* 1. 

Hands-On Lab

Test Driven Development in Visual Studio 2010

* 1. Lab version: 1.0.0
  2. Last updated: 1/4/2011
  3. 

Contents

[Overview 3](#_Toc281931458)

[Exercise 1: Red, Green… 5](#_Toc281931459)

[Task 1 – Creating a New Project 5](#_Toc281931460)

[Task 2 – Defining a Context for the Tests. 7](#_Toc281931461)

[Task 3 – Writing the First Test: Designing and Defining the SimpleStack 12](#_Toc281931462)

[Task 4 – Adding More Functionality Driven by Tests 21](#_Toc281931463)

[Exercise 2: Refactor! 24](#_Toc281931464)

[Task 1 – Moving from Passing the Test to Building Functionality 24](#_Toc281931465)

[Task 2 – Reorganizing the Code 30](#_Toc281931466)

[Summary 34](#_Toc281931467)

Overview

* 1. Test Driven Development (TDD), also frequently referred to as Test Driven Design, is a development methodology where developers create software by first writing a unit test, then writing the actual system code to make the unit test pass. The unit test can be viewed as a small specification around how the system should behave; writing it first helps the developer to focus on only writing enough code to make the test pass, thereby helping ensure a tight, lightweight system which is specifically focused meeting on the documented requirements.
  2. TDD follows a cadence of “Red, Green, Refactor.” Red refers to the visual display of a failing test – the test you write first will not pass because you have not yet written any code for it. Green refers to the step of writing just enough code in your system to make your unit test pass – your test runner’s UI will now show that test passing with a green icon. Refactor refers to the step of refactoring your code so it is tighter, cleaner, and more flexible. This cycle is repeated constantly throughout a TDD developer’s workday.
  3. **Note:** This lab is not meant to teach you how to work with TDD; instead, it is intended to highlight Visual Studio 2010’s support for working in TDD. If you are interested in learning more about the TDD methodology and its benefits then you should look to the following books as a starting point:
  4. *Test Driven Development in Microsoft .NET* by James Newkirk and Alexei Voronstov, ISBN 0735619484
  5. *Pragmatic Unit Testing in C# with NUnit, 2nd Edition* by Andy Hunt, Dave Thomas, and Matt Hargett, ISBN 0977616673
  6. A critical aspect of TDD is developer cadence. Getting into a productive development rhythm (the elusive “zone”) involves having the right mindset, but also working with tools that assist the developer in working rapidly and with as little friction as possible.
  7. Visual Studio 2010 brings with it several enhancements to help cut development friction and enable the developer to focus on the task: writing high-quality code. In the following exercises, you will learn several of the new features that the TDD developer can use to enhance his/her development cadence. VS10 helps your cadence by cutting the number of keystrokes to accomplish frequently performed tasks, speeds navigation through your solution, and enables you to use test frameworks other than MSTest.
  8. To demonstrate the new features you will build an implementation of a stack in a test-first manner, showing how tests drive the design and implementation of a **SimpleStack** class.
  9. **Note:** In computer science, a stack is an abstract data type and data structure based on the principle of Last In First Out (LIFO).
  10. If you are unfamiliar with the concept of a stack as a data structure, or just need a quick refresher, please see the full [Wikipedia article](http://en.wikipedia.org/wiki/Stack_(data_structure)) for a more thorough definition.

# Objectives

* 1. In this Hands-On Lab, you will learn how to:
  + Use the new Smart Tag actions to save keystrokes by letting the IDE generate small bits of code for you.
  + Use Quick Search to move around in the code base with just a few keystrokes.

# System Requirements

* 1. You must have the following items to complete this lab:
  + Microsoft Visual Studio 2010
  + .NET Framework 4.0

# Setup

* 1. All the requisites for this lab are verified using the **Configuration Wizard**. To make sure that everything is correctly configured, follow these steps.
  2. Run the **Configuration Wizard** for the Training Kit if you have not done it previously. To do this, browse to **Source\Setup** folder of this lab, and double-click the **Dependencies.dep** file. Install any pre-requisites that are missing (rescanning if necessary) and complete the wizard.
     1. **Note:** The Configuration Wizard is used for checking dependencies and setting up the environment. If the Configuration Wizard is not installed on your machine, you must install it running the DependencyChecker.msi file located on the %VS2010TKInstallationFolder%\Assets\DependencyChecker folder (e.g. by default the Training Kit is installed under C:\VS2010TrainingKit).
     2. For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. The **Dependencies.dep** file launches the Visual Studio installer file that installs the code snippets.

# Exercises

* 1. This Hands-On Lab comprises the following exercises:
  2. Start test-driving your design by writing the tests and them making them go from Red (failing) to Green (passing).
  3. Refactor your code, run the tests, and ensure they all pass.
  4. Estimated time to complete this lab: **60 minutes**.
  5. **Note:** This lab shows code examples in both C# and VB.NET; however, screen shots reflect only C# actions. Each action shown in the screenshots has a similar set of steps for VB.NET.

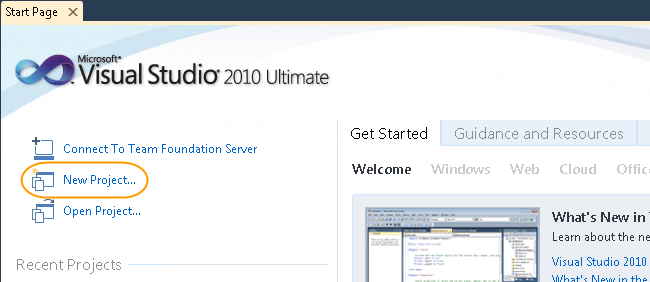
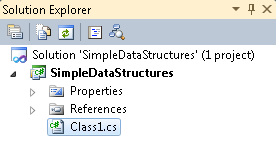
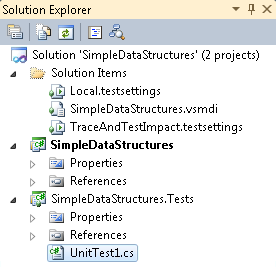
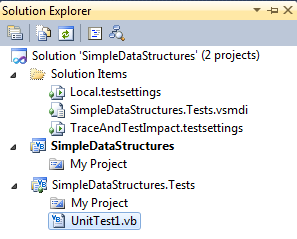
# Next Step

* 1. Exercise 1: Red, Green…

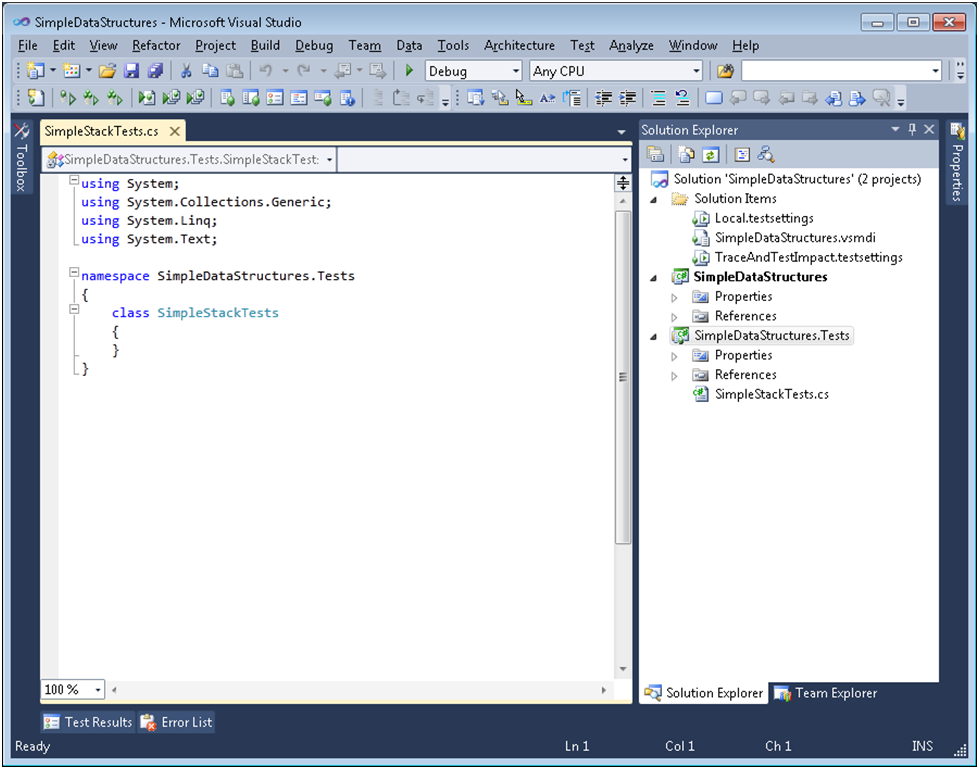
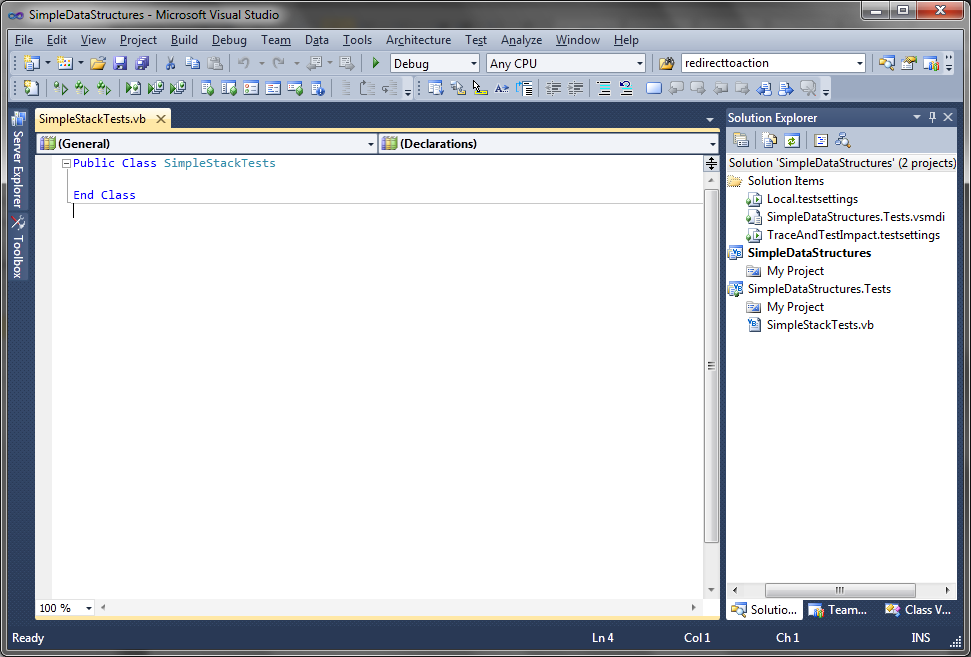
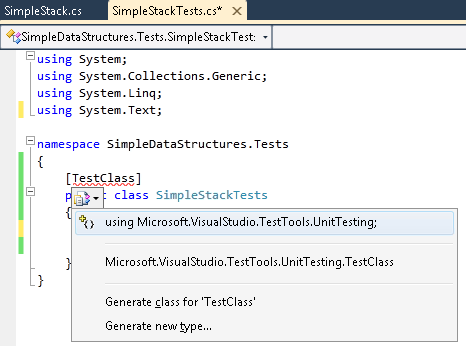
Exercise 1: Red, Green…

* 1. In this exercise, you will use the new features of Visual Studio 2010, which help support a TDD workflow. You will walk through Smart Tag actions, test runner enhancements, and navigation features.

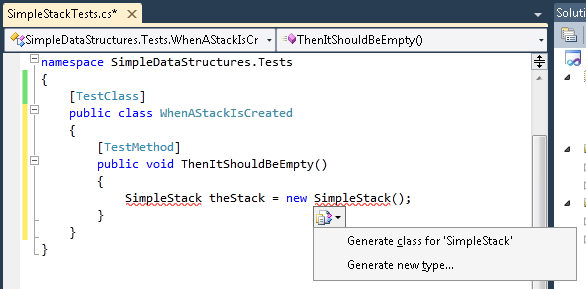
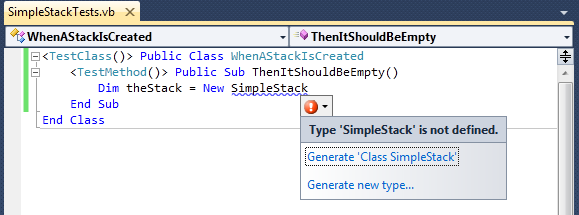
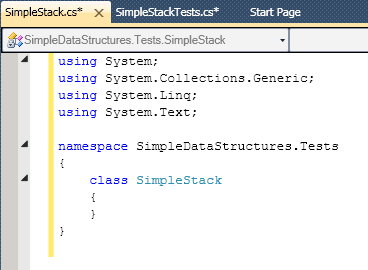
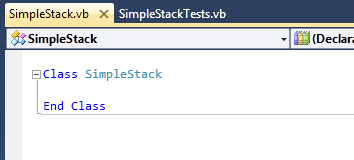
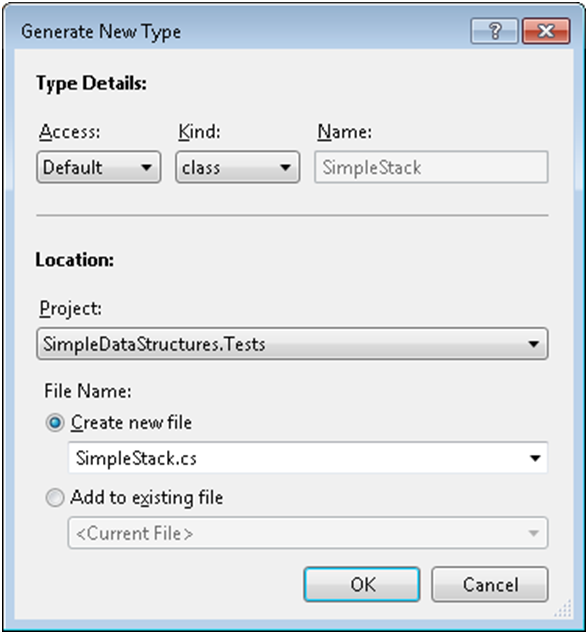
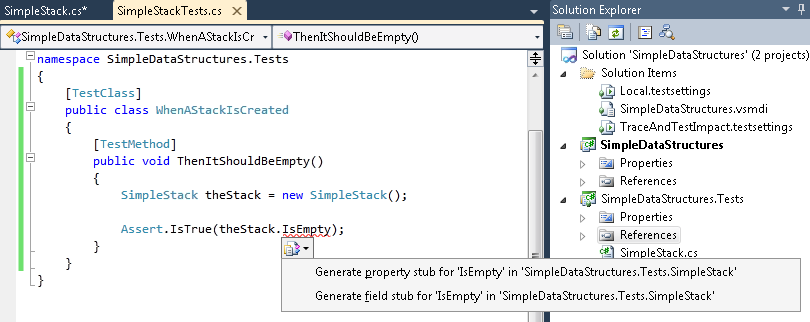
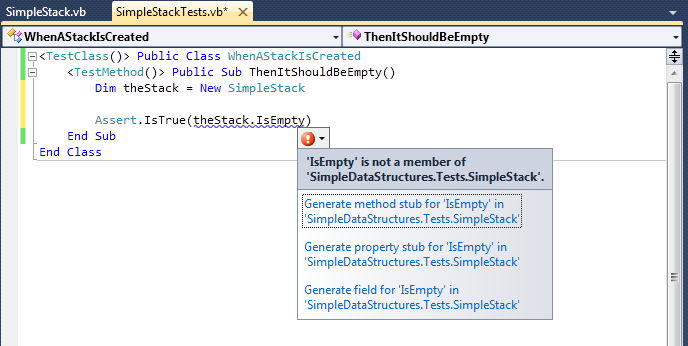
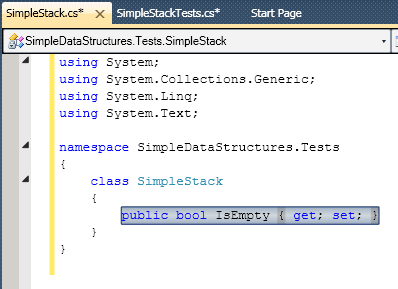
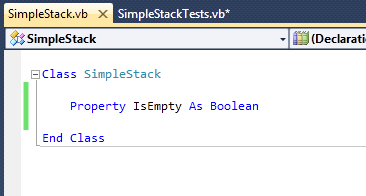
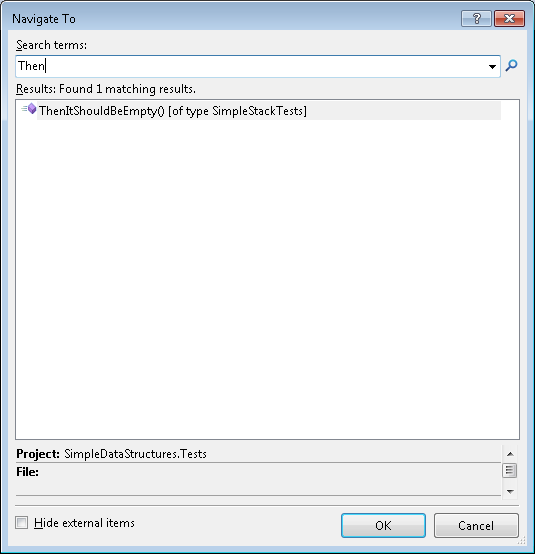
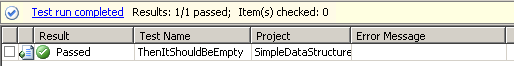
Task 1 – Creating a New Project

* 1. In this task, you will create a C# or VB Class Library and Test Project that will be used to explore the TDD features added to Visual Studio.
  2. Open Microsoft Visual Studio from **Start** | **All Programs** | **Microsoft Visual Studio 2010** | **Microsoft Visual Studio 2010**.
  3. From the Visual Studio Start Page click **Projects** and then click the **New Project** button.
     1. 
     2. Figure
     3. Creating the project
  4. In the **New Project** dialog, select the **Visual C#** or **VB** project type. Make sure that **.NET Framework 4.0** is selected, and then select the **Class Library** template.
  5. Change the name of the project to **SimpleDataStructures**, make sure the Create directory for solution checkbox is checked, and click **OK**.
     1. **Note:** The default target version of the .NET Framework in test projects is the .NET Framework 4. However, Visual Studio 2010 SP1 Beta adds basic support for unit tests that target the .NET Framework 3.5. For more information, see <http://msdn.microsoft.com/library/gg442059.aspx>.
  6. Delete the default Class1.cs file (or Class1.vb) that the New Project Wizard created.
     1. 
     2. Figure
     3. Deleting the default file created by the New Project Wizard
  7. In the Solution Explorer, right-click on the **SimpleDataStructures** solution and click **Add | New Project…** and select the **Visual C# | Test** or **Visual Basic | Test** project type. Make sure that **.NET Framework 4.0** is selected, and then select **Test Project**.
  8. Change the project’s name to **SimpleDataStructures.Tests** and click **OK**.
  9. Delete UnitTest1.cs file (or UnitTest1.vb) that the New Project Wizard created.
     1. 
     2. Figure
     3. Deleting the default test file generated by the New Project Wizard (C#)
     4. 
     5. Figure
     6. Deleting the default test file generated by the New Project Wizard (VB)

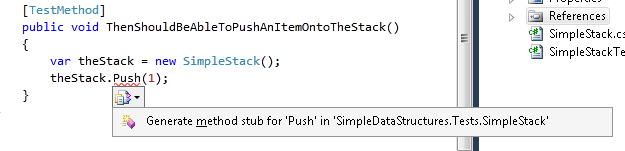
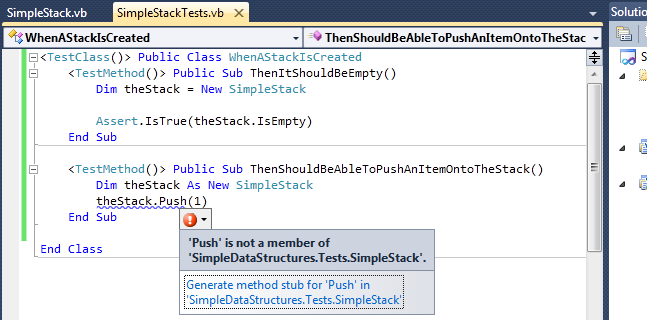
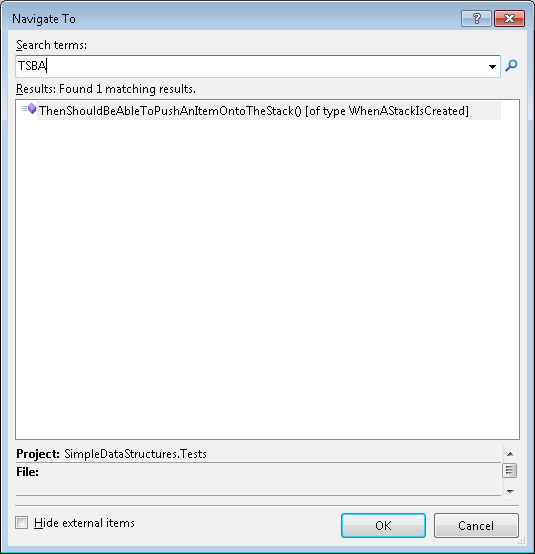
Task 2 – Defining a Context for the Tests.

* 1. In this task, you will write a couple of tests to start driving out the design of your **SimpleStack** class.
  2. You will start by creating a new file to hold some tests for your **SimpleStack** class and then jump into writing out the first test. Along the way, you will use the new **Smart Tag** actions to help us wire up some of the necessary import statements, do some micro-code generation, and more.
  3. From the **Solution Explorer**, right-click on the **SimpleDataStructures.Tests** project and select **Add | Class**. Be sure **Visual C# Items | Code** (or **Common Items | Code** for VB.NET) is selected and then choose the **Class** Template.
  4. Change the name to **SimpleStackTests** and click the **Add** button.
  5. Visual Studio will open the new SimpleStackTests.cs file (or SimpleStackTests.vb). Your solution should look like **Error! Reference source not found.**.
     1. 
     2. Figure
     3. The new SimpleStackTests.cs file(C#)
     4. 
     5. Figure
     6. The new SimpleStackTests.vb file(VB)
  6. Visual Studio automatically makes newly generated classes internal by default, but the class must be public in order for the Test Runner to find and run the tests. Add the **public** keyword as seen in the following code.
     1. C#
     2. namespace SimpleDataStructures.Tests
     3. {
     4. **public class SimpleStackTests**
     5. {
     6. }
     7. }
     8. **Note:** Visual Basic creates classes with public scope by default, and class files do not include namespaces unless you need to manually alter them from the default.
     9. Visual Basic
     10. **Public Class SimpleStackTests**
     12. End Class
  7. Next, you need to decorate this class with the **TestClass** Attribute so the Visual Studio Test Runner will recognize this class as a test fixture. Add the **TestClass** attribute to the **SimpleStackTests** class.
     1. C#
     2. namespace SimpleDataStructures.Tests
     3. {
     4. **[TestClass]**
     5. public class SimpleStackTests
     6. {
     7. }
     8. }
     9. Visual Basic
     10. **<TestClass()>** Public Class SimpleStackTests
     12. End Class
  8. Notice there is a problem – Visual Studio does not know what the **TestClass** attribute is because the file is missing a using statement.
  9. You can use a new Smart Tag action, added in Visual Studio 2010, to automatically add the correct **using** (**Imports** if using VB.Net) statement. Press **CTRL + .** to trigger the Smart Tag context menu, and select the first option.
     1. 
     2. Figure
     3. Using the Smart Tag to add a missing using statement
  10. You have a test fixture defined, but before you start writing tests, you need to get some context around what you will be writing the tests for.
      1. **Note:** “Context” in TDD is used for describing the current conditions and state of the **SimpleStack**. “Context” enables us to clearly describe what the test is meant to check. This is a great benefit in having the tests read like regular descriptive text instead of like software code.
      2. In addition, you are using longer and more descriptive names for the test fixtures and test methods. The goal is to communicate the essence of the requirement in the naming of tests, and let the test body (code) communicate the mechanics.
      3. While this may seem odd and go against some pre-conceived notions about writing code, it is common practice amongst many [TDD](http://en.wikipedia.org/wiki/Test_Driven_Development) & [BDD](http://en.wikipedia.org/wiki/Behavior_driven_development) practitioners.
  11. Begin by assuming the stack has just been created, meaning it is in a default state. To set this context, rename the **SimpleStackTests** class to something more descriptive, like so:
      1. C#
      2. **public class WhenAStackIsCreated**
      3. {
      5. }
      6. Visual Basic
      7. **<TestClass()> Public Class WhenAStackIsCreated**
      9. End Class

Task 3 – Writing the First Test: Designing and Defining the SimpleStack

* 1. In this task, you will start designing and building the **SimpleStack** class by writing a test using the context defined in Task 2. Along the way, you will use the new Smart Tag actions to help wire up some of the necessary using statements, do some micro-code generation, and more.
  2. Begin by creating a new test method that specifies that a newly created stack should be empty. Decorate the method with the **TestMethodAttribute**. Note the relationship of the test class and test method names: **WhenAStackIsCreated.ThenItShouldBeEmpty()**. This naming convention helps us keep a clear understanding of what each test is checking.
     1. (Code Snippet – *TDD Lab - Ex1 ThenItShouldBeEmpty C#*)
     2. C#
     3. **[TestMethod]**
     4. **public void ThenItShouldBeEmpty()**
     5. **{**
     6. **}**
     7. (Code Snippet – *TDD Lab - Ex1 ThenItShouldBeEmpty VB*)
     8. Visual Basic
     9. **<TestMethod()> Public Sub ThenItShouldBeEmpty()**
     10. **End Sub**
  3. Next, you need to declare and initialize a new **SimpleStack**, and then verify that its **IsEmpty** property returns true. To begin, just declare and initialize a new **SimpleStack**.
     1. C#
     2. [TestMethod]
     3. public void ThenItShouldBeEmpty()
     4. {
     5. **SimpleStack theStack = new SimpleStack();**
     6. }
     7. Visual Basic
     8. <TestMethod()> Public Sub ThenItShouldBeEmpty()
     9. **Dim theStack As New SimpleStack**
     10. End Sub
  4. Notice that Visual Studio will again give you a Smart Tag, but for a different reason than before. In this case, Visual Studio does not know what the **SimpleStack** class is, so it will offer to generate a stub for you.
     1. 
     2. Figure
     3. Smart Tag options for an as-yet-undefined class (C#)
     4. 
     5. Figure
     6. Smart Tag options for an as-yet-undefined class (VB)
  5. Select Generate class for “**SimpleStack**” from the list (or Generate ‘Class **SimpleStack**’ if using VB.Net). A new window will open with a stub of the new class generated for you. Open the file called SimpleStack.cs or SimpleStack.vb from Solution Explorer.
     1. 
     2. Figure
     3. The generated stub of SimpleStack class(C#)
     4. 
     5. Figure
     6. The generated stub of SimpleStack class(VB)
     7. **Note:** The Smart Tag above has a second option, “Generate other…” which will open a dialog, allowing you to customize the to-be-generated code. Had you chosen that option you would have seen the following:
     8. 
     9. Take special note of the Project location option. By default, the Smart Tag will generate the stub code in the current project, but that may not be the behavior you desire. The Project location field gives you the ability to create the generated code in a completely different location.
     10. As an important side note, TDD purists will use the Smart Tag approach, generating the code directly in the same location where they are currently working. This subtle design approach encourages you the developer to make your fundamental design choices as late in the process as possible. In your small solution it is obvious where the generated code should live; however, in larger projects you might have 20 or 30 different components. Your code might need to be accessible by a number of different components. How can you know this early in the development cycle exactly which components may need access to the code you are developing?
     11. Leaving the decision of where to move the code until late in the cycle means you have gotten a much clearer picture of potential dependencies in the system, and are better able to make smart choices about where to place code you have written. All this falls back to the fundamental tenet: Do the simplest thing possible to make the test pass.
     12. (For more on the concept of making critical decisions at the last responsible moment, you highly recommend Tom and Mary Poppendeick’s *Lean Software Development*.)
     13. In this case, you triggered the Smart Tag from within your **SimpleDataStructures.Tests** project, but you likely want the **SimpleStack** implementation to live in the **SimpleDataStructures** project and namespace. You could have done that by selecting the corresponding Project Location in this dialog.
  6. Going back to the test, you will verify that the stack is empty by asserting its **IsEmpty** property is true.
     1. C#
     2. [TestMethod]
     3. public void ThenItShouldBeEmpty()
     4. {
     5. SimpleStack theStack = new SimpleStack();
     7. **Assert.IsTrue(theStack.IsEmpty);**
     8. }
     9. Visual Basic
     10. <TestMethod()> Public Sub ThenItShouldBeEmpty()
     11. Dim theStack As New SimpleStack
     12. **Assert.IsTrue(theStack.IsEmpty)**
     13. End Sub
  7. Again, you will see a Smart Tag, this time offering to Generate property stub for **IsEmpty** in **SimpleDataStructures.Tests.SimpleStack**. Selecting this option will generate a public auto-Property in the **SimpleStack** class.
     1. 
     2. Figure
     3. The Smart Tag showing the option to create a method stub(C#)
     4. 
     5. Figure
     6. The Smart Tag showing the option to create a method stub (VB)
  8. Move to the SimpleStack.cs file (or SimpleStack.vb) and you will see the stubbed property.
     1. 
     2. Figure
     3. The stubbed IsEmpty property (C#)
     4. 
     5. Figure
     6. The stubbed IsEmpty property (VB)
  9. Now run the test by right-clicking the test method and selecting **Run Tests** (or type Ctrl+R, T). You should observe the assertion that **IsEmpty** returns true fails, and therefore, the test fails too.
  10. In TDD fashion, you will now do the simplest thing possible to get the test to pass, you will simply return true for the getter!
      1. **Note:** This may seem a silly, non-sensical step; however, it is critical for a number of reasons. First, you are checking the test you just wrote. If your test does not pass, then you have created an error in your test – checking for false when you may have meant to check for true, for example. Secondly, this enforces the concept of doing the simplest thing possible to make the test pass.
      2. As part of doing the simplest thing possible, you also changed this from an auto-Property, which is both read and write, to a read-only Property. You did this because it is the minimum functionality, and code, required to pass the test. The test does not specify having to write to the stack, so you are not going to allow your code to have that functionality. You are keeping the code lean, which improves legibility and decreases your exposure to bugs. (Less code == less chance for bugs!)
      3. C#
      4. class SimpleStack
      5. {
      6. public bool IsEmpty
      7. {
      8. **get { return true; }**
      9. }
      10. }
      11. (Code Snippet – *TDD Lab - Ex1 IsEmpty Returns True VB*)
      12. Visual Basic
      13. Public Class SimpleStack
      14. **Public ReadOnly Property IsEmpty() As Boolean**
      15. **Get**
      16. **Return True**
      17. **End Get**
      18. **End Property**
      19. End Class
  11. The next step is to go back to the test; however, you do not need to move your hands off the keyboard and reach for the mouse. Instead, use the new Quick Search features to navigate the code base in just a few key strokes. Hit **CTRL + ,** (that is Control and a comma) to trigger the Quick Search dialog, and then start typing the name of a File, Class, or Method. Quick Search will filter the list of available locations in the code base for you. When you have filtered enough and located the spot in the code you want to go, use the **Up** and **Down** arrow keys to highlight the option, and hit the **Enter** key.
      1. 
      2. Figure
      3. Use Quick Search to navigate directly to a particular method
      4. **Note:** Visual Studio will navigate to that location in the code base, place the cursor at the start of the Class or method name, and you are ready to go!
  12. Now that you are back in the test method, go ahead and re-run the test. This time it should pass!
      1. 
      2. Figure
      3. The test passes!

Task 4 – Adding More Functionality Driven by Tests

* 1. At this point, you have a nearly empty shell of a **SimpleStack** implementation. There is only one working member: a read-only property. You can continue adding functionality by driving out the design with tests. Now, look to add the ability to put items onto, and take items off the **SimpleStack**.
  2. Add a new test that specifies a new method on **SimpleStack**, the **Push** method, which takes in an integer:
     1. (Code Snippet – *TDD Lab - Ex1 ThenShouldBeAbleToPushAnItemOntoTheStack C#*)
     2. C#
     3. **[TestMethod]**
     4. **public void ThenShouldBeAbleToPushAnItemOntoTheStack()**
     5. **{**
     6. **var theStack = new SimpleStack();**
     7. **theStack.Push(1);**
     8. **}**
     9. (Code Snippet – *TDD Lab - Ex1 ThenShouldBeAbleToPushAnItemOntoTheStack VB*)
     10. Visual Basic
     11. **<TestMethod()> Public Sub ThenShouldBeAbleToPushAnItemOntoTheStack()**
     12. **Dim theStack As New SimpleStack**
     13. **theStack.Push(1)**
     14. **End Sub**
  3. Under the **Push** method call, you should notice a Smart Tag. Trigger the Smart Tag by hitting CTRL + . and you will see it is offering to generate a method stub for the **Push** method. Hit **Enter** and let Visual Studio generate the stub.
     1. 
     2. Figure
     3. Use the Smart Tag to generate method stubs(C#)
     4. 
     5. Figure
     6. Use the Smart Tag to generate method stubs(VB)
  4. Now run the test and make sure it fails.
     1. **Note:** It may seem odd to run the test at this point because you know the code that Visual Studio stubbed in for us should cause the test to fail. However, an important part of TDD is the cadence of Red – Green – Refactor, and a key step in that cadence is making sure the test actually does fail, as you expect.
     2. Why is this key? Because there will inevitably be cases where a test passes by coincidence, despite what you know should happen. In those cases, you need to make sure your tests are specifying the correct behaviors and fix them if not.
  5. Your new goal is to get this test to pass. Being true to TDD, you will do the absolute simplest thing you can to get the test to go green: you will get rid of the **NotImplementedException** in the **Push** method! At this point, you are not going to add any implementation in the method – the test you have only checks that you can successfully push something on to the stack.
     1. C#
     2. internal void Push(int p)
     3. {
     5. }
     6. Visual Basic
     7. Sub Push(ByVal p As Integer)
     9. End Sub
  6. Navigate back to the test using Quick Search (CTRL + ,) by entering “TSBA” into the search box.
  7. Notice that Quick Search will filter the results by matching the capital letters TSBA to the capital letters in the test name you are looking for – **ThenShouldBeAbleToPushAnItemOntoTheStack**.
     1. 
     2. Figure
     3. Use Quick Search to filter and navigate using just the capital letters of a class, method, or file name
  8. Re-run the test and it should pass. Green! Unfortunately, the push method is not very useful at this point, so you will work on that next.

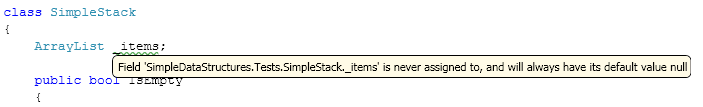
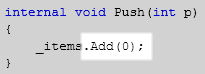
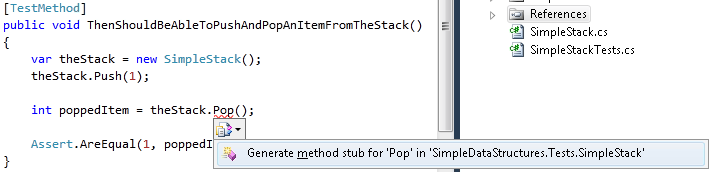
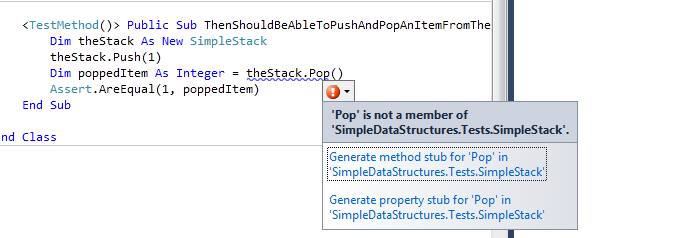
# Next Step

* 1. Exercise 2: Refactor!

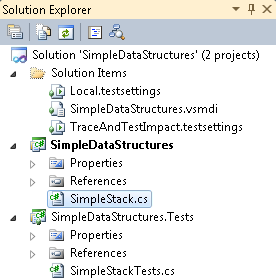
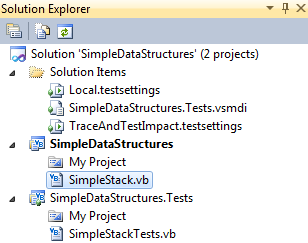
Exercise 2: Refactor!

* 1. While the system code you just created passes the test, it is obvious that it is completely nonsensical. Continue iterating through your work, refactoring from this simplest example to something that actually makes sense in addition to passing the test. Now you will move on to the refactoring stage.

Task 1 – Moving from Passing the Test to Building Functionality

* 1. Open Microsoft Visual Studio 2010 from **Start** | **All Programs** | **Microsoft Visual Studio 2010** | **Microsoft Visual Studio 2010**.
  2. Open the **SimpleDataStructures.sln** solution file. By default, this file is located in the folder: ***\Source\Ex02-Refactor\begin\*** and choose the language of your preference. Optionally, you can continue working the solution you created in the previous exercise.
  3. You need to be able to verify that pushing an item onto the stack actually alters the stack, that is, after the **Push** operation; the stack holds an item as you expect. One way of verifying that might be to make sure that the stack is no longer empty after pushing an item onto the stack.
  4. Update the test accordingly:
     1. C#
     2. [TestMethod]
     3. public void ThenShouldBeAbleToPushAnItemOntoTheStack()
     4. {
     5. var theStack = new SimpleStack();
     6. theStack.Push(1);
     8. **Assert.IsFalse(theStack.IsEmpty);**
     9. }
     10. Visual Basic
     11. <TestMethod()> Public Sub ThenShouldBeAbleToPushAnItemOntoTheStack()
     12. Dim theStack As New SimpleStack
     13. theStack.Push(1)
     14. **Assert.IsFalse(theStack.IsEmpty)**
     15. End Sub
  5. Now run the test and it should fail. What causes the failure?
  6. Remember, you did the simplest thing you could in the implementation of **IsEmpty** and always returned true. Now you need to make some more design decisions about how **IsEmpty** should really behave, and what that means for the **Push** method as well.
     1. **Note:** Test-Driven Development is also commonly called Test-Driven Design for the very reasons you are starting to see – you are defining how your system should behave and designing it to deliver that behavior in a very iterative manner.
  7. After navigating back to the **SimpleStack** class and looking closer, it appears the stack is going to need a mechanism for storing some state – specifically the state of what item(s) the stack holding onto.
  8. Storing the items in a list would allow us to determine if the stack **IsEmpty** by checking the size of the list. It would also give us somewhere to store the pushed items.
     1. (Code Snippet – *TDD Lab - Ex2 Refactor SimpleStack C#*)
     2. C#
     3. class SimpleStack
     4. {
     5. **ArrayList \_items;**
     7. **public bool IsEmpty**
     8. **{**
     9. **get { return \_items.Count == 0; }**
     10. **}**
     12. **internal void Push(int p)**
     13. **{**
     14. **\_items.Add(0);**
     15. **}**
     16. }
     17. (Code Snippet – *TDD Lab - Ex2 Refactor SimpleStack VB*)
     18. Visual Basic
     19. Class SimpleStack
     20. **Private \_items As ArrayList**
     21. **Public ReadOnly Property IsEmpty() As Boolean**
     22. **Get**
     23. **Return (Me.\_items.Count = 0)**
     24. **End Get**
     25. **End Property**
     26. **Friend Sub Push(ByVal p As Integer)**
     27. **Me.\_items.Add(0)**
     28. **End Sub**
     29. End Class
  9. Notice the **ArrayList** type is not recognized. Press **CTRL+.** and select the option to import the **System.Collections** namespace.
  10. Now run the tests. You may be surprised to see they continue to fail. The tests are failing because you have not initialized your **ArrayList**. Look closely and notice Visual Studio is telling you that with the little green squiggly underline:
      1. 
      2. Figure
      3. The green squiggly line and tooltip calling attention to an uninitialized field(C#)
  11. Add a default constructor and initialize the list in the constructor, as shown in the following code.
      1. (Code Snippet – *TDD Lab - Ex2 SimpleStack Constructor C#*)
      2. C#
      3. class SimpleStack
      4. {
      5. ArrayList \_items;
      7. **public SimpleStack()**
      8. **{**
      9. **\_items = new ArrayList();**
      10. **}**
      11. …
      12. }
      13. (Code Snippet – *TDD Lab - Ex2 SimpleStack Constructor VB*)
      14. Visual Basic
      15. Class SimpleStack
      16. Private \_items As ArrayList
      17. **Public Sub New()**
      18. **Me.\_items = New ArrayList**
      19. **End Sub**
      20. …
      21. End Class
  12. Run the tests again and this time they will pass with green colors!
  13. How do you know you have stored the correct item? All you have done is push something onto the stack. In this case, in an earlier step in the lab you stuck to doing the simplest thing and just added a zero (0) onto the stack.
      1. 
      2. Figure
      3. Do the simplest thing possible to pass a test
  14. That is not too useful. You need to drive out something a bit more functional.
  15. Add a new test that will first push an item onto the stack, then pop an item off, and make sure you have the same item that was pushed onto the stack.
      1. (Code Snippet – *TDD Lab - Ex2 ThenShouldBeAbleToPushAndPopAnItemFromTheStack C#*)
      2. C#
      3. **[TestMethod]**
      4. **public void ThenShouldBeAbleToPushAndPopAnItemFromTheStack()**
      5. **{**
      6. **var theStack = new SimpleStack();**
      7. **theStack.Push(1);**
      9. **int poppedItem = theStack.Pop();**
      11. **Assert.AreEqual(1, poppedItem);**
      12. **}**
      13. (Code Snippet – *TDD Lab - Ex2 ThenShouldBeAbleToPushAndPopAnItemFromTheStack VB*)
      14. Visual Basic
      15. **<TestMethod()> Public Sub ThenShouldBeAbleToPushAndPopAnItemFromTheStack ()**
      16. **Dim theStack As New SimpleStack**
      17. **theStack.Push(1)**
      18. **Dim poppedItem As Integer = theStack.Pop()**
      19. **Assert.AreEqual(1, poppedItem)**
      20. **End Sub**
  16. Again, notice the Smart Tag that appears below the newly defined **Pop** method.
      1. 
      2. Figure
      3. Use Smart Tags to generate method stubs, complete with the correct return type (C#)
      4. 
      5. Figure
      6. Use Smart Tags to generate method stubs, complete with the correct return type (VB)
  17. Trigger the Smart Tag and let it generate a method stub for the **Pop** method, and then navigate to the new stubbed method using Quick Search (CTRL + ,).
  18. Notice that Visual Studio was able to determine that **Pop** needed to return a value, and it inferred the correct return type – in this case, an **int**.
      1. C#
      2. internal int Pop()
      3. {
      4. throw new NotImplementedException();
      5. }
      6. Visual Basic
      7. Function Pop() As Integer
      8. Throw New NotImplementedException
      9. End Function
  19. Run the test and it will fail due to the **NotImplementedException**, but now that you have defined what **Pop** should look like and how it should behave in this context, you can add the implementation details to get the test passing.
      1. (Code Snippet – *TDD Lab - Ex2 Pop Implementation C#*)
      2. C#
      3. internal int Pop()
      4. {
      5. **int value = (int) \_items[0];**
      6. **\_items.RemoveAt(0);**
      8. **return value;**
      9. }
      10. (Code Snippet – *TDD Lab - Ex2 Pop Implementation VB*)
      11. Visual Basic
      12. Function Pop() As Integer
      13. **Dim value As Integer = CInt(Me.\_items.Item(0))**
      14. **Me.\_items.RemoveAt(0)**
      15. **Return value**
      16. End Function
  20. Running the test once again results in a failure. But why? The **Pop** method looks good, but the test fails due to your naïve **Push** implementation. Time to fix that!
      1. C#
      2. internal void Push(int p)
      3. {
      4. **\_items.Add(p);**
      5. }
      6. Visual Basic
      7. Friend Sub Push(ByVal p As Integer)
      8. **Me.\_items.Add(p)**
      9. End Sub
  21. Run the tests and all three will be green!
  22. **Note:** This lab is not meant to do a deep dive into the principles and practice of Test-Driven Development, nor on how to properly implement a stack.
  23. However, if you were to continue designing and building out the **SimpleStack**, you would need to continue to think about how the stack would handle trying to call Pop on an empty stack. Moreover, what happens when you try to operate against a full stack? Or a partially filled stack?
  24. To tackle those design questions you would need to introduce some new test fixtures, complete with more tests, into your SimpleStackTests.cs file.
  25. In addition, you may have noticed there is also a bug in your implementation! You are pushing on to the end of the **Stack**, but popping from the front! This issue would have been caught in short order as you continued to flesh out your design with tests for pushing and popping multiple items.

Task 2 – Reorganizing the Code

* 1. At some point, you will determine its time to move the **SimpleStack** implementation out of the **SimpleDataStructures.Tests** project and into the **SimpleDataStructures** project and namespace. You could have done that when you created the class (see: Exercise 1, Task #3, Step #2) by using the “Generate Other…” Smart Tag action. However, you opted to do the simplest thing and let Visual Studio use the default settings, which generated the class in the project you were in, when the Smart Tag was triggered, the **SimpleDataStructures.Tests** project.
  2. Using the **Visual Studio Solution Explorer**, select the SimpleStack.cs file (or SimpleStack.vb) from the **SimpleDataStructures.Tests** project. Right-click and select **Cut**, then right-click in the **SimpleDataStructures** project and select **Paste**. At this point the Solution Explorer should look similar to this:
     1. 
     2. Figure
     3. The solution structure after moving the SimpleStack.cs file(C#)
     4. 
     5. Figure
     6. The solution structure after moving the SimpleStack.cs file(VB)
  3. Open the SimpleStack.cs file and change the namespace to **SimpleDataStructures**. Press Ctrl-S to save the file. This is not required if using VB.Net.
     1. C#
     2. **namespace SimpleDataStructures**
  4. Navigate to the tests and you will notice all of the red-squiggly lines. Navigate back to the SimpleStack.cs file (or SimpleStack.vb) again and look at the definition of the SimpleStack class. When Visual Studio generated the class, it did the simplest thing it could and created the class as internal by default because that is the visibility that was needed at the time. Now that you have refactored the class out into a different project and namespace, you need to adjust its visibility.
     1. C#
     2. namespace SimpleDataStructures
     3. {
     4. **public class SimpleStack**
     5. {
     6. …
     7. }
     8. }
     9. Visual Basic
     10. **Public Class SimpleStack**
     11. …
     12. End Class
  5. Along the same lines, you also need to change the visibility of the **Push** and **Pop** methods to be public.
     1. C#
     2. **public void Push(int p)**
     3. {
     4. \_items.Add(p);
     5. }
     7. **public int Pop()**
     8. {
     9. int value = (int)\_items[0];
     10. \_items.RemoveAt(0);
     12. return value;
     13. }
     14. Visual Basic
     15. **Public Sub Push(ByVal p As Integer)**
     16. Me.\_items.Add(p)
     17. End Sub
     18. **Public Function Pop() As Integer**
     19. Dim value As Integer = CInt(Me.\_items.Item(0))
     20. Me.\_items.RemoveAt(0)
     21. Return value
     22. End Function
  6. Finally, you also need to add reference the **SimpleDataStructures** project from the **SimpleDataStructures.Tests** project. Right-click on the **References** in the **Solution Explorer**, and select **Add Reference**. Click the **Projects** tab in the **Add Reference** dialog, select the **SimpleDataStructures** project, and then click **OK**.
     1. 
     2. Figure
     3. Add a reference to the SimpleDataStructures project
  7. Run the tests again and observe they all pass!

# Next Step

Summary

Summary

* 1. In this lab, you have explored some of the new IDE enhancements targeted at the test-first developer. You have seen navigation improvements designed to help you get around your code base more quickly and easily. You have seen the IDE’s ability to help you stub out methods you are targeting in your test classes, even pushing that code into completely different files. You have even seen how simple it is to see the status of your code and design via the test runner.
  2. Finally, and perhaps most importantly, if you are new to test driven design/development, hopefully you have gotten a taste of how Visual Studio can help you move into this exciting approach to creating lean, tight, high-quality software.