy can be used to help make sure your organization is following a mandated development process.

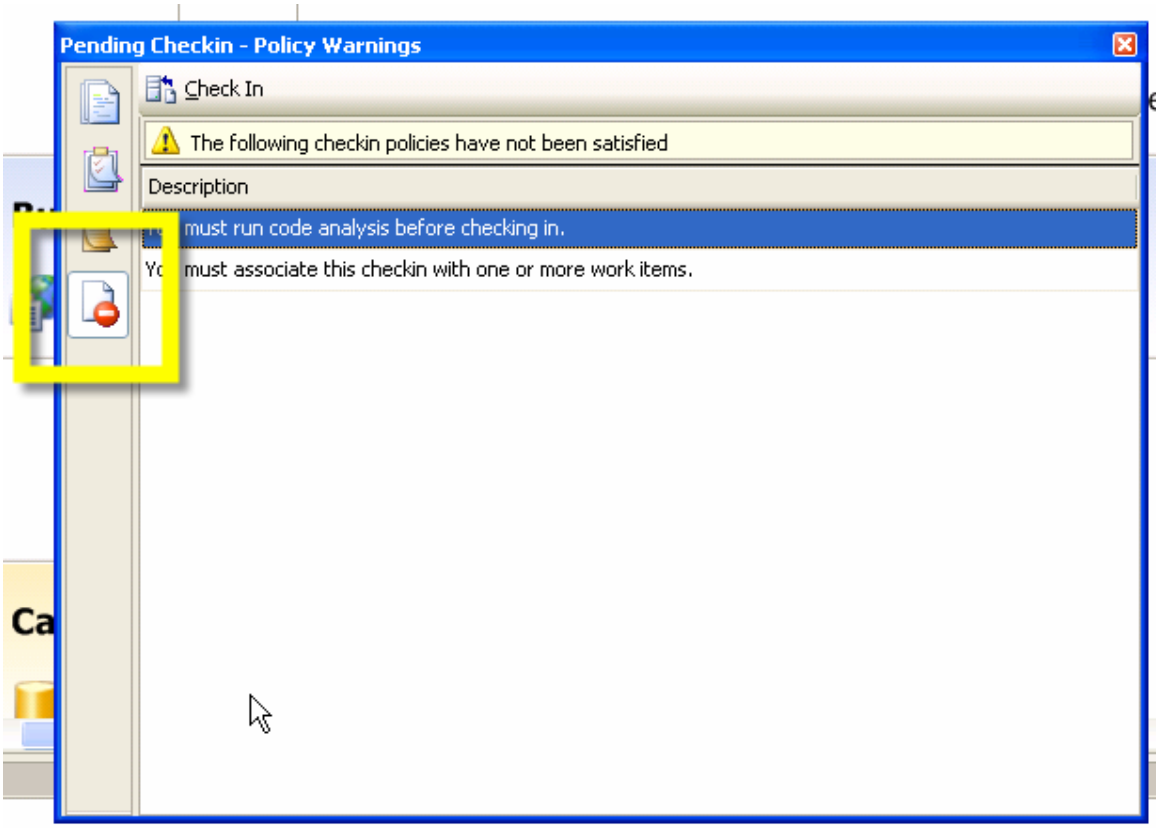
Actions	We've violated a couple of check in policies Press the **Cancel** button in the **Policy Failure** dialog box



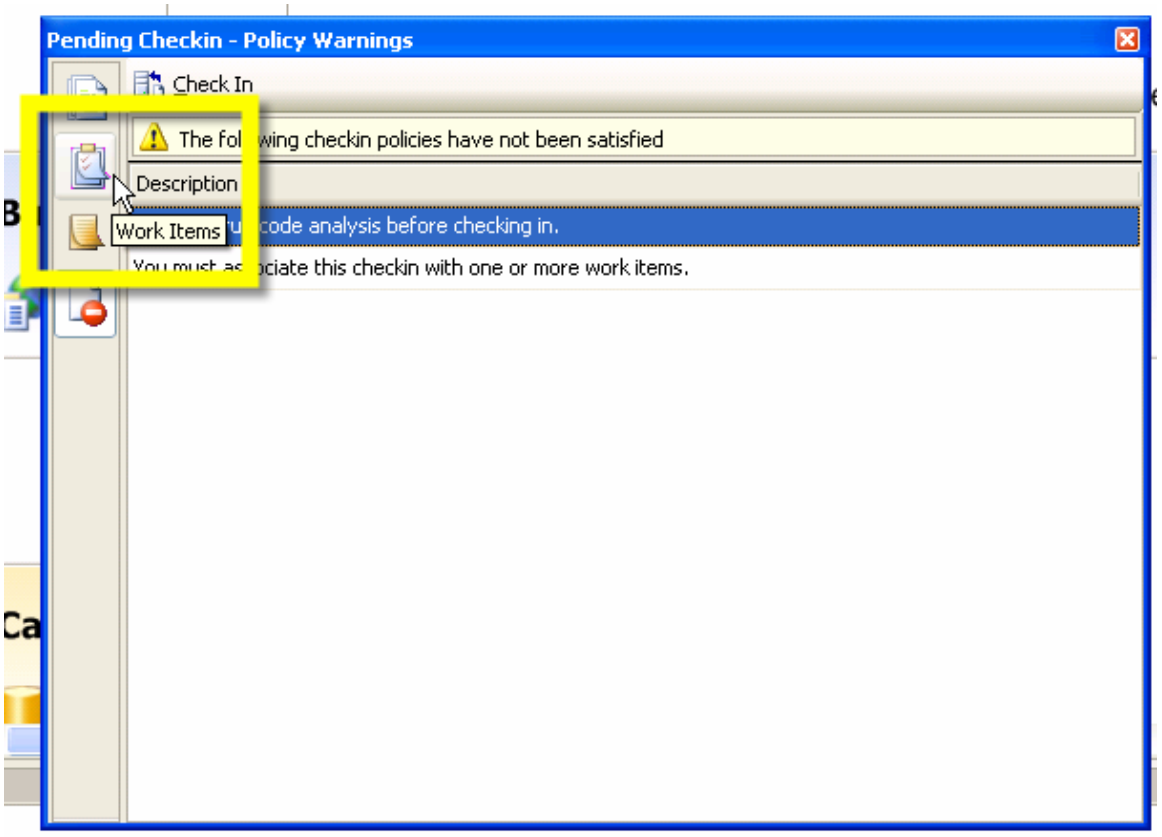
We haven't run code analysis or associated work items with our changes, so we can't check our changes in yet.

Let's take care of these policy violations, starting with the work items. We will associate this check in with the work item that we started this hands on lab with.

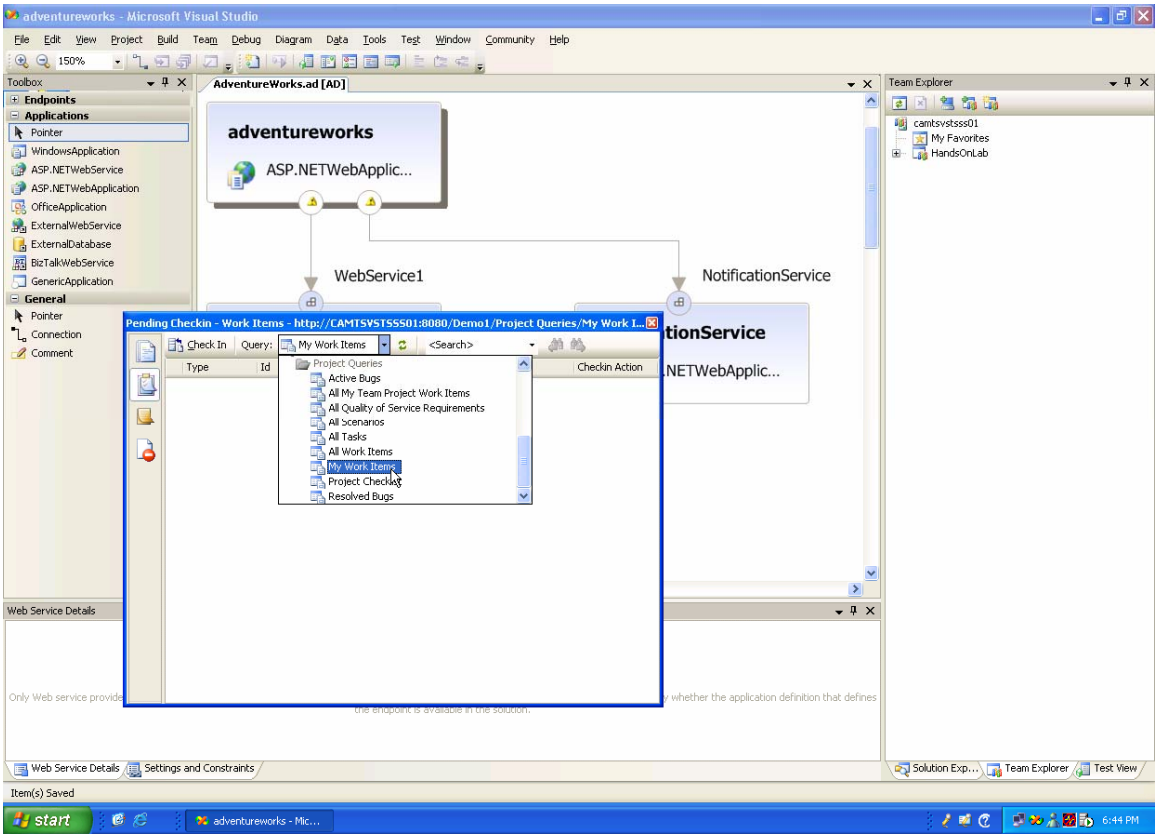
Actions	Press the **Work Items** button in the **Policy Failure** dialog box



Actions	Press the **Work Items** button



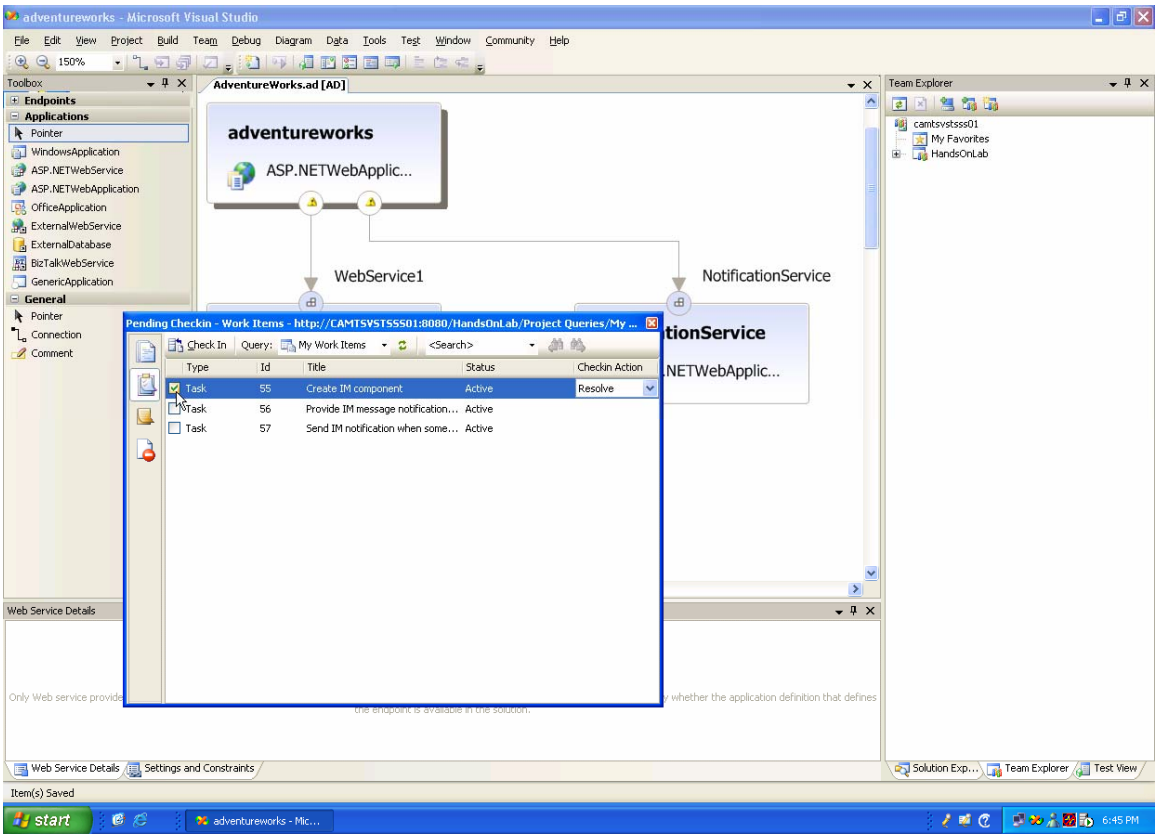
Actions	Select the **My Work Items** query under the **HandsOnLabs** project



Our work items policy has been satisfied, now let's run code analysis to resolve the second policy violation.

Code analysis is like a grammar checker for your source code; in the next section, we will see how it can flush out potential problems in our source code.

Actions	Select the **Create IM Component** task Go back to the **Policy Warnings** and see that one violation has been satisfied Close the **Pending Checkins** dialog box – we are going to run **Code Analysis** to resolve the remaining violation



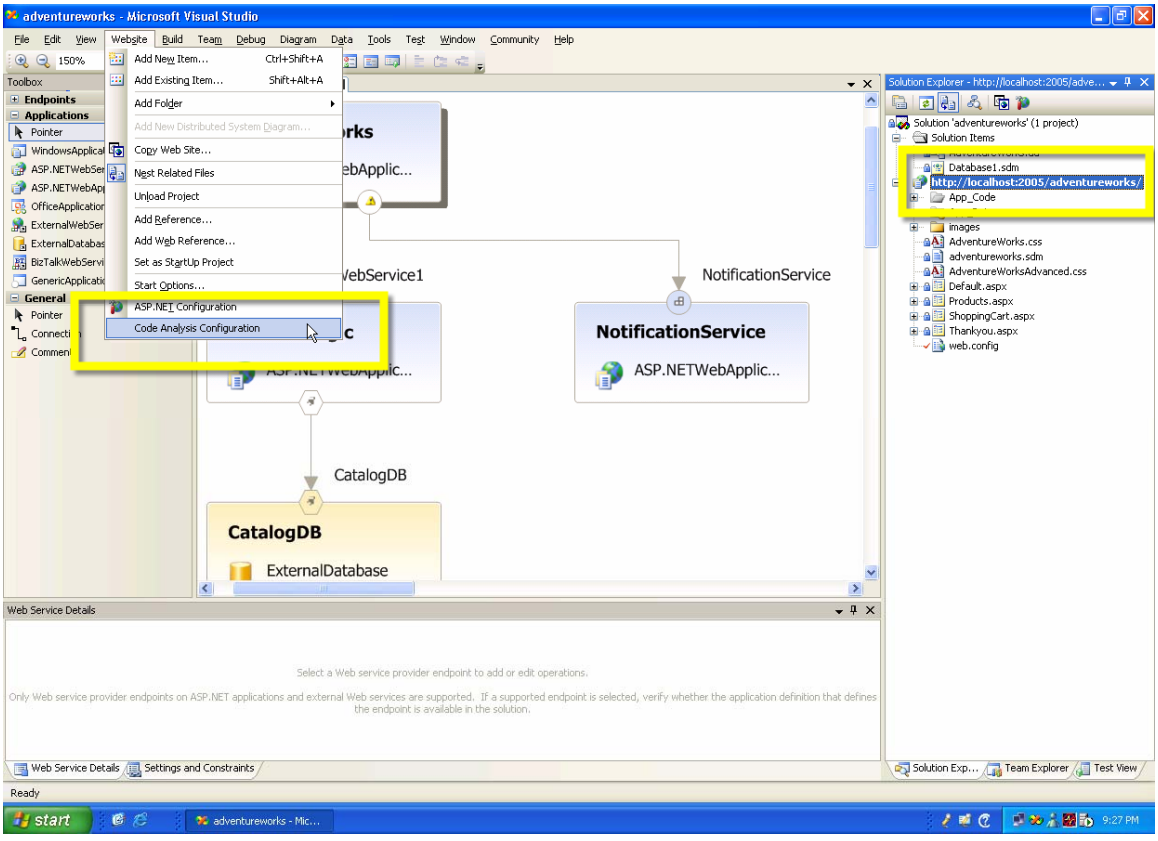
Exercise - Code Analysis

Microsoft has learned some really good lessons about security over the years – these lessons are integrated right into Visual Studio Team System.

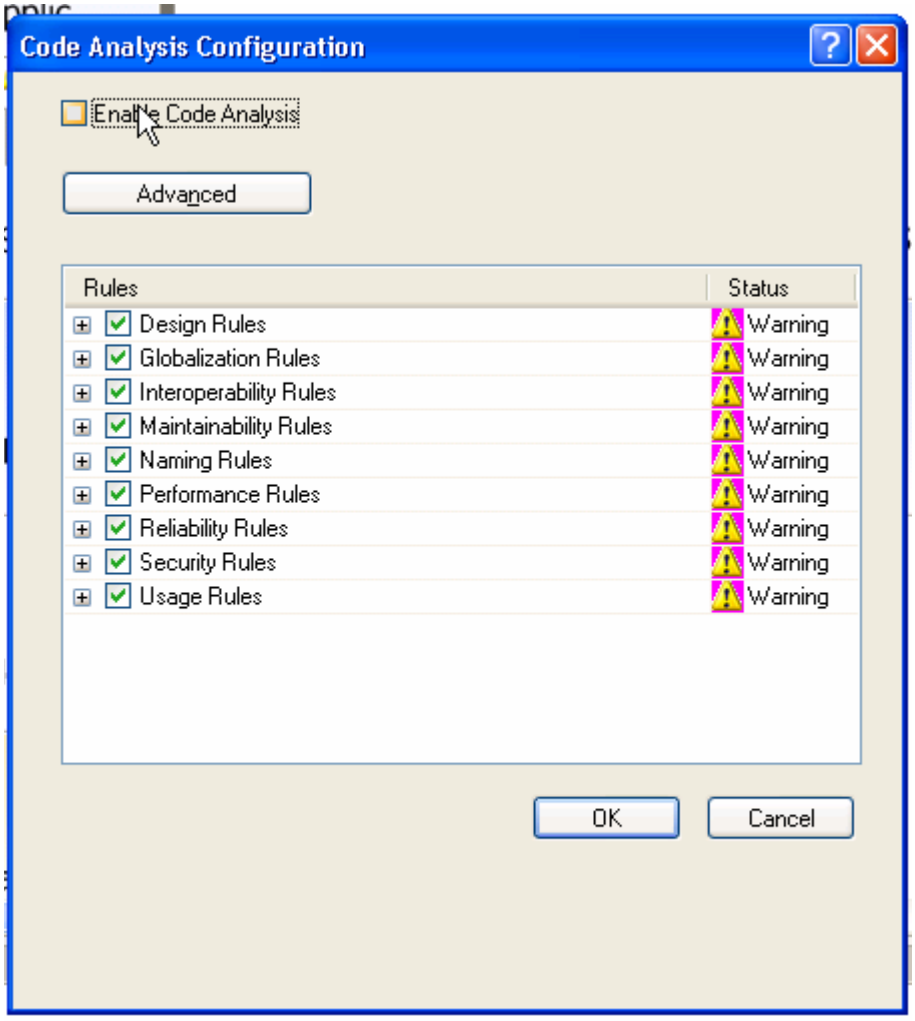
We can build my project with Visual Studio Team System and have all of our source code analyzed for me. Known problems, such as security problems, get flushed out and shown to us.

Code analysis is an integrated part of Visual Studio Team System; we just have to enable it for our project.

Actions	Make sure that you have your **http://localhost:2005/adventureworks** project selected. Choose the **Website -> Code Analysis Configuration** menu option (the **Website** menu is not shown until you select a web project)

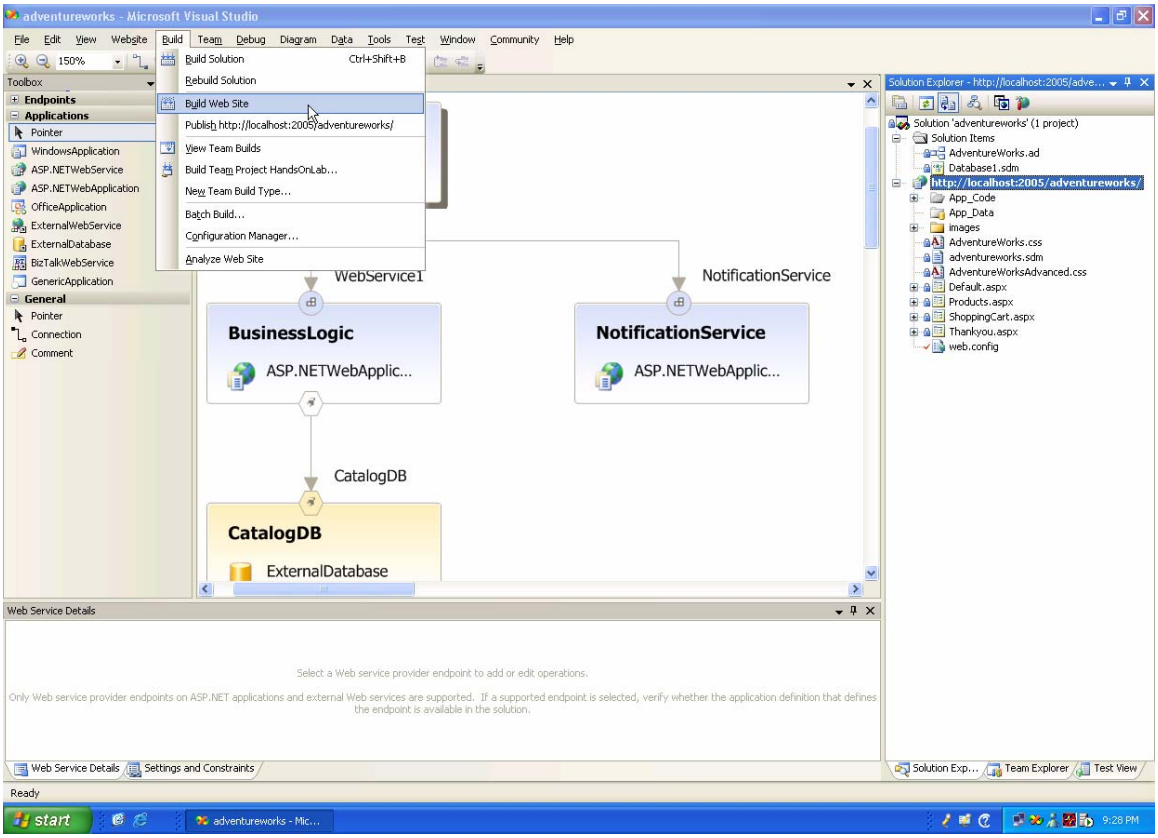


Actions	There are a number of code analysis rules that we can enable; in our case, let's enable all of them. Select the **Enable Code Analysis** check box Press **OK**

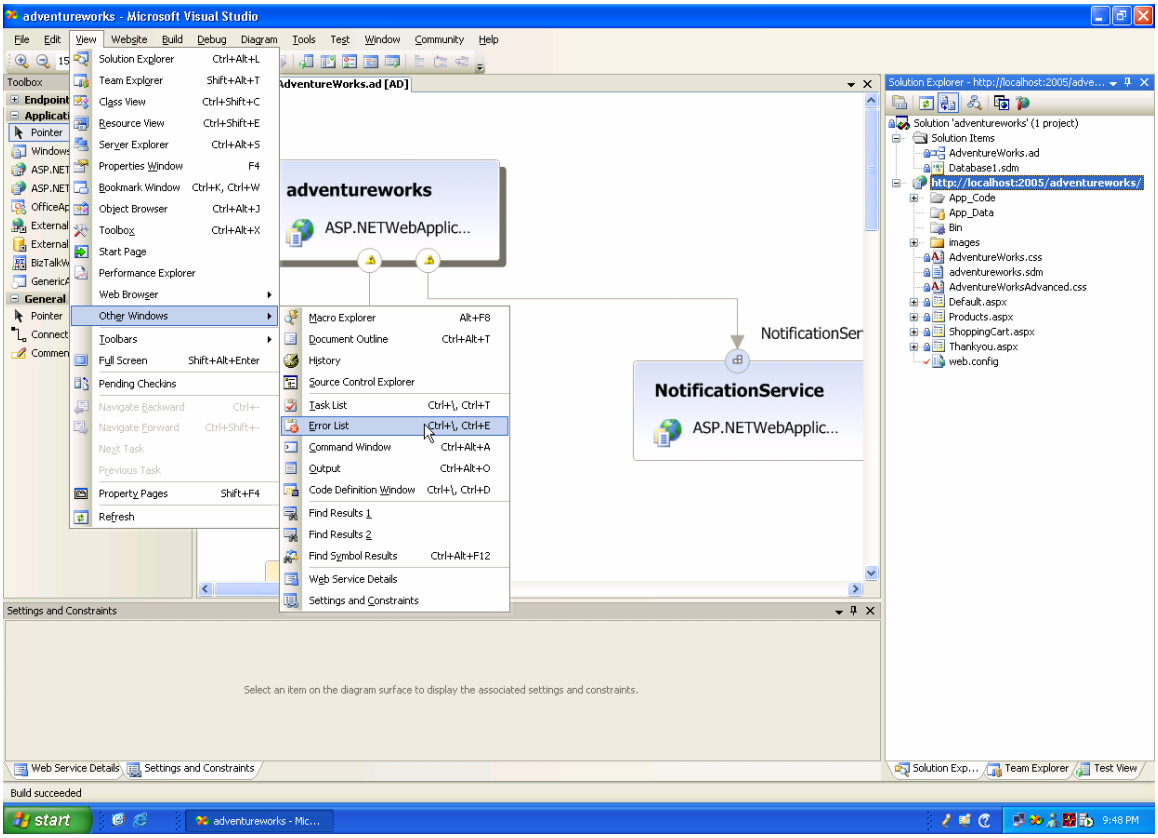


With code analysis enabled, we will rebuild our project. Instead of getting compiler errors (if there are any), we will also get warnings about our source code where common problematic patterns are being recognized.

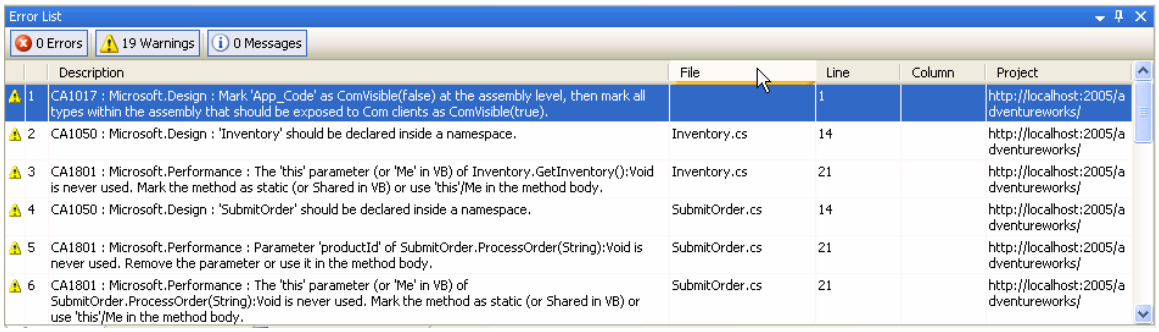
Actions	Choose the **Build -> Build Web Site** menu option



Actions	Choose the **View -> Other Windows -> Error List** menu option



Actions	The **Error List** shows all of the warnings that code analysis flushed out

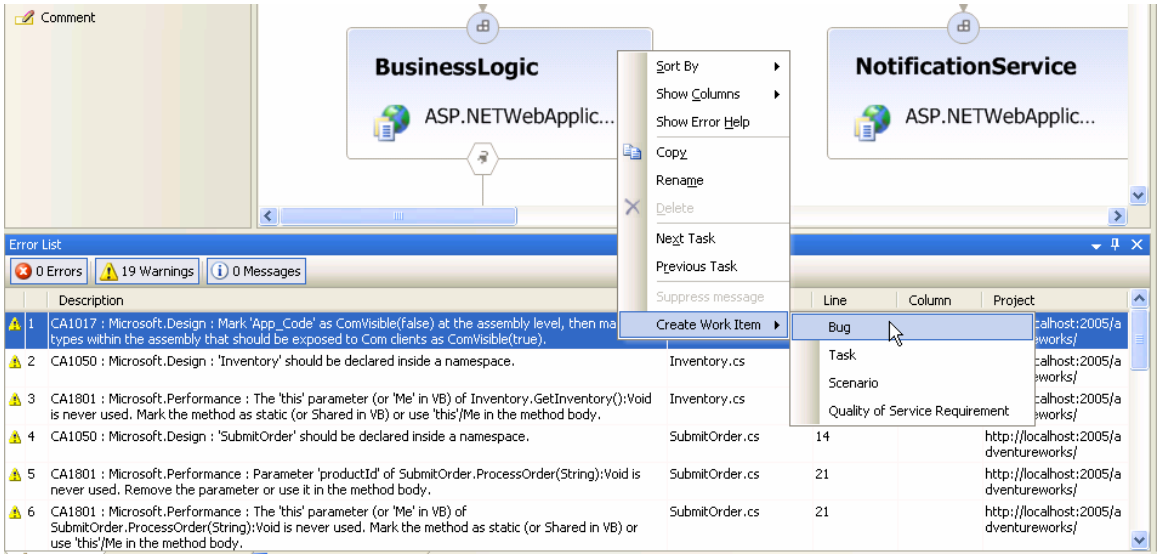


We won't fix all of these issues right now; the integration in Visual Studio Team System allows us to quickly convert these warnings into bugs.

Visual Studio Team System will detect that we have run code analysis and allow our check in to proceed.

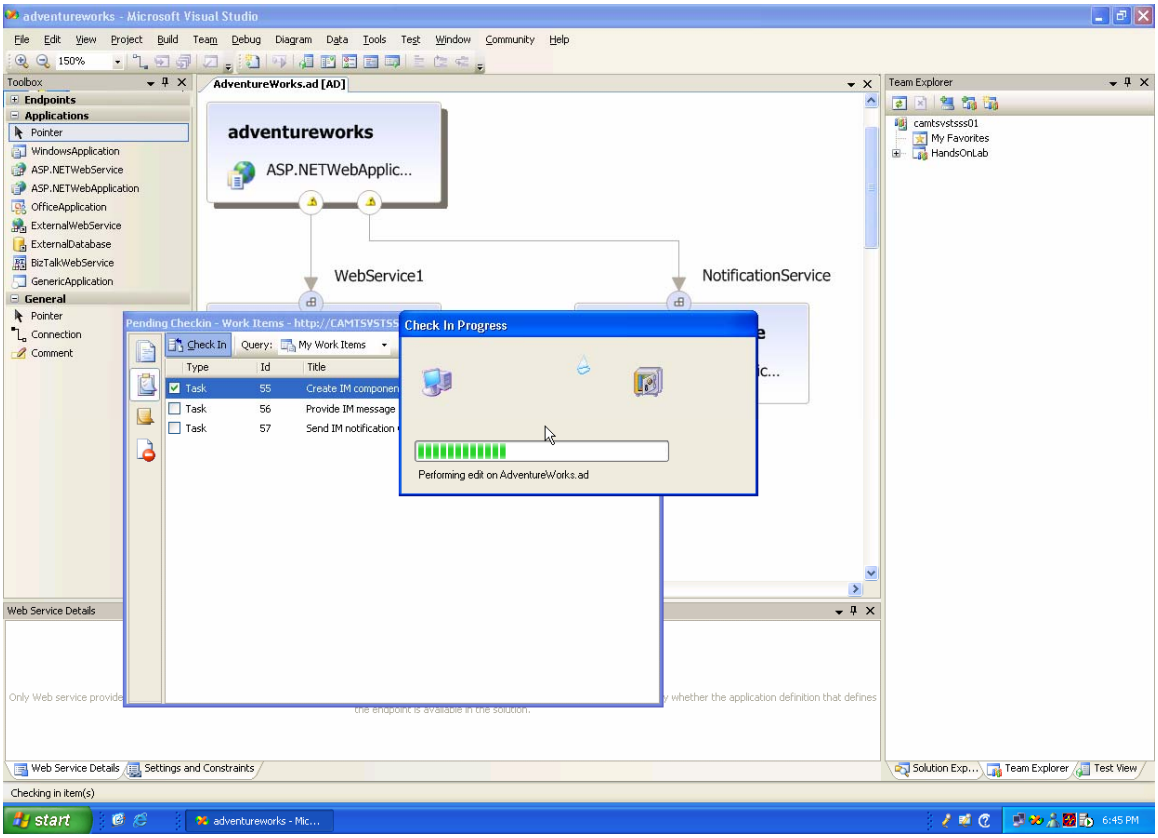
Visual Studio Team System provides a set of very useful check in policies; this mechanism is extensible as well. In the next section, we will create our own check in policy and integrate it with Visual Studio Team System.

Actions	Right-click on the warnings and choose **Create Work Item -> Bug** Remember to save the bugs that you create



Actions	Bring up the **Pending Checkins** dialog box again If the code analysis warning is still there, select it, right-click and choose **Evaluate**

Actions	Press the **Check In** button again and wait for files to be added



Exercise - Custom Check-in Policy

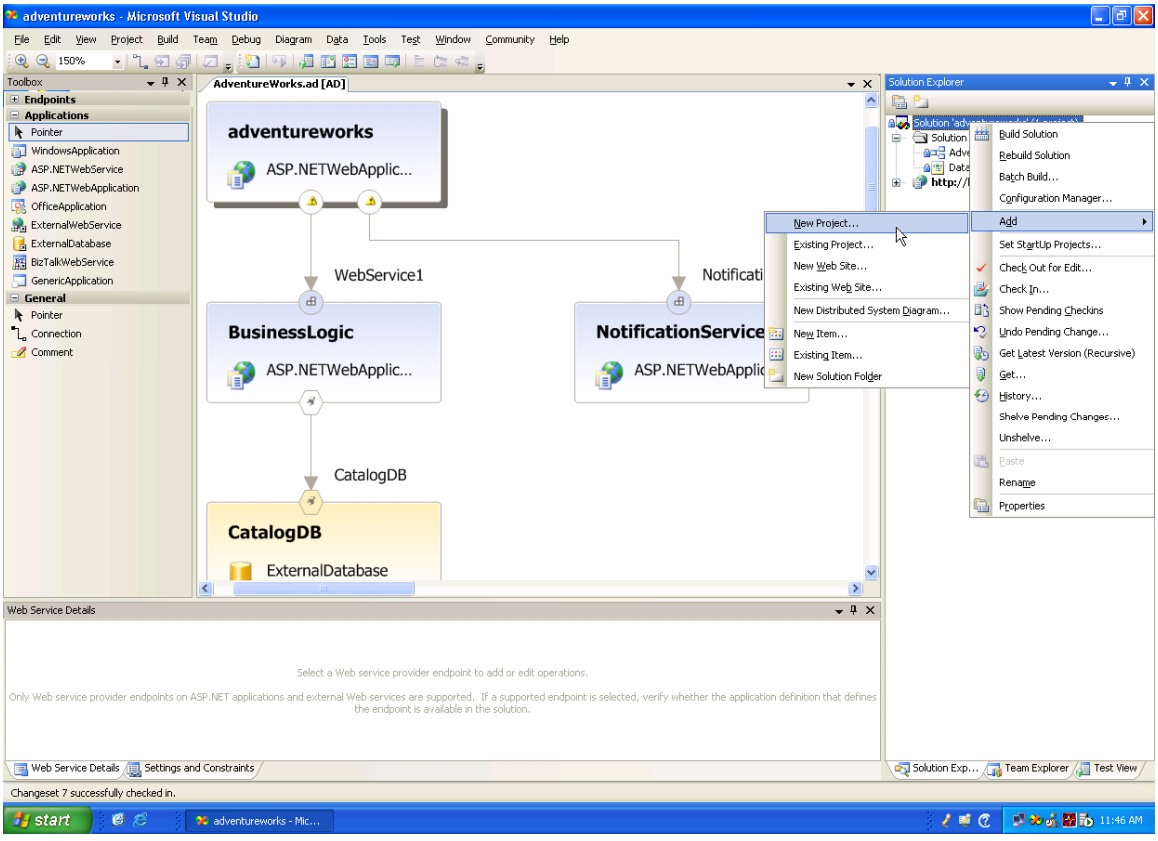
One of the core design principles behind Visual Studio Team System was to design its functionality as a platform, rather than just a product. This would allow customers to tweak Team System to suite their organizations, as well as allow partners to build their businesses upon Team System.

We will create a check in policy that makes sure we add a comment to each check in.

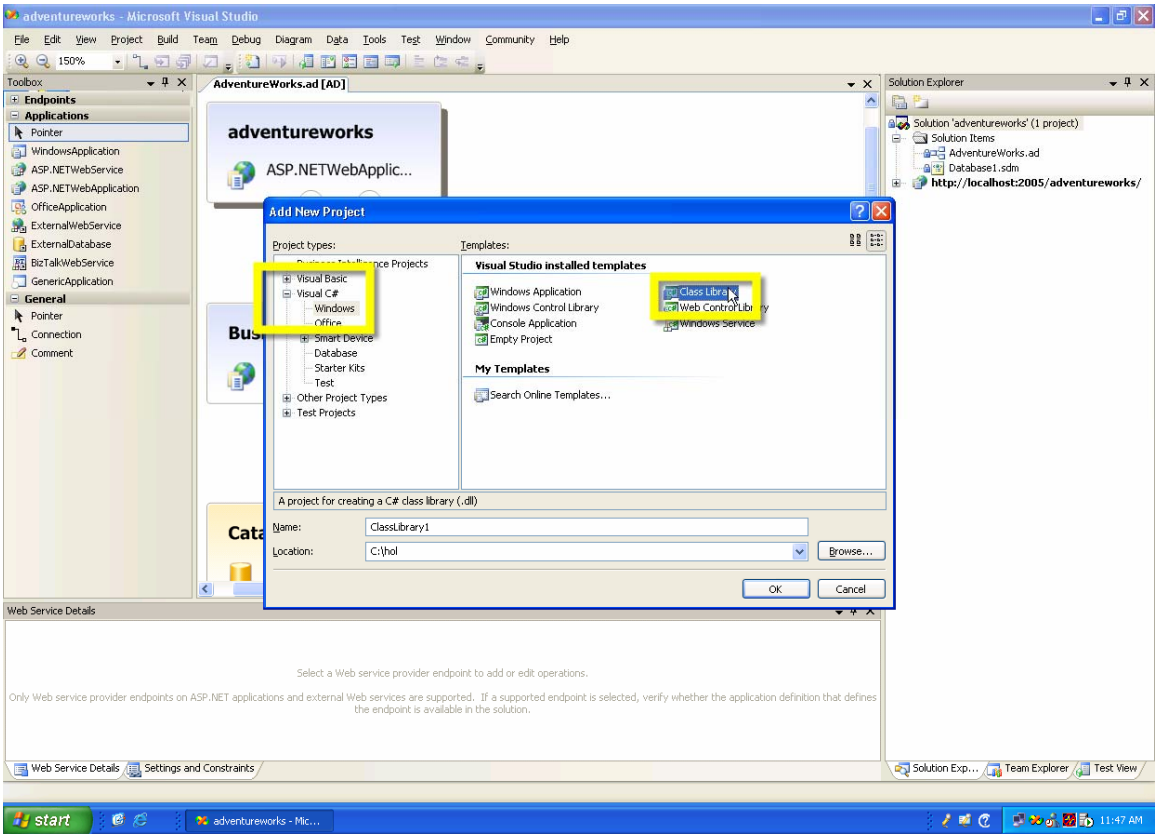
Creating custom check in policies is quite simple – they are simply .NET assemblies that implement certain interfaces. Team System is made aware of these policies through a registration mechanism.

Let's start by creating our .NET assembly project.

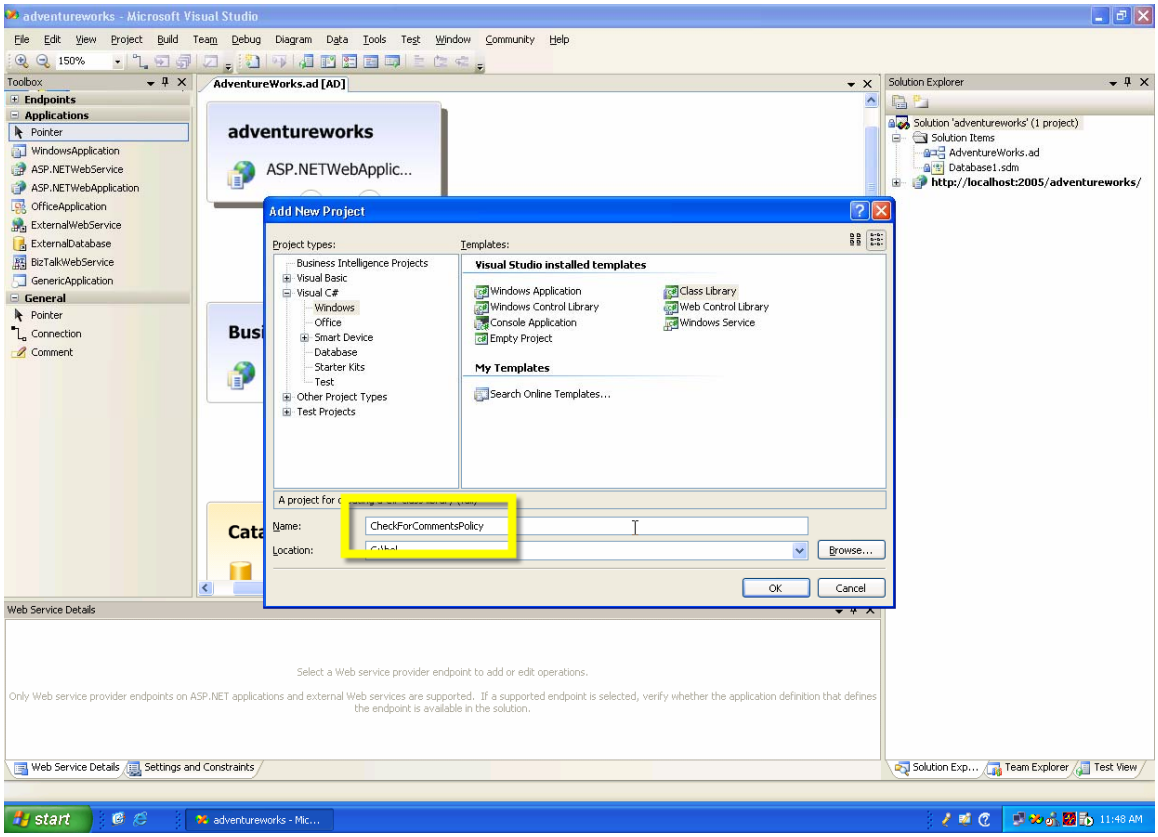
Actions	From the **Solution Explorer** right-click on your **adventureworks** solution node and choose **Add -> New Project**



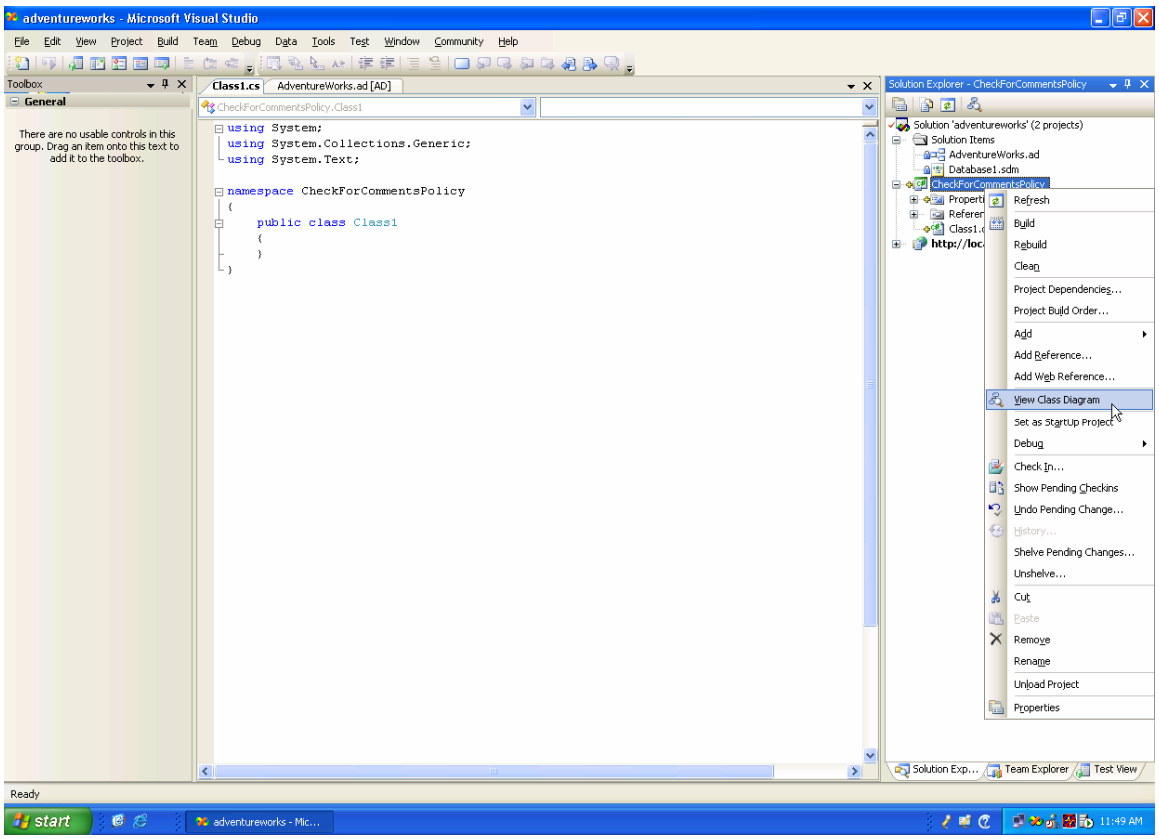
Actions	Create a **Visual C# -> Windows -> Class Library** project



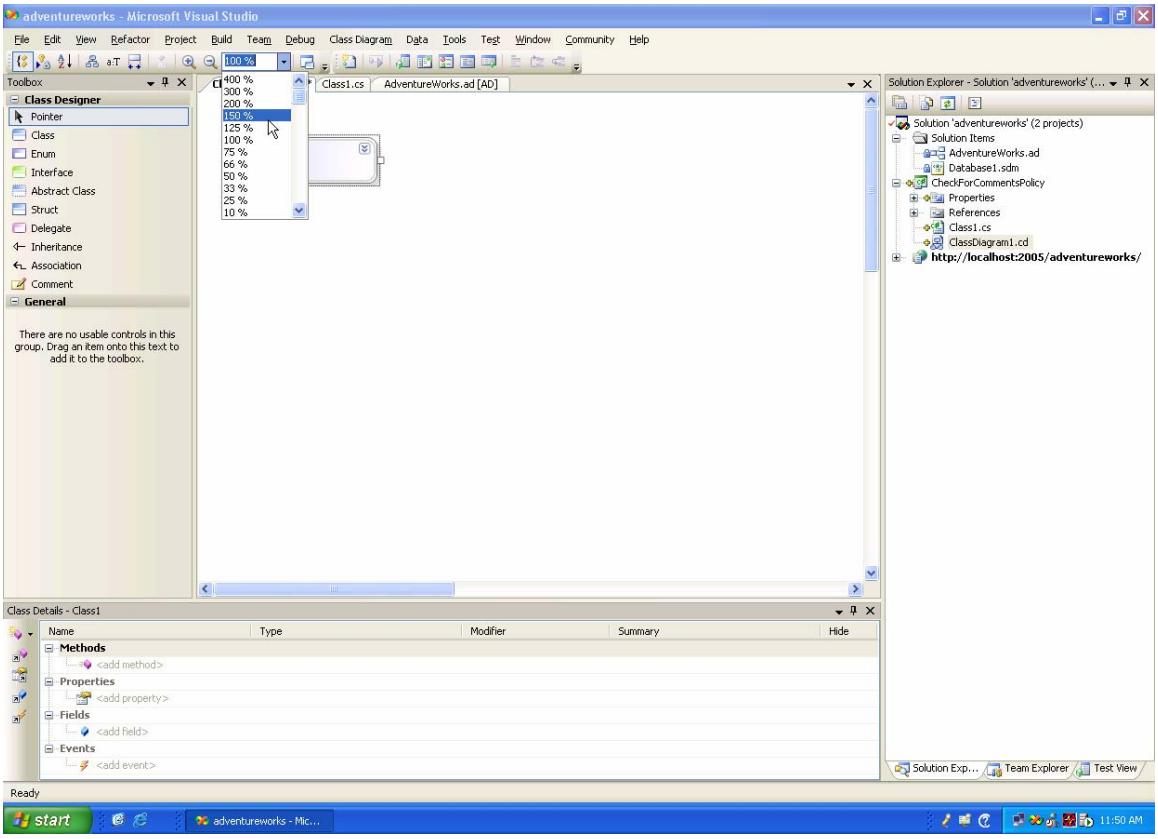
Actions	Name this project **CheckForCommentsPolicy** Press **OK**



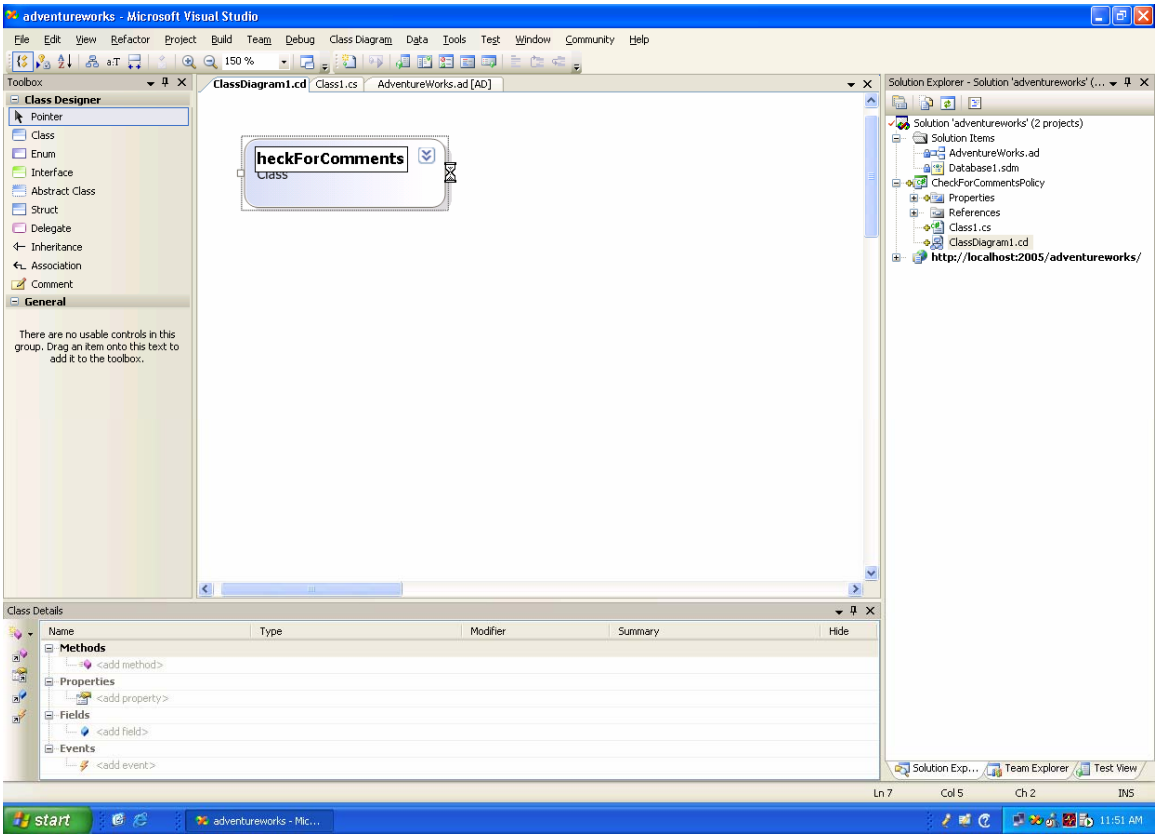
Actions	From the **Solution Explorer** right-click on your new **CheckForComments** project and choose **View Class Diagram**



Actions	Set the **Zoom** to **150%**



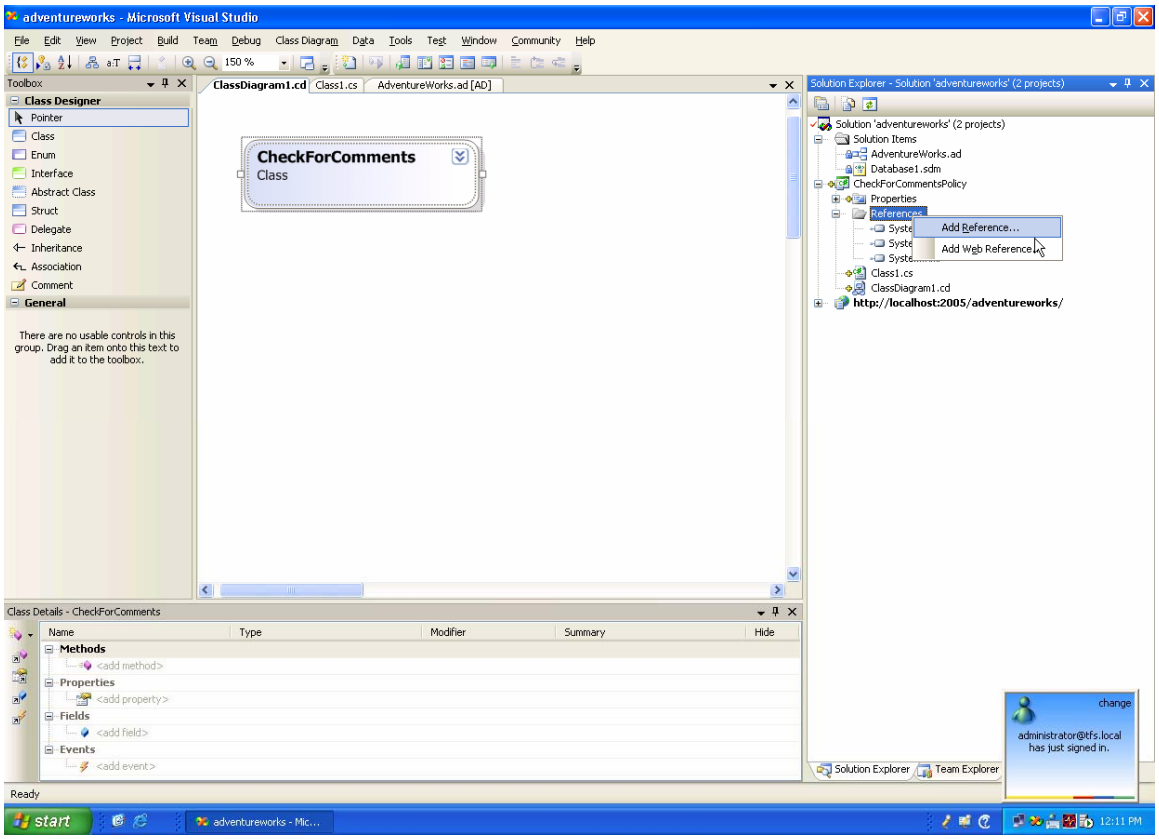
Actions	Rename your class from **Class1** to **CheckForComments**



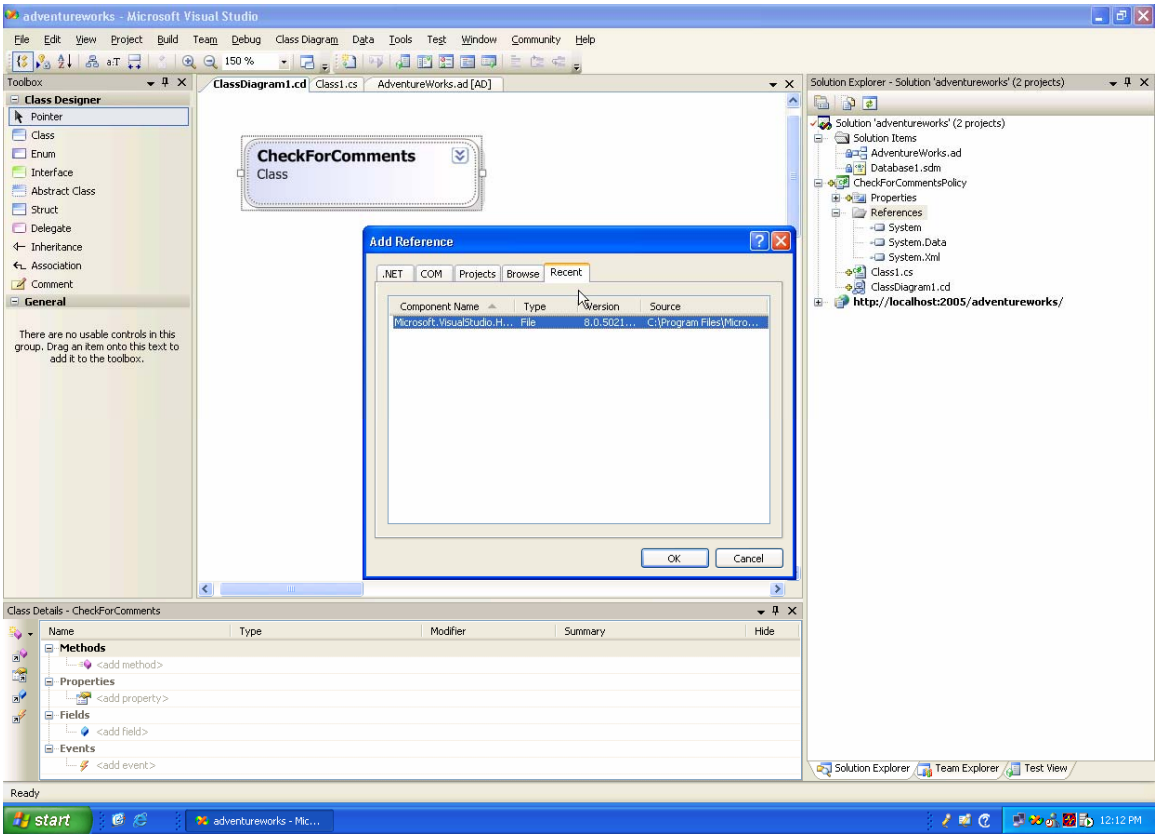
The interfaces that we need to implement to become a check in policy are exposed through an object model that Team System provides. The Class Designer will help us implement those interfaces by generating default implementations for us.

Let's add a reference to the object model that we need, then we will implement the required interfaces.

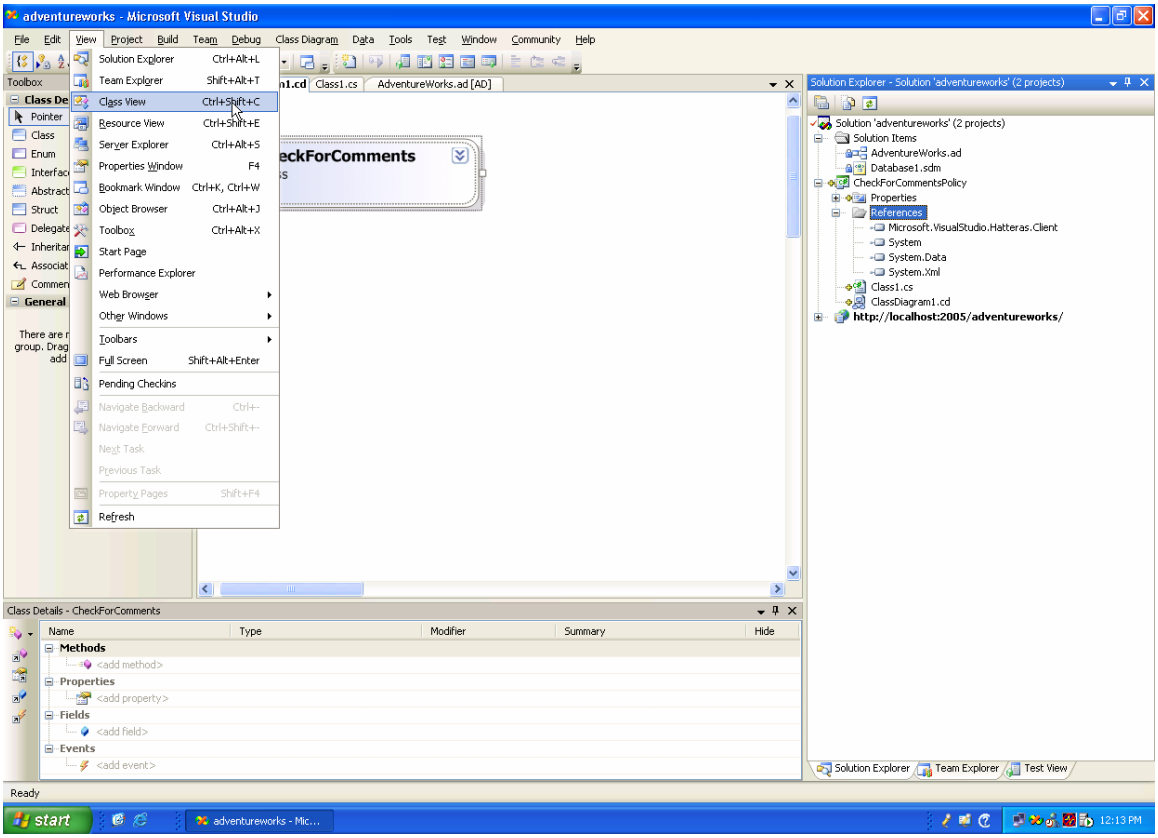
Actions	Right-click on the **Reference** node under the **CheckForCommentsPolicy** project node Choose **Add Reference**



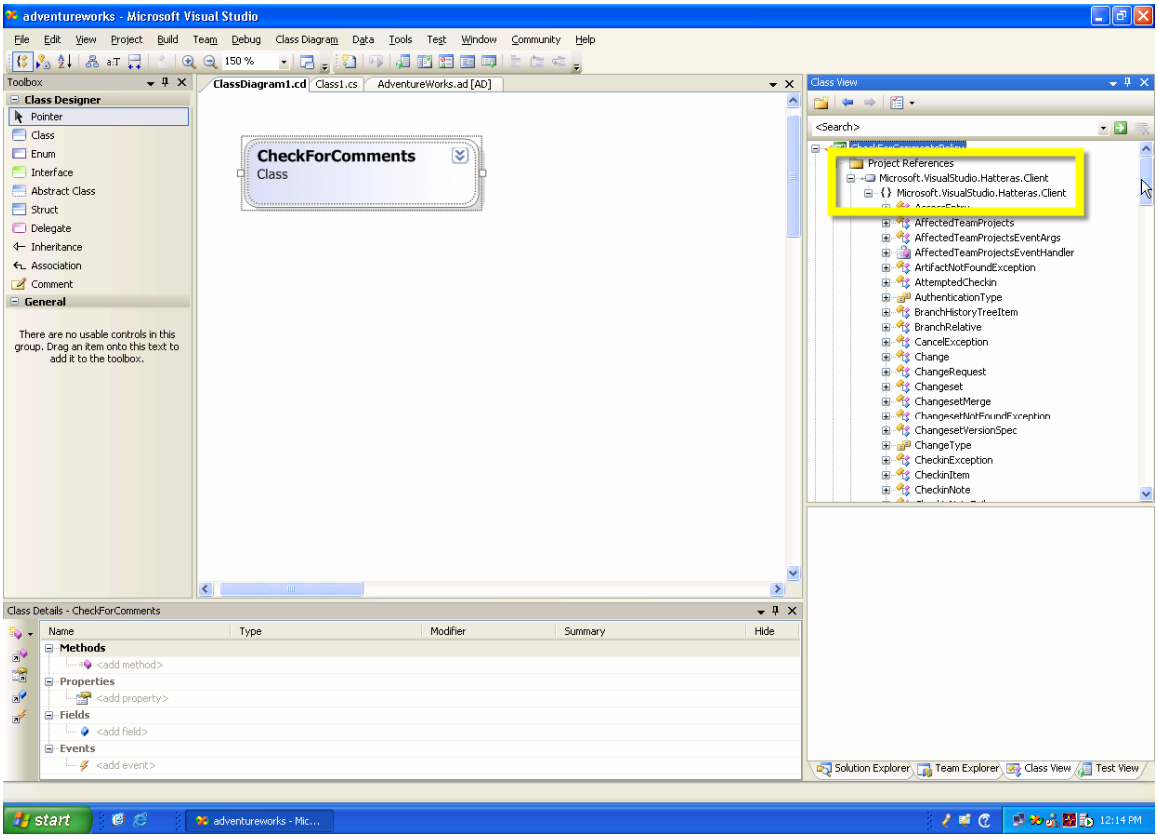
Actions	Click on the **Recent** tab Choose the **Microsoft.VisualStudio.Hatteras.Client.dll** file Press **OK**



Actions	Choose the **View -> Class View** menu option

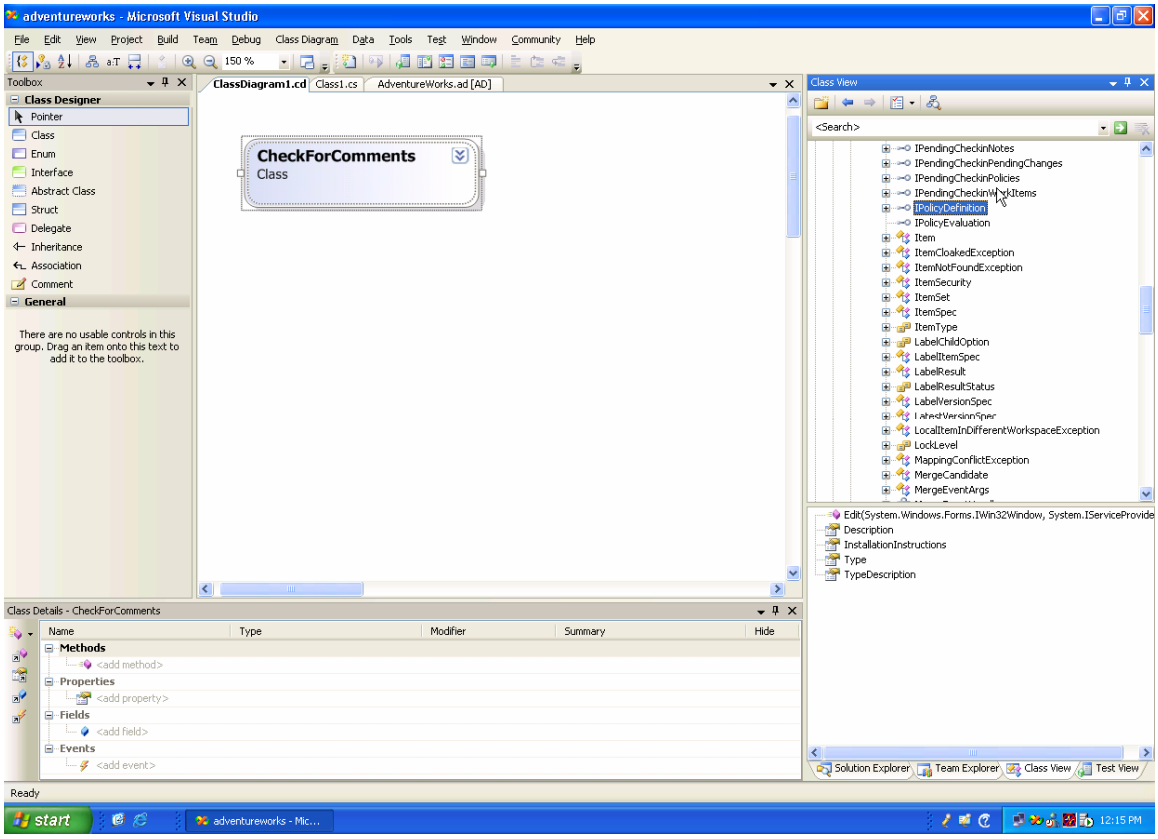


Actions	Expand the **Microsoft.VisualStudio.Hatteras.Client** nodes as shown

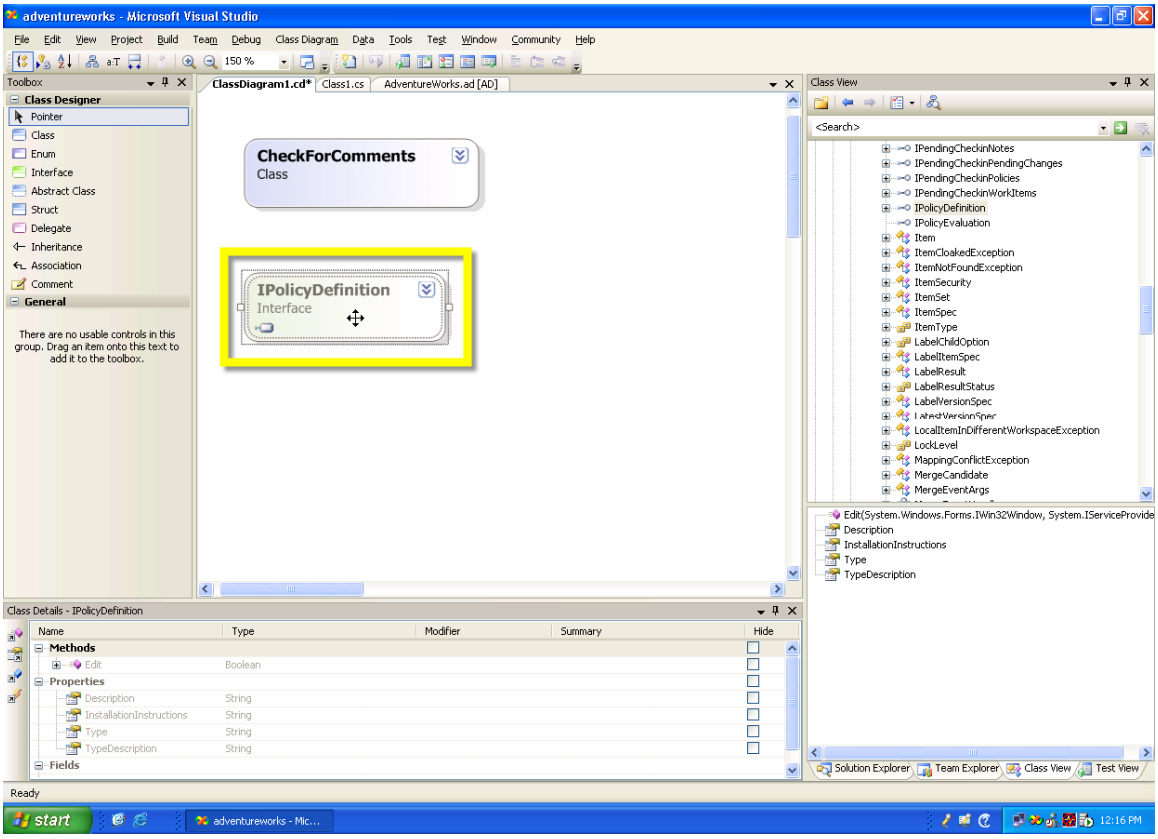


IPolicyDefinition and IPolicyEvaluation are the interfaces that we need; Class Designer will help us implement them.

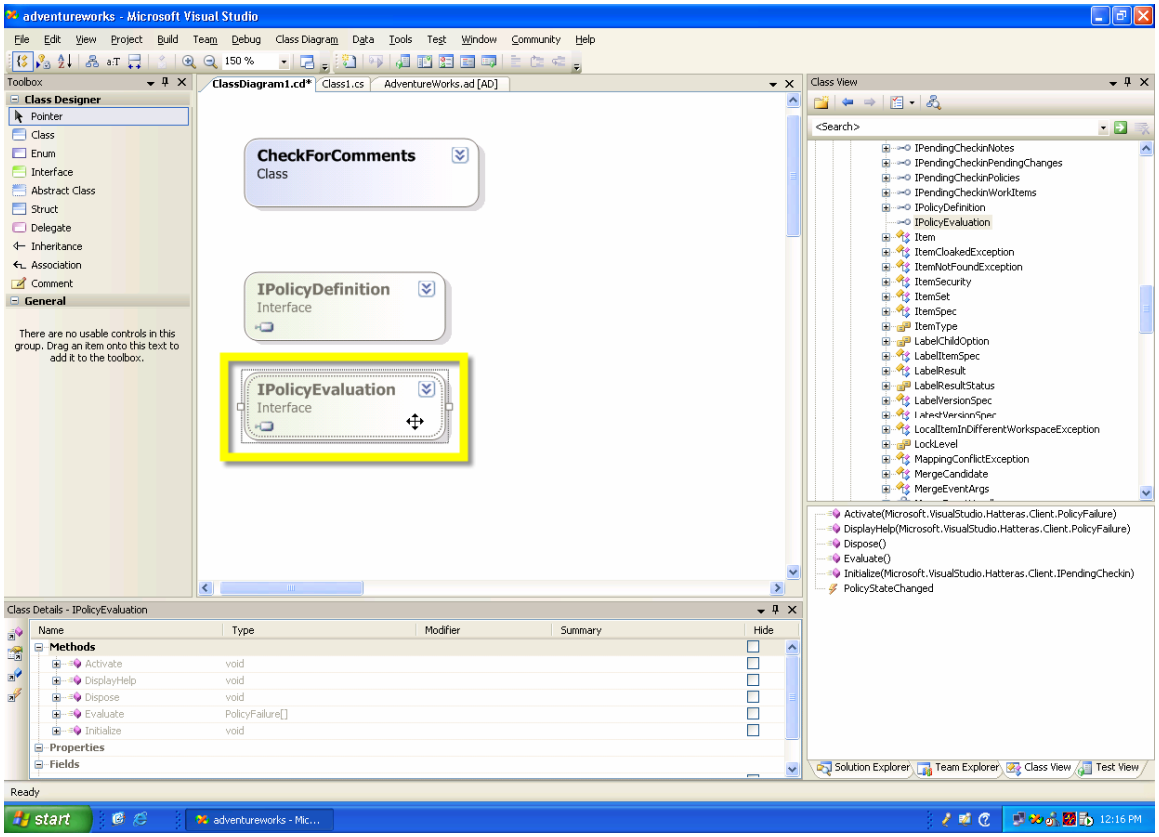
Actions	Select **IPolicyDefinition**



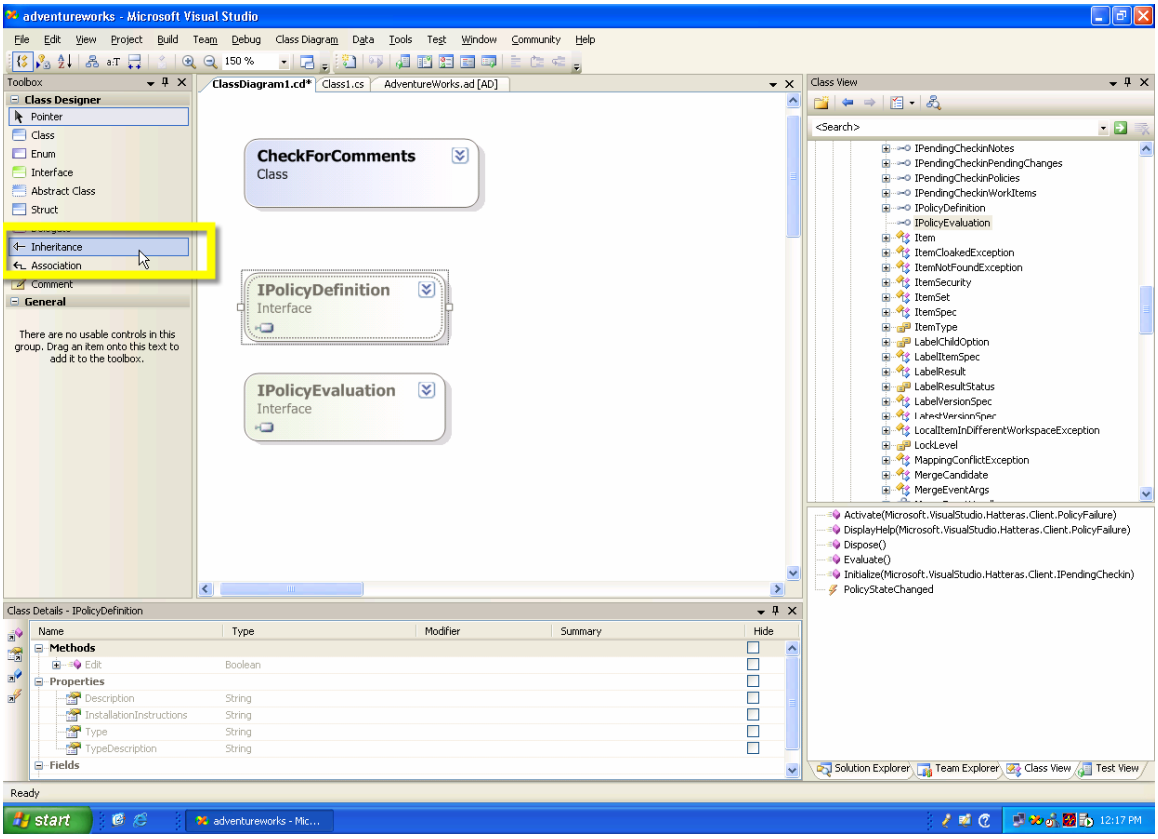
Actions	Drag it onto your **design surface**



Actions	Select **IPolicyEvaluation** Drag and drop it onto your **design surface**

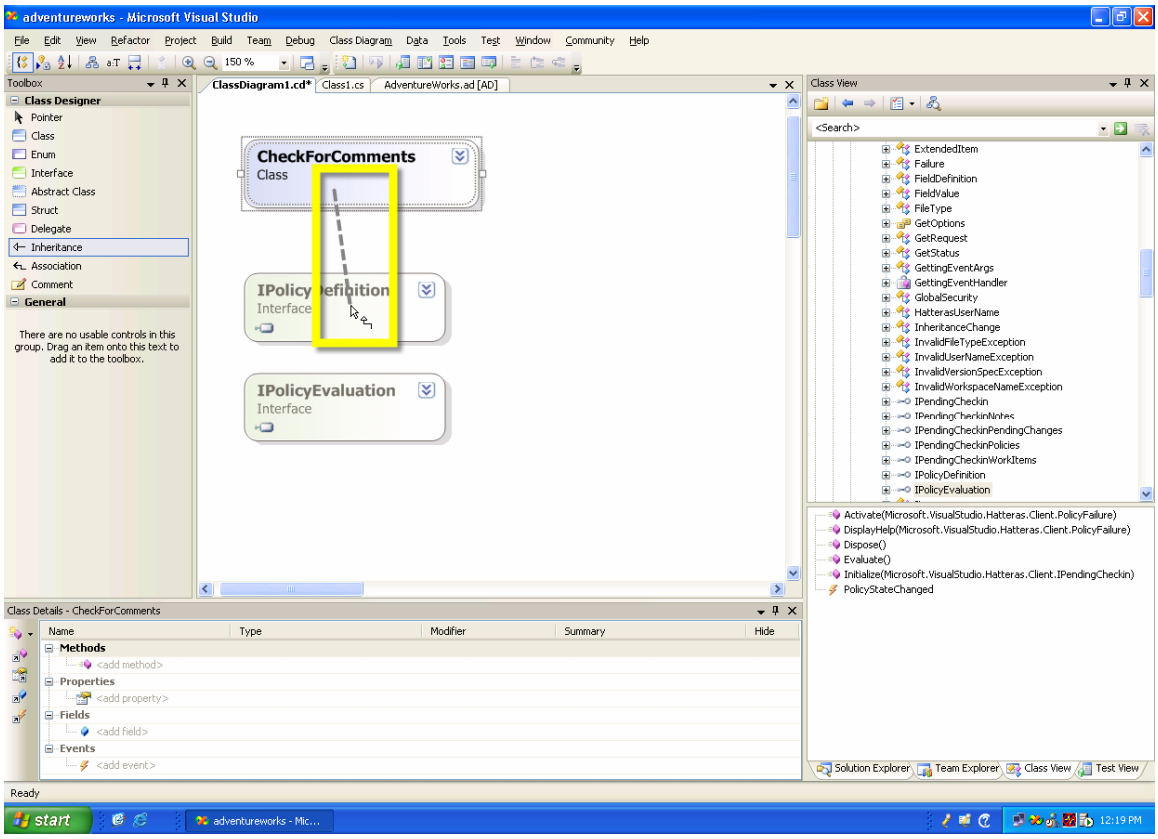


Actions	Select **Inheritance** on your **Toolbox**

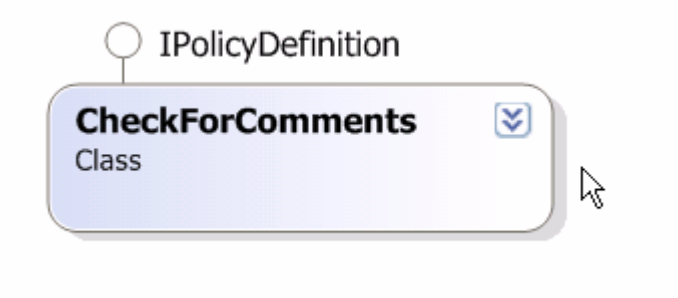


Actions	Your **cursor** should change into the icon shown below
	

Actions	Press and hold your **left** mouse button Make a dragging motion from **CheckForComments** to **IPolicyDefinition**

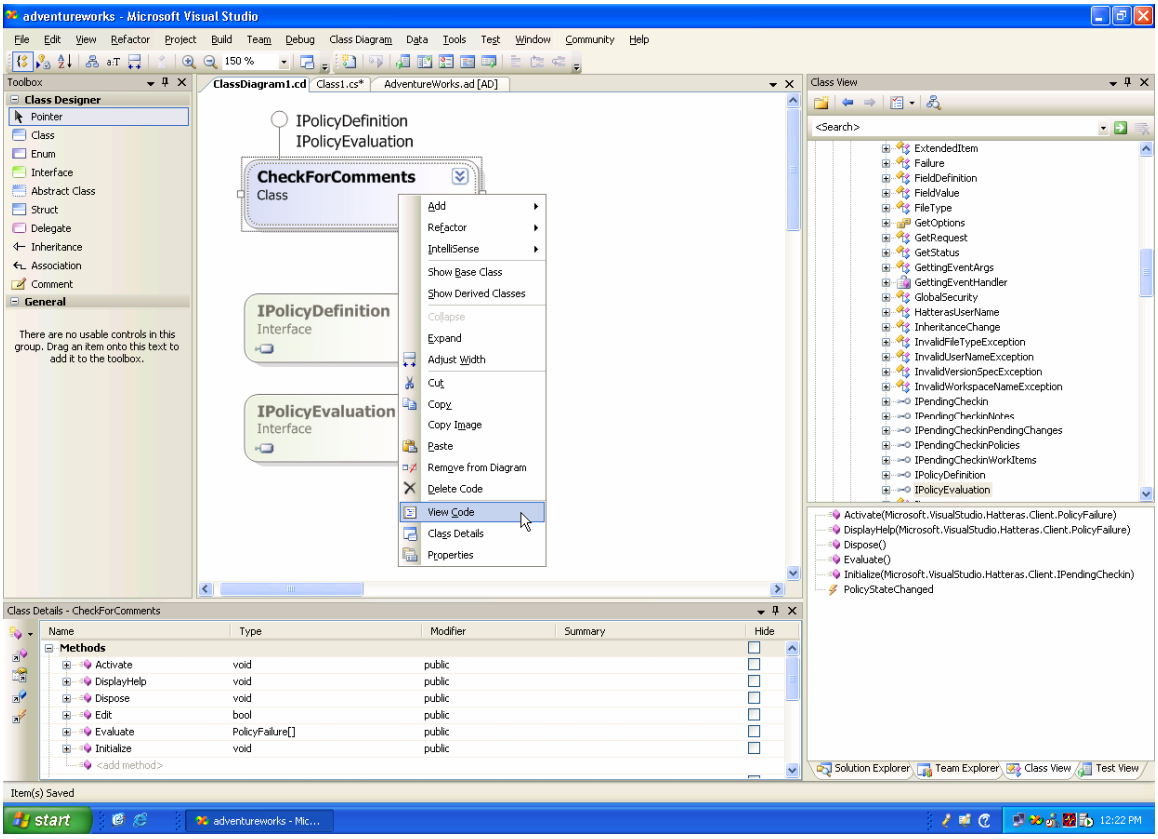


Actions	**CheckForComments** should look as shown



Actions	Do the same with **IPolicyEvaluation** **CheckForComments** should look as shown below

Actions	Select **CheckForComments** and choose **View Code**



Actions	All of the methods for each of the **interfaces** you chose to implement are stubbed out for you

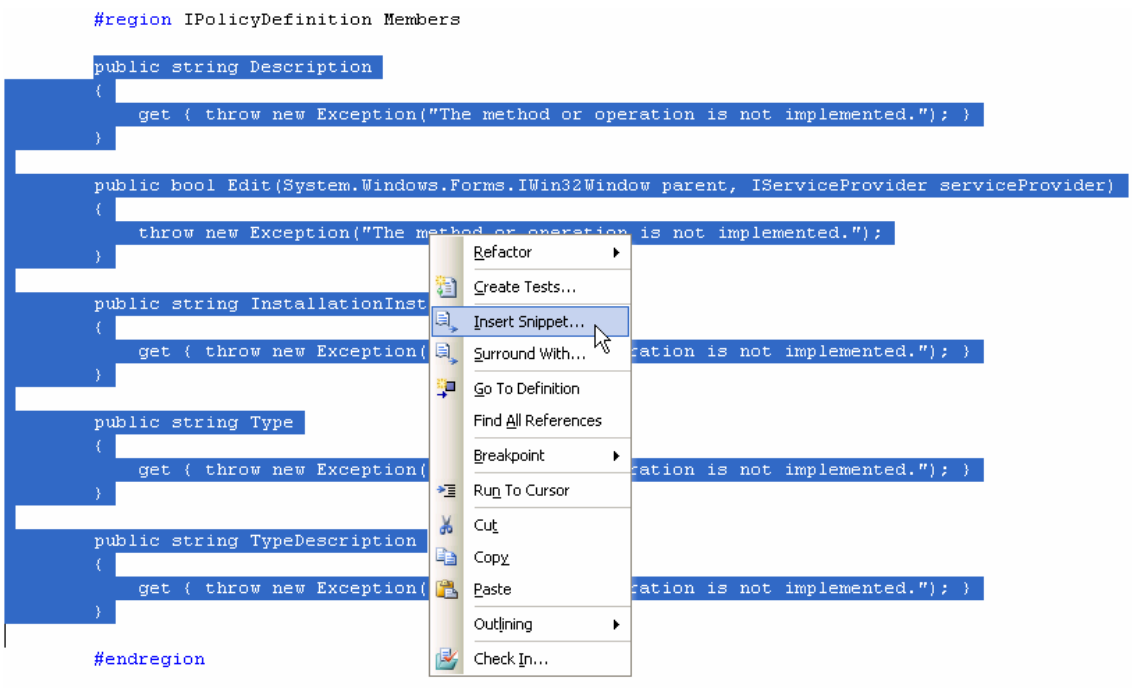
```
namespace CheckForCommentsPolicy
{
    public class CheckForComments : Microsoft.VisualStudio.Hatteras.Client.IPolicyDefinition, Mic
    {
        #region IPolicyDefinition Members

        public string Description
        {
            get { throw new Exception("The method or operation is not implemented."); }
        }

        public bool Edit(System.Windows.Forms.IWin32Window parent, IServiceProvider serviceProvid
        {
            throw new Exception("The method or operation is not implemented.");
        }

        public string InstallationInstructions
        {
            get { throw new Exception("The method or operation is not implemented."); }
        }

        public string Type
        {
            get { throw new Exception("The method or operation is not implemented."); }
        }

        public string TypeDescription
        {
            get { throw new Exception("The method or operation is not implemented."); }
        }

        #endregion

        #region IPolicyEvaluation Members

        public void Activate(Microsoft.VisualStudio.Hatteras.Client.PolicyFailure failure)
        {
```

Now we are going to implement the code behind these interfaces. Let's start with **IPolicyDefinition.**

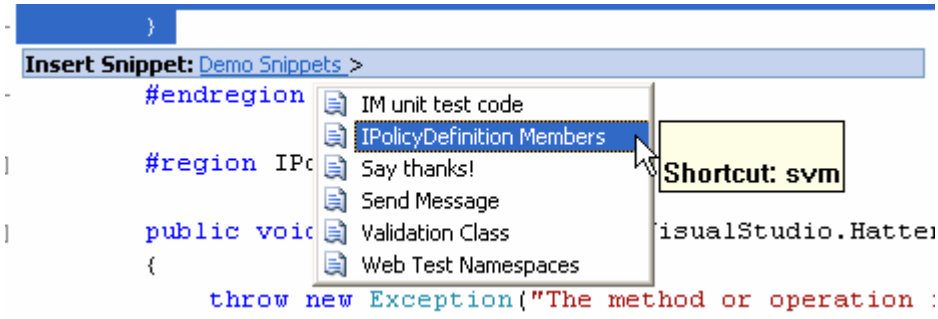
Actions	Select the code as shown

```
#region IPolicyDefinition Members

public string Description
{
    get { throw new Exception("The method or operation is not implemented."); }
}

public bool Edit(System.Windows.Forms.IWin32Window parent, IServiceProvider serviceProvider)
{
    throw new Exception("The method or operation is not implemented.");
}

public string InstallationInstructions
{
    get { throw new Exception("The method or operation is not implemented."); }
}

public string Type
{
    get { throw new Exception("The method or operation is not implemented."); }
}

public string TypeDescription
{
    get { throw new Exception("The method or operation is not implemented."); }
}

#endregion
```

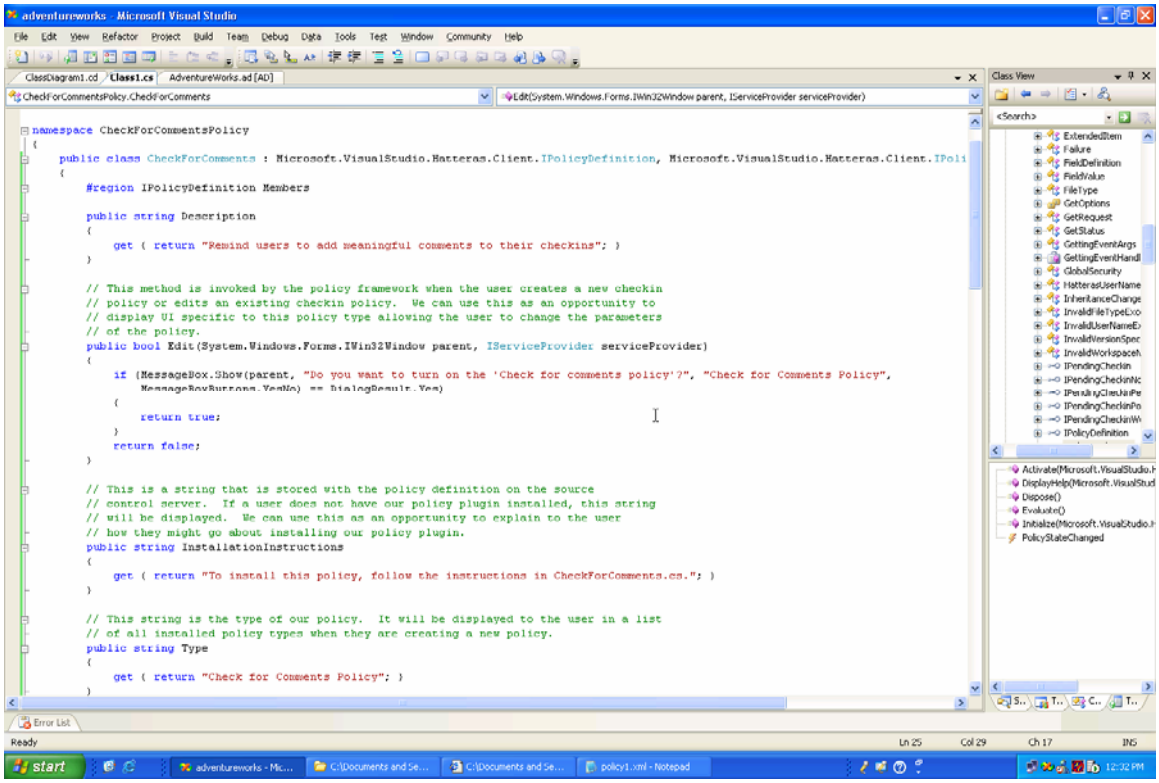
Actions	Right-click and choose **Insert Snippet…**



Actions	Choose **Demo Snippets -> IPolicyDefinition Members**



Actions	**IPolicyDefinition** is now implemented Each method has explanatory comments associated with it

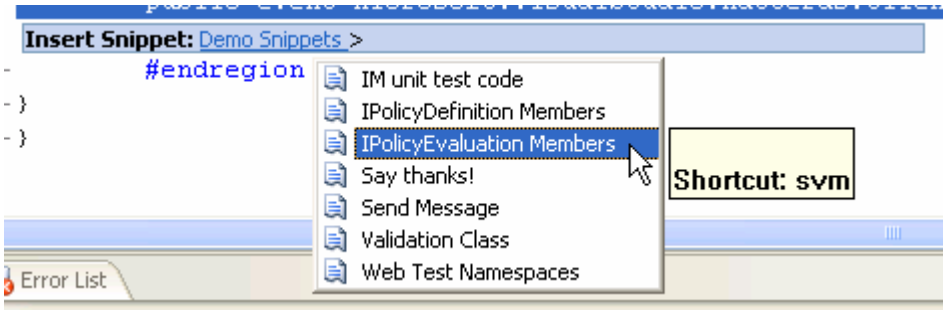


Now let's do the same thing for IPolicyEvaluation

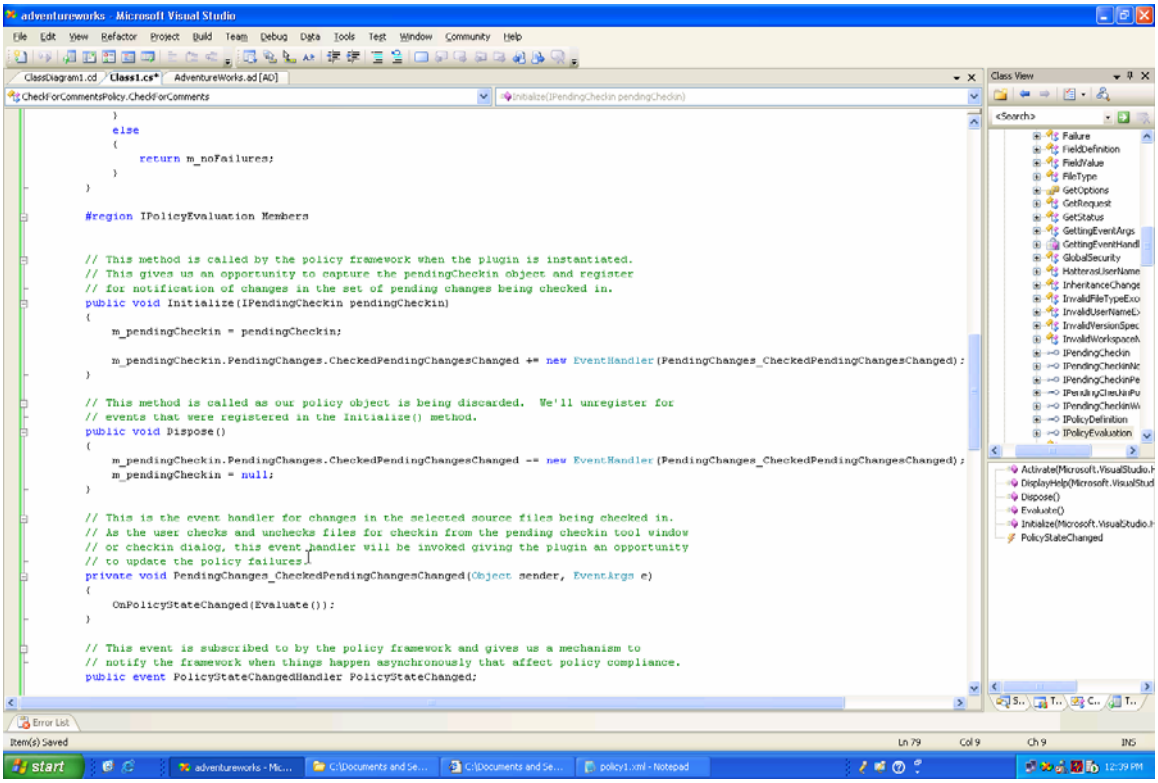
Actions	Find and select the **IPolicyEvaluation** members

```
#region IPolicyEvaluation Members

public void Activate(Microsoft.VisualStudio.Hatteras.Client.PolicyFailure failure)
{
    throw new Exception("The method or operation is not implemented.");
}

public void DisplayHelp(Microsoft.VisualStudio.Hatteras.Client.PolicyFailure failure)
{
    throw new Exception("The method or operation is not implemented.");
}

public void Dispose()
{
    throw new Exception("The method or operation is not implemented.");
}

public Microsoft.VisualStudio.Hatteras.Client.PolicyFailure[] Evaluate()
{
    throw new Exception("The method or operation is not implemented.");
}

public void Initialize(Microsoft.VisualStudio.Hatteras.Client.IPendingCheckin pendingCheckin)
{
    throw new Exception("The method or operation is not implemented.");
}

public event Microsoft.VisualStudio.Hatteras.Client.PolicyStateChangedHandler PolicyStateChanged;

#endregion
```

Actions	Right-click and choose **Insert Snippet…** Choose **Demo Snippets -> IPolicyEvaluation Members**

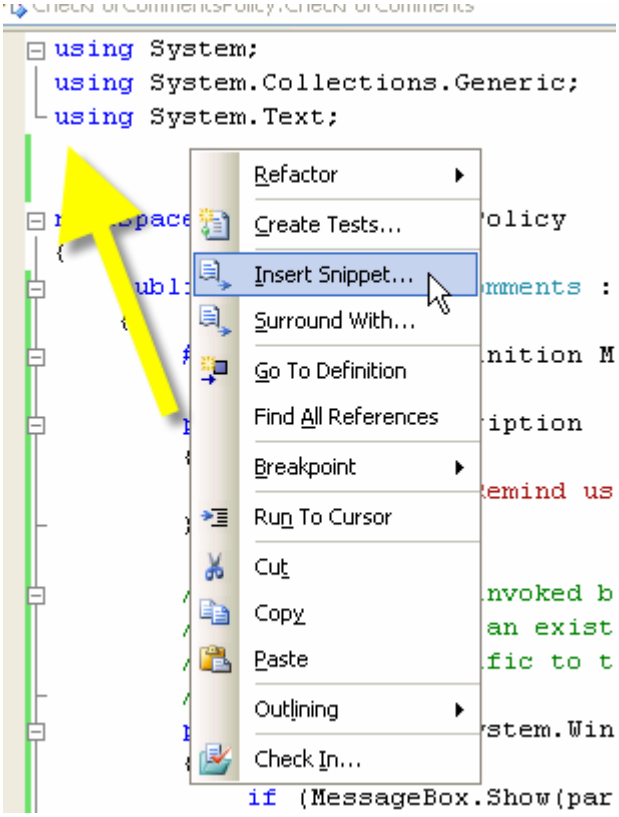


Actions	**IPolicyEvaluation** is now implemented Each method has explanatory comments associated with it

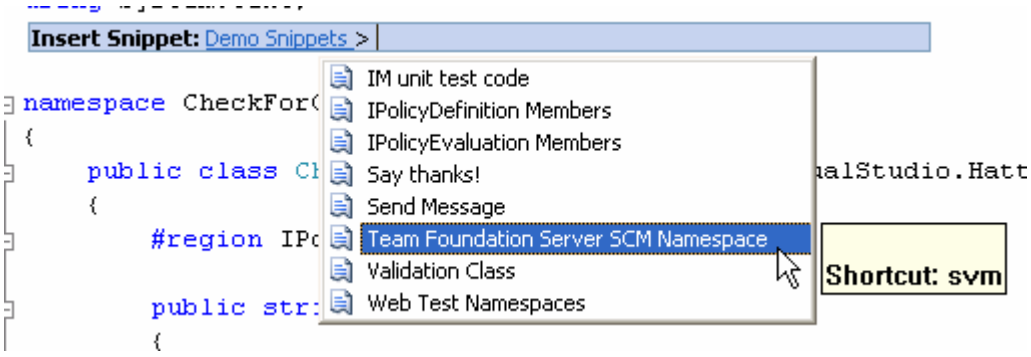


There are a few more odds and ends we need to do to implement our check in policy.

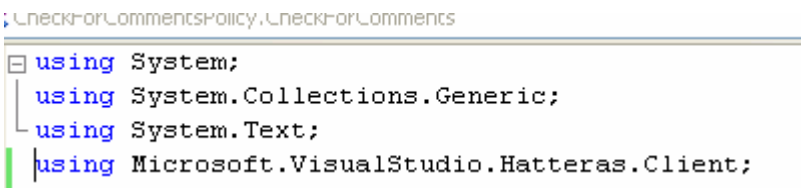
Actions	Put your cursor on the next line after the **using System.Text;** line Right-click and choose **Insert Snippet…**



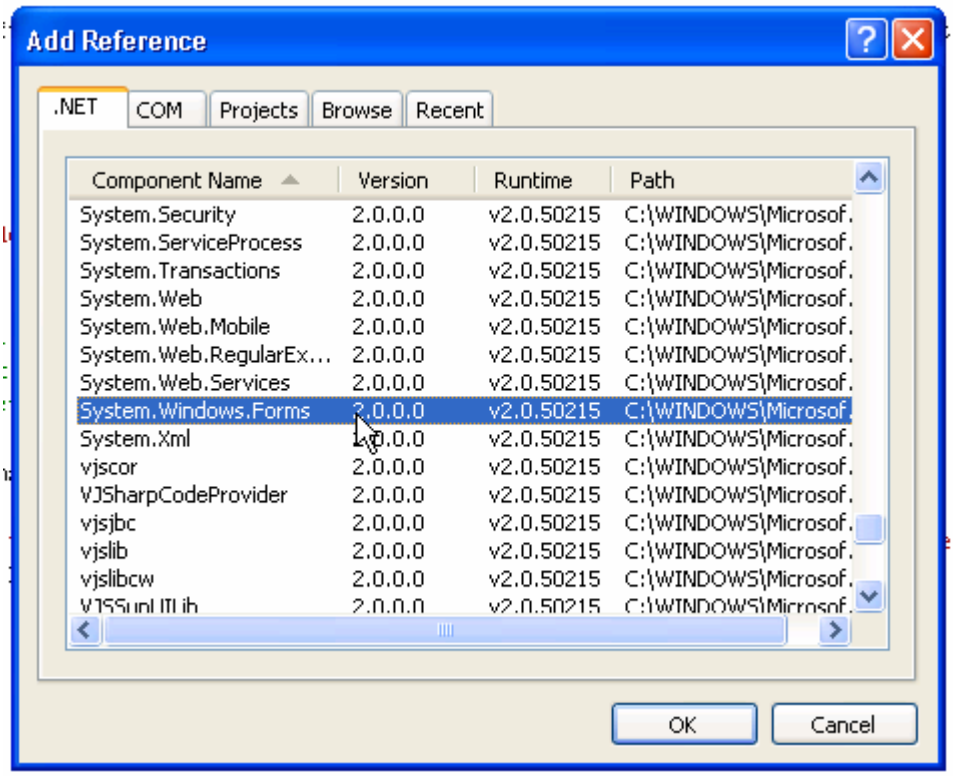
Actions	Choose **Demo Snippets -> Team Foundation Server SCM Namespace**



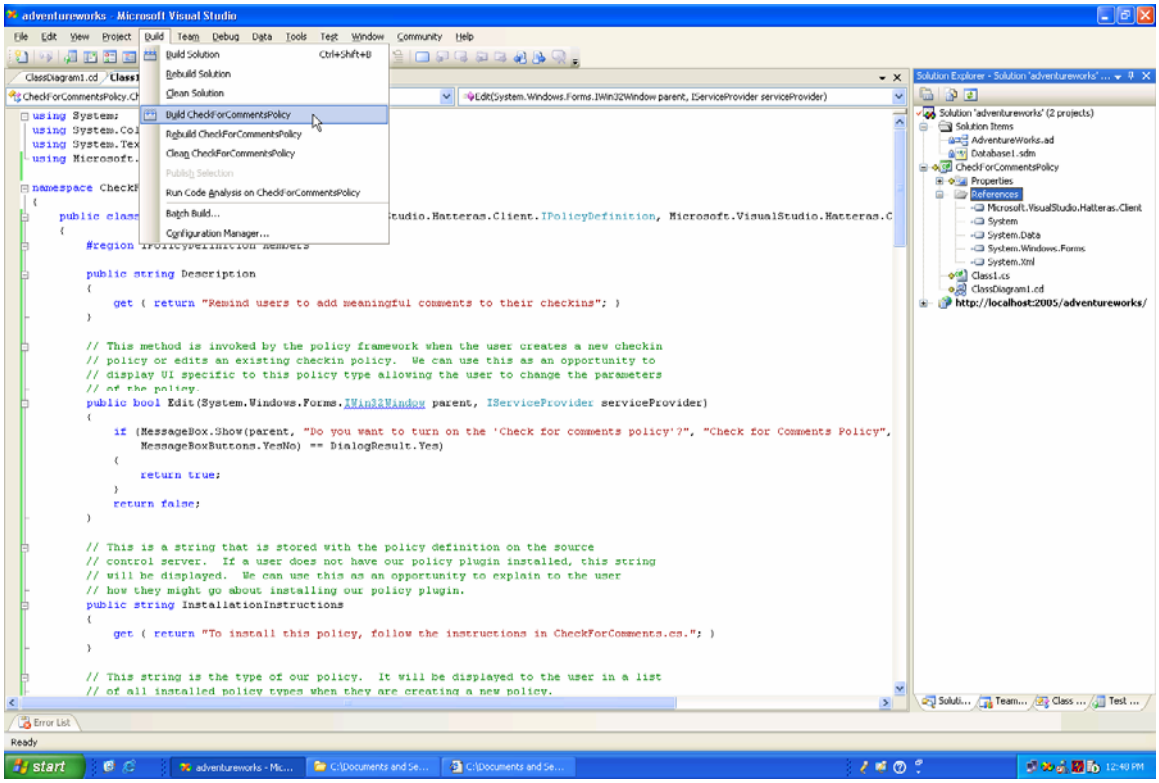
Actions	Your namespace inclusions should look as shown



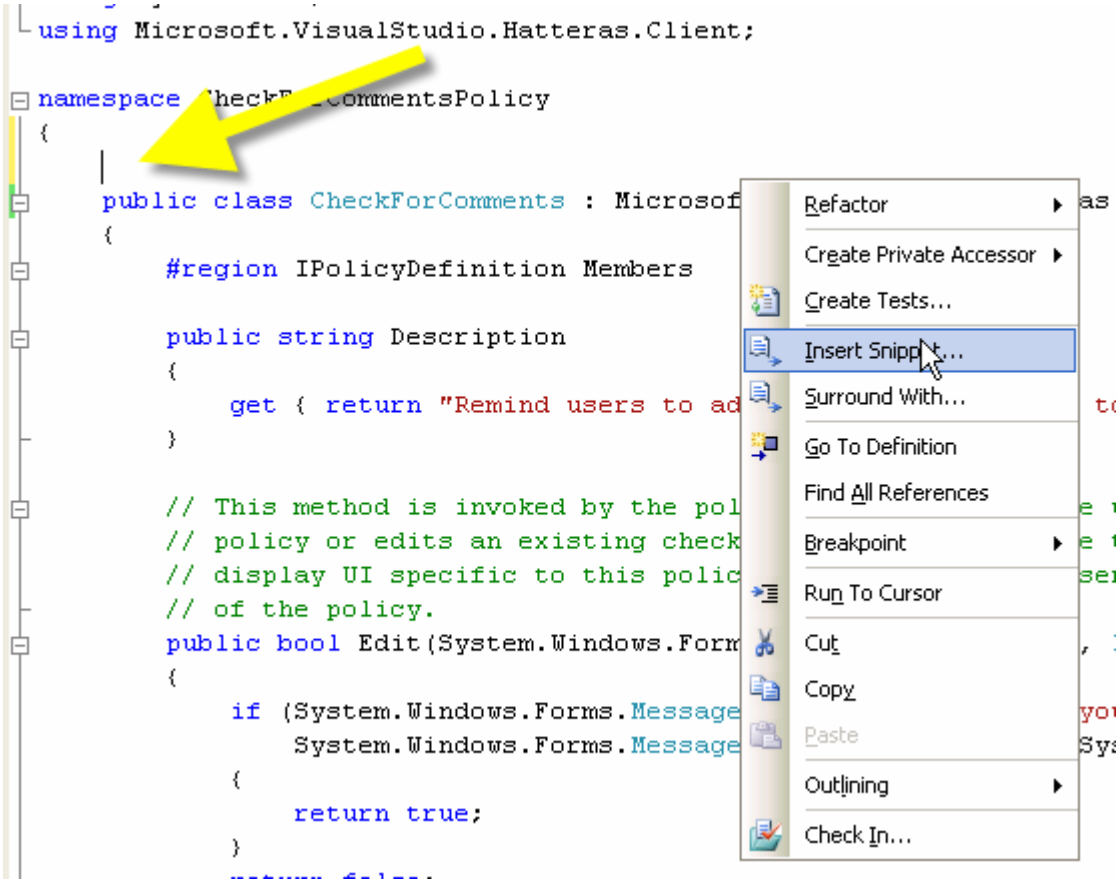
Actions	Right-click on the **Reference** node under the **CheckForCommentsPolicy** project node Choose **Add Reference** Click on the **.NET** tab Choose **System.Windows.Forms** Press **OK**



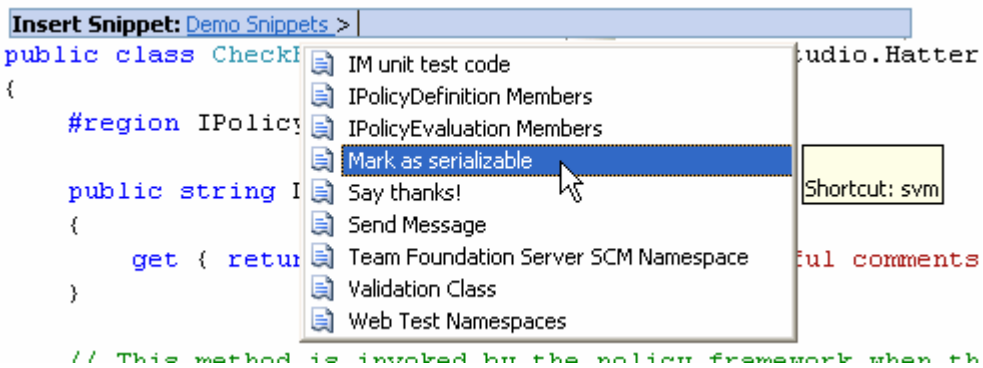
Actions	Choose the **Build -> Build CheckForCommentsPolicy** menu option



Actions	Put your cursor in the location shown Right-click and choose **Insert Snippet…**



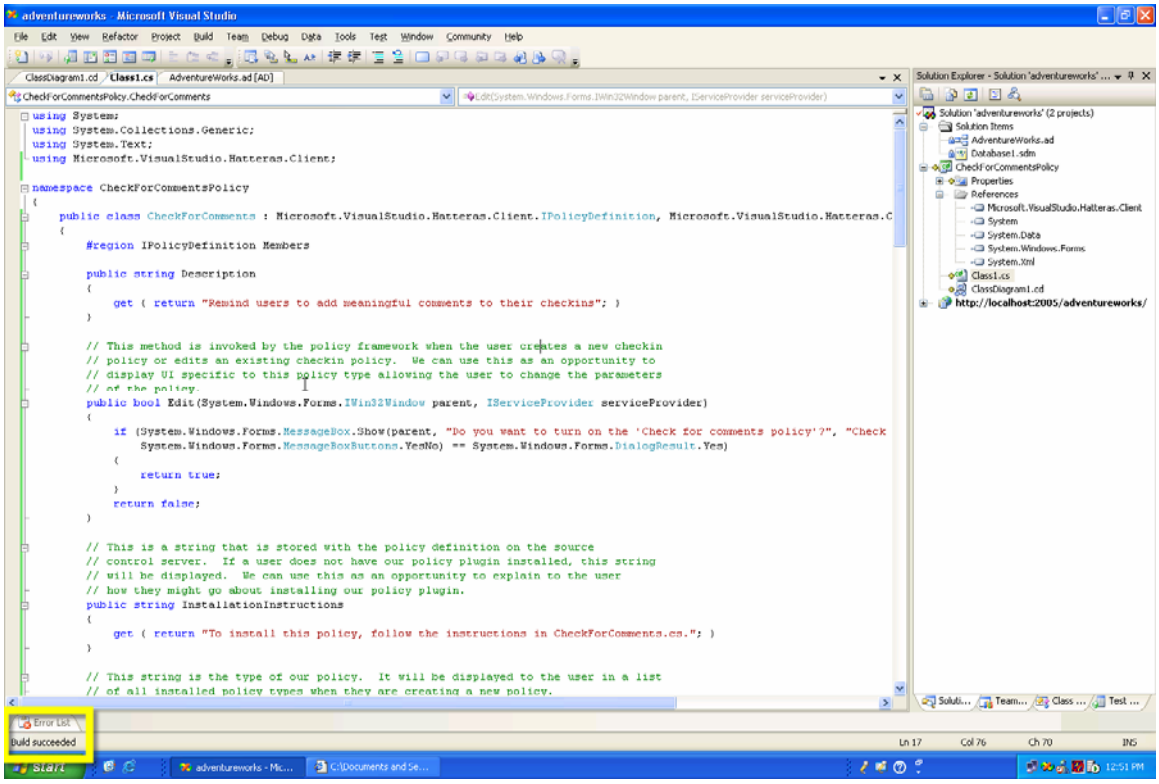
Actions	Choose **Demo Snippets -> Mark as serializable**



Actions	The attribute as shown should be added to your code

```
[Serializable]
public class CheckForComments : Microsoft.VisualStudio.Hattera
{
    #region IPolicyDefinition Members

    public string Description
    {
        get { return "Remind users to add meaningful comments
    }
```

Actions	Your **build** should be successful



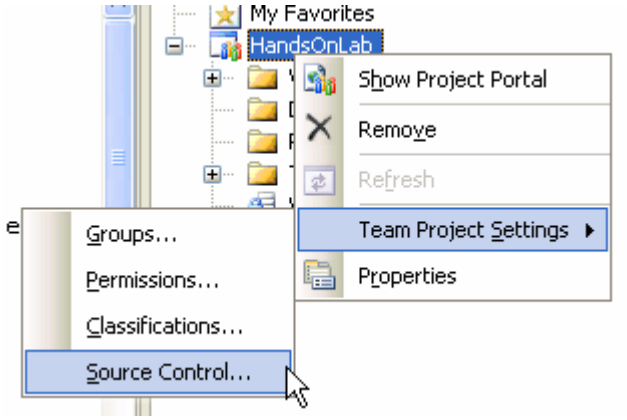
Once our policy has been built, we need to register it with Team Server. We have the register entries already; we just need to run the .reg file.

Actions	Go to your **Desktop** and double-click on the **CheckForComments.reg** file Answer **Yes** and **OK** to the two dialogs that appear after you double-click This will register your policy with **Visual Studio Team System**

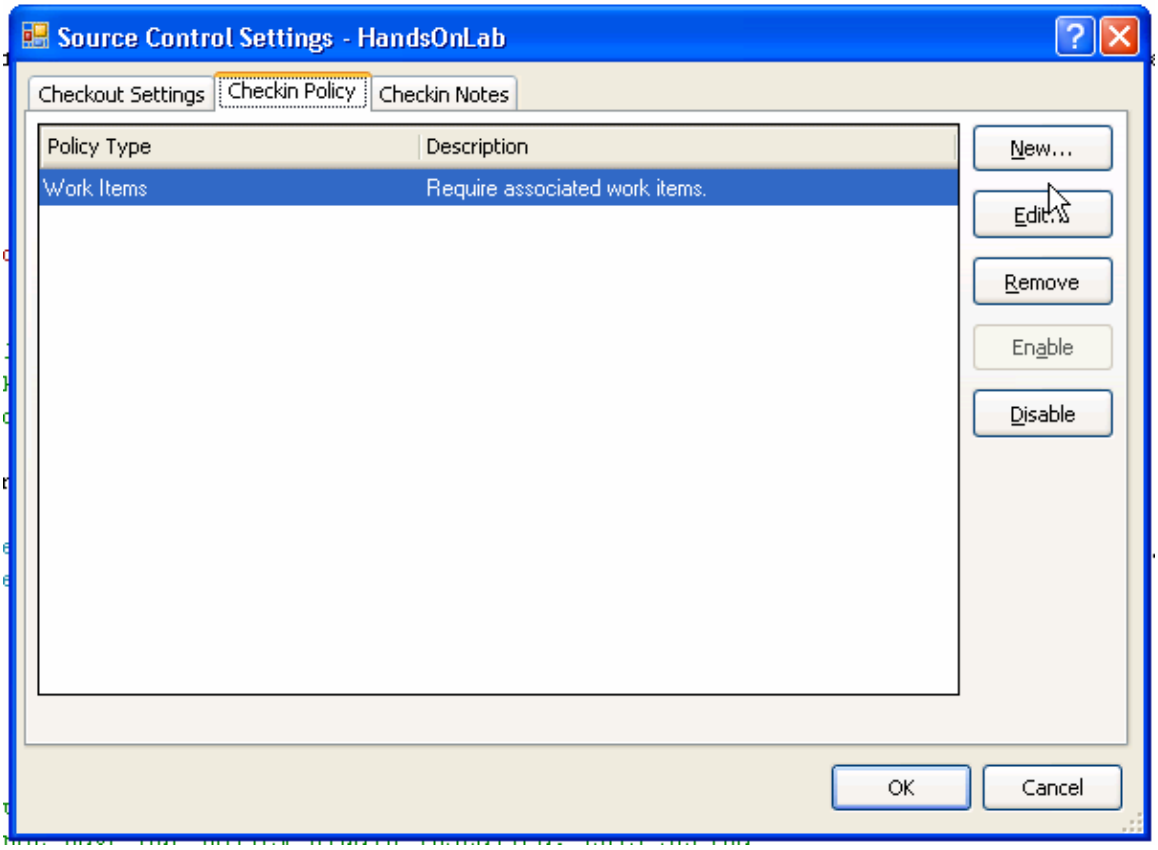
Our custom check in policy is created and registered; now let's add it to our source control project.

We can test our new policy by trying to do a check in.

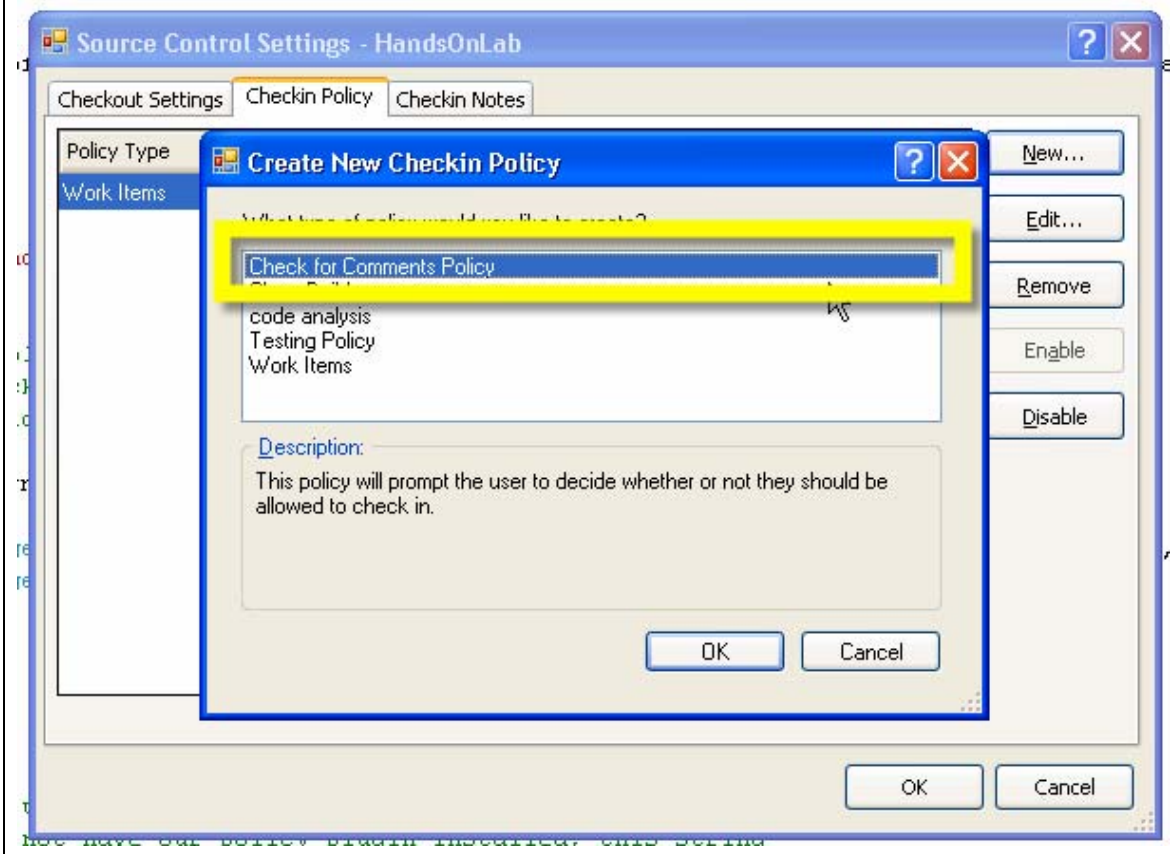
Actions	From the **Team Explorer** right-click on the **HandsOnLab** node Choose **Team Project Settings -> Source Control…**



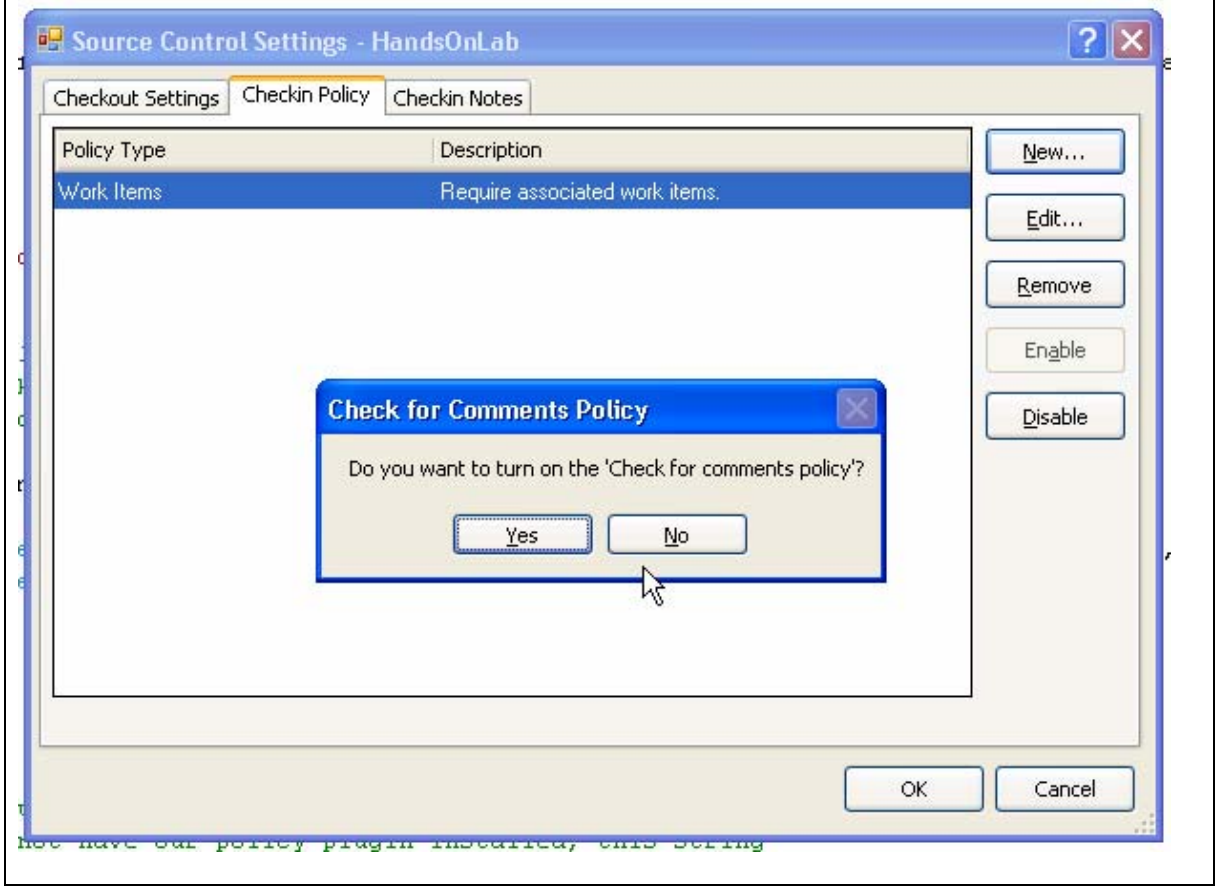
Actions	Click on the **Checkin Policy** tab



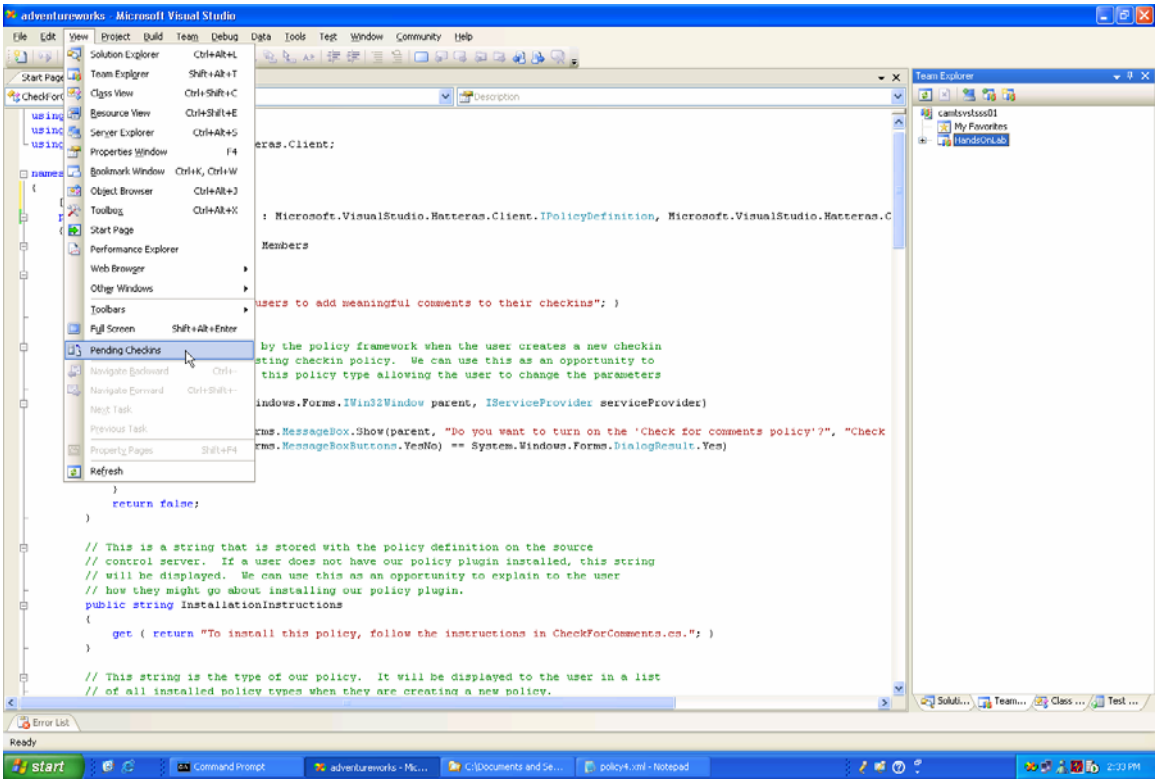
Actions	Press **New …** Our **Check For Comments Policy** should be listed Select it and press **OK**



Actions	This dialog box is really just an example of how you can add a user interface to allow your policy to be configured. Press **Yes** Press **OK** to close the **Source Control Settings** dialog box

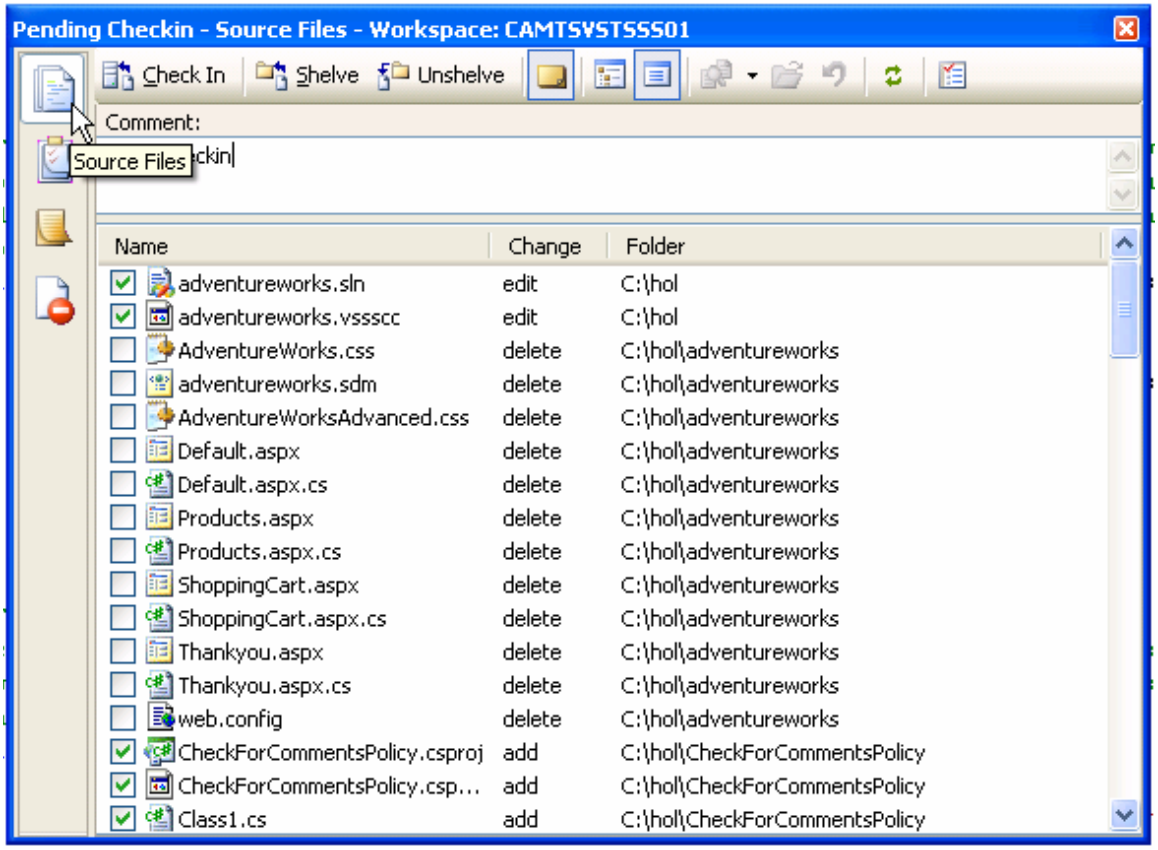


Actions	Choose the **View – Pending Checkins** menu option

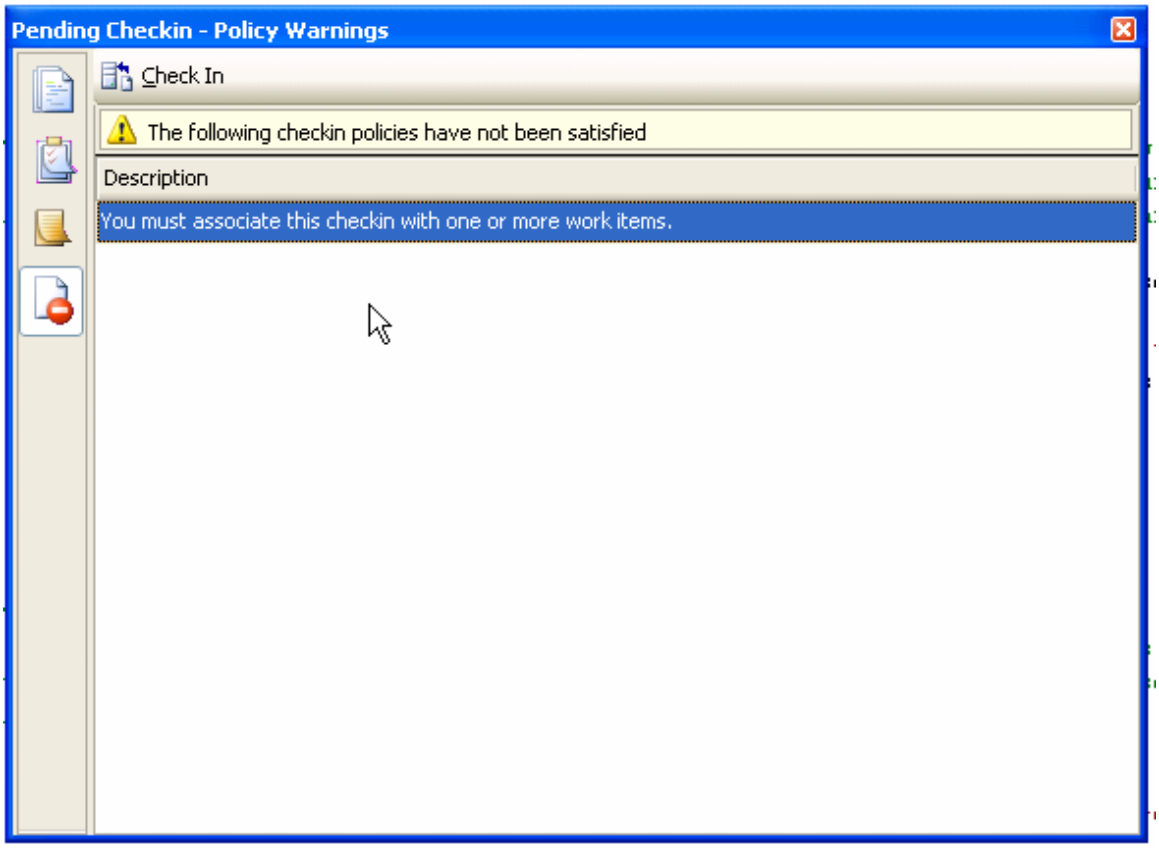


Actions	Press **Policy Warnings** Notice the two policy violations that you have; one of them is the policy that we created

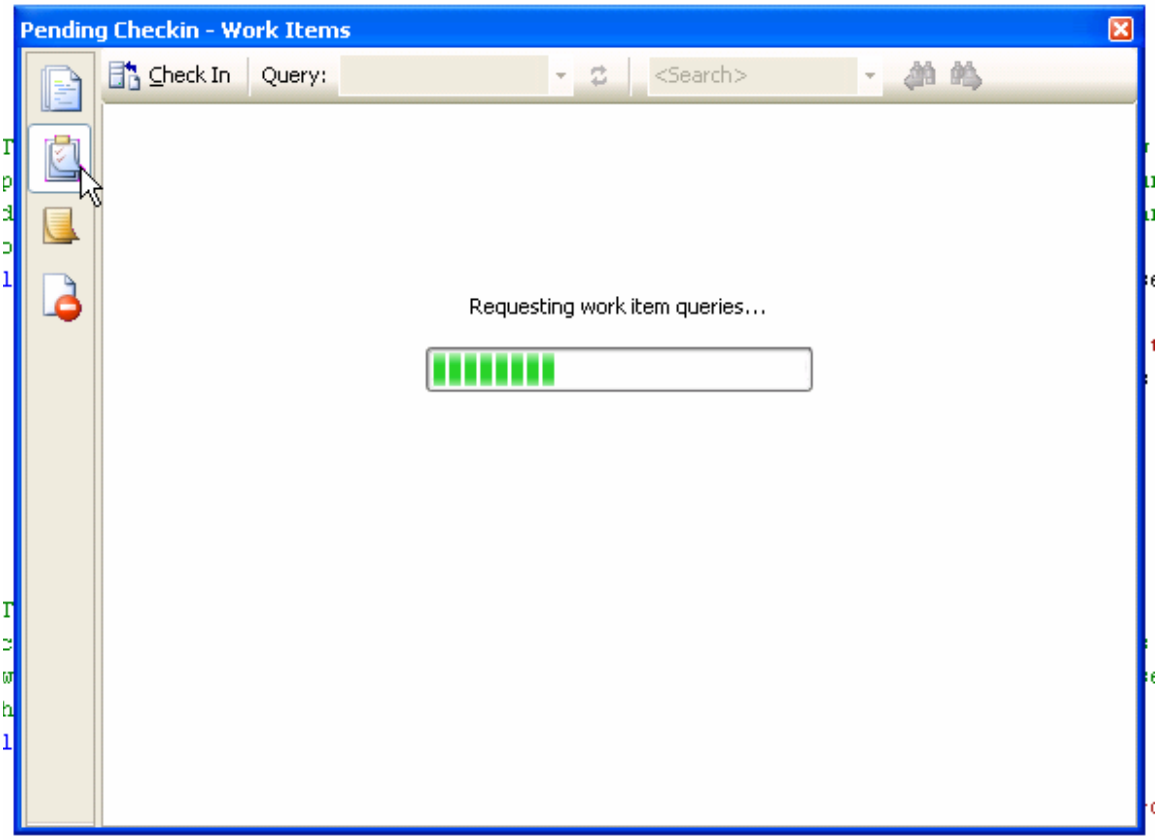
Actions	Press **Source Files** Enter a comment



Actions	Press **Policy Warnings** again and notice that our new policy has been satisfied **Note –** If you still see a policy warning, right-click on it and choose **Evaluate**



Actions	Click on **Work Items** to find a work item to associate this check in to satisfy our other check in policy as we did before.



This was a very simple example, but it illustrates the integration and extensibility that is part of Visual Studio Team System. Check in policy is one of the many facets of Team System that helps your whole team write more robust and reliable code in a productive manner.

We have our IM component written, tested and checked in, now it's time to flesh out the rest of our application.

Exercise - Implement Architecture

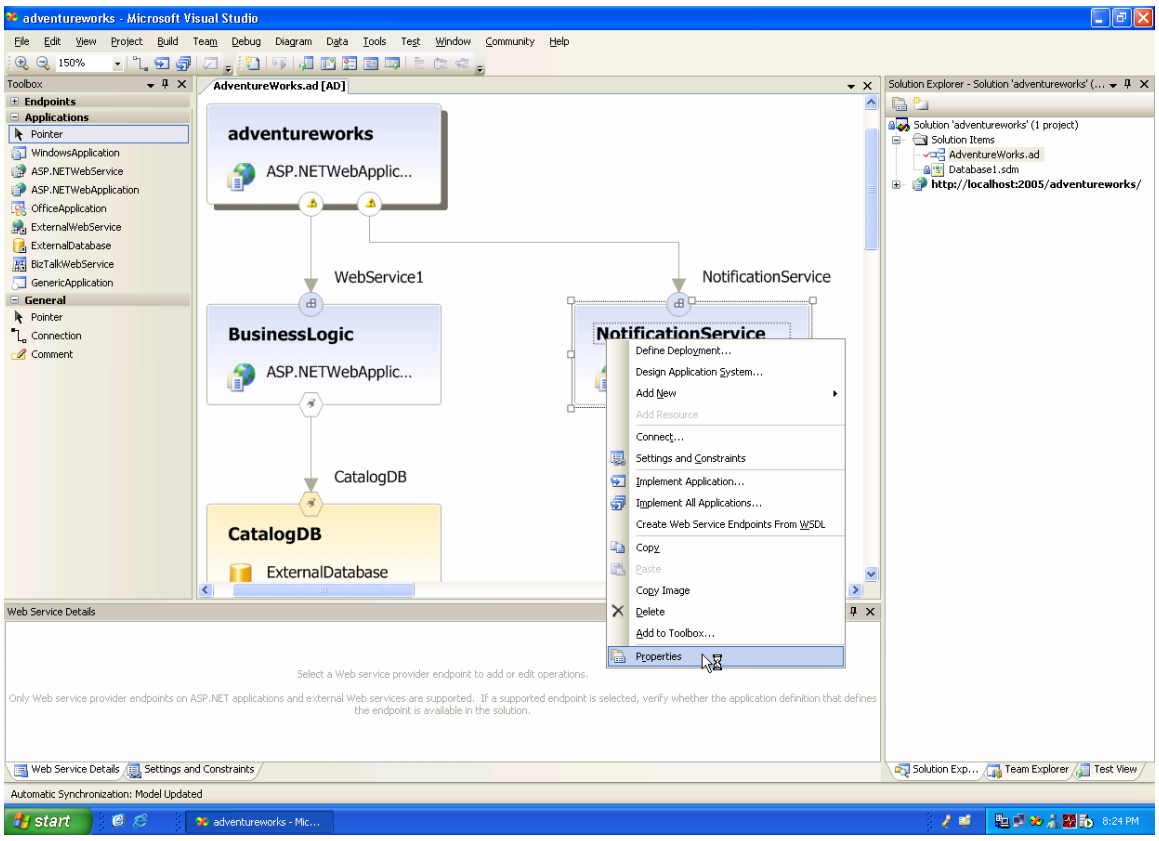
This section of the lab is able to standalone, but you will have to use the default implementation of the XML Web Service, rather than calling into the Notification component if you did not complete the Exercise - Test Driven Development section.

So far, we've looked at traditional forms of writing robust code, but another aspect is to writing robust and reliable software is being able to see where our code fits into the bigger picture.

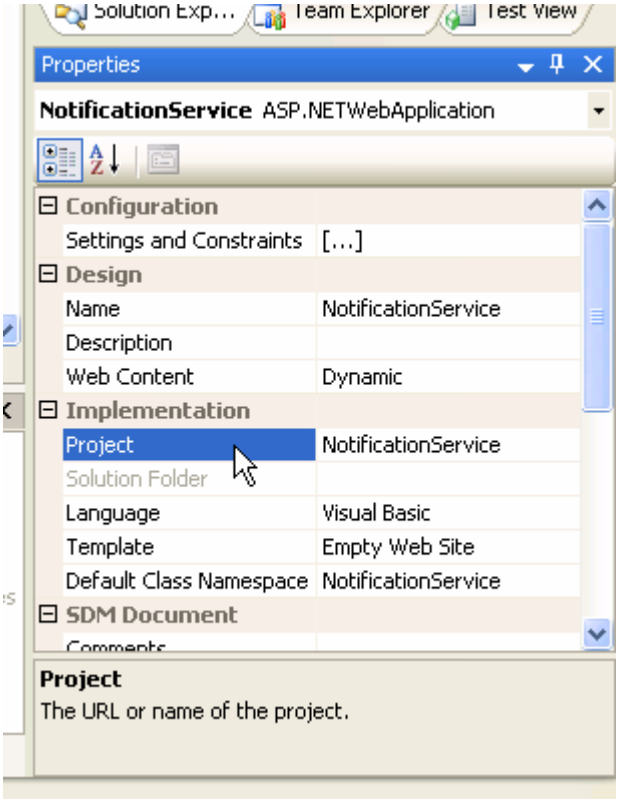
Team System helps us design our applications visually – as with the Class Designer, the model and the source code is always kept in sync.

We will implement a new XML Web Service by taking its design, and generating the code for an ASP.NET web service hosted in IIS.

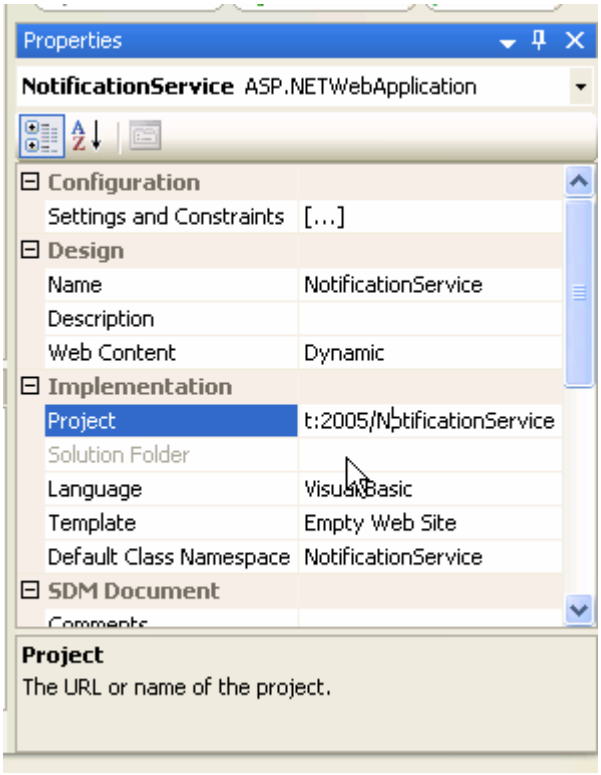
Actions	Switch to the **AdventureWorks.ad** file Right-click on the **NotificationService** item and choose **Properties**



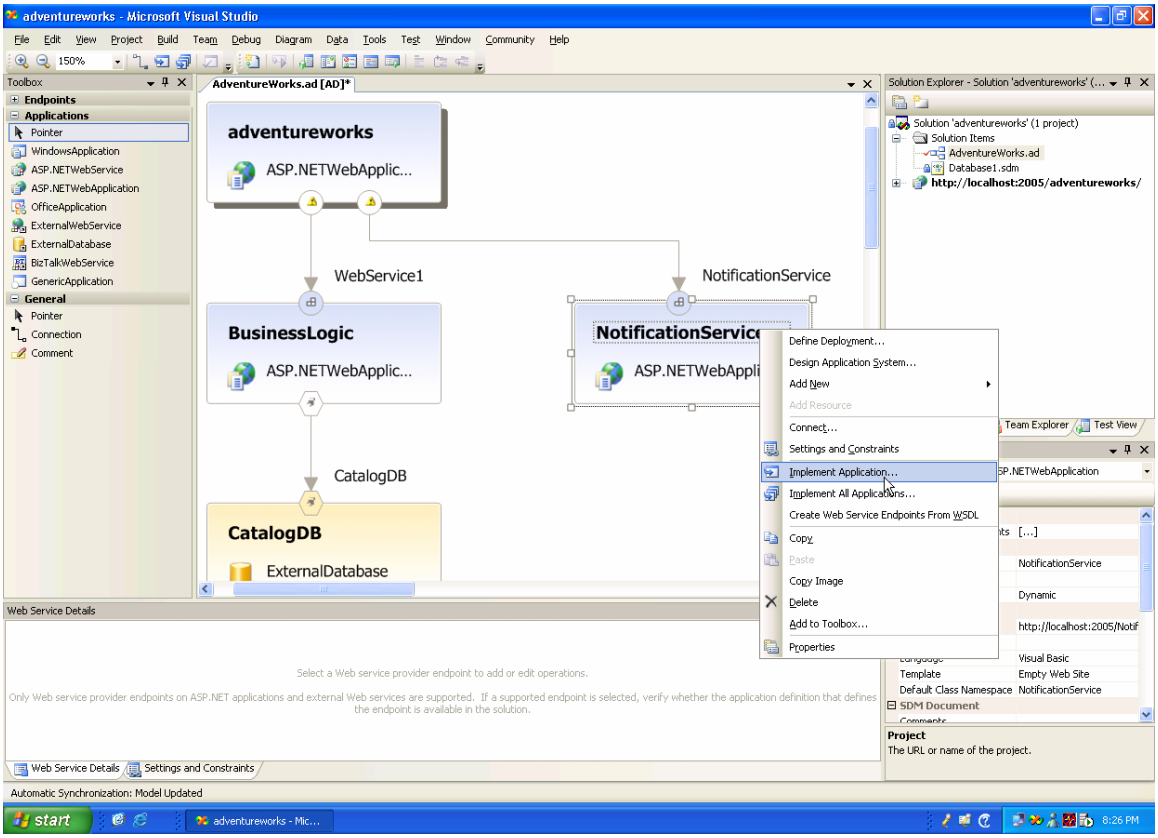
Actions	In the **Properties** grid, select the **Project** property



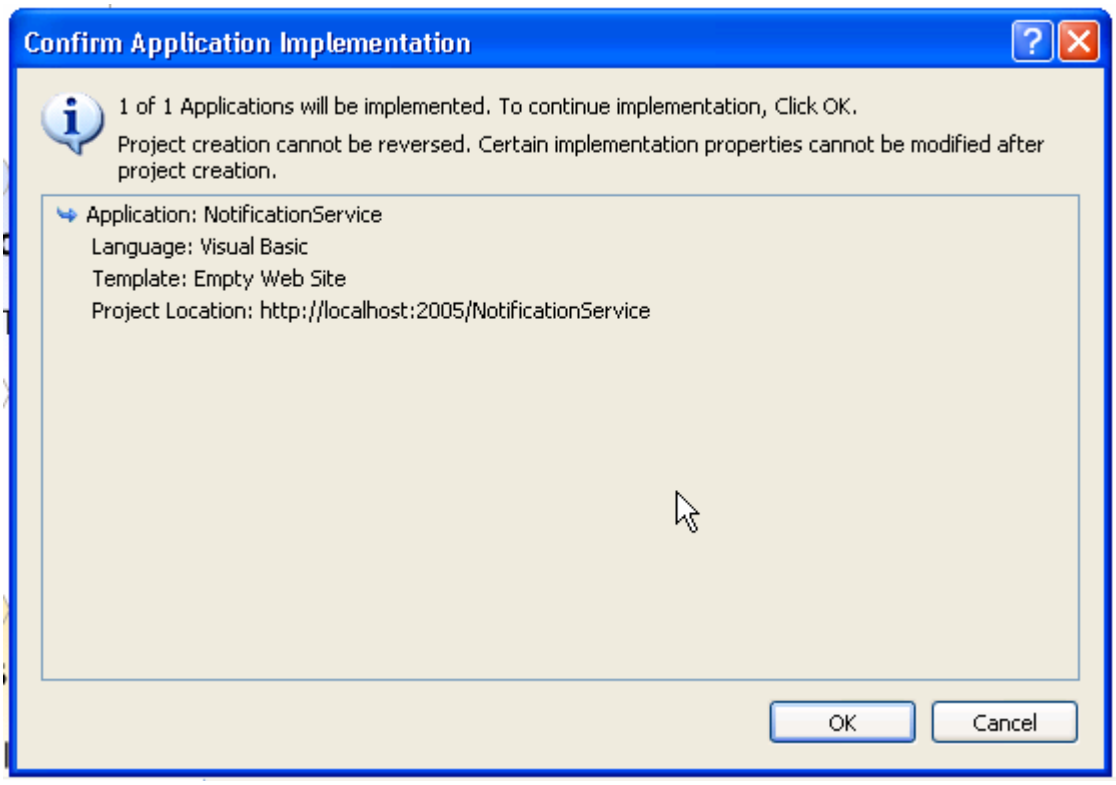
Actions	Change the value to **http://localhost:2005/notifyservice**



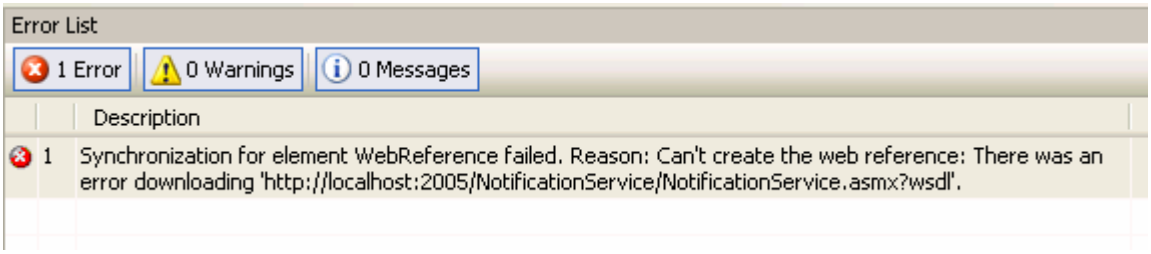
Actions	Right-click on **NotificationService** and choose **Implement Application…**



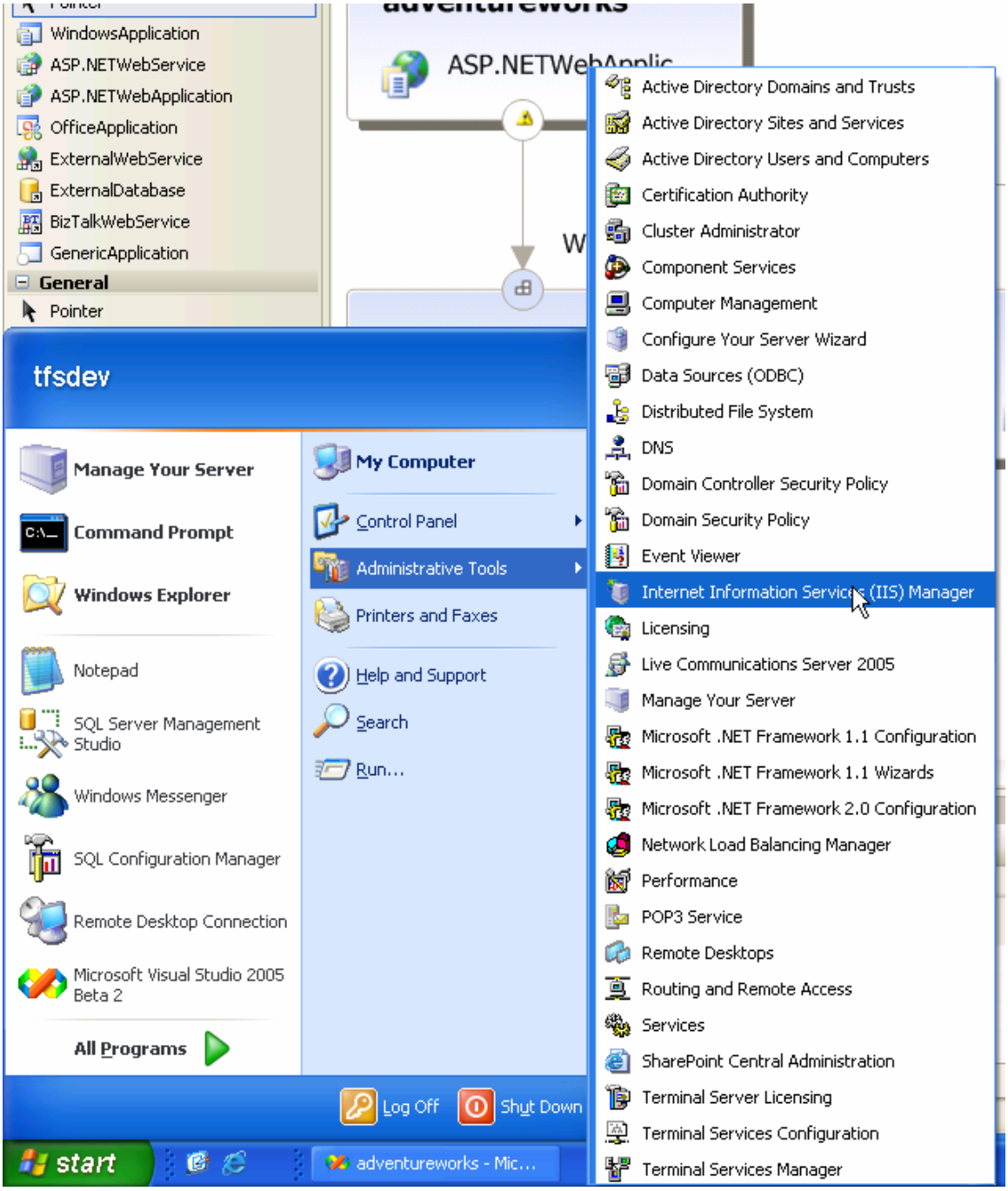
Actions	Press **OK** It will take a few moments to generate your new project



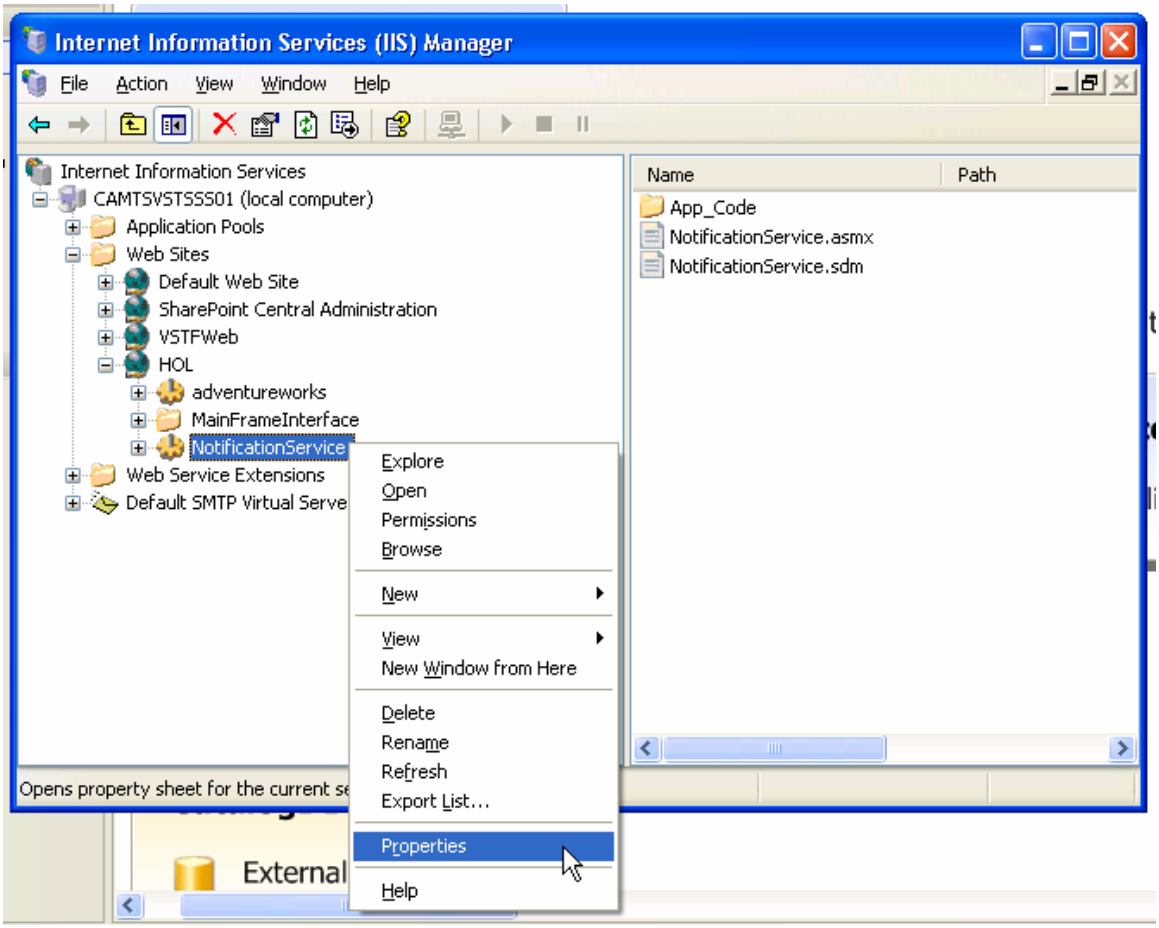
Actions	After **NotificationService** has been generated – you will see this error This is a known issue in Beta 2 – the following actions are workarounds to this issue



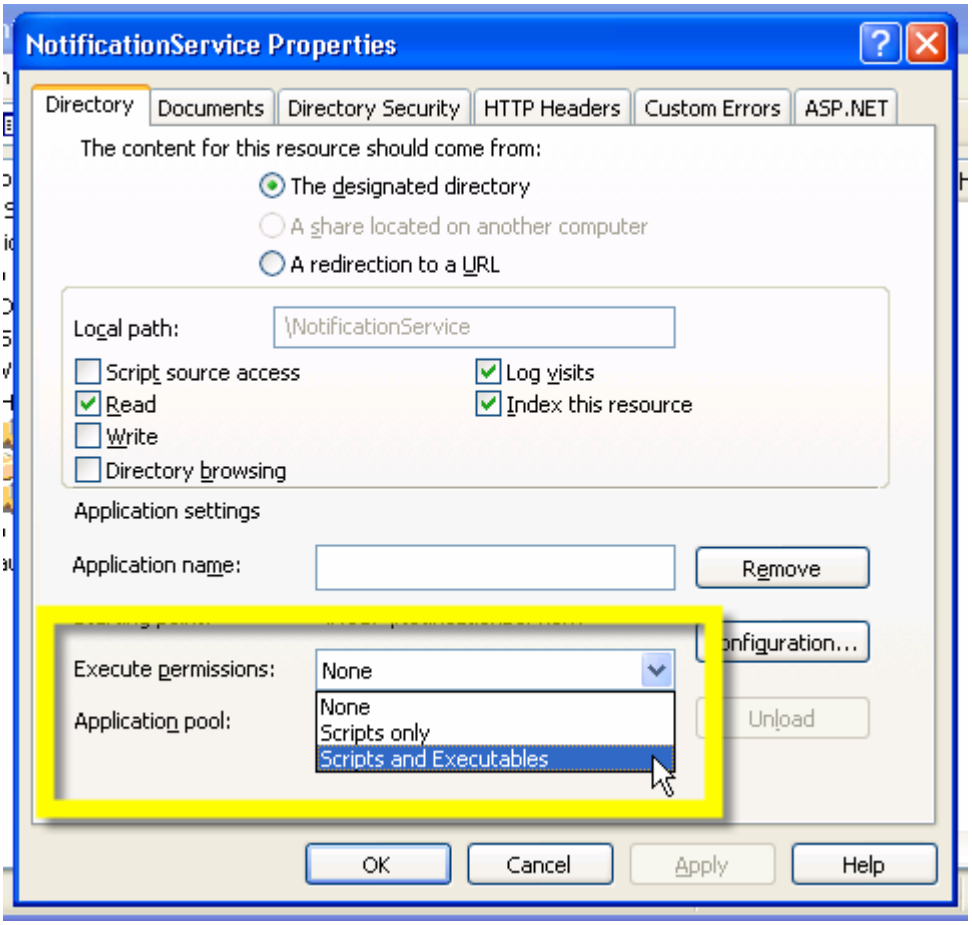
Actions	Start **Internet Information Services (IIS) Manager**



Actions	Find the **NotificationService** virtual directory and select it Right-click and choose **Properties**

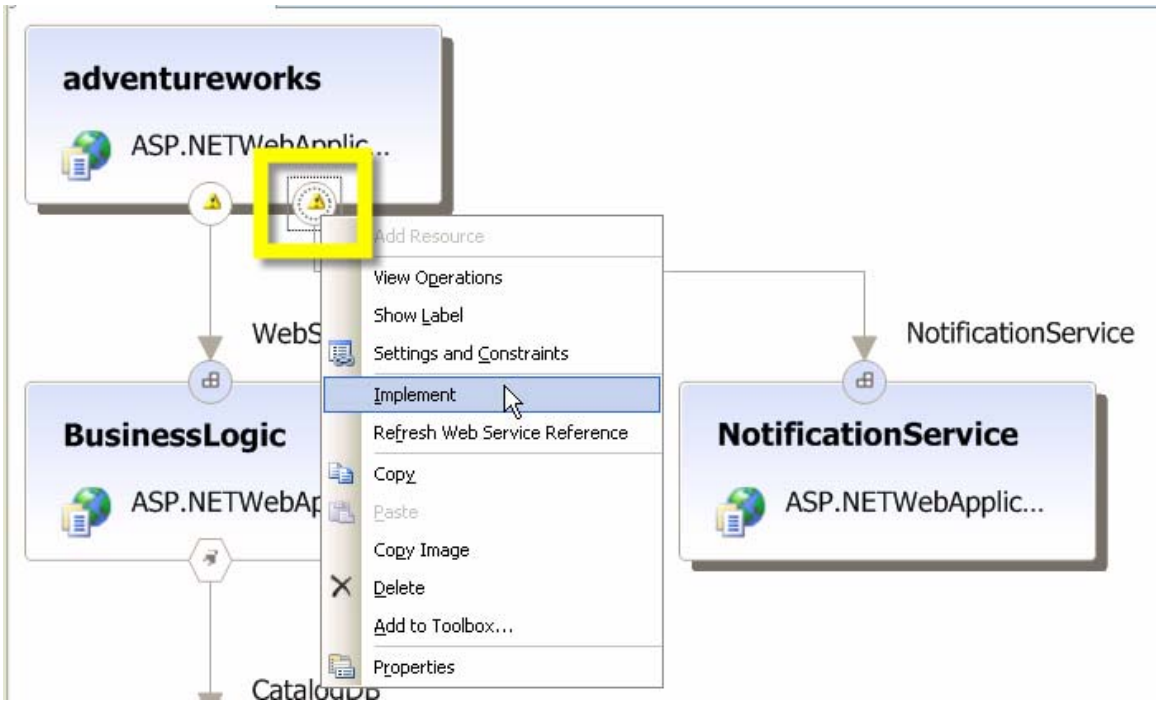


Actions	Change the **Execute Permissions** property to **Scripts and Executables** Press **OK** to close this dialog Close **Internet Information Services (IIS) Manager**

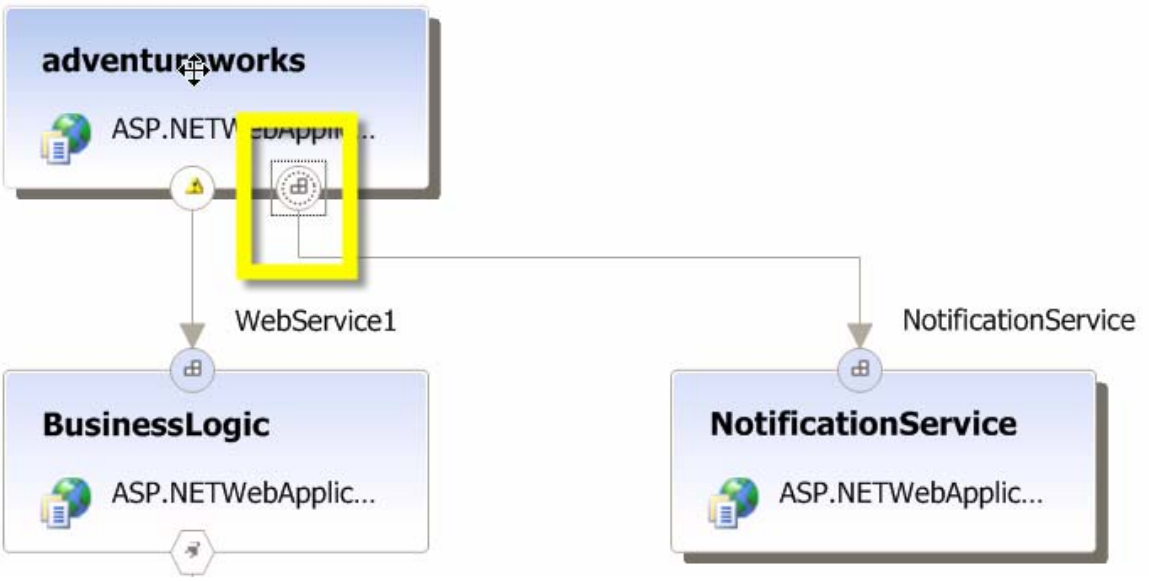


Once our ASP.NET web service has been implemented, we will have a skeleton project to work with. We will flesh out the operations of this web service to send IM messages; we will use the IM component that we created earlier.

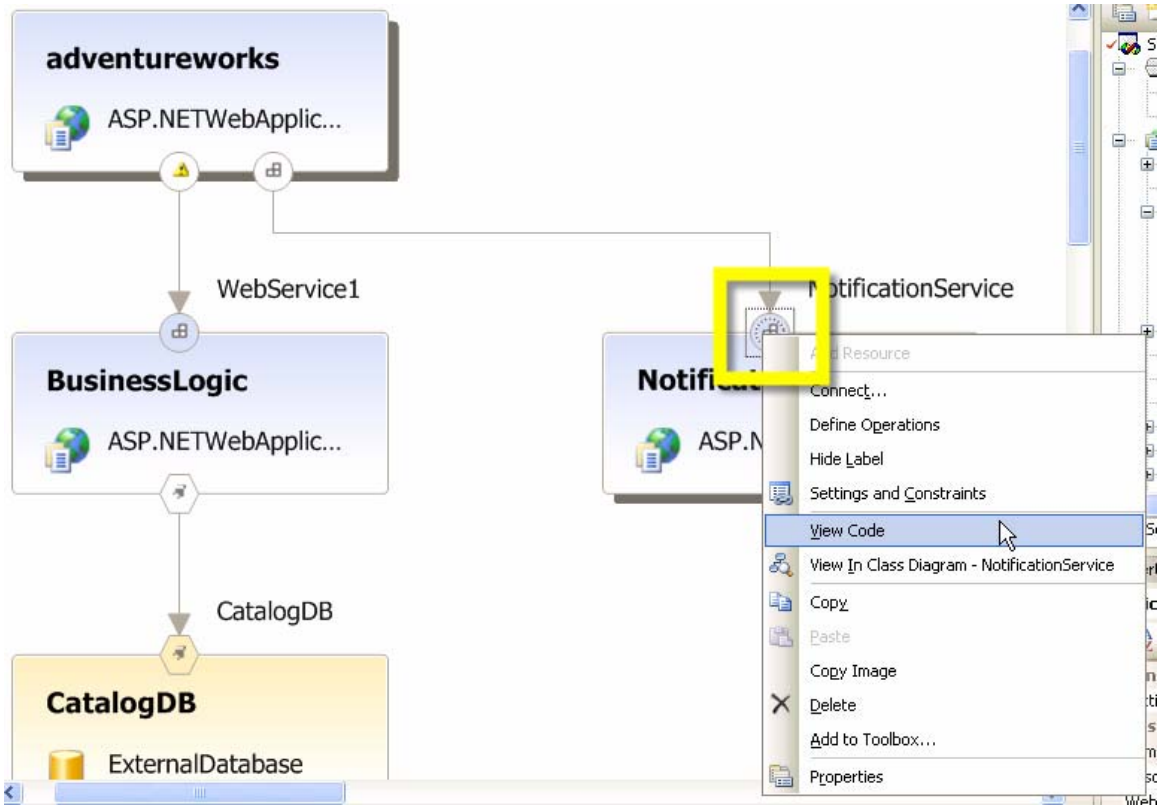
Actions	Select the **end point** on **adventureworks** Right-click and choose **Implement**



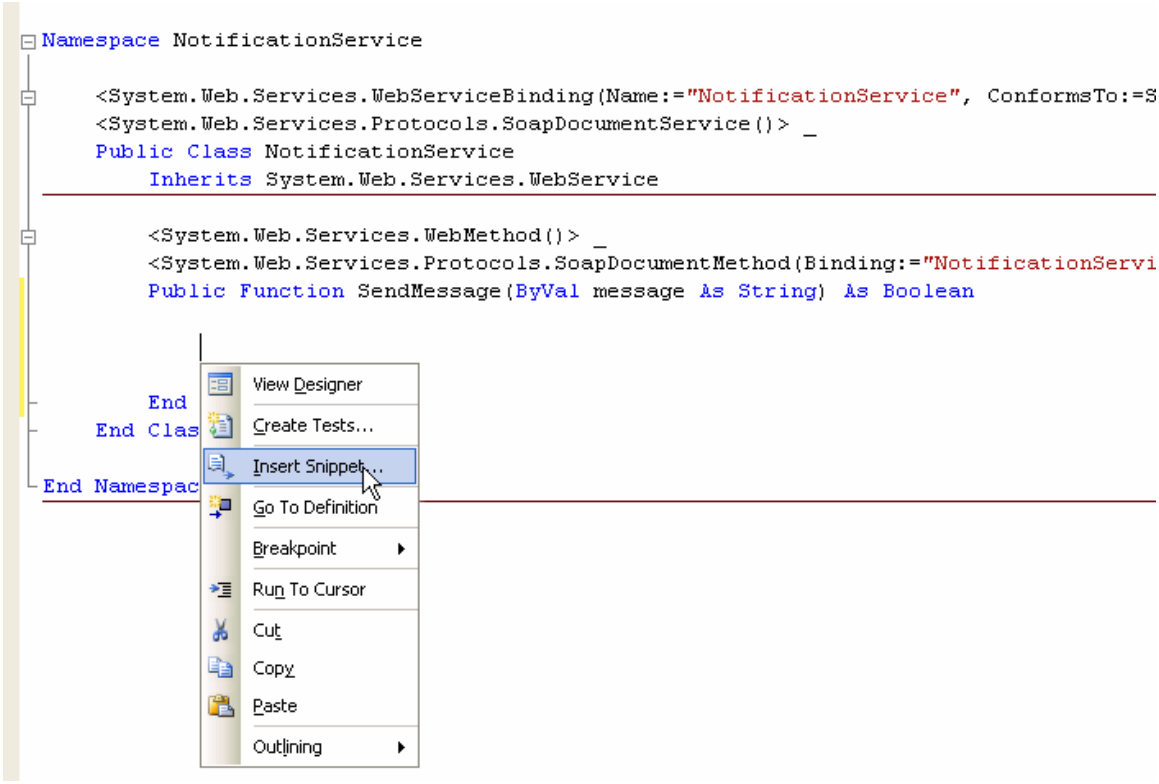
Actions	Once the **end point** changes to the icon as shown, you have successfully worked around the issue.



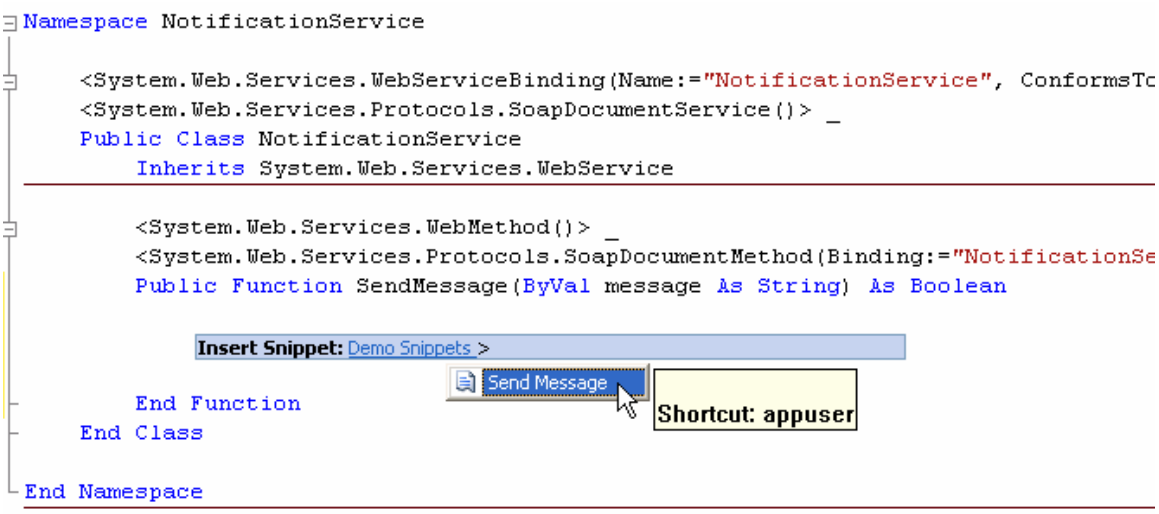
Actions	Select the **end point** on the **NotificationService** Right-click and choose **View Code**



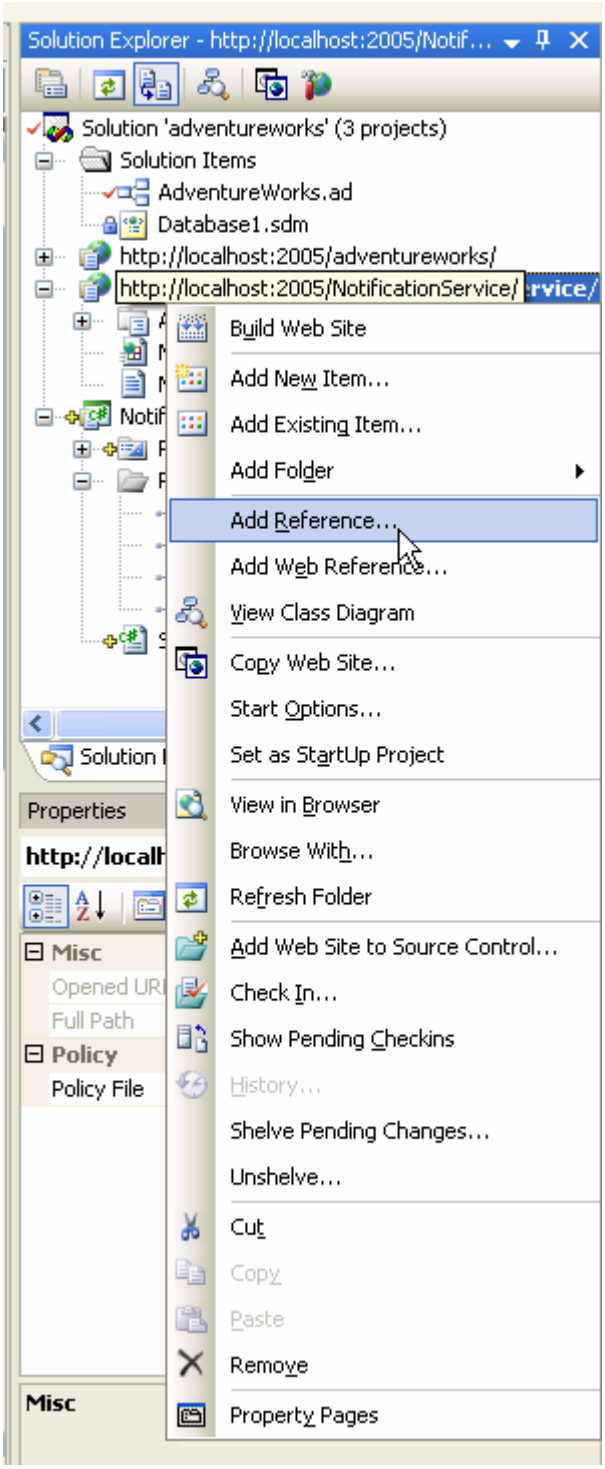
Actions	Inside the **SendMessage** method, right-click and choose **Insert Snippet …**



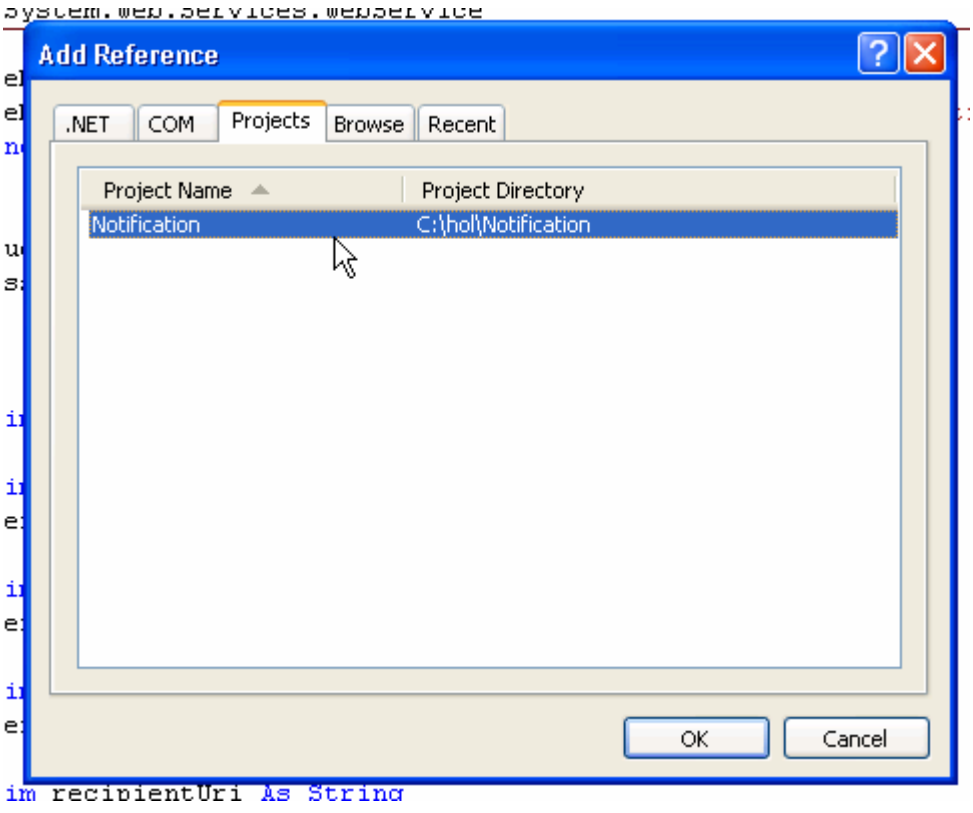
Actions	Choose the **Demo Snippets -> Send Message** snippet



Actions	Select the **http://localhost:2005/NotificationService** project Right-click and choose **Add Reference…**

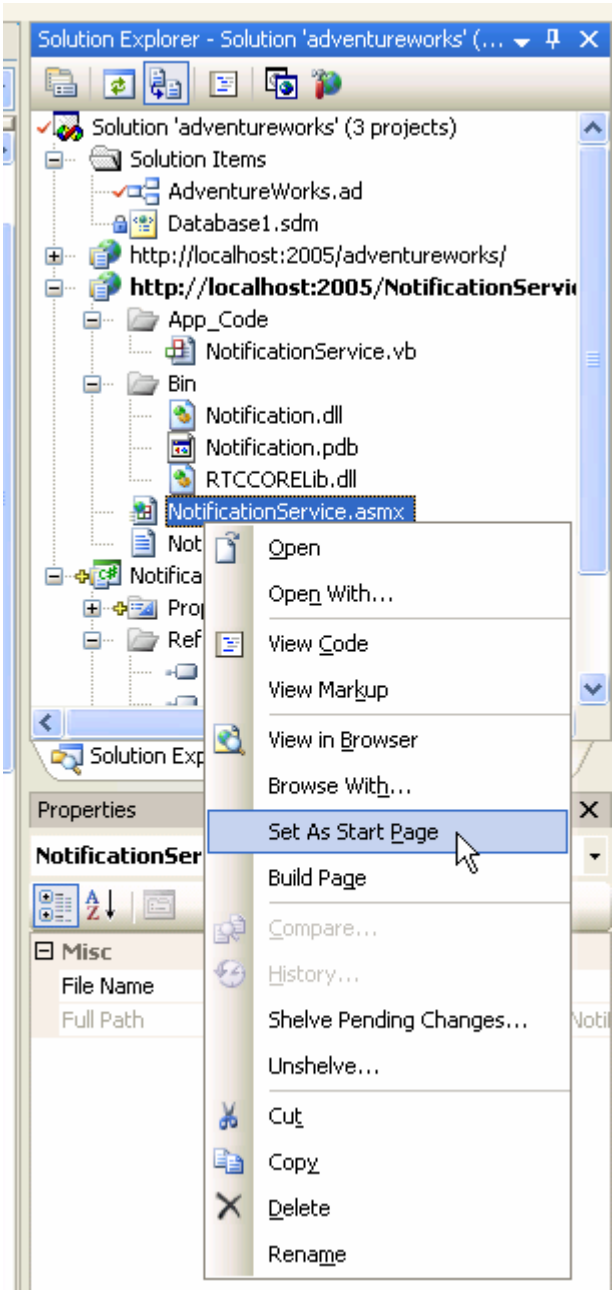


Actions	Under the **Projects** tab, choose **Notification** and press **OK**

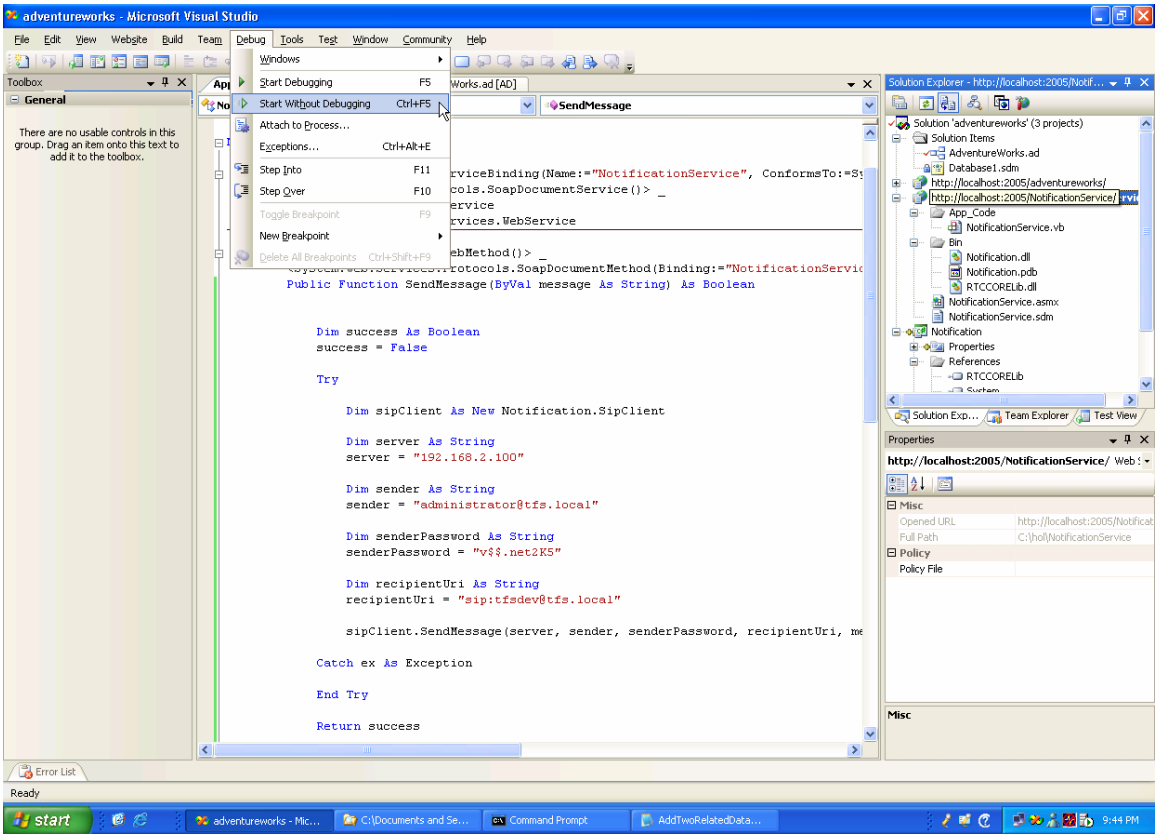


With our XML Web Service written, let's give it a try. We can invoke XML Web Services right within Internet Explorer.

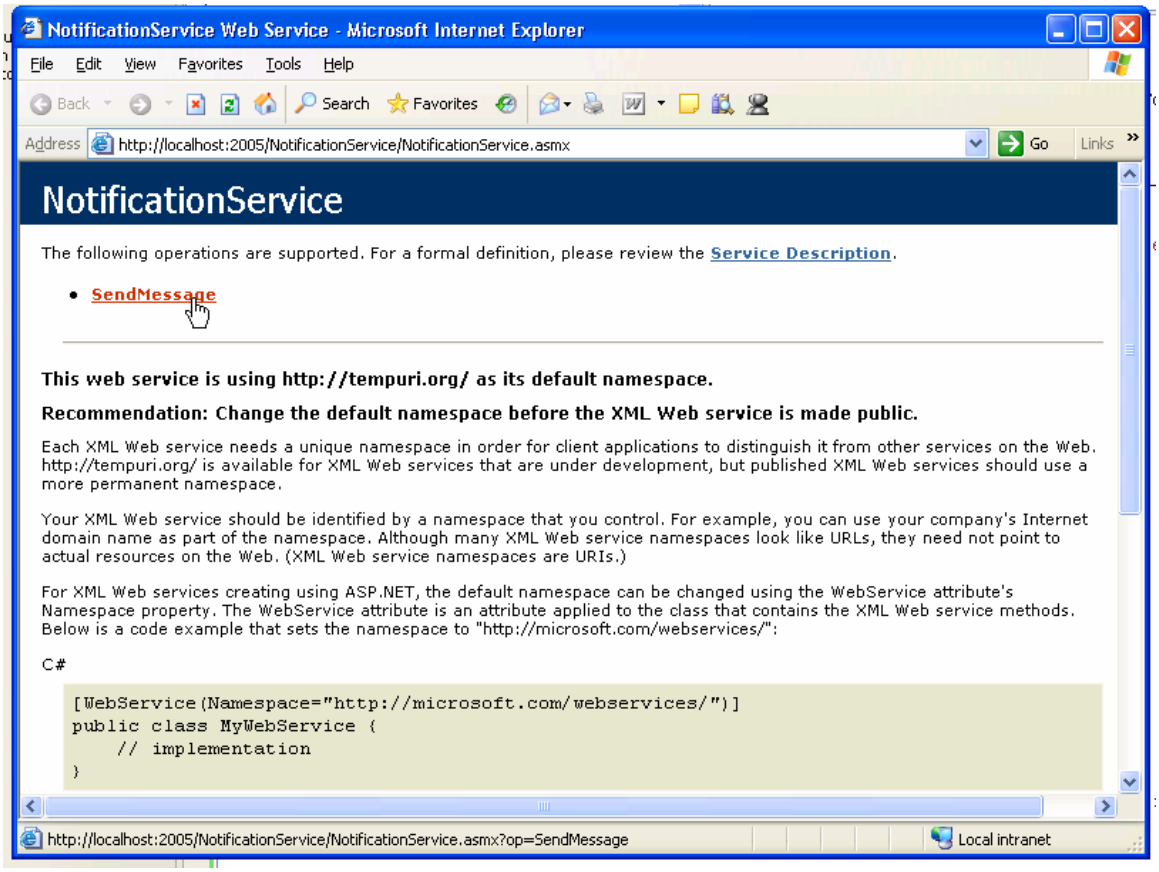
Actions	Select the **NotificationService.asmx** file, right-click and choose **Set As Start Page**



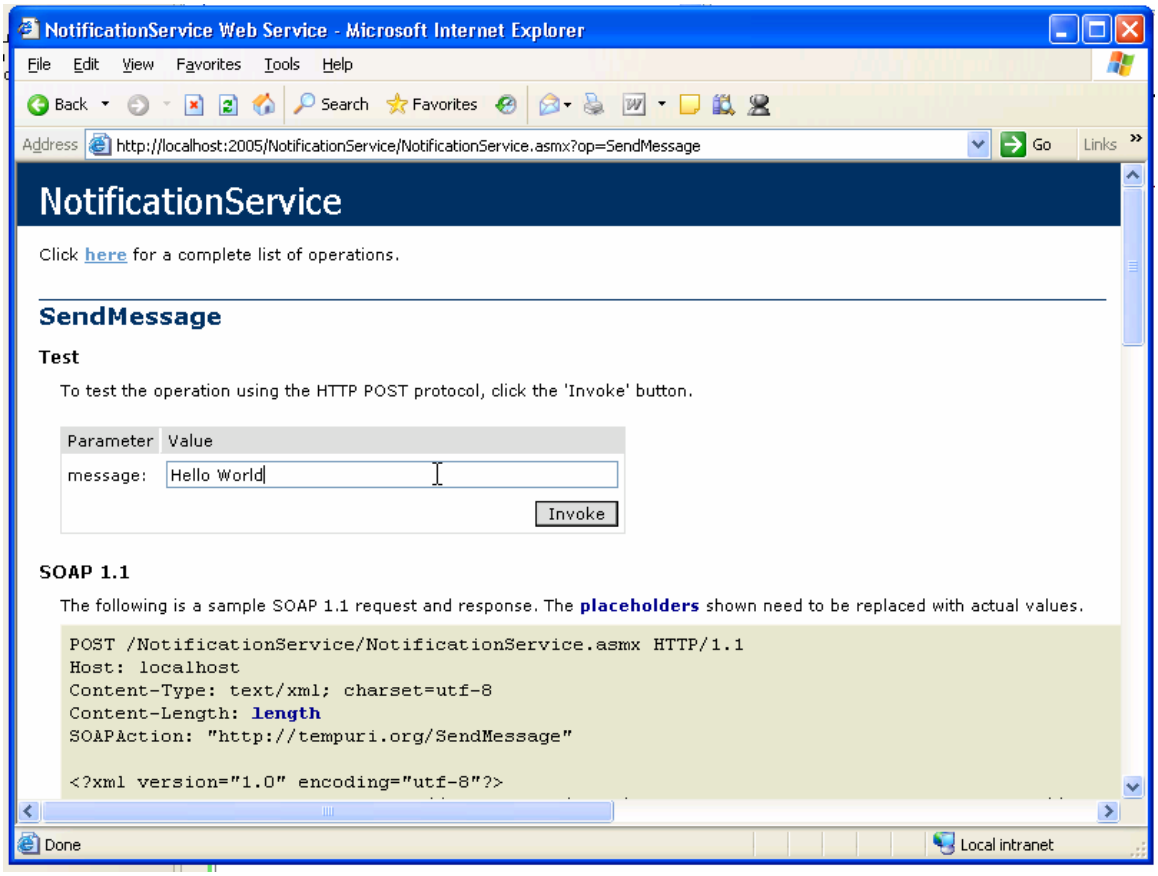
Actions	Choose the **Debug -> Start Without Debugging** menu option



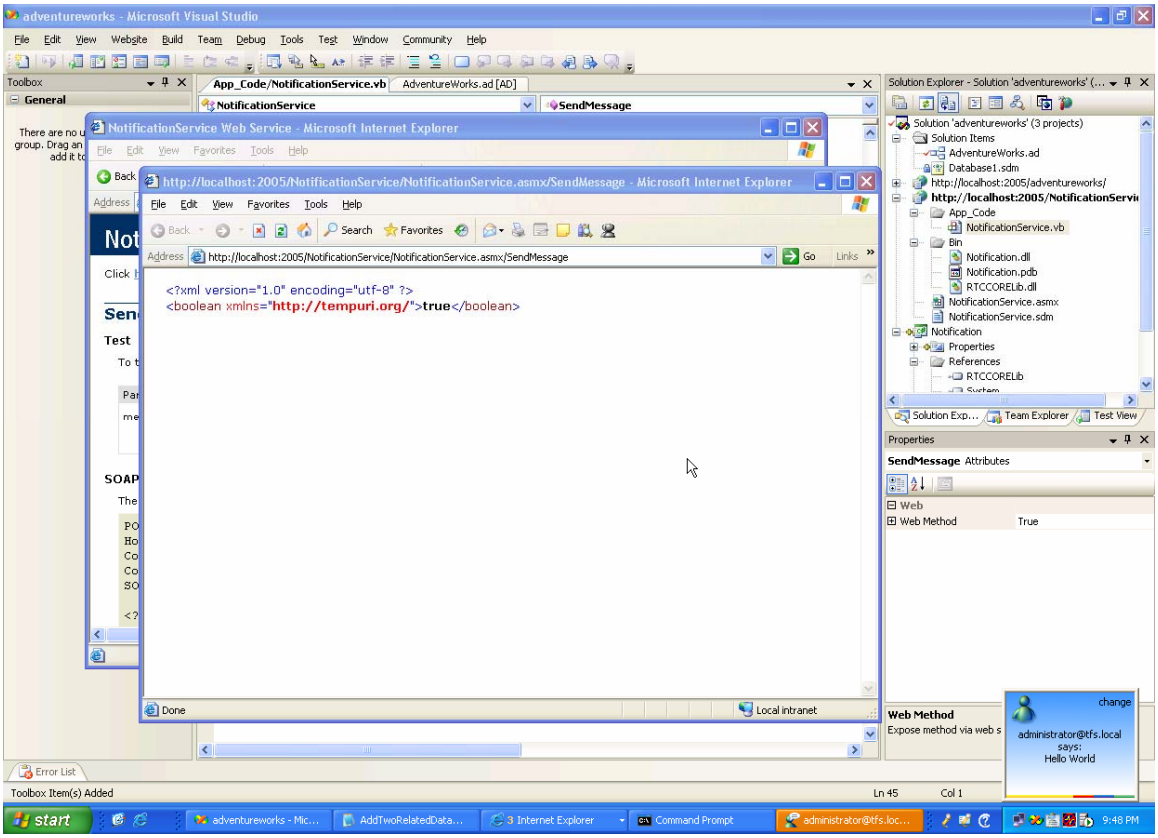
Actions	Press the **SendMessage** link



Actions	Type **Hello World** and press the **Invoke** button



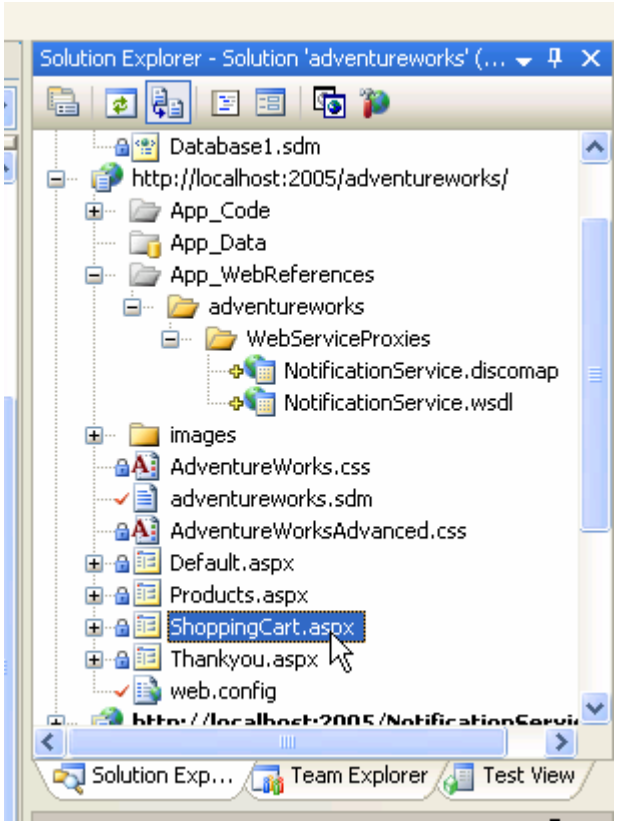
Actions	You should get your IM message! Close all your instances of **Internet Explorer**



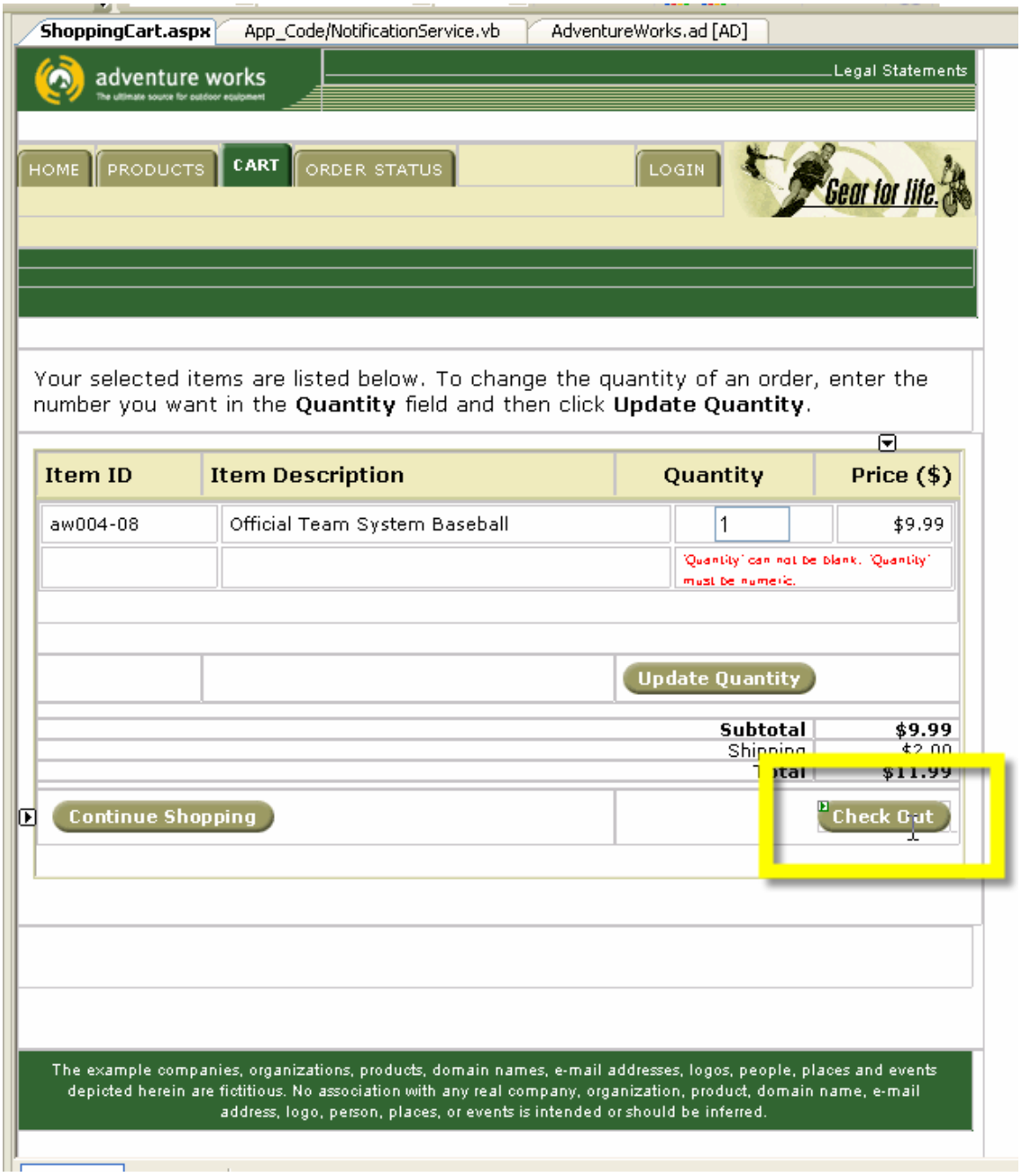
Our XML Web Service is ready, so now we just have to hook it up to our ASP.NET application.

Visual Studio Team System helps make this a productive process by generating all of the web references we need. All we need to do is find the place where we want to call our new web service.

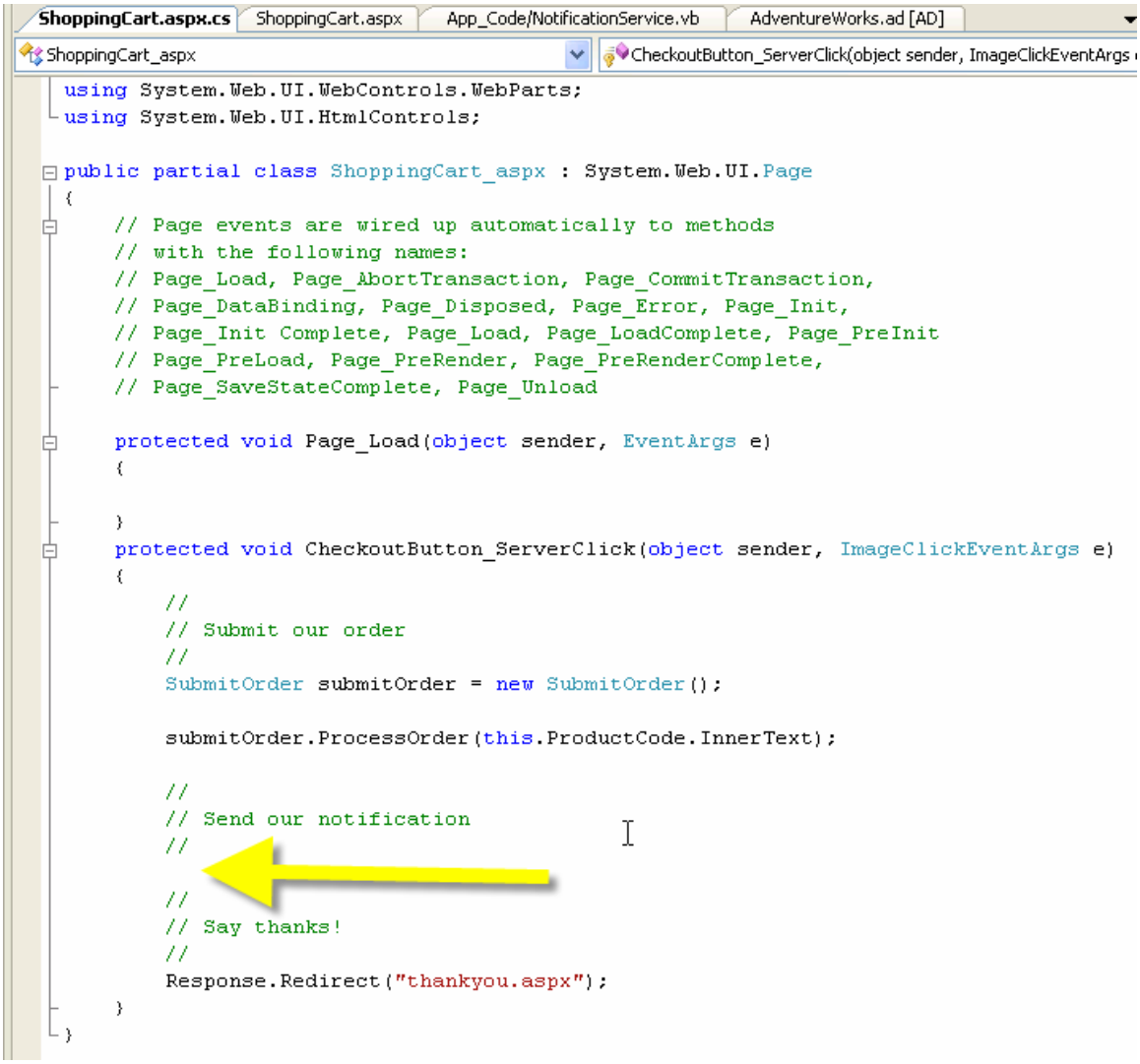
Actions	Select the **shoppingcart.aspx** file under your **http://localhost:2005/adventureworks/** project Right-click and choose **View Designer**



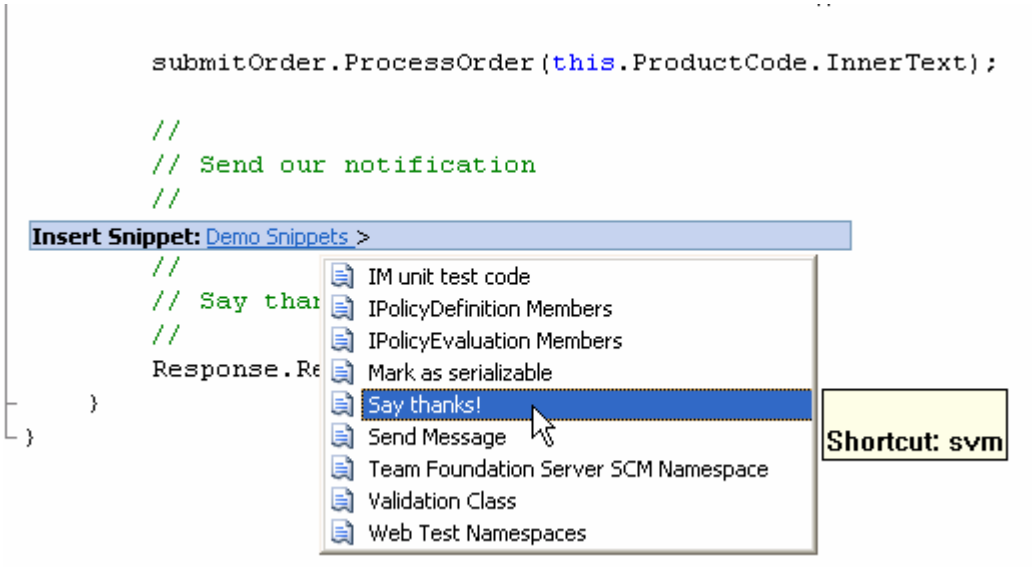
Actions	Double-click on the **Check Out** button



Actions	Put your cursor in the location shown

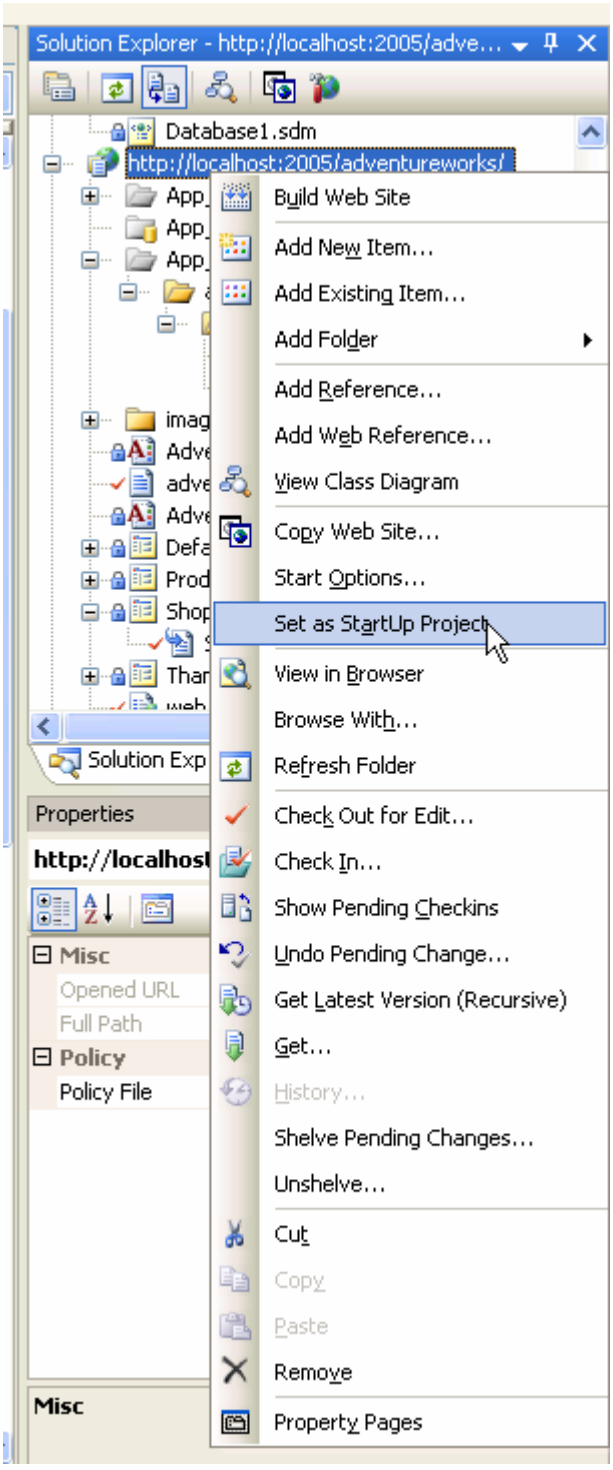


Actions	Right-click and choose **Insert Snippet** Choose **Demo Snippets -> Say Thanks!**

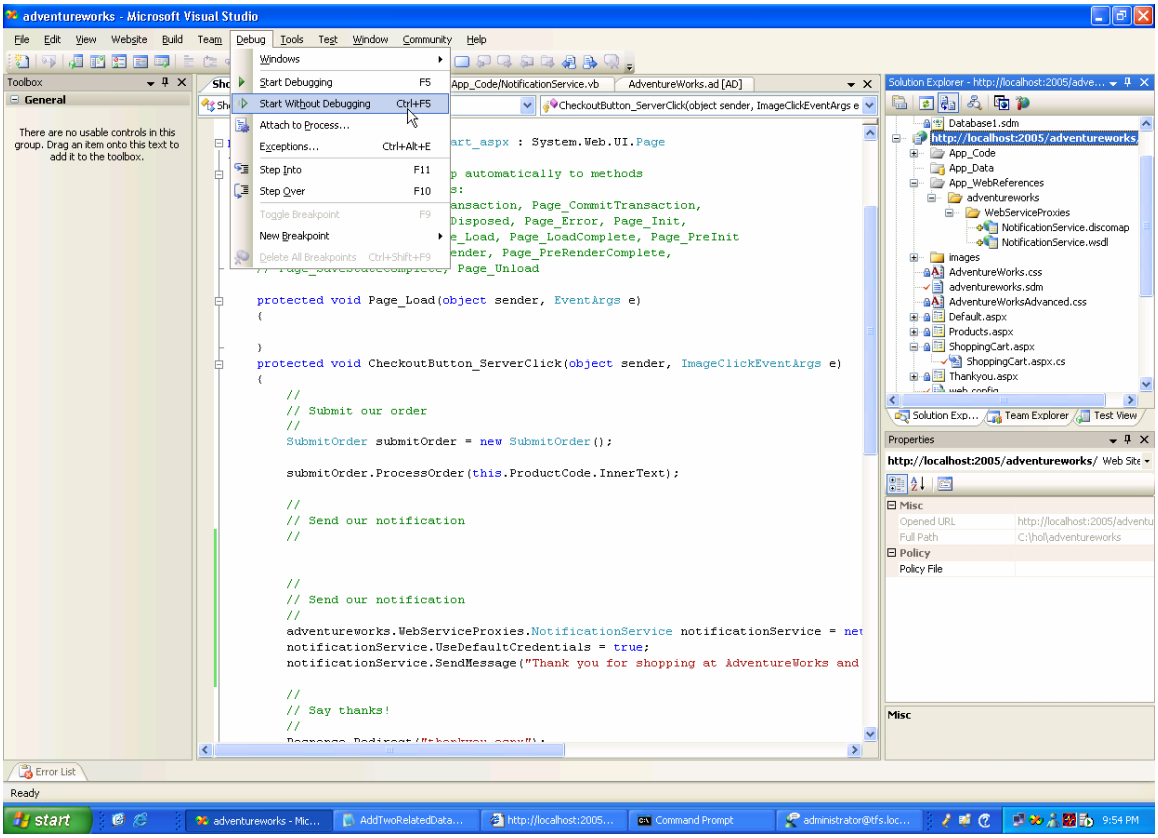


With our code written, let's try to use our new feature.

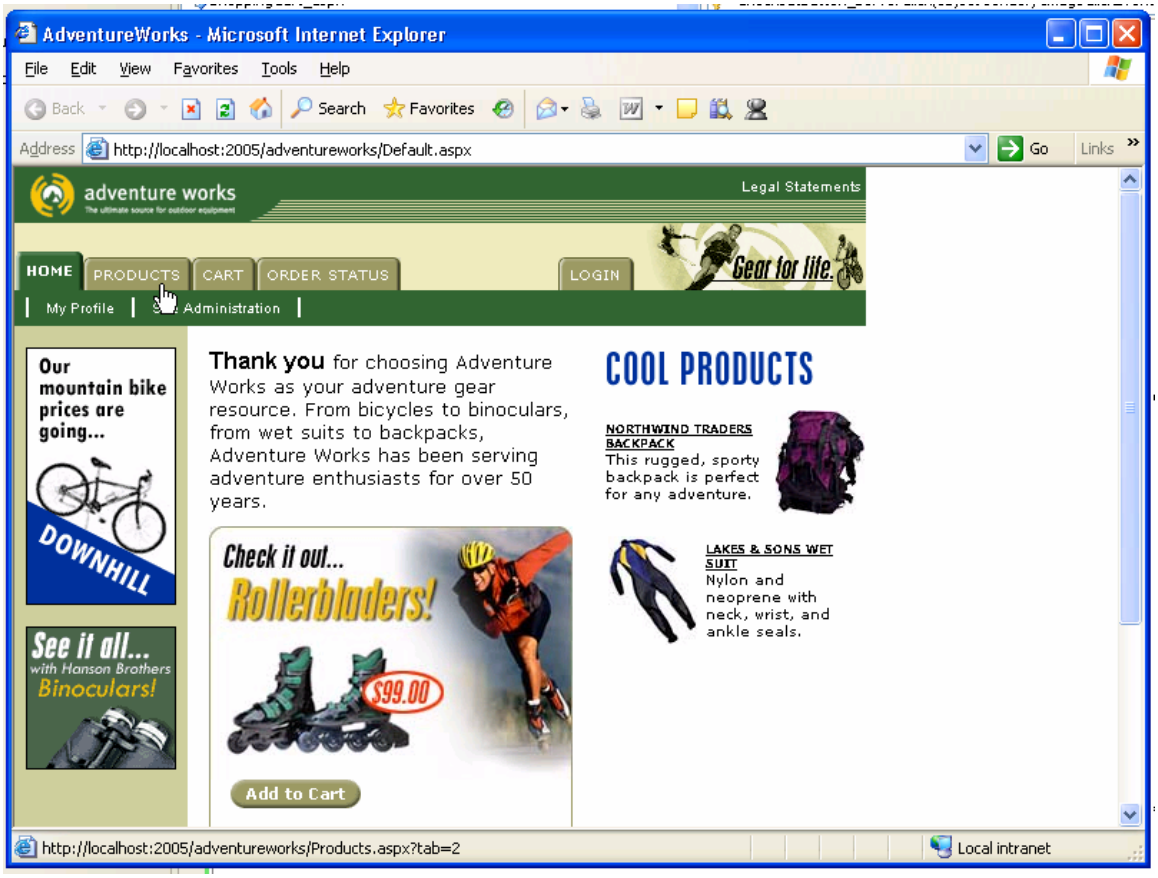
Actions	Select **http://localhost:2005/adventureworks** Right-click and choose **Set as StartUp Project**



Actions	Choose the **Debug -> Start Without Debugging** menu option

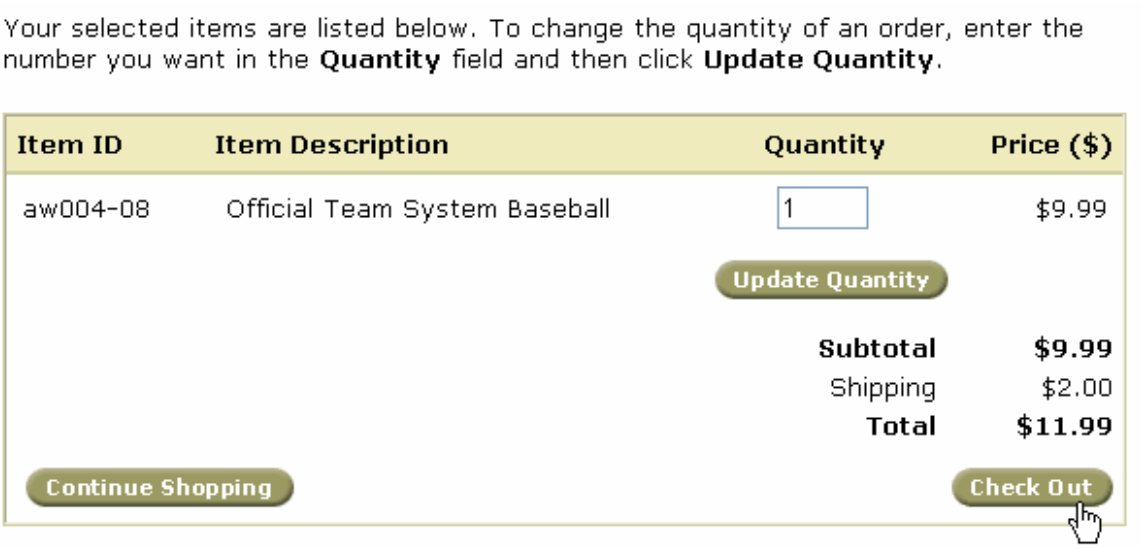


Actions	Press the **Products** tab

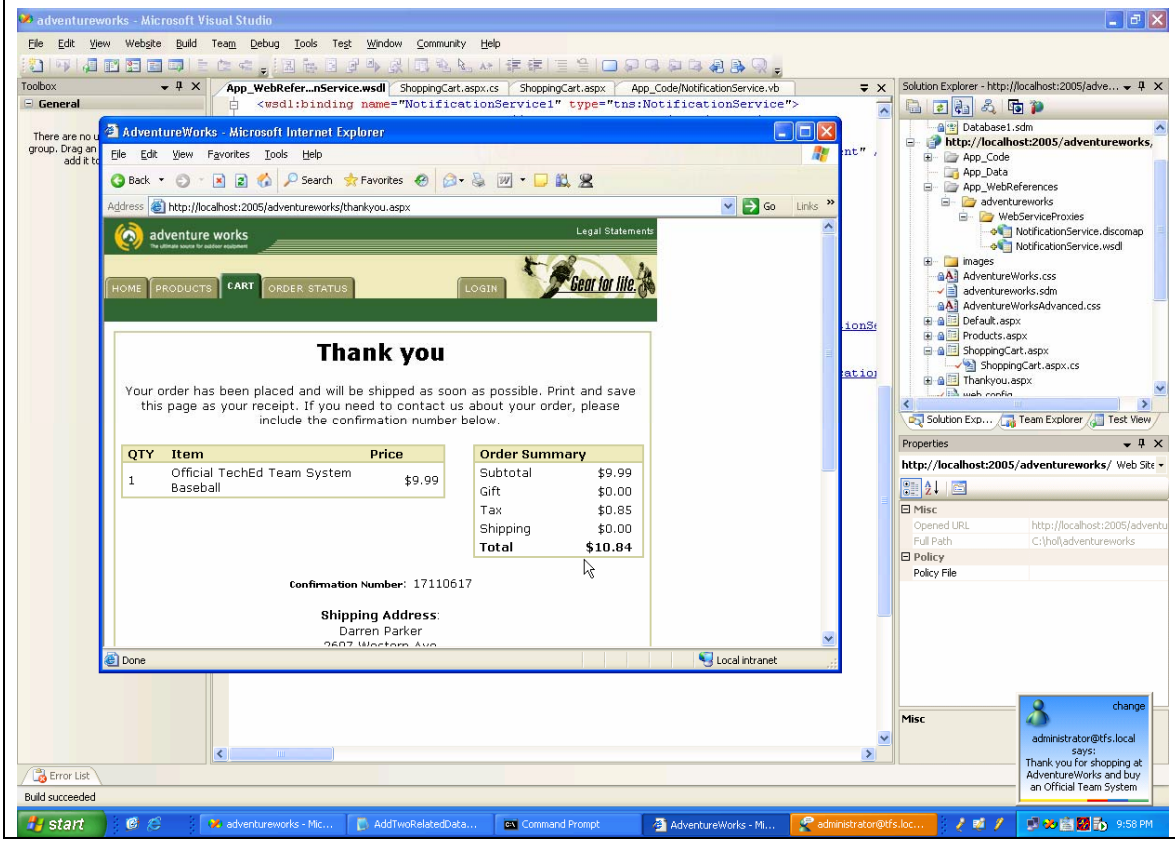


Actions	Press the **Add to Cart** button

Actions	Press the **Check Out** button



Actions	You should get an IM thanking you for your purchase

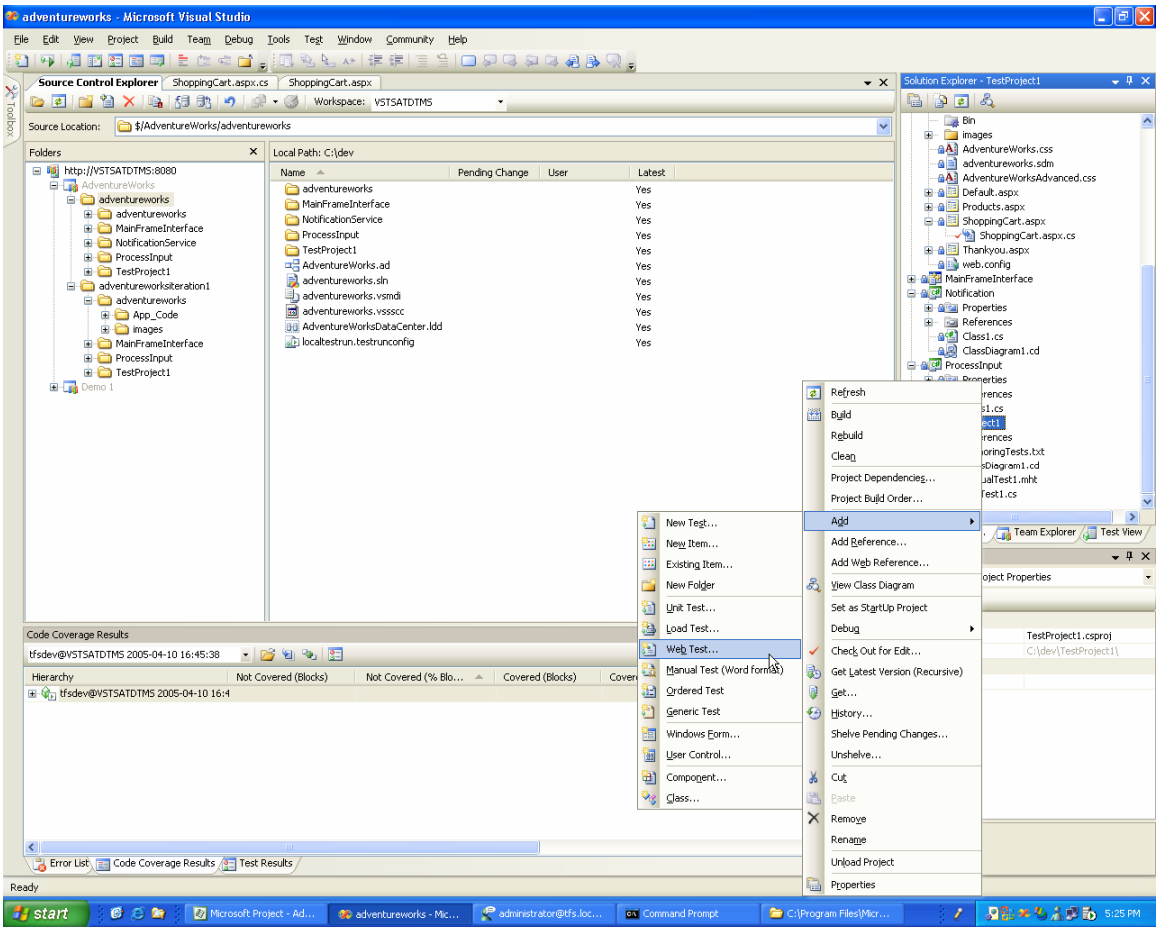


We have verified the functionality of our application – now let's make that verification a part of our development process. Visual Studio Team System supports the notion of a web test, which we will see in the following section.

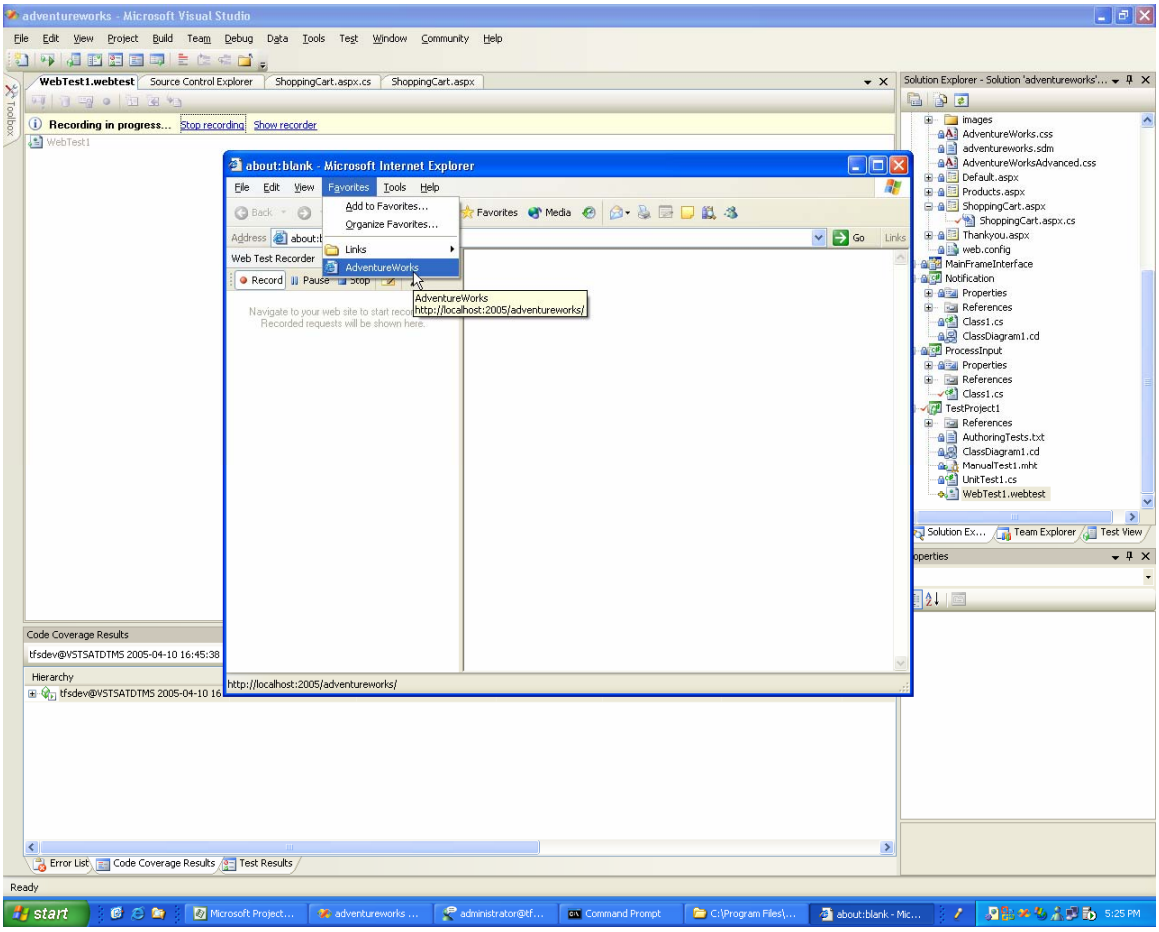
Exercise - Web Testing

Visual Studio Team System supports a wide range of test types; we'll create a web test that allows us to record the way we use our application. We'll be able to play it back later.

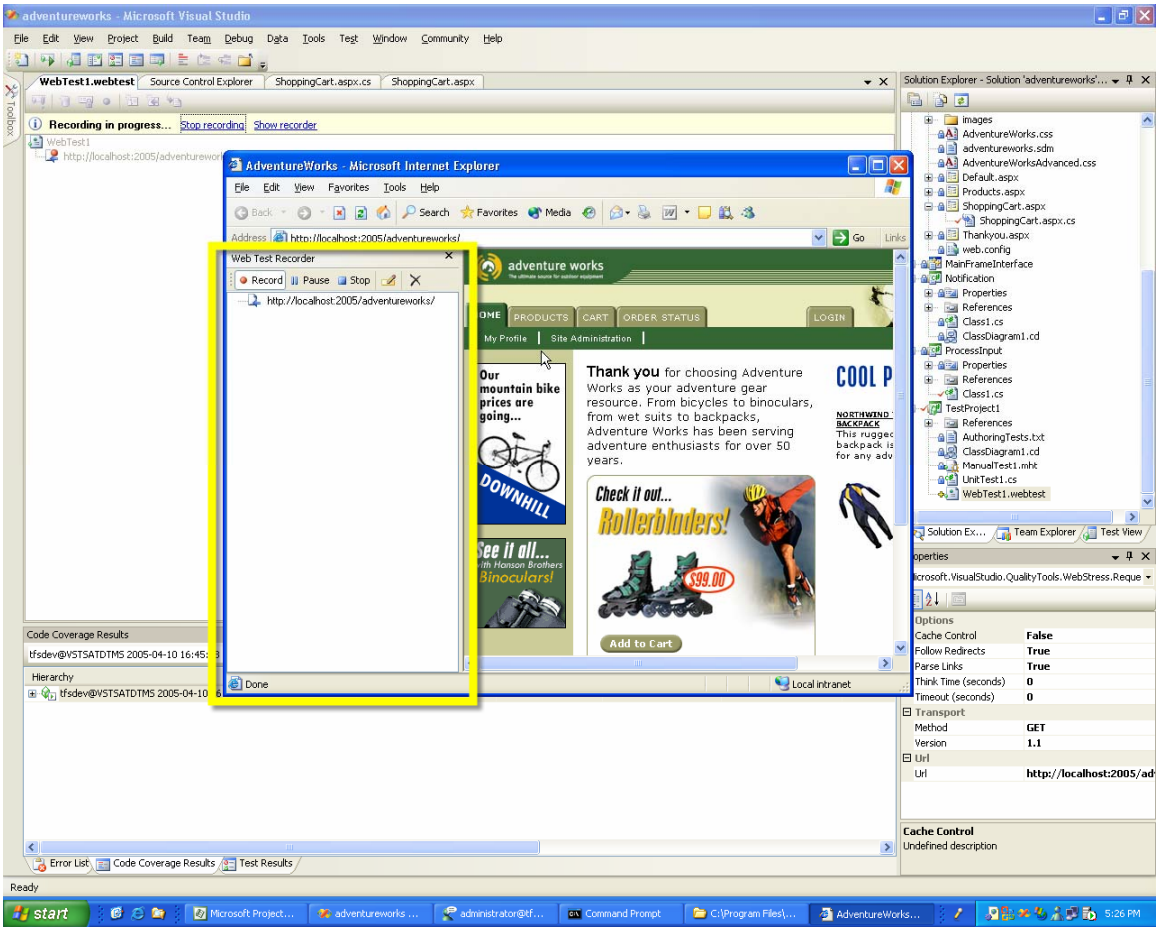
Actions	If you are doing this section of the lab independently, you will have to add a **Test Project** to your solution; see the Exercise - Test Driven Development for details. Select your **Test Project**, right-click and choose **Add -> Web Test**



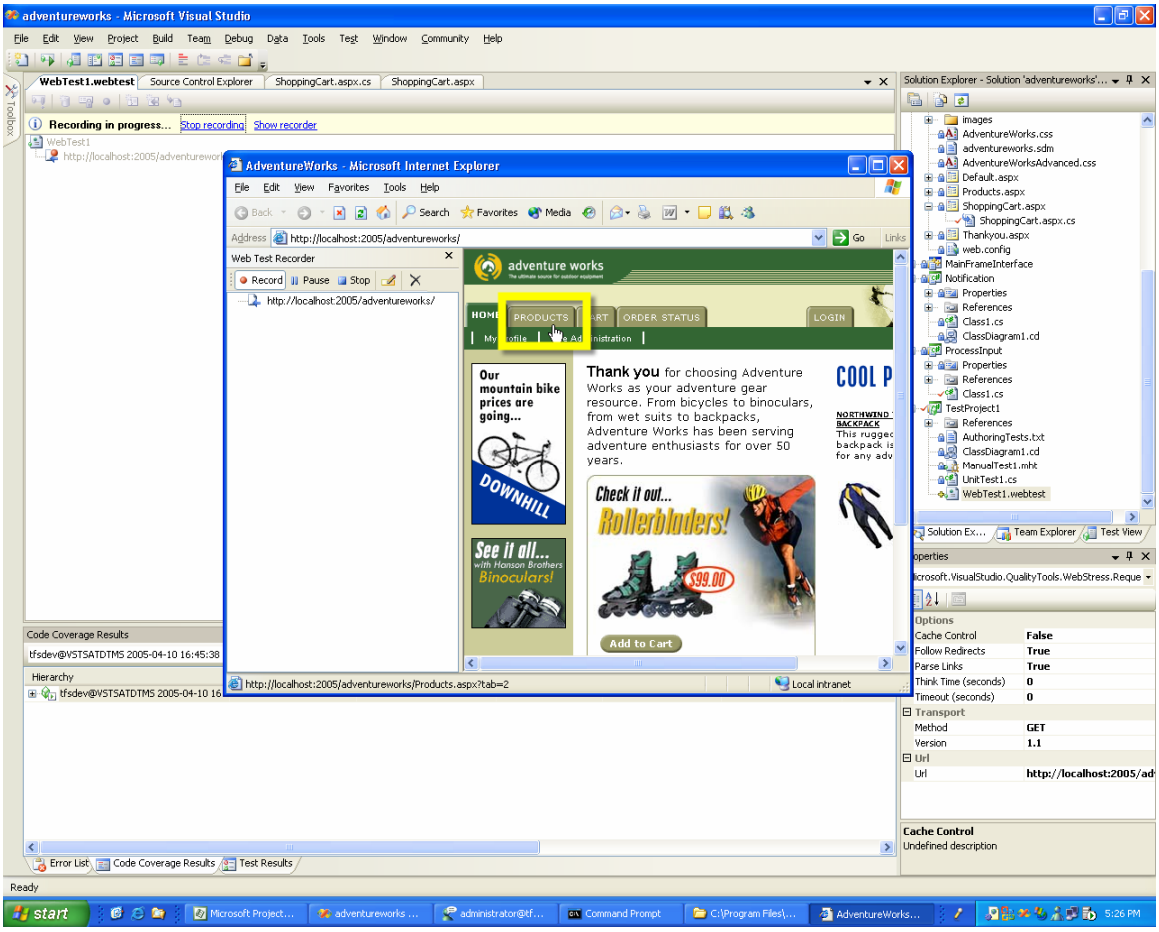
Actions	When **Internet Explorer** opens, choose **Favorites -> AdventureWorks**



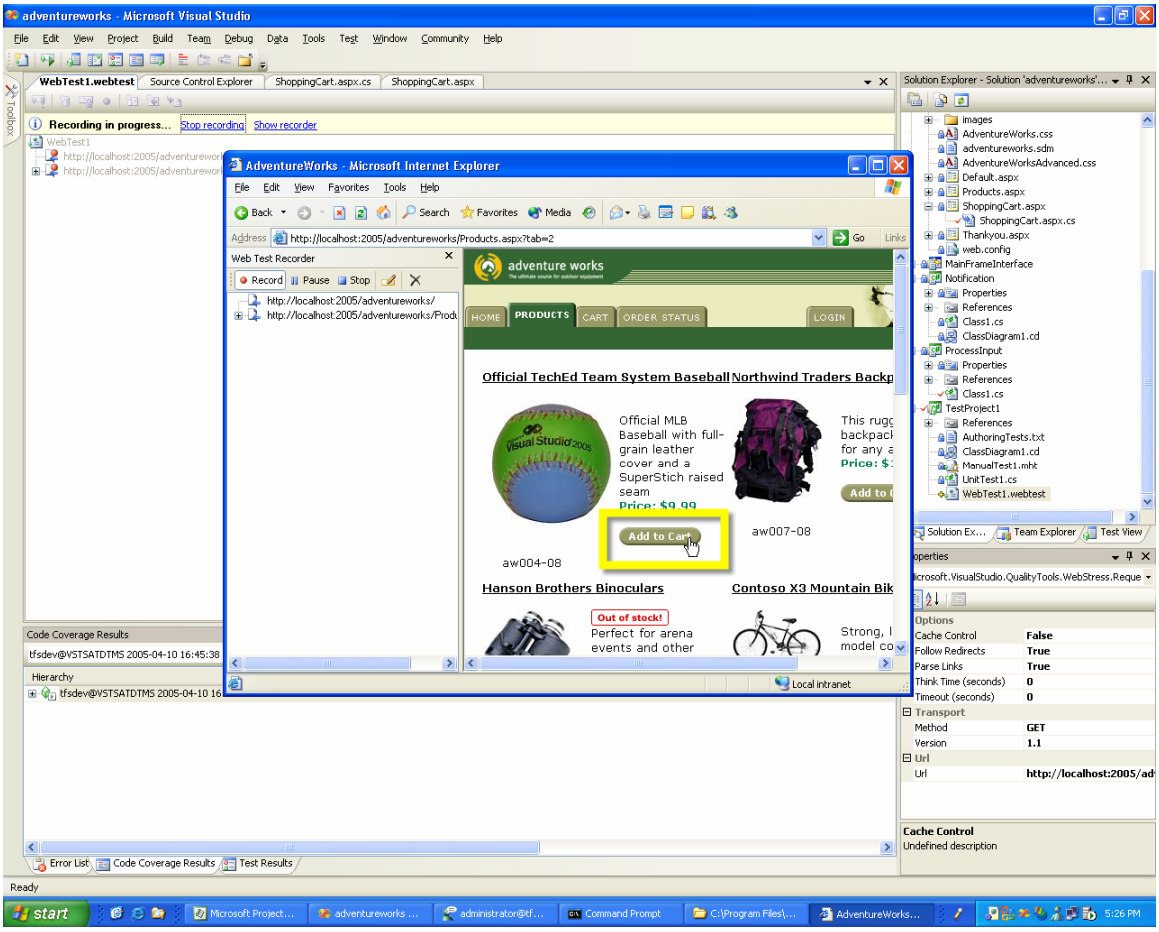
Actions	Notice the window that will record your requests



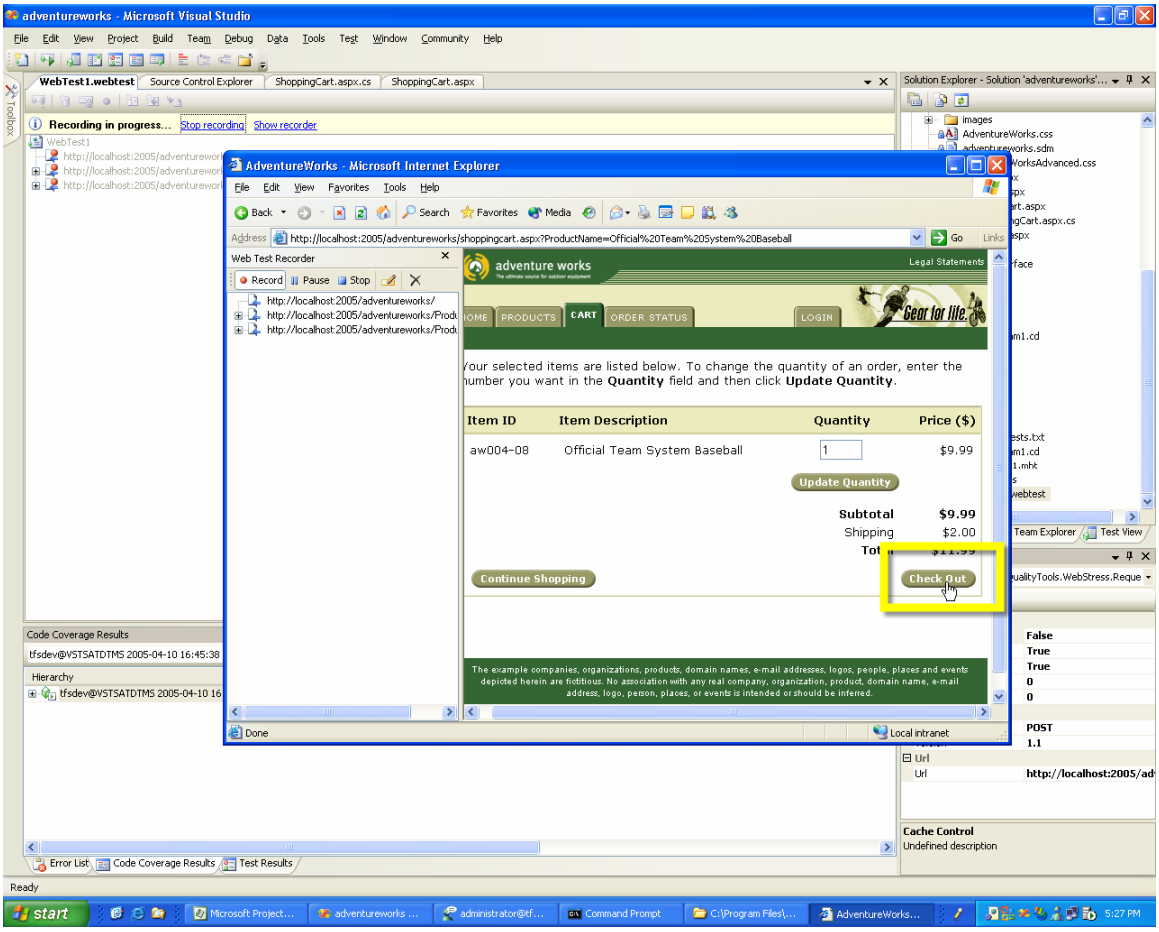
Actions	Click the **Products** tab



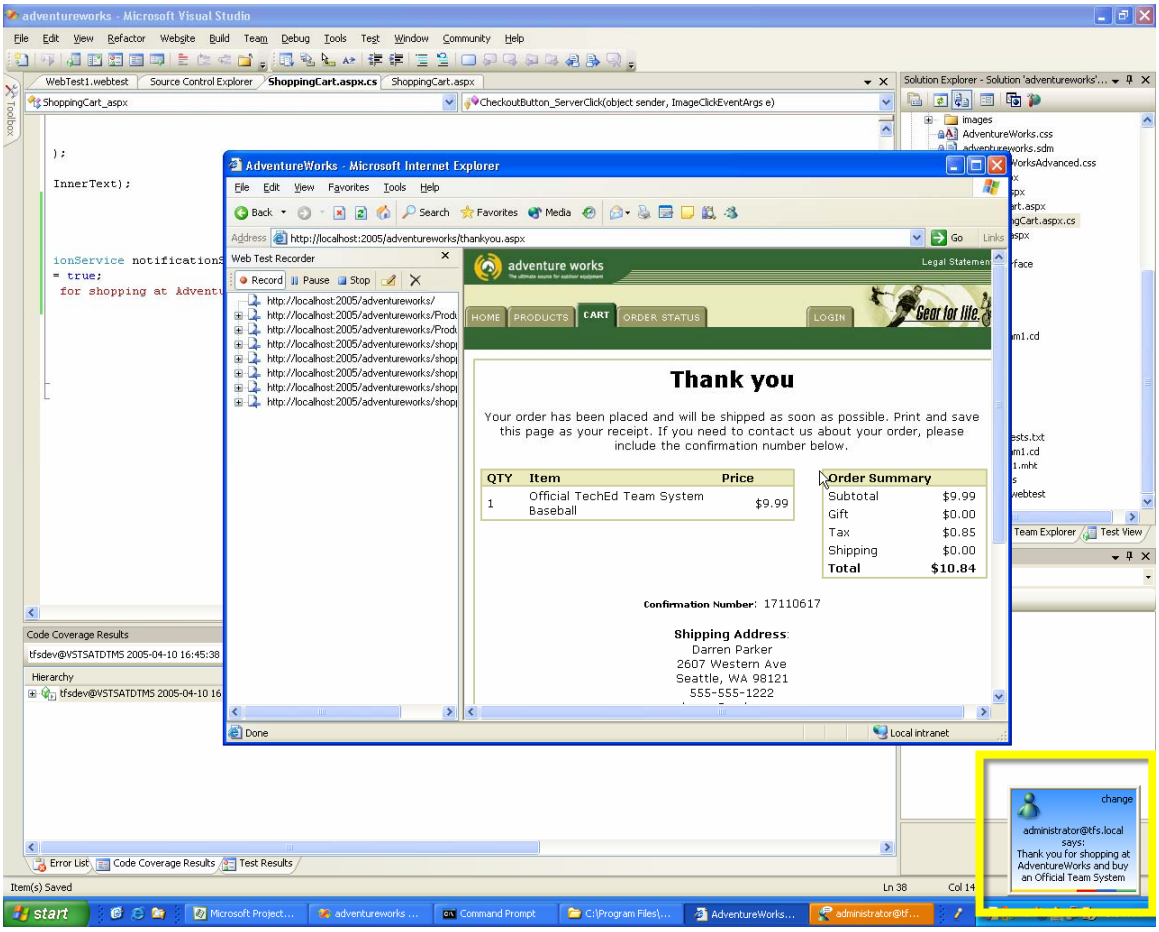
Actions	Press the **Add to Cart** button



Actions	Press the **Check Out** button



Actions	Notice the IM message that is sent Press the **Stop** button on the recording panel tool bar



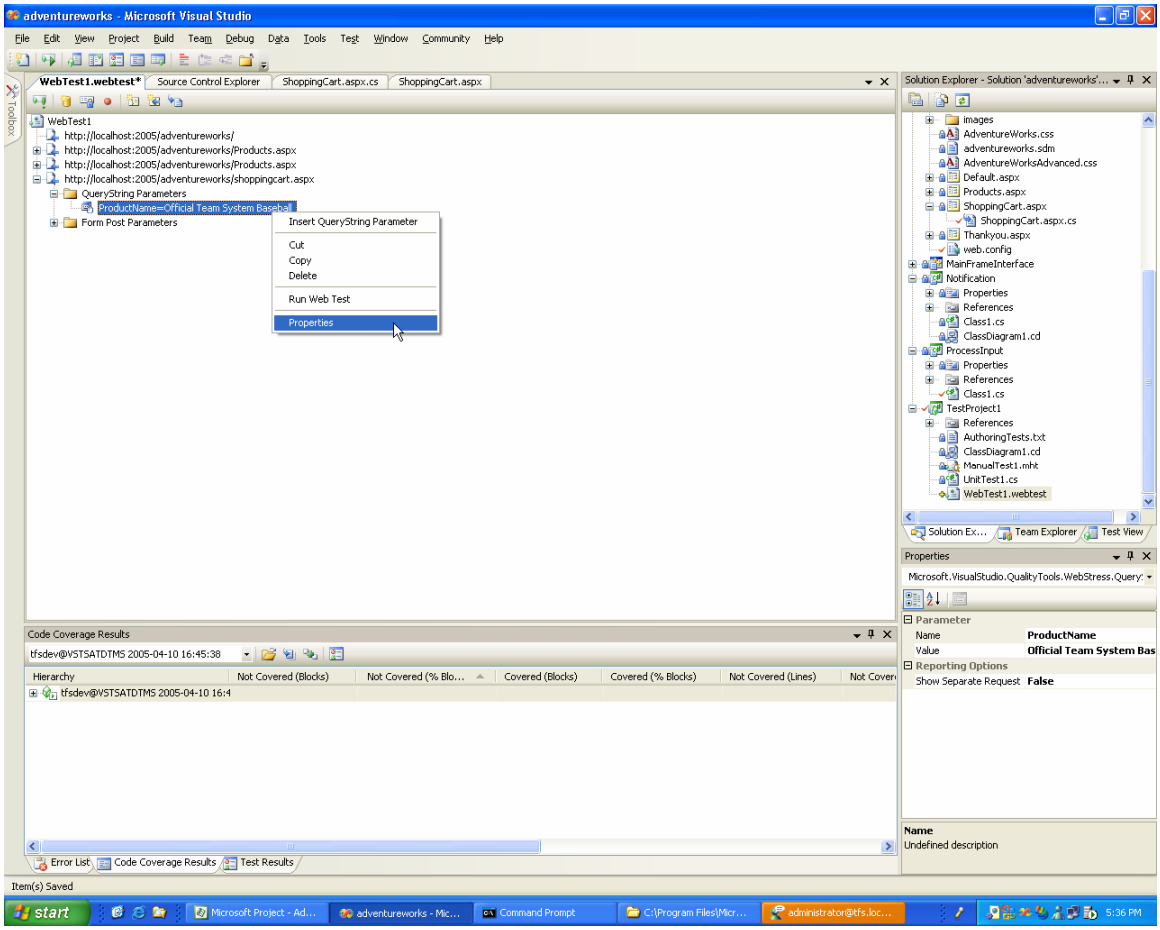
Now we have a test that verifies the functionality of our web application; however, there are a few problems with our test. Our test always buys the same item (baseballs), it only represents one user, and we haven't defined what a 'pass' or a 'fail' means.

We will address each of these problems in the following sections.

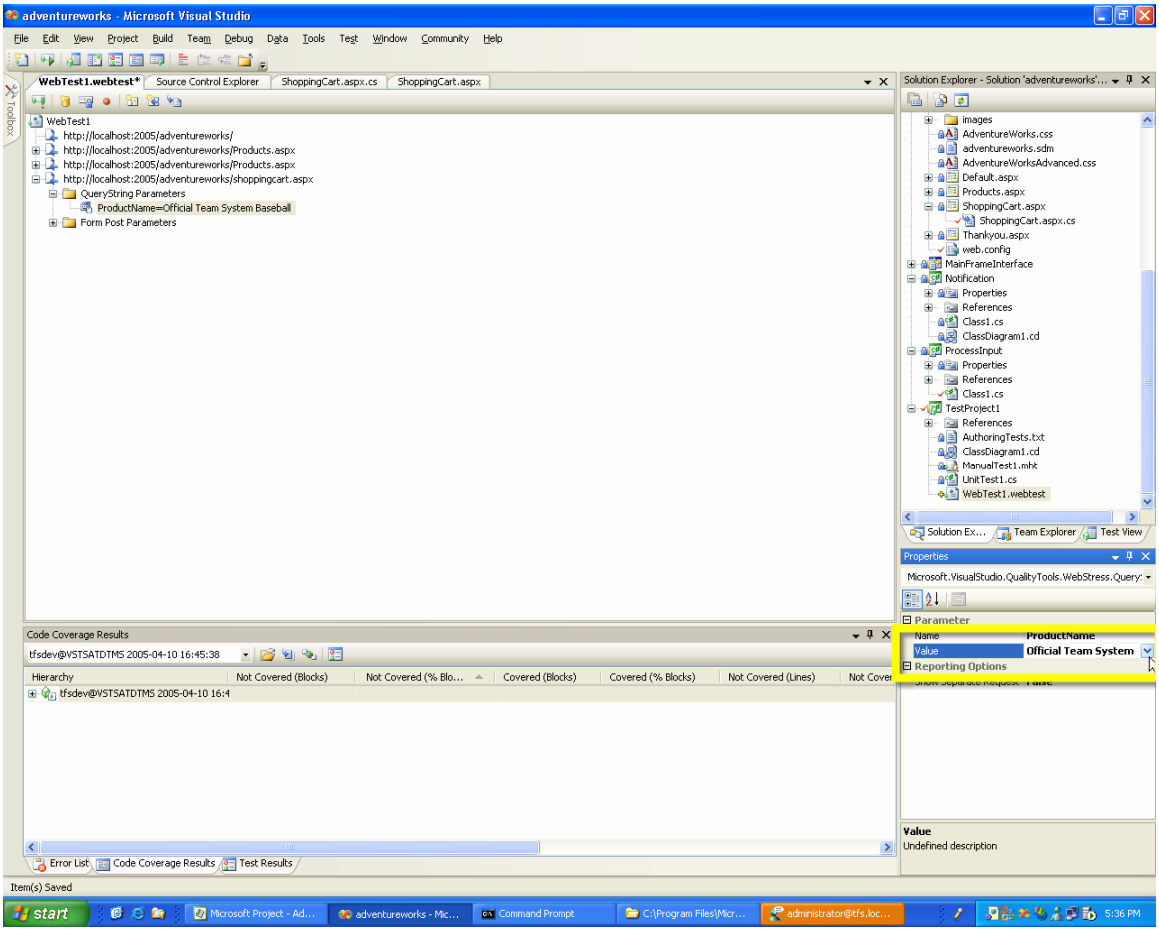
Exercise - Data Driven Testing

Almost all of the test types in Visual Studio Team System can be driven with data pulled from a database. We are going to replace the hard coded 'Team System Baseball' that we recorded with a value that we pull from a database.

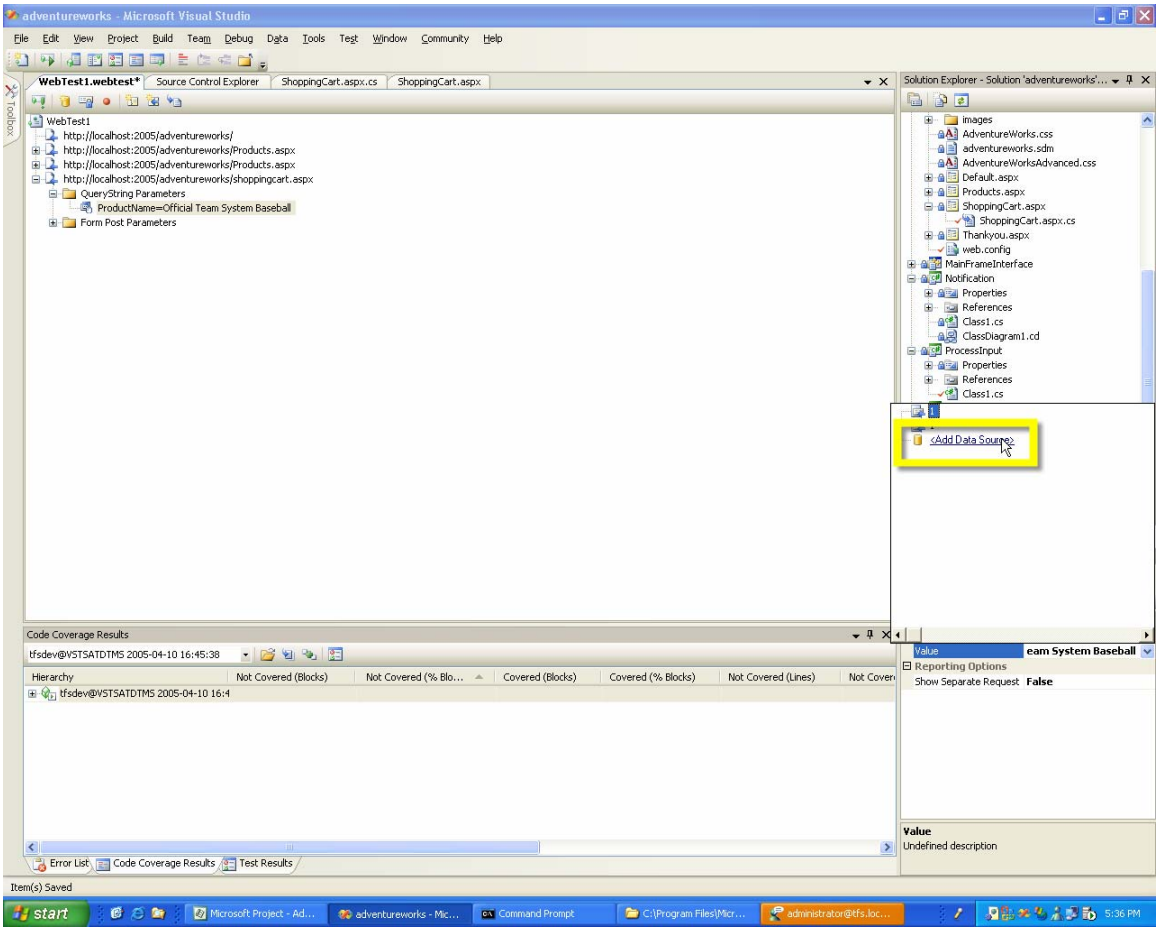
Actions	Right-click on the **Product Name** node and choose **Properties**



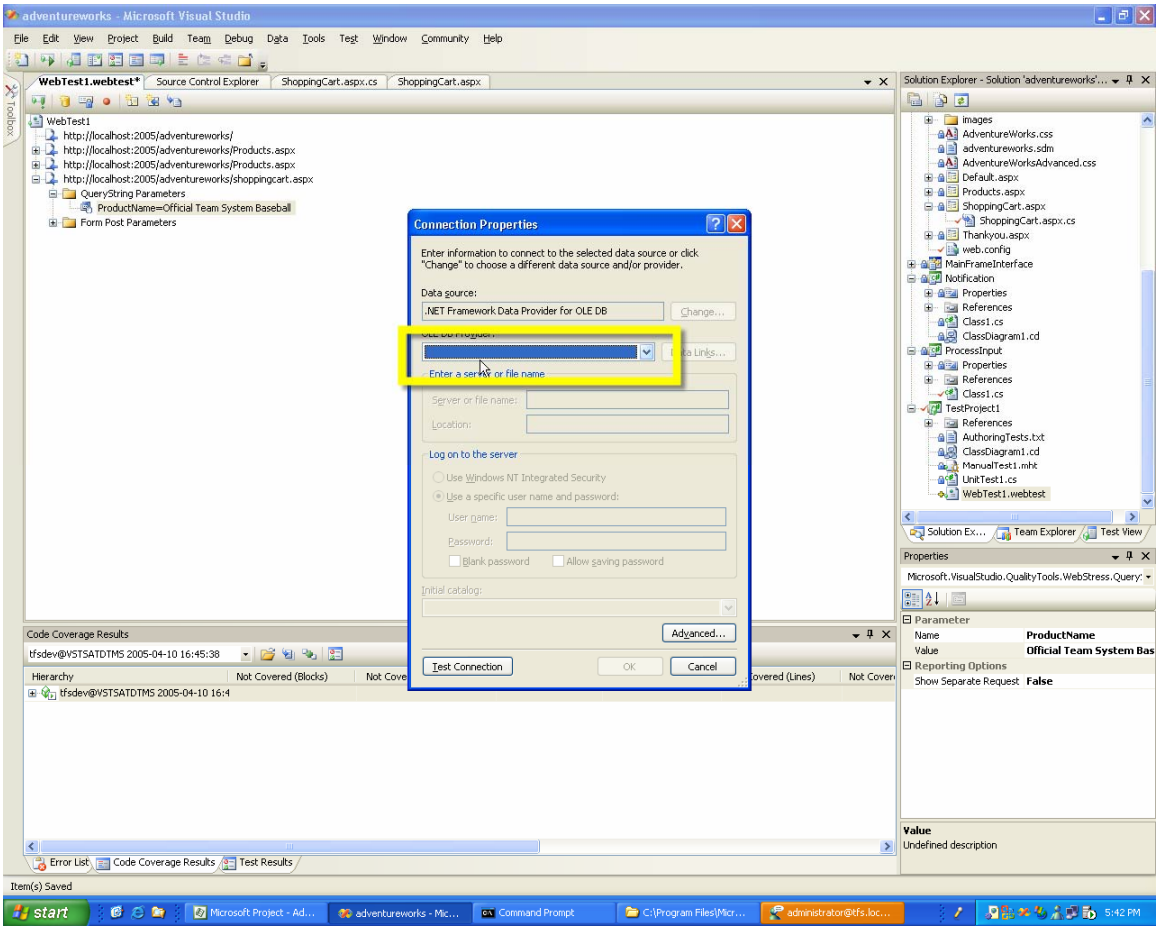
Actions	Pull the value picker down for the **Value** option



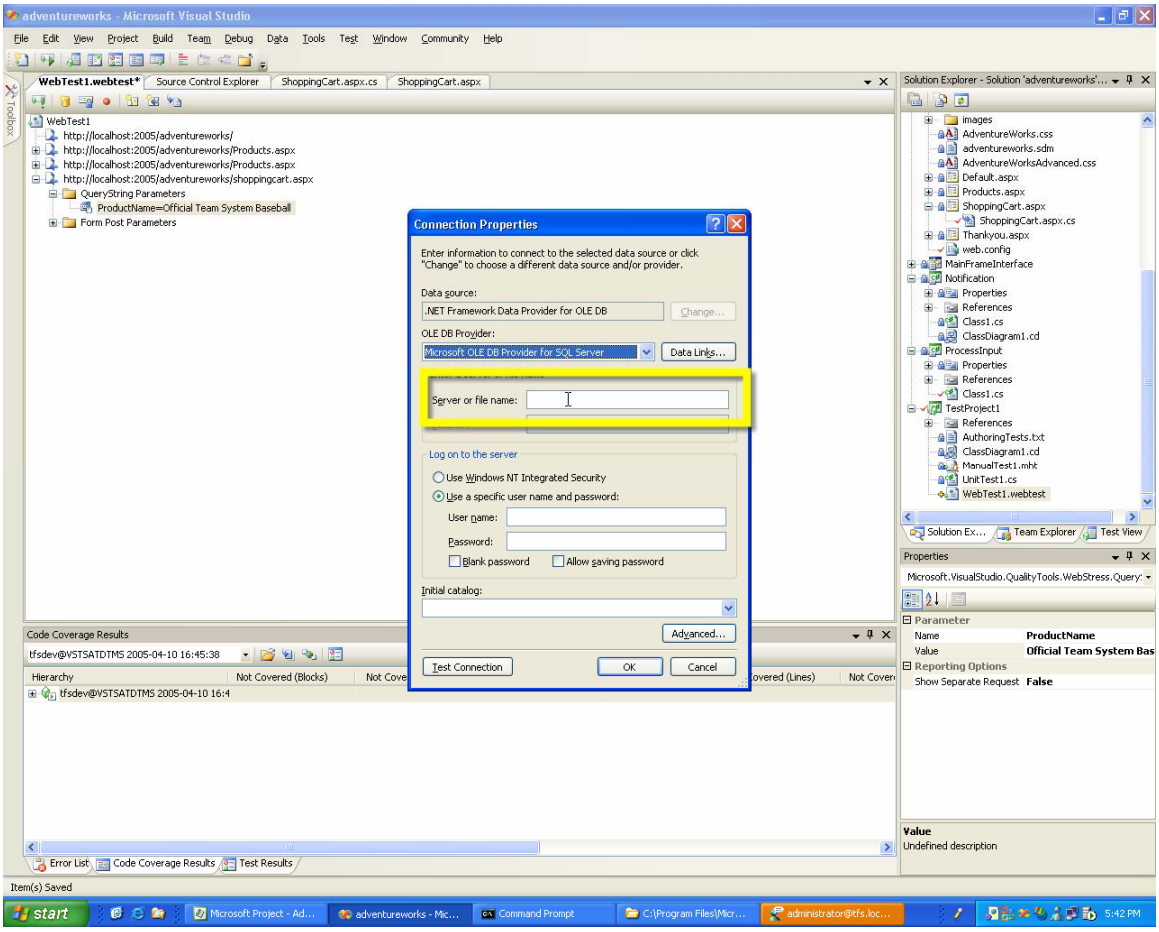
Actions	Choose the **Add Data Source** option



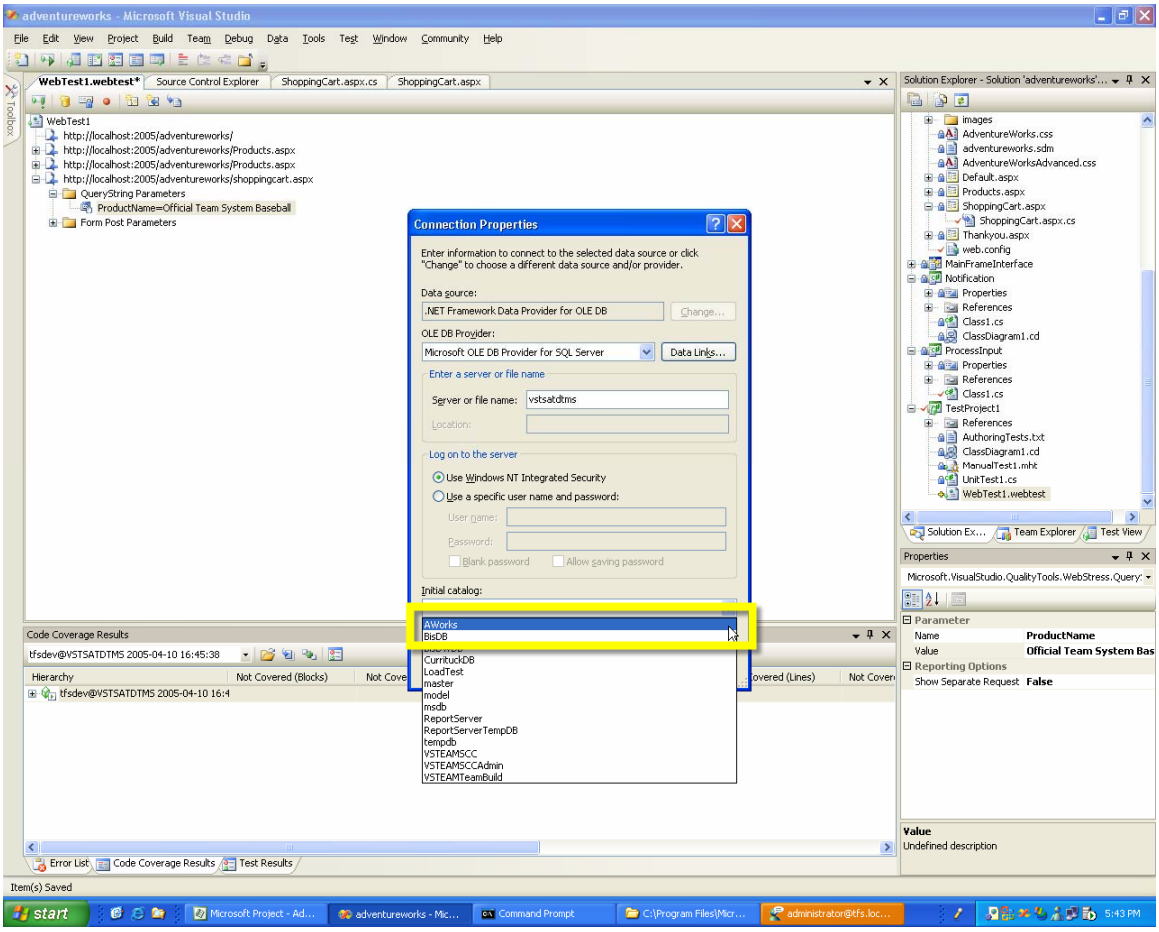
Actions	Choose the **Microsoft OLE DB Provider for SQL Server** as the **OLE DB Provider**



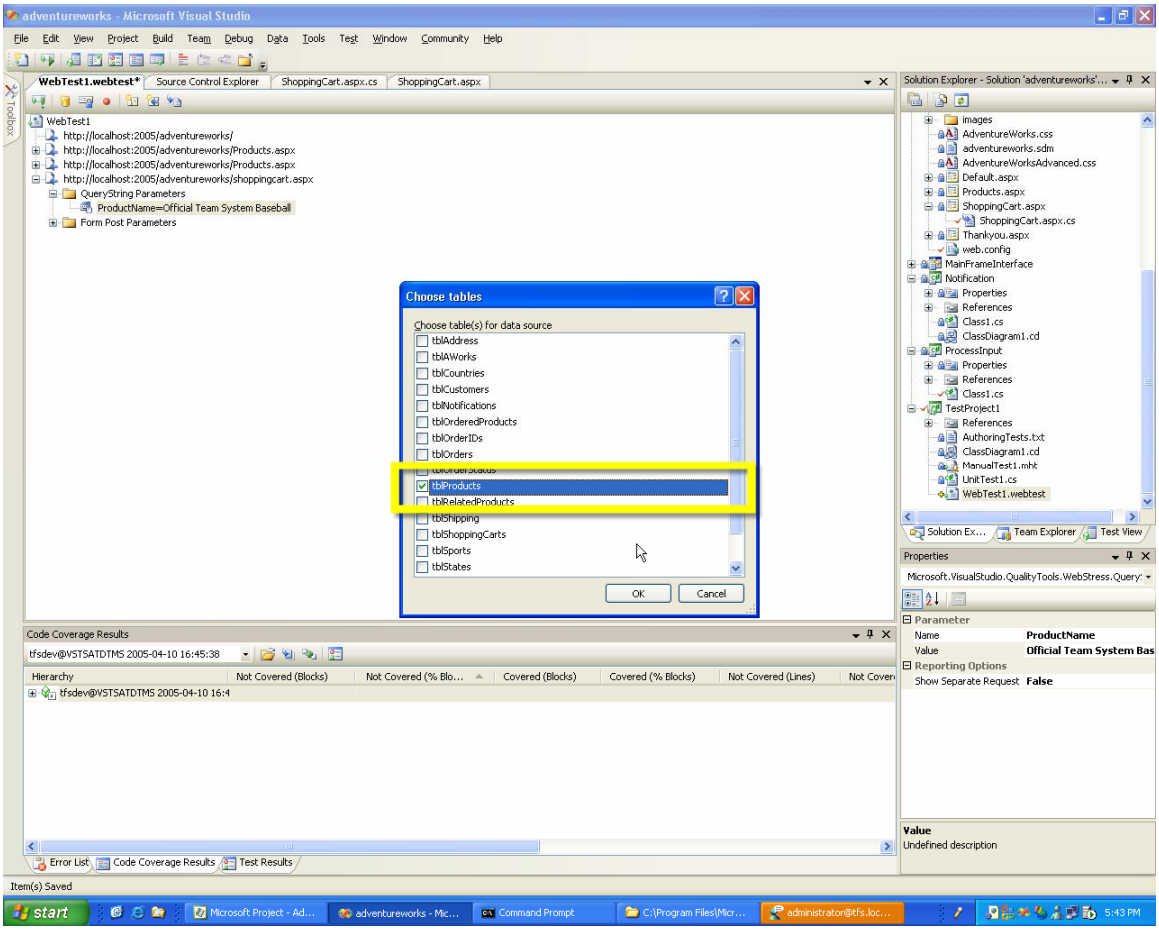
Actions	Enter **CAMTSVSTSSS01** as the **Server or file name**



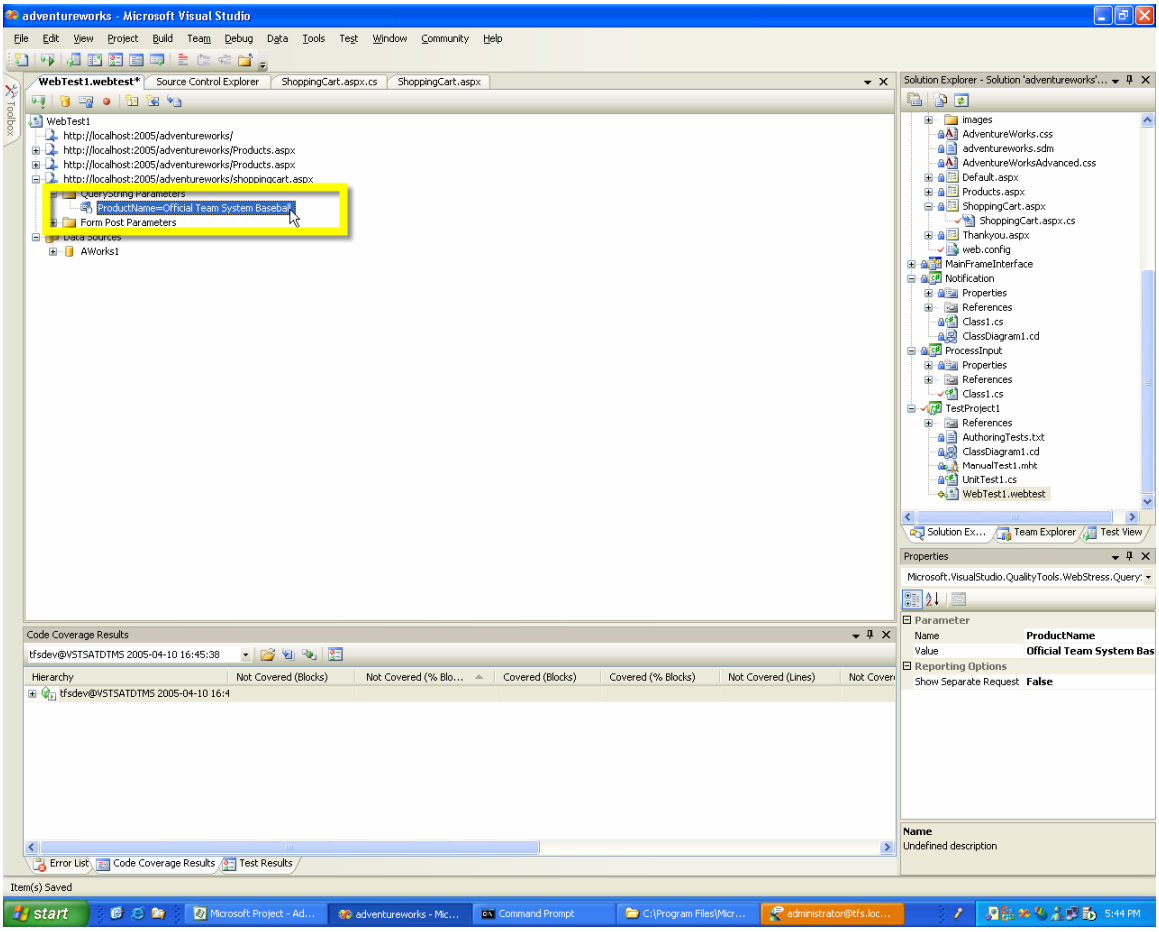
Actions	Select the **Use Windows NT Integrated Security** option Enter **CAMTSVSTSSS01** as the Initial Catalog (the image shown incorrectly shows 'AWorks' Press the **OK** button



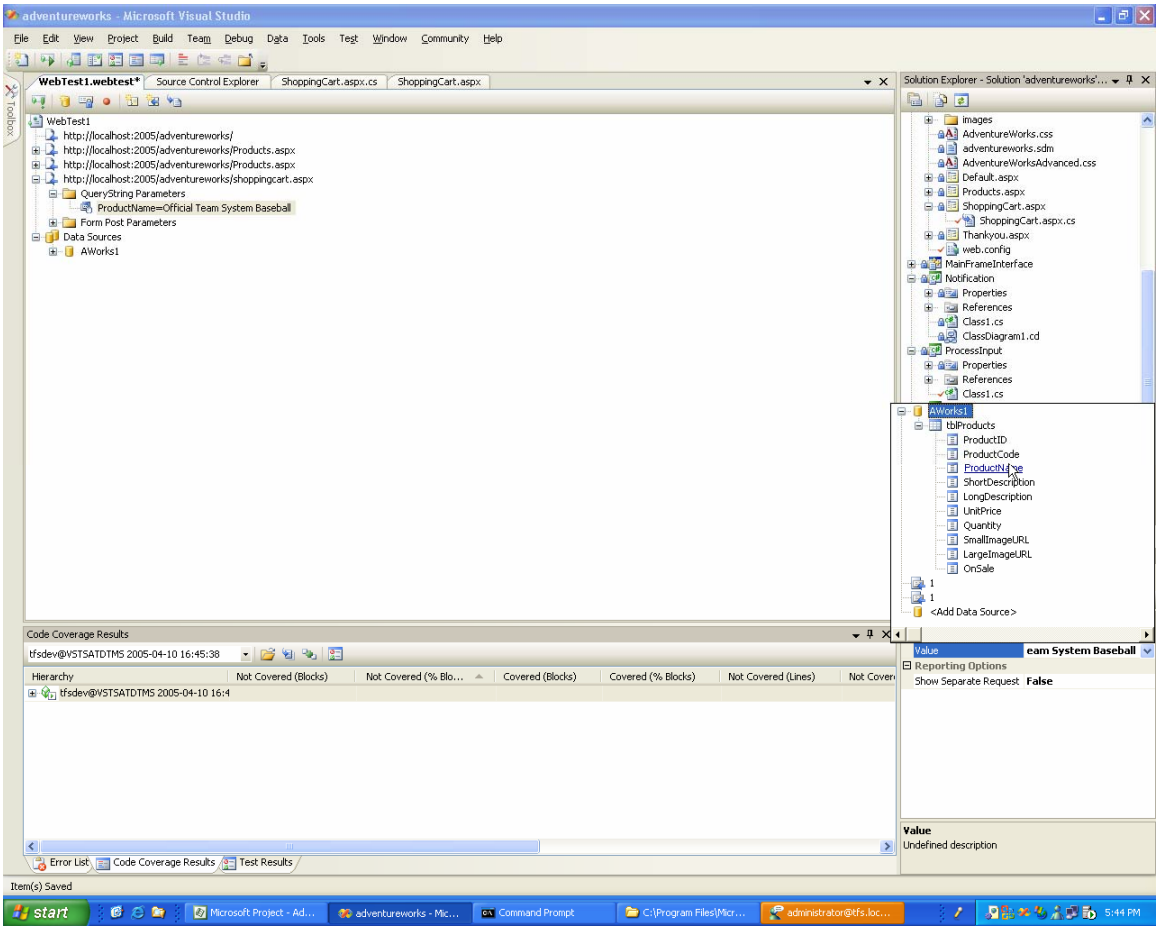
Actions	Choose the **tblProducts** table Press **OK**



Actions	Select the first **QueryString** parameter again



Actions	Pull down its value picker Expand the **CAMTSVSTSSS01** node and choose the **ProductName** value Run



Actions	In the **Test View** panel, select your web test and press **Run Tests** Your test should pass – you should see a different item purchased; if you don't try running the test again.

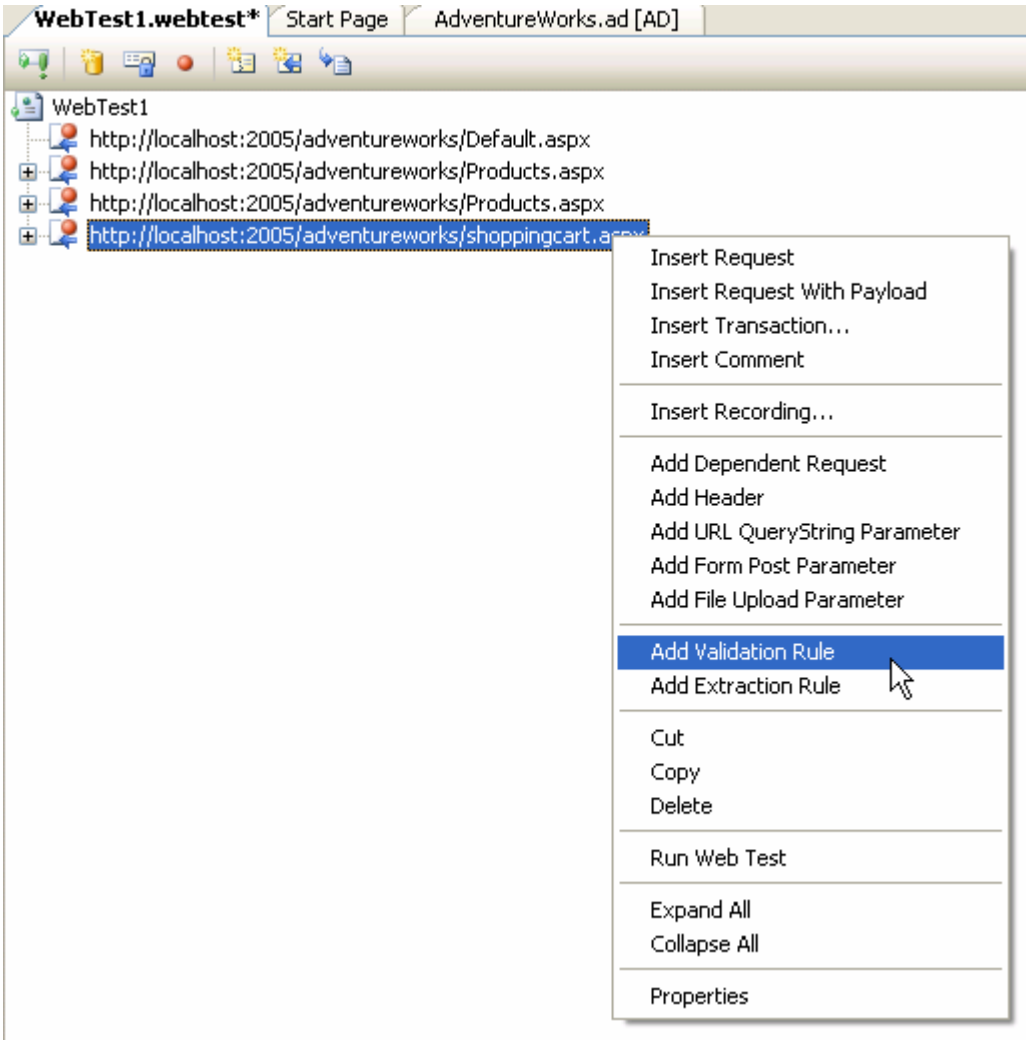
Now our tests run with more realistic data, we still need to define whether our test passes or fails. In order to do that, we have to inspect the response that we get back from our requests. In the next section, we will see how to do just that.

Exercise - Validating a Web Test

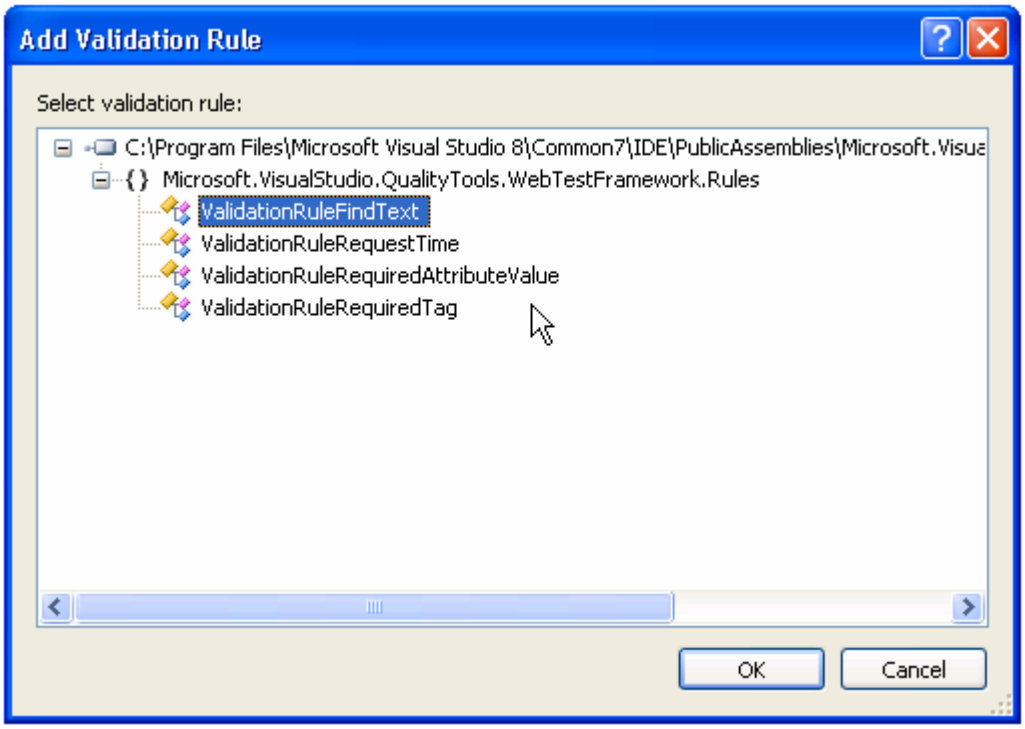
Visual Studio Team System provides validation rules to help you inspect the response from an HTTP request to see if it matches your criteria. You can verify whether a request contains certain text, HTML tags, or comes back in a certain amount of time.

Of course, this mechanism is extensible. Instead of using a built-in validation rule, we will create our own custom validation rule.

Actions	Select the last request in your web test Right-click and choose **Add Validation Rule**

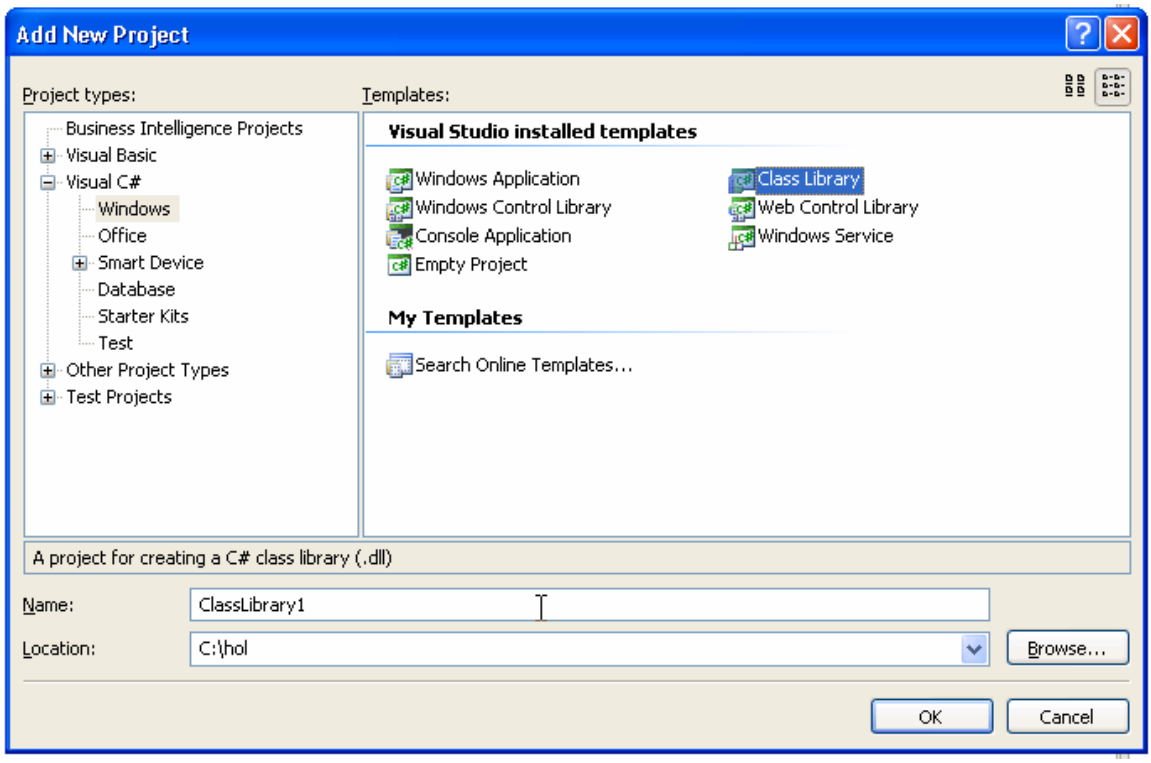


Actions	These are the standard built-in validation rules Let's create our own; press **Cancel**

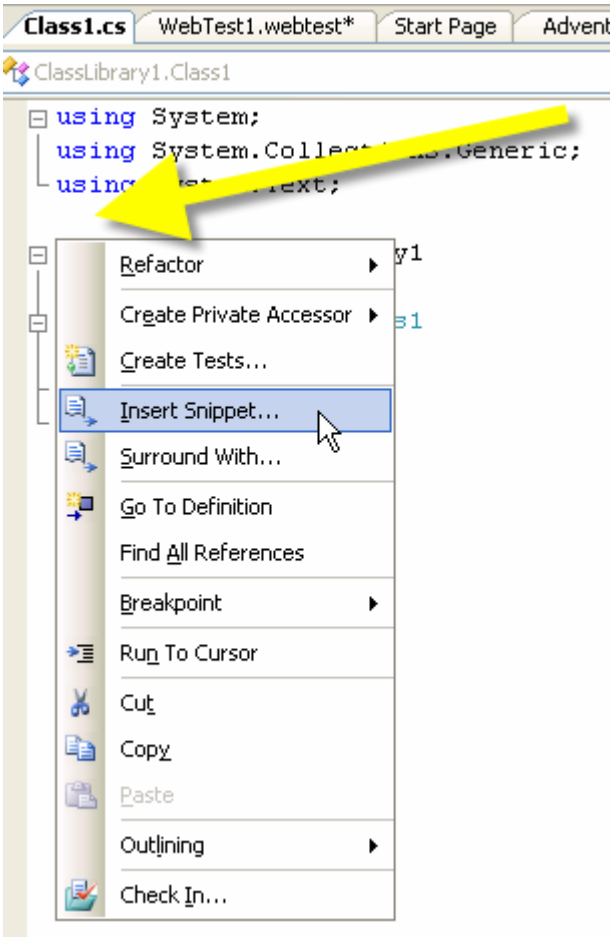


Validation rules are just .NET assemblies that implement certain interfaces. Like before, we will create our .NET assembly project, add a reference to an object model that Visual Studio Team System publishes and write the code for our validation rule.

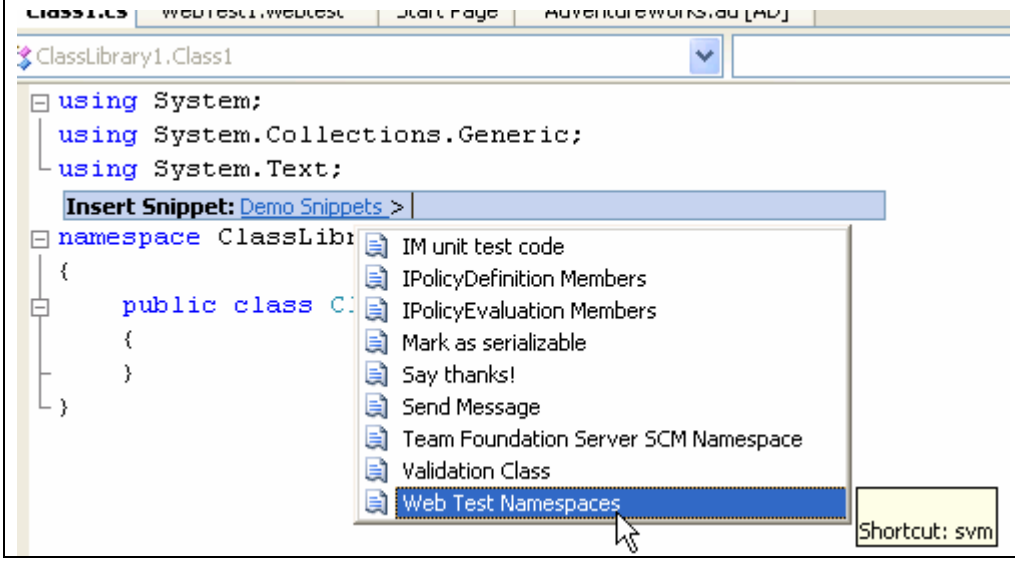
Actions	Add a new **Visual C# -> Windows -> Class Library** project to your solution



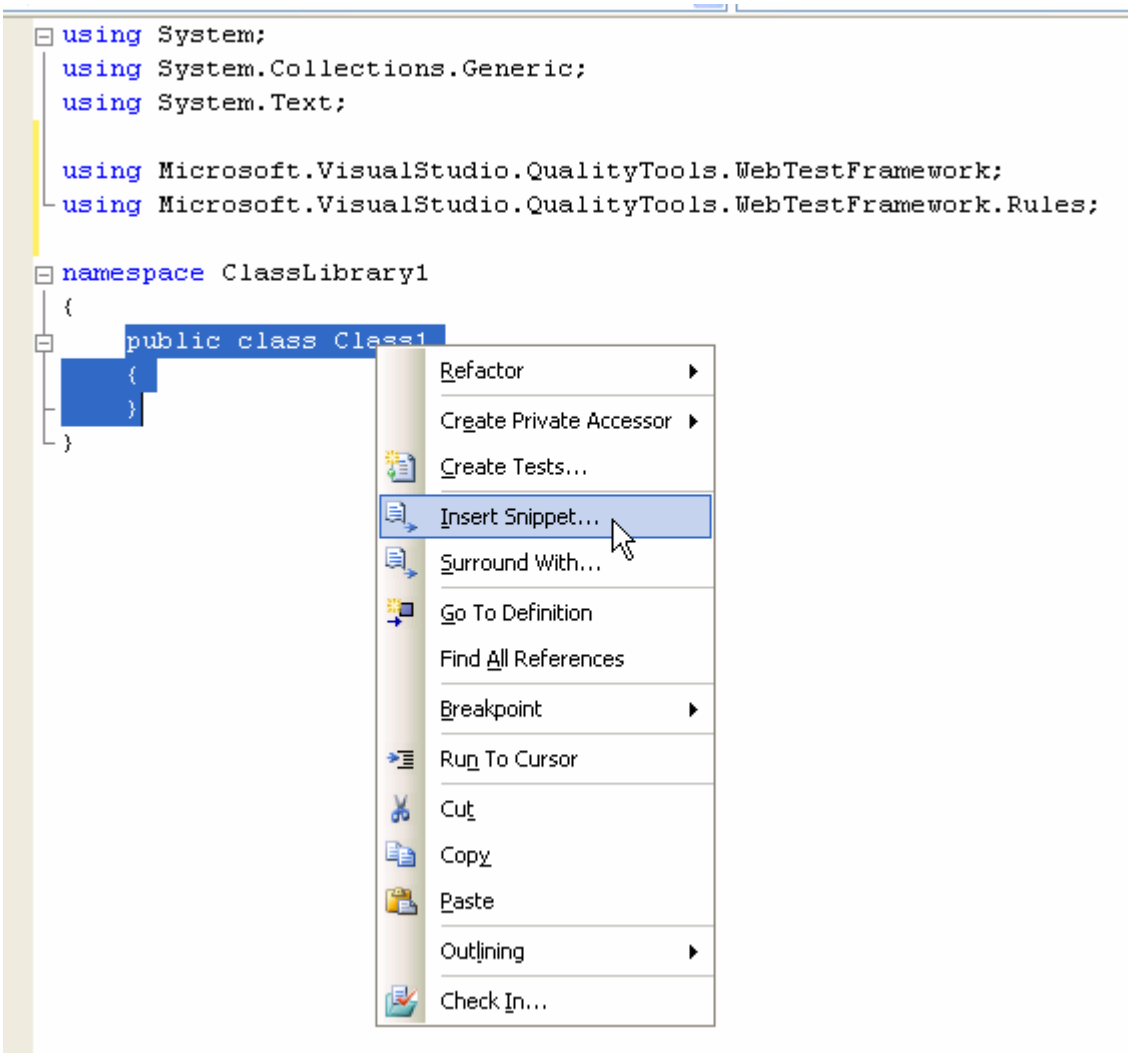
Actions	Put your cursor in the area shown, right-click and choose **Insert Snippet**



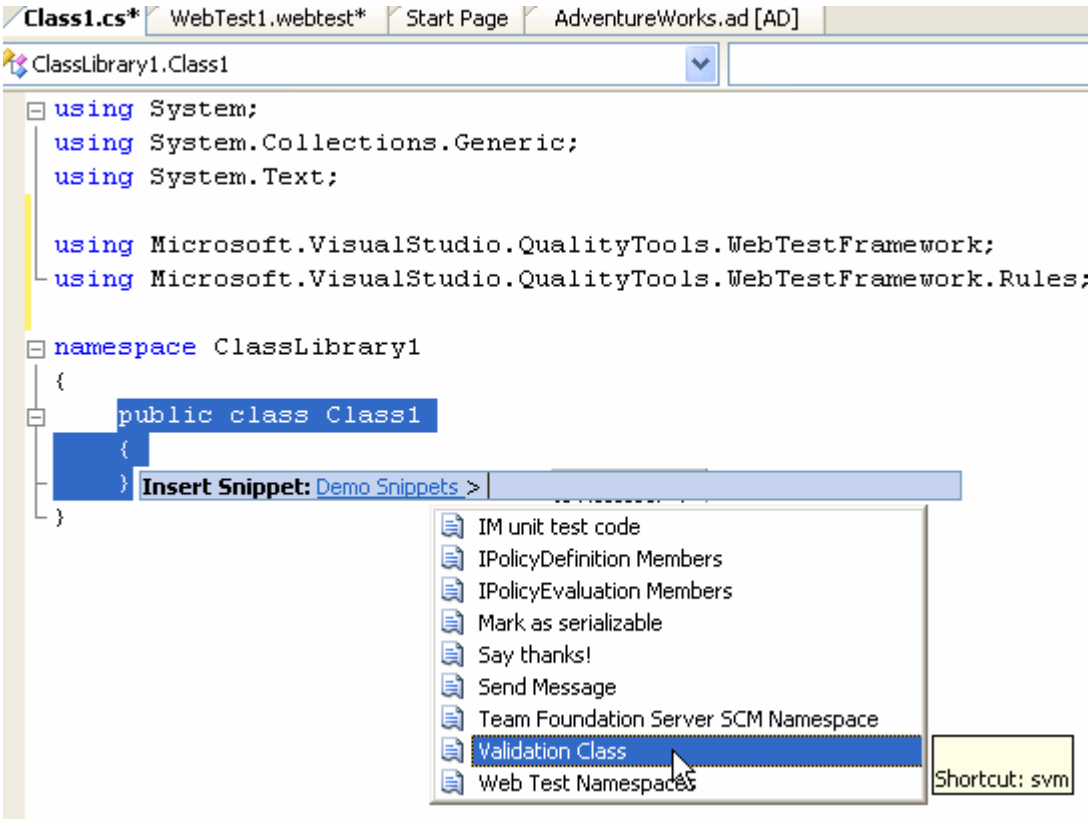
Actions	Pick **Demo Snippets -> Web Test Namespaces**



Actions	Select the code shown Right-click and choose **Insert Snippet…**



Actions	Pick **Demo Snippet -> Validation Class**

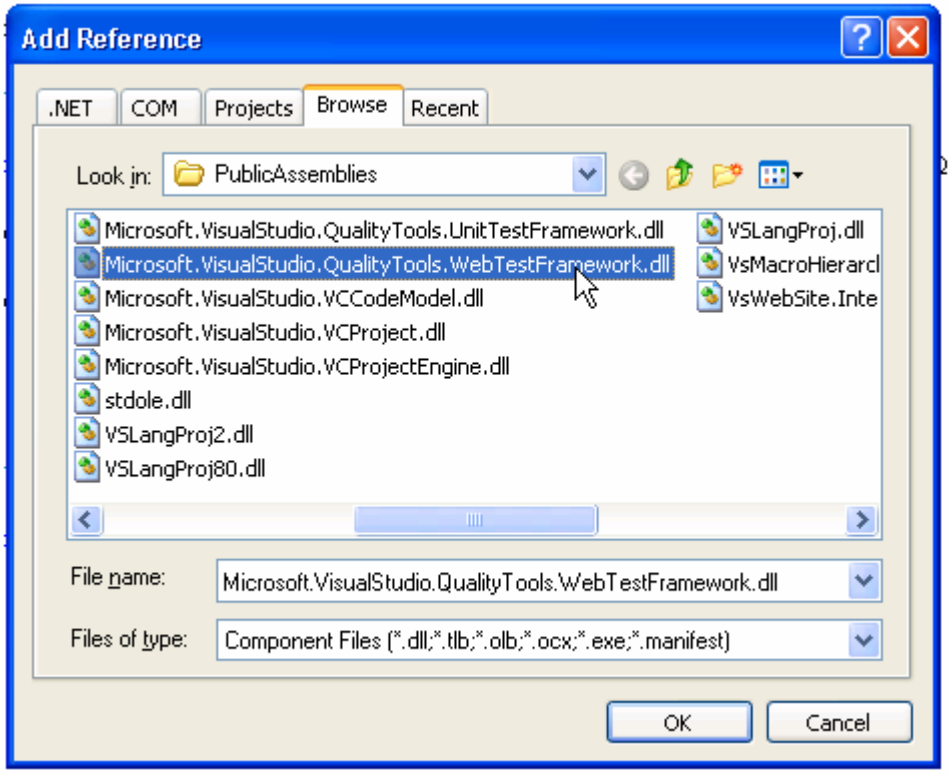


Our validation rule will look for the occurrence of 'Thank You' in the response; this should be in the page we see after we purchase an item.

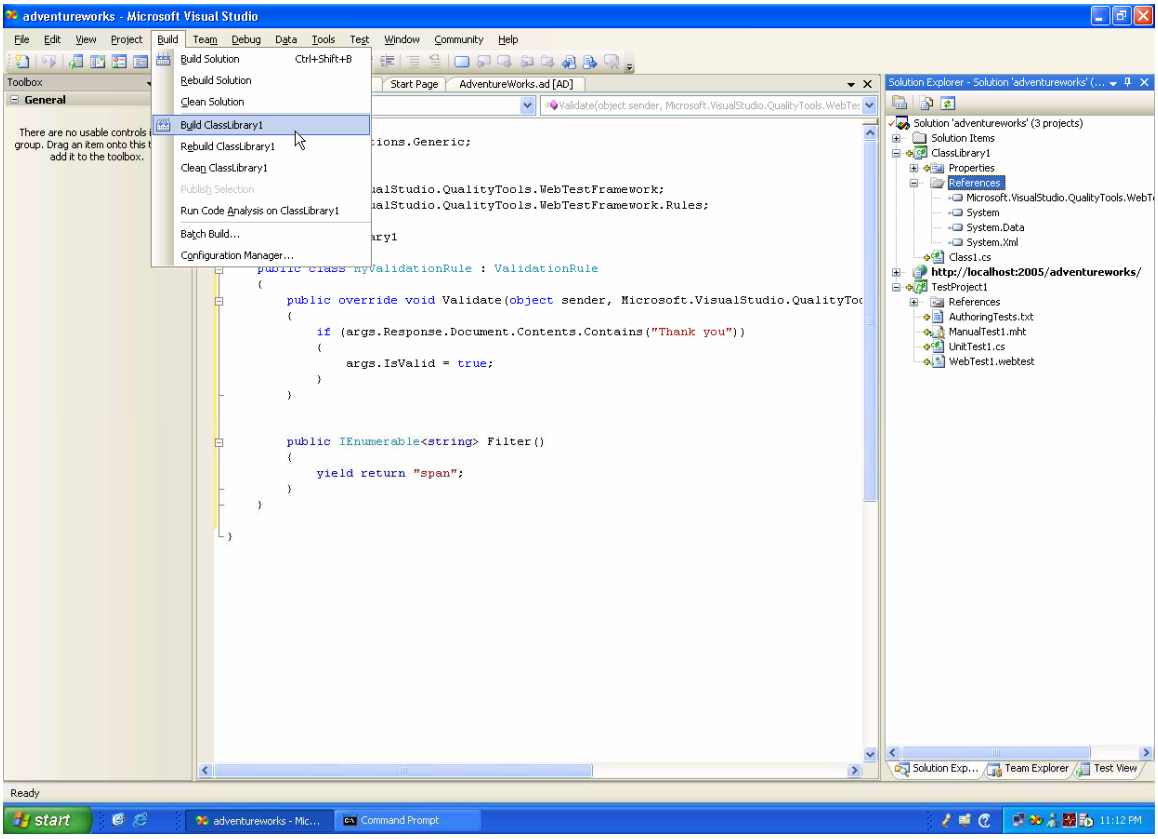
Actions	Your code should look as shown

```
using System;
using System.Collections.Generic;
using System.Text;

using Microsoft.VisualStudio.QualityTools.WebTestFramework;
using Microsoft.VisualStudio.QualityTools.WebTestFramework.Rules;

namespace ClassLibrary1
{
    public class MyValidationRule : ValidationRule
    {
        public override void Validate(object sender, Microsoft.VisualStudio.QualityToo
        {
            if (args.Response.Document.Contents.Contains("Thank you"))
            {
                args.IsValid = true;
            }
        }


        public IEnumerable<string> Filter()
        {
            yield return "span";
        }
    }

}
```

Actions	Select your project, right-click and choose **Add Reference** Browse and choose: **C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\Public Assemblies\Microsoft.VisualStudio.QualityTools.WebTestFramework.dll** Press **OK**

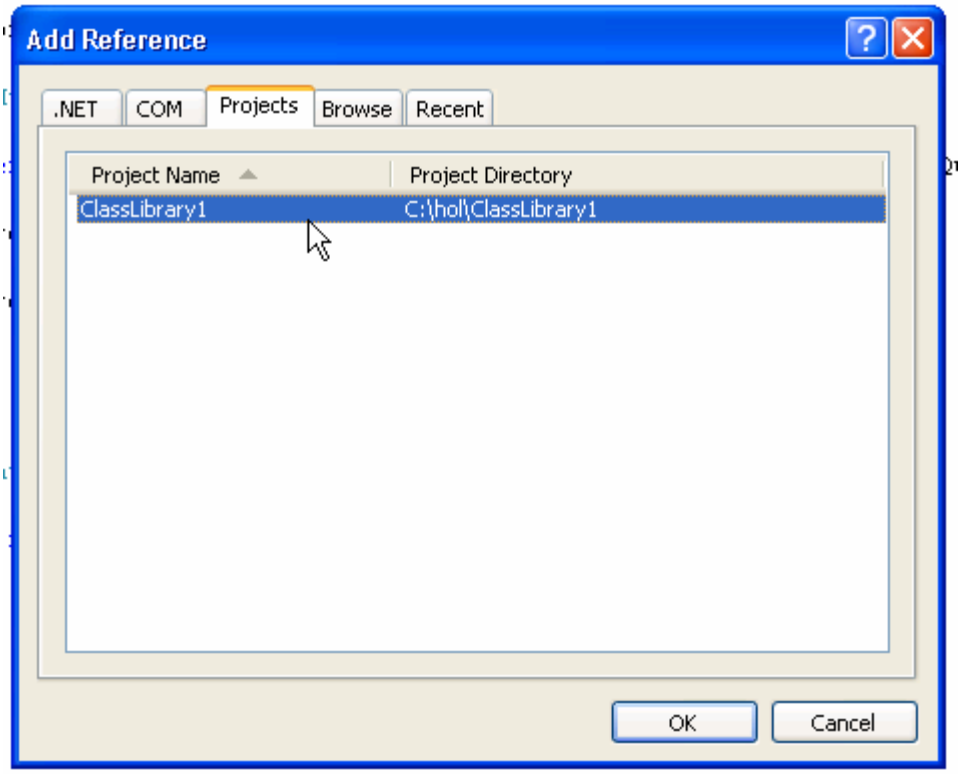


Actions	Choose the **Build -> Build ClassLibrary1** menu option

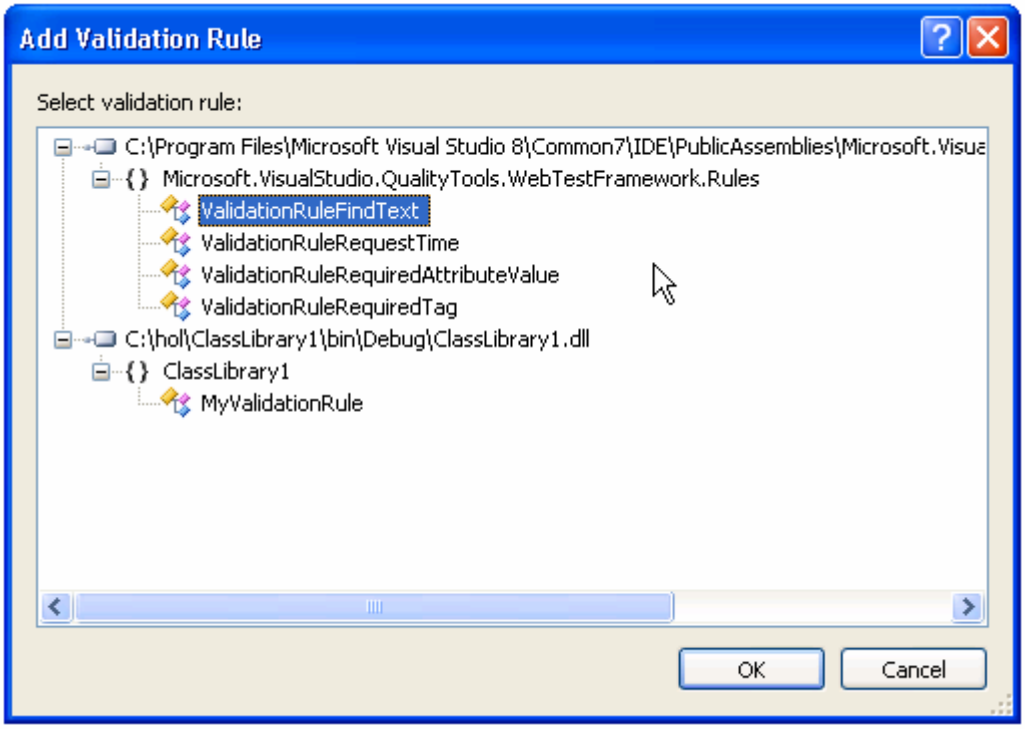


Adding a reference from our validation rule project to our test project is the registration mechanism for validation rules.

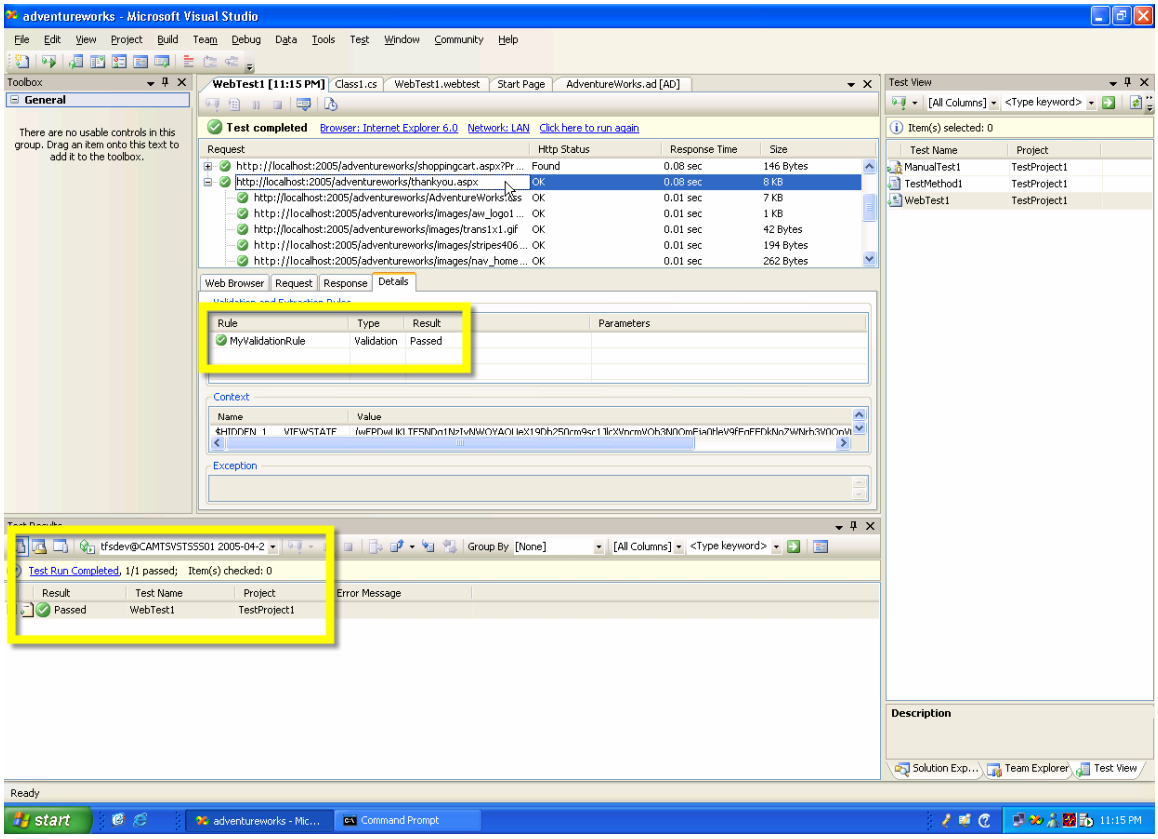
Actions	Select your **Test Project**, and add a reference to the **ClassLibrary1** project



Actions	Go back to the last request in your web test, right-click and choose **Validation Rules** Your new Validation Rule should be shown; select it and press **OK**



Actions	When your test finishes, double-click on it's results to view its details The **Details** tab of the last request in your web test will show the results of your validation rule



We have one more issue with our web testing, we are only simulating one user, and this is not very realistic. In the next section, we will see how to multiply the affects of our web test to simulate a more realistic level of stress.

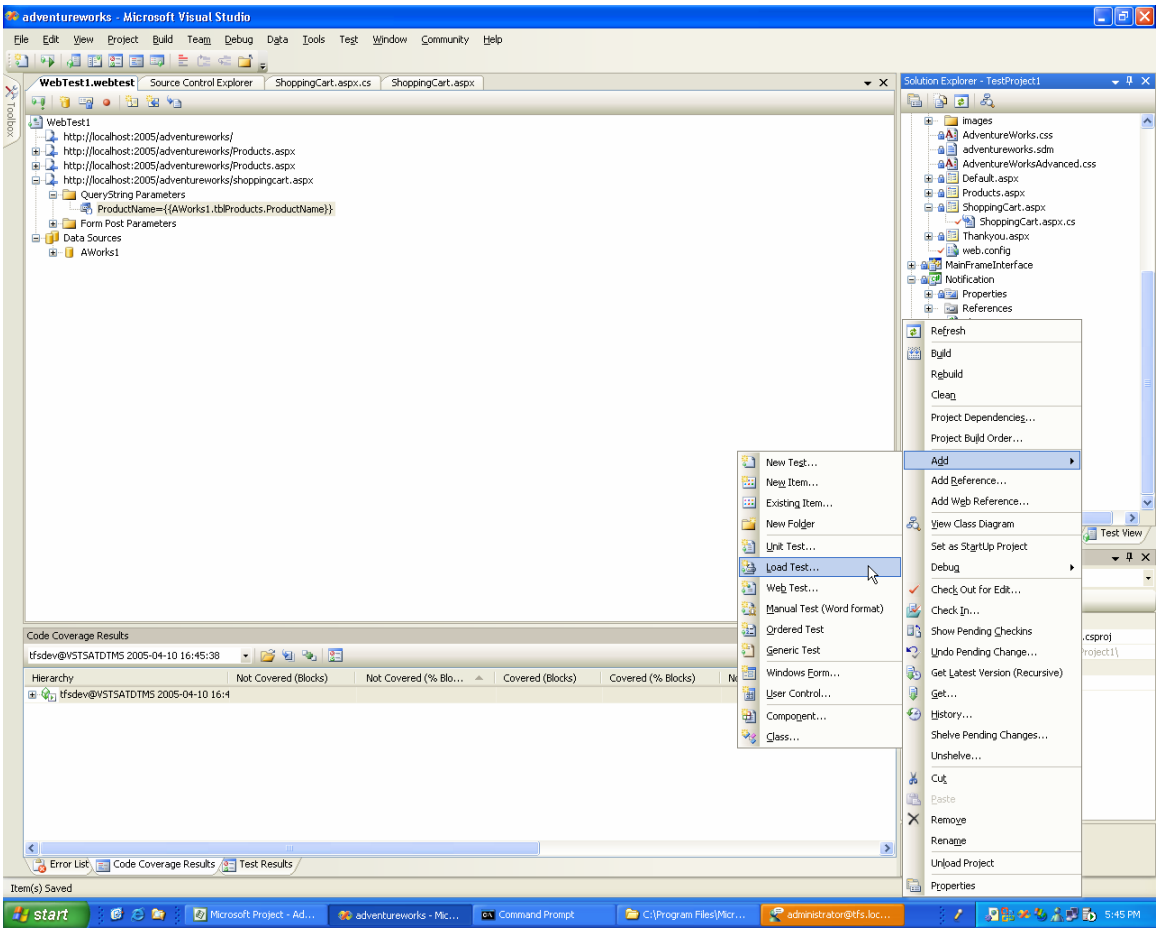
Exercise - Load Testing

We've done a great job of making sure our code works for a single user; but we're going to get a lot of traffic to our re-vamped AdventureWorks site.

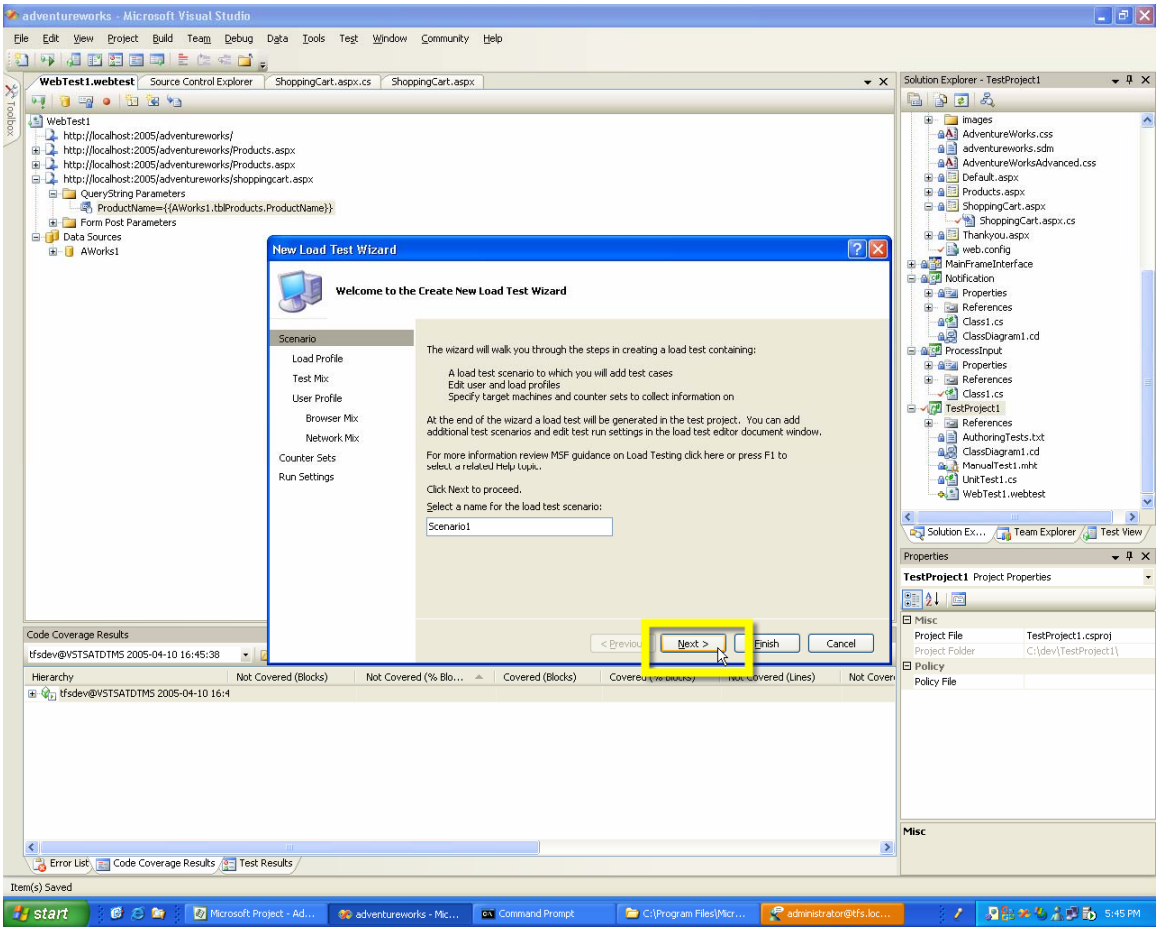
From right inside Visual Studio Team System, we can make sure that my web site is going to stand up to a lot of traffic and stress.

Load testing is just another type of test that Visual Studio Team System supports.

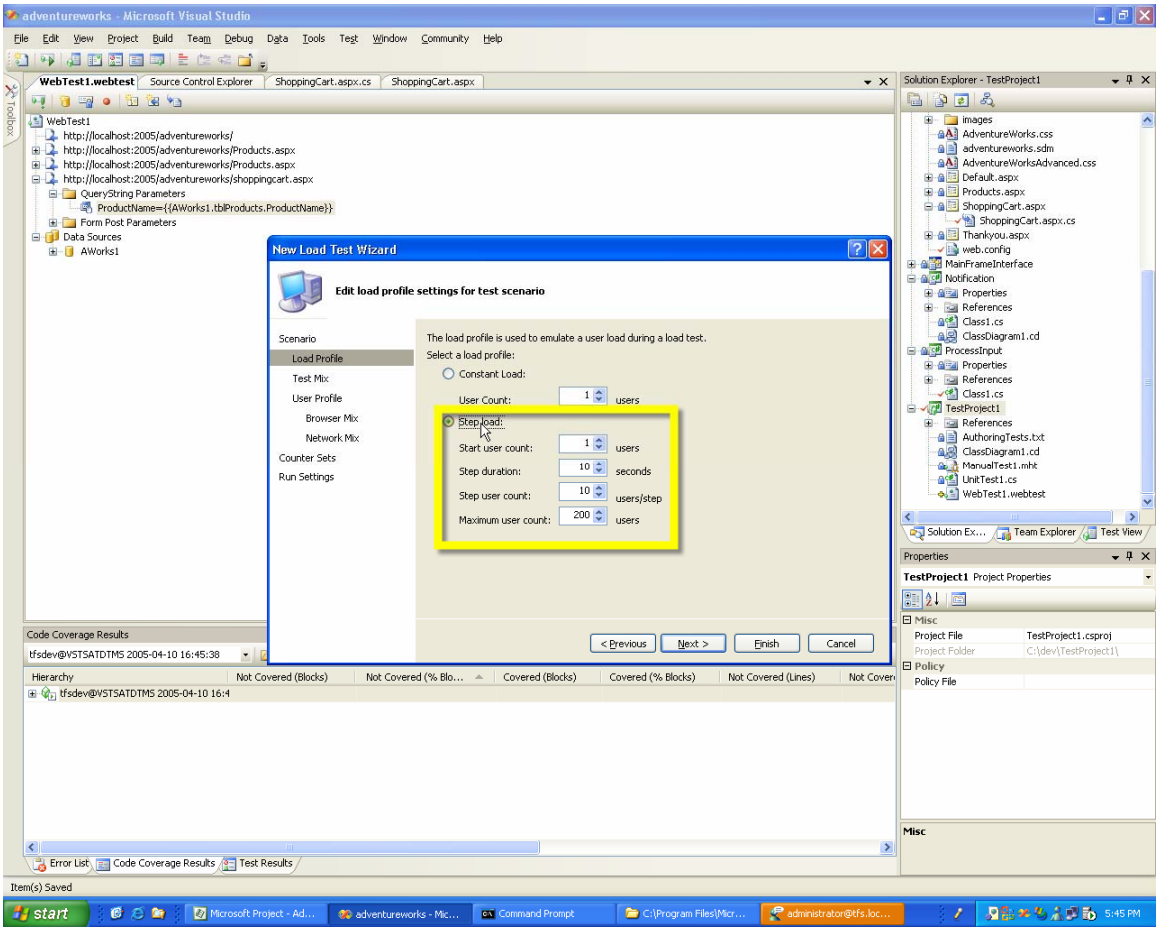
Actions	Select your test project Right-click and choose the **Add -> Load Test** option



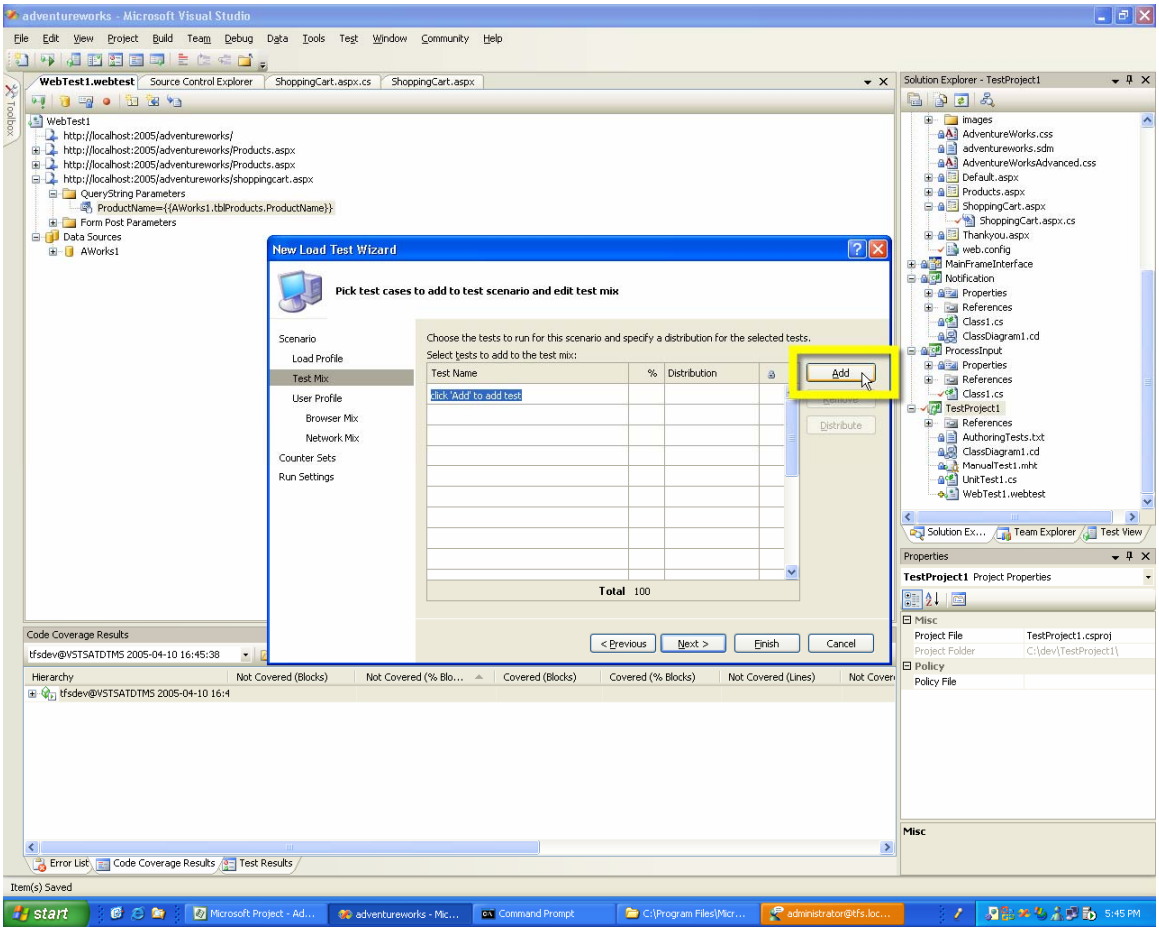
Actions	Name your **load test** anything that you'd like Press **Next**



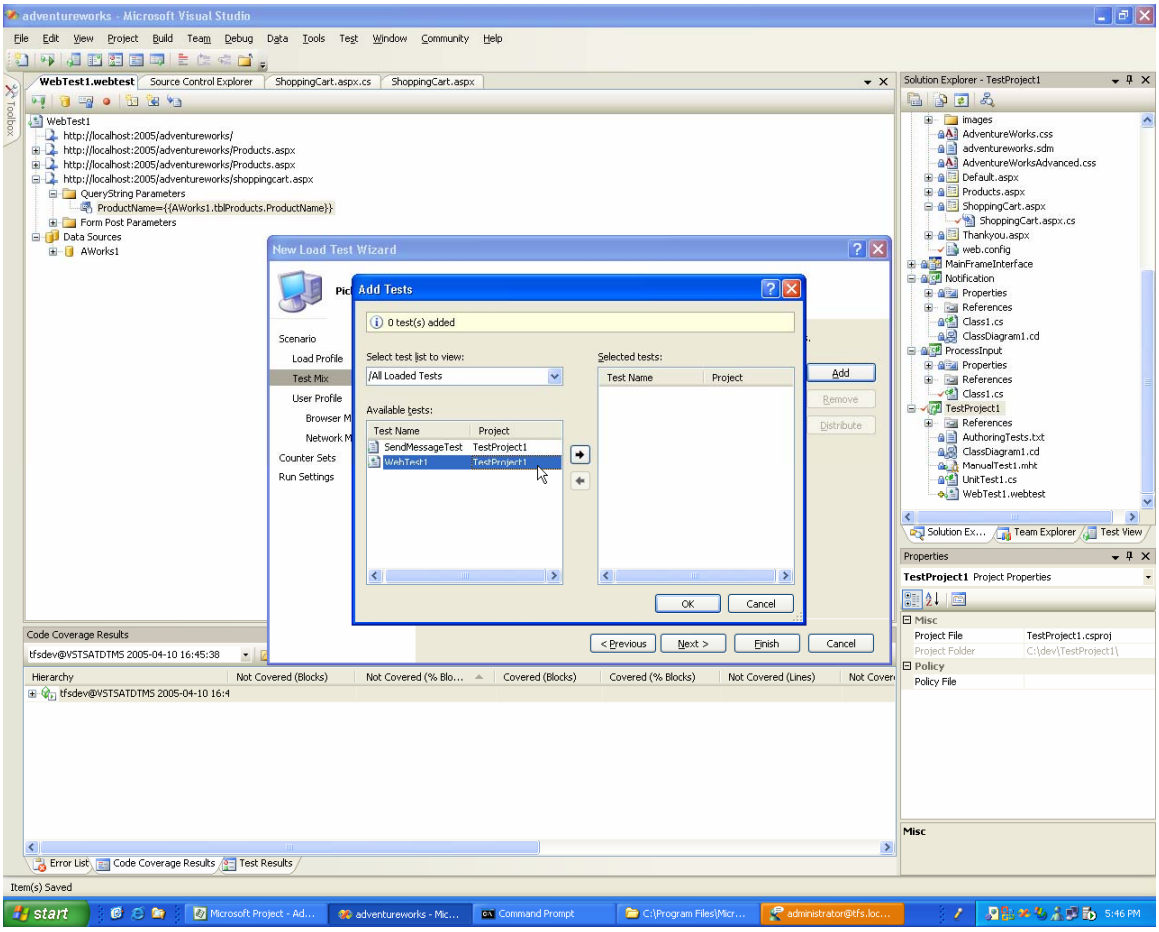
Actions	Choose the **Step Load** option Press **Next**



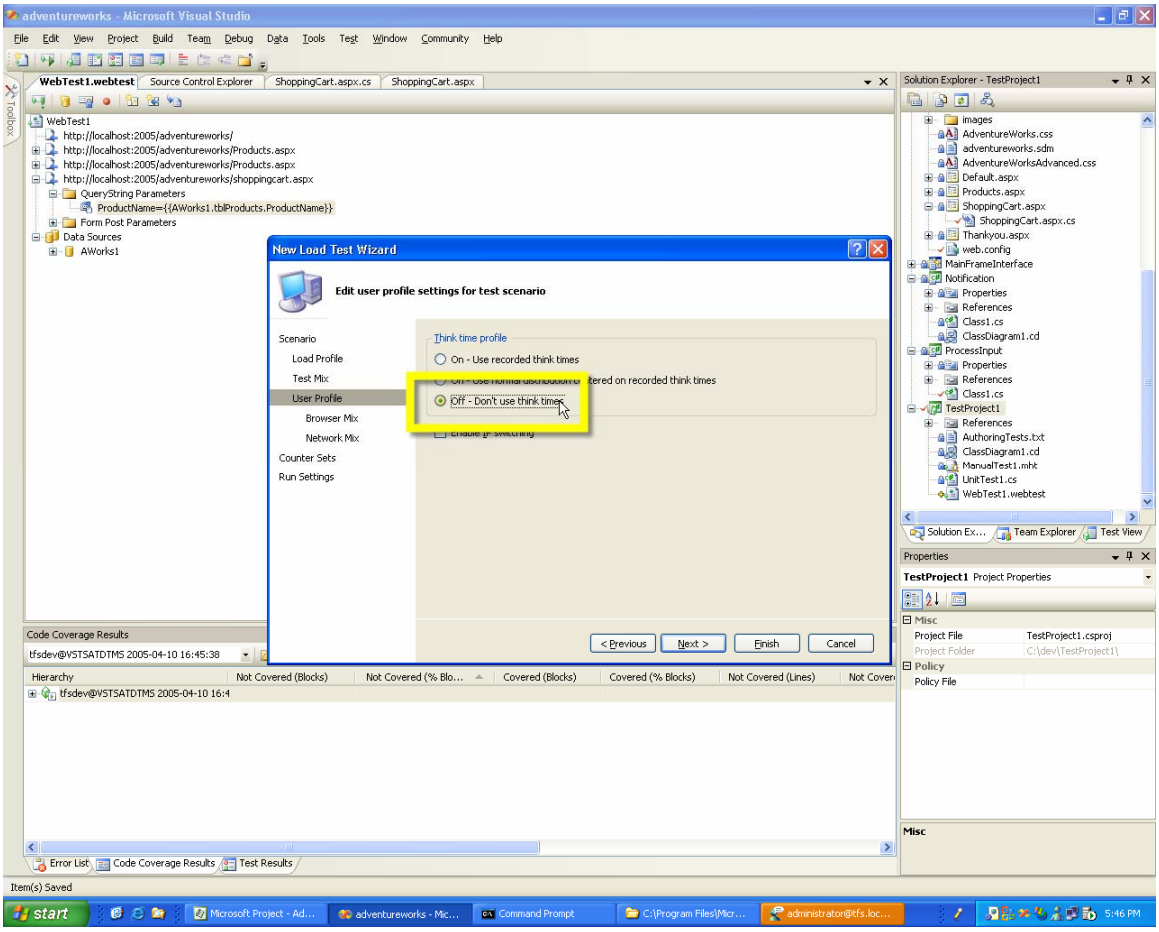
Actions	Press **Add**



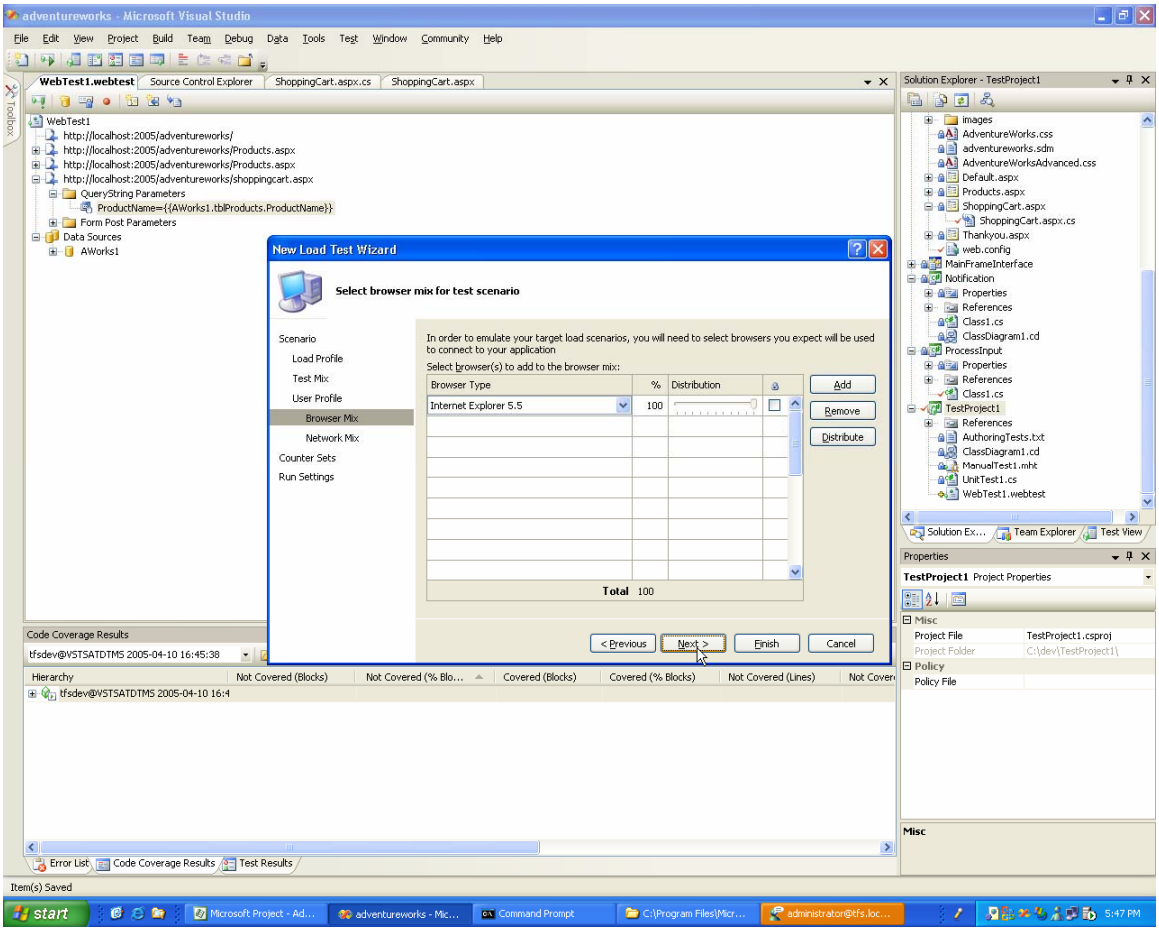
Actions	Double-click on your web test Press **Next**



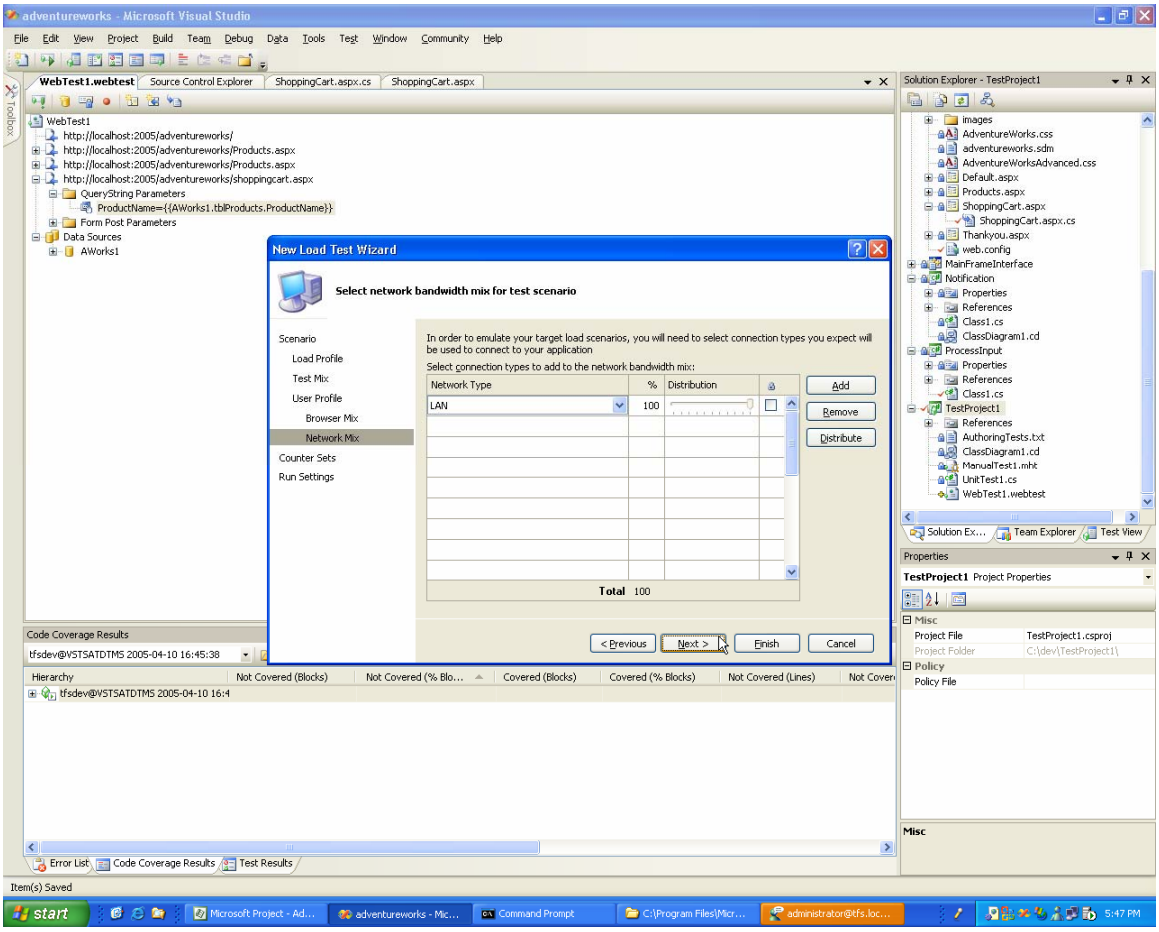
Actions	Choose the **Off – Don't use Think Times option** Press **Next**



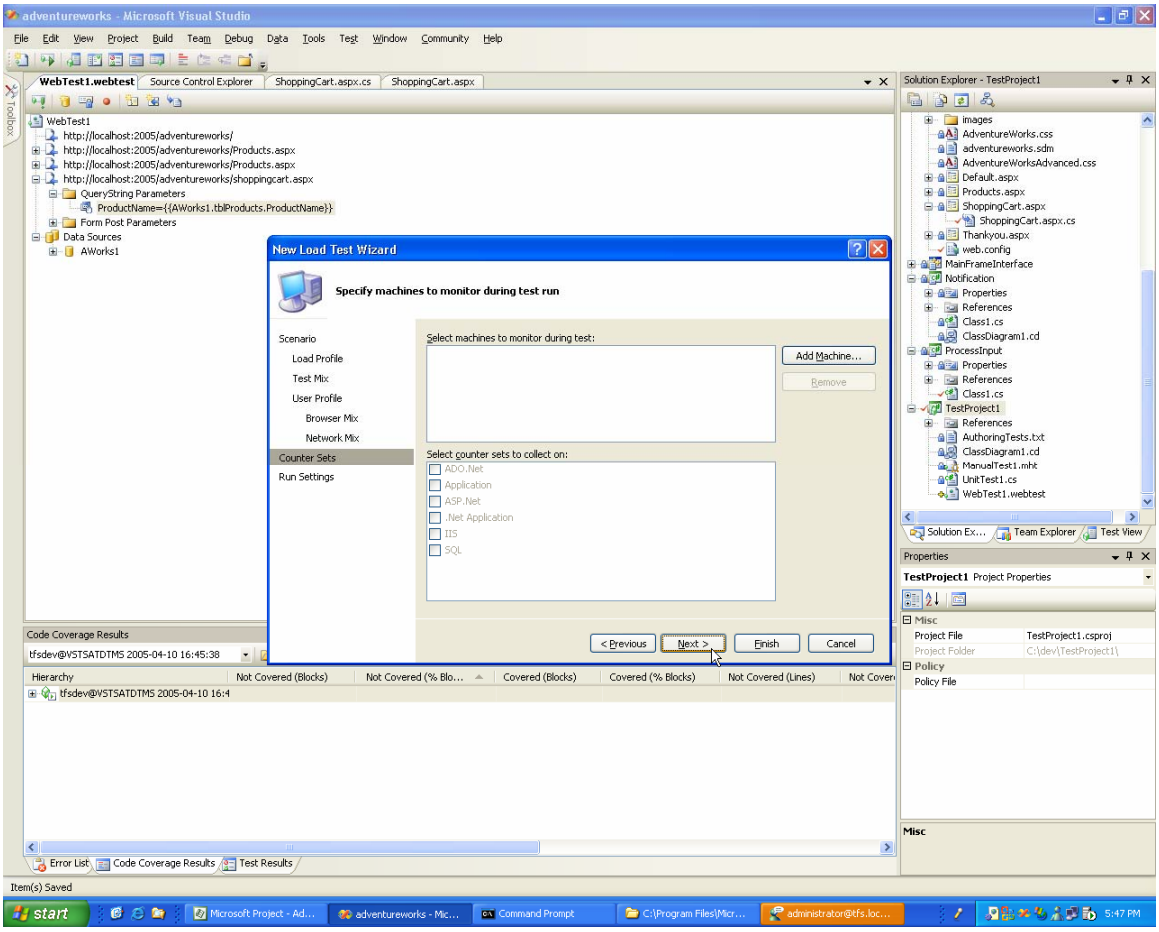
Actions	Choose any spread of browsers that you would like Press **Next**



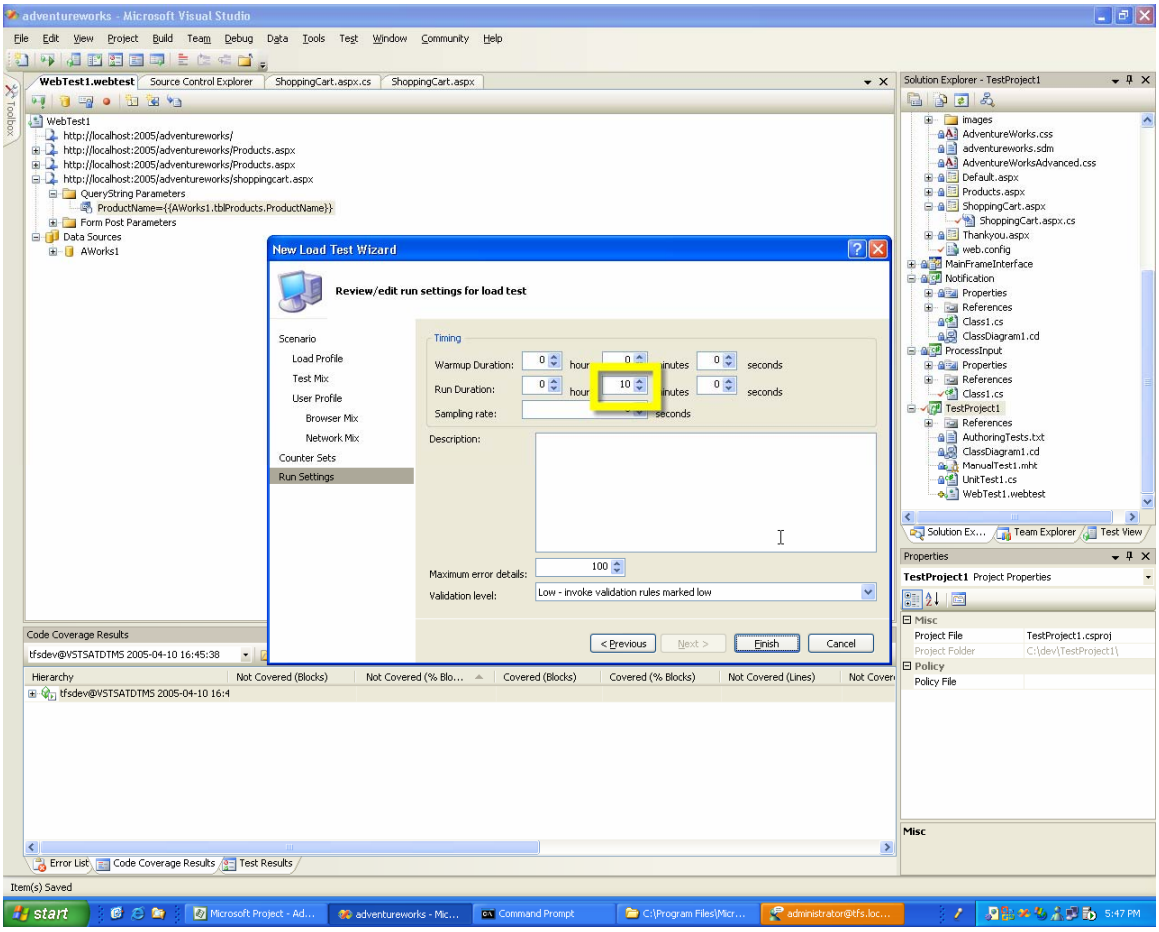
Actions	Choose any spread of network connection speeds that you would like Press **Next**



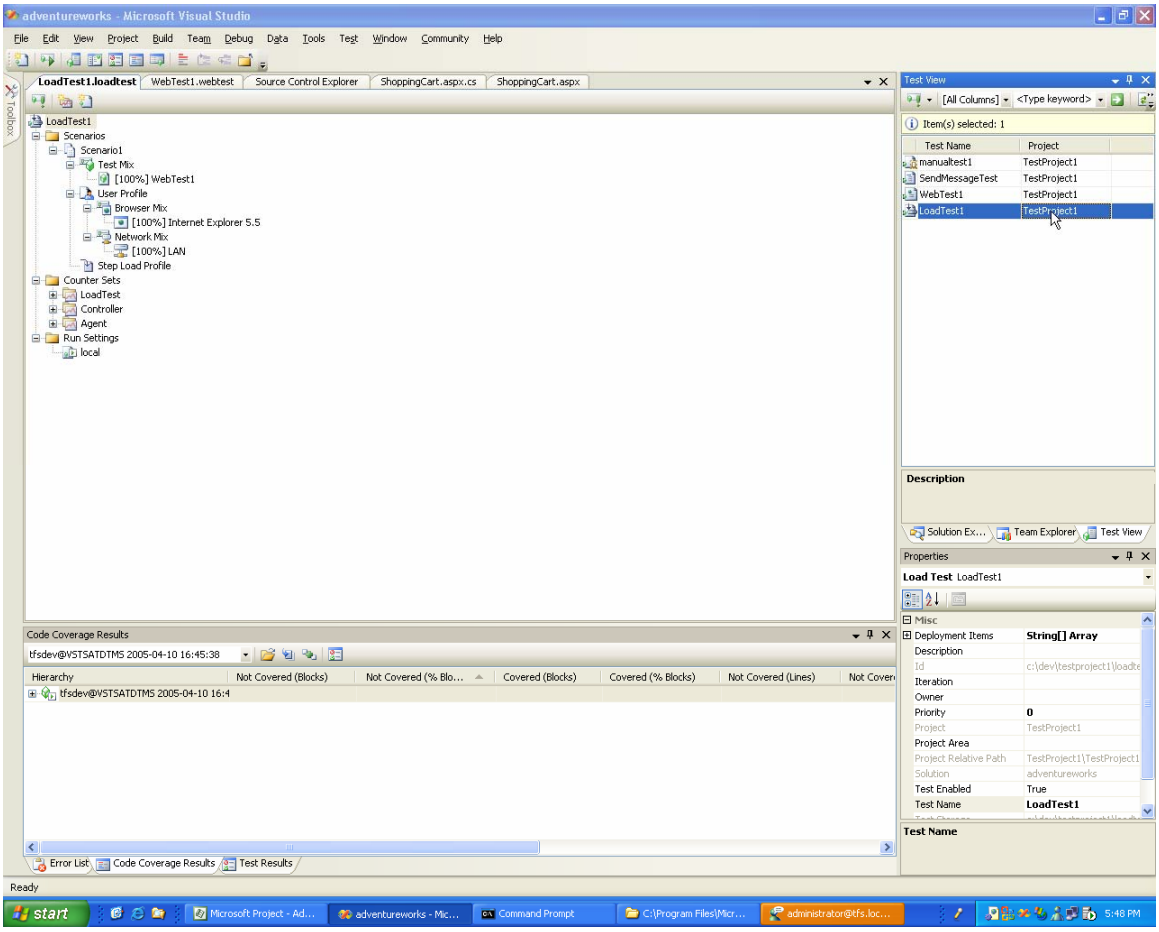
Actions	Press **Next**



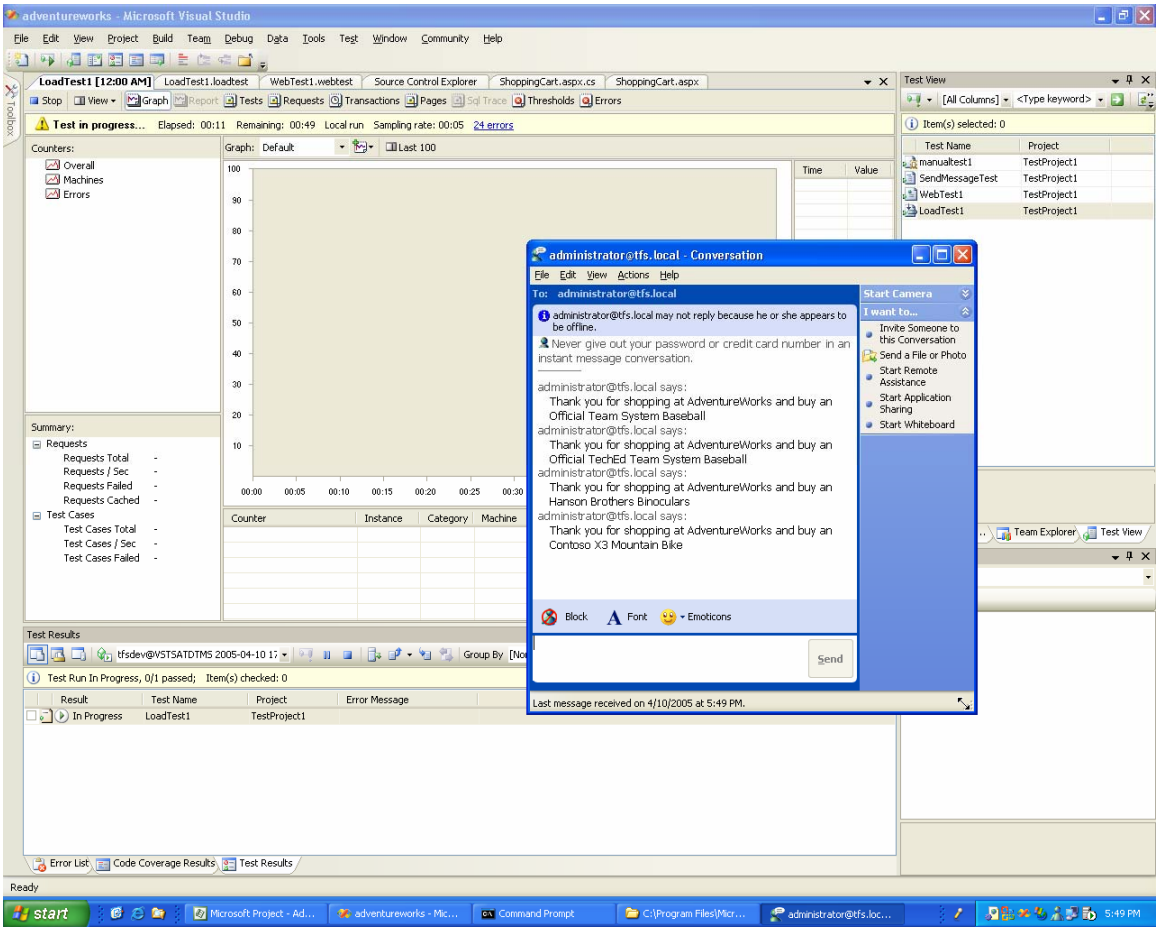
Actions	Set the **Run Duration** to 1 minute Press **Finish**



Actions	Go to your **Test View** Select your **load test** and press **Run Tests**



Actions	It will take a few minutes for your load test to run As it runs, you will see Windows Performance Counters charted



End of Lab

Visit http://lab.msdn.microsoft.com/teamsystem/ for more information