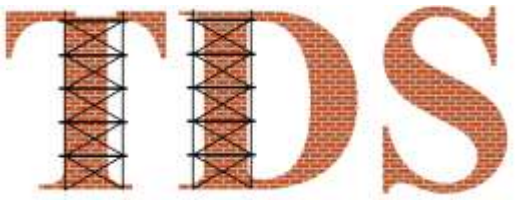


Test-Driven Scaffolding
User's Guide

Version 2.0

August 2017

Vincent R. Johns

About the author: Vincent Johns has written programs for a variety of computers. His software projects have included writing programs to cut cams using numerically controlled machine tools, writing a multitasking operating system for a Univac 9300 computer, and developing automated testing software for avionics and flight-control systems on the B-2 airplane. A project to translate the Classic Adventure program from Fortran to C#, involving numerous methods that needed to be unit-tested, gave rise to TDS as presented in this *TDS User's Guide*. He and his wife live in Oklahoma City with two cats who enjoy helping with the computer but are better at offering encouragement than they are at operating keyboards. 🐈 🐈 💻

Note: The "TDS" image on the cover is intended to suggest the building of a brick wall, employing scaffolding, which is to be removed when the wall is complete.

Table of Contents

1 Introduction 9
1.1 Navigation 9
1.2 Purpose 9
1.3 Sneak preview 9
1.4 Background 10
1.5 What is this "TDS" software supposed to do?? 12
1.5.1 Quickly/easily define test methods for exercising working code 12
1.5.2 Support debugging and testing working code 12
1.5.3 Optionally generate simple test reports 12
1.5.4 Support test-first and/or code-first development 12
1.5.5 Easily remove TDS code 12
1.6 About this document 13
1.6.1 Use the Tutorial as an introduction 13
1.6.2 Don't try to read this document like a novel 13
1.6.3 TDS is written in C# 13
1.6.4 Extra details are intentionally included in this manual 13
1.7 Packing list 14
1.8 Why is this called "Test-Driven Scaffolding"? 14
1.8.1 Test-Driven Development (TDD) is a well-known development methodology 14
1.8.2 Scaffolding helps with construction. 16
1.8.3 Concurrent development 16
1.8.4 Which is better? 16
1.9 Using the code and this document 17
1.10 Takeaways 17
1.10.1 Why should one use TDS? 18
1.10.2 Purpose of TDS 18
1.10.3 Features of TDS 18
1.10.4 Steps for customizing/setting up 21
1.10.5 "I'll take care of that pretty soon." 22
2 Introduction to the Tutorial and example projects 23
2.1 Short-circuit the long explanation 23

2.1.1 Really quick overview 23
2.2 What to expect 23
2.2.1 Results of running the Tutorial 23
2.2.2 Using the Tutorial as a foundation 24
2.2.3 Apology 24
2.3 Usage notes 25
2.3.1 Essential steps 25
2.3.2 Typography 25
2.3.3 Copying code 25
2.3.4 Navigational aids in this document 26
3 Overview 29
3.1 Add a TDS Project to your Solution 29
3.2 Set references 29
3.3 Edit TDS.cs 29
3.3.1 Link to the namespaces of testable types 30
3.3.2 Specify which static variables need to be activated 30
3.3.3 Identify active TDS source-code files 30
3.3.4 Delete unused TDS method definition 30
3.3.5 Identify active TDS test methods 30
3.4 Do a TDS "smoke test" 30
3.5 Add a TDS test method 30
3.6 Run TDS for tracing and debugging 31
3.7 Run TDS for testing (optional) 31
3.8 Hide TDS when done 31
4 Tutorial 32
4.1 Learning objectives 32
4.2 Tutorial road map 32
4.3 Set up Visual Studio and TDS files [20 minutes] 34
4.3.1 Intended environment 34
4.3.2 Check that Visual Studio ("VS") is installed 34
4.3.3 Create an empty file folder for your VS Solution 34
4.3.4 Extract the contents of the TdsSource.zip file 35

4.3.5 Configure Visual Studio 35
4.3.6 Set up simulated existing working code 36
4.4 Set up TDS 38
4.4.1 Add TDS code to the Solution [6 minutes] 38
4.4.2 Hide "unhandled exception" messages 39
4.4.3 Run a "smoke test" of TDS [4 minutes] 41
4.4.4 Import the code snippet file [5 minutes] 44
4.4.5 TDS is ready to use 46
4.5 Use alternate unit-test platforms 46
4.5.1 NUnit demonstration [15 minutes] 47
4.5.2 Microsoft Unit Test platform demonstration [15 minutes] 54
4.5.3 Return to the TDS platform [3 minutes] 57
4.6 Exercise an existing TDS method [12 minutes] 57
4.6.1 Set up the Task List window 58
4.6.2 Buggify: Raise the wrong exception 58
4.6.3 Buggify: Ignore an illegal argument 58
4.6.4 Buggify: Return a false calculated value 59
4.6.5 Bugs are gone; signal "Inconclusive" result 59
4.6.6 Buggify: Cause an exception to fail to be raised 60
4.7 Run the working code without TDS 60
4.8 Create a new TDS method 61
4.8.1 Clean up the test report 61
4.8.2 Create and run a TDS test [22 minutes] 62
4.8.3 Modify and test the working code [35 minutes] 66
4.8.4 Resume testing elsewhere [4 minutes] 73
4.8.5 Find existing TDS methods [3 minutes] 74
4.8.6 Use named objects in `testValues[]` [6 minutes] 75
4.8.7 Filter test methods and test cases [15 minutes] 76
4.8.8 Test inaccessible function members (optional) [15 minutes] 79
4.9 Add a #define symbol 83
4.10 Create a new TDS method file 85
4.10.1 Make a copy and customize it [15 minutes] 85

4.10.2 Add a TDS method [6 minutes] 87
4.10.3 Add the working-code stub to be debugged/tested [20 minutes] 87
4.11 Automate the testing 88
4.11.1 Set up script files [5 minutes] 89
4.11.2 Run tests independently of VS or NUnit 89
4.12 Hide TDS when done 93
4.12.1 Consider keeping the TDS methods after the Solution is complete 93
4.13 What's next? 94
4.14 Comments on the Tutorial 94
4.14.1 Purpose 95
4.14.2 Requirements statement for code being developed 96
4.14.3 Assert statements in TDS and other test platforms 97
4.14.4 Filtering test cases in TDS 98
4.14.5 Calling static constructors 99
4.14.6 Running NUnit 100
4.14.7 Setting up a stand-alone TDS Project 101
4.14.8 Ignoring unhandled exceptions 106
4.14.9 Documentation (XML) comments 107
4.14.10 Adding properties to testValues[] 115
4.14.11 Using named types in testValues[] 117
4.14.12 Using Debug mode 118
4.14.13 Multiple TDS source files 119
4.14.14 Example project name 120
4.14.15 Customizing TDS 121
4.14.16 Task List (`//TODO:`) comments 122
4.14.17 Navigating in Visual Studio 124
4.14.18 Use of "////" in comments 125
5 Examples – long, picky details 126
5.1 Example: Adding a new method, Succ(), to a new Solution 127
5.1.1 What we shall do in this example 127
5.1.2 Overview of this example 127
5.1.3 Learning objectives 133

5.1.4 Requirements for "Successor" calculation 133
5.1.5 Set up a new function member and its TDS method 134
5.1.6 Convert TDS method to a test procedure 145
5.1.7 Test the new method 148
5.1.8 Refine the new function member and its TDS method 149
5.2 Example: Adding a new method, Fib(), to a new Solution 150
5.2.1 Overview of this example 150
5.2.2 Learning objectives 151
5.2.3 Statement of purpose of the code in this example 151
5.2.4 Analyze the problem mathematically 151
5.2.5 Requirements for Fibonacci sequence calculation 158
5.2.6 Set up a new function-member stub and its TDS method 158
5.2.7 Overview of using alternate calculations 173
5.2.8 Convert the TDS method to a test procedure 173
5.2.9 Test the new method 184
5.3 Example: Modifying an XElement via a new method 209
5.3.1 Overview of this example 209
5.3.2 Learning objectives 210
5.3.3 Statement of purpose of the code in this example 210
5.3.4 Analysis 210
5.3.5 Requirements for the **`InsertSymbol()`** method 211
5.3.6 Set up a new function-member stub and its TDS method 211
5.3.7 Begin coding the method 215
5.3.8 Do a smoke test on the new TDS method 216
5.3.9 Set up for validation 217
5.3.10 Delete the **`throw`** 220
5.3.11 Add XML comments 220
5.3.12 Calculate the summary string 221
5.3.13 Update the call to **`InsertSymbol()`** 222
5.3.14 Check the revised code 223
5.3.15 Add a parameter specifying editing 223
5.3.16 Add specifications for insertions of additional **`<Symbol>`**s 231

5.3.17 Begin automatic testing 233
5.3.18 Do some housekeeping (refactor subexpressions) 258
5.3.19 Re-enable all TDS tests 270
5.3.20 Summary 270
5.3.21 Test using NUnit 271
5.4 Example: Testing a Visual Basic Project 271
5.4.1 Create & run an example Project3 271
5.4.2 Add the TDS Project to the Solution 273
5.4.3 Construct a TDS method 273
5.4.4 Run the TDS tests 274
5.4.5 Clean up the code 274
5.4.6 Re-enable all TDS tests 274
6 That's all for now 275
7 Glossary 276
8 Subject Index 279

1 Introduction

1.1 Navigation

For suggestions on navigating in this *TDS User's Guide*, please see section 2.3.4 below. If your viewing software supports hyperlinks, you may go there by clicking on its number here, "2.3.4".

1.2 Purpose

This package is intended to make it almost trivially easy to develop unit-test methods for function members in a Visual Studio Solution. Having added to the Solution a Project (called "TDS") that provides some optional reporting services, one may use a C# code snippet, "TdsTest", to insert a generic unit-test method into the TDS Project. This "TDS method", suitably customized, may call a function member elsewhere in the Solution to assist in tracing and debugging it, or may perform simple unit tests on the function member. The TDS method is also compatible with NUnit and with Visual Studio Test, and may perform unit tests under their control instead of using the built-in TDS test reporting facility.

This *TDS User's Guide* provides detailed instructions and examples of intended usage. For a more detailed description of how one might choose to benefit from TDS, please see section 1.5 below. Of the accompanying files listed in section 1.7, only TDS.cs and TestMethodSnippet.snippet are essential parts of the package; the others provide examples of possible ways to use the system.

1.3 Sneak preview

The next few sections of the *TDS User's Guide* discuss what to expect from using the TDS software to assist with exercising and unit-testing computer programs developed with the help of Microsoft Visual Studio 2017 (and some other versions of Visual Studio). If you are more interested in "how" to use it than in "why" it might interest you, you may prefer to skip to section 2 now.

The *TDS User's Guide* includes some discussion of test platforms NUnit and Visual Studio Test, with both of which the TDS methods are compatible. I view those as management systems for unit testing (and TDS can perform some of their functions, too, including generating basic test reports), whereas TDS is intended to make creating new unit-test methods easy to do; this is apparently not a primary purpose of those other platforms. Many examples are presented here to demonstrate the intended process, and if you don't like the suggested system, suggestions are also included on how to customize it to your (and your team's) own preferences. Although some setup is initially required, the purpose is, ideally, to enable you to define a basic TDS method, ready to run, along with its corresponding stub of a function member, in four or five minutes. The idea is to avoid much of the tedium of setting up the test method, and thus to avoid the temptation to omt or postpone creating the test method. This guide and the accompanying files are offered in the hope that they will make it easier, faster, and less painful to do a Solution's needed testing.

The code that you can develop and exercise with the help of TDS may be written in any of various languages that allow a method written in C# to invoke it (as shown in section 5.4, "Example: Testing a Visual Basic Project"); however, most of the examples of working code in this *TDS User's Guide* are written in C# version 5.0, and the TDS code itself is written in C# version 5.0 and utilizes the Desktop template in Visual Studio[1]. The

[1] With some reluctance, I have omitted some newer features of C# from the TDS code, in an effort to make the code usable with earlier versions of the compiler, based on the notion that someone without access to the newer versions might otherwise be unable to use the code as written, and the present code works with both. If I publish a revised

TDS methods that you will develop will also be expressed as C# code. If you plan to use Visual Studio Community 2017, install it using the workload ".NET desktop development", or modify its current installation to add that workload..-

The TDS files intended to accompany this *TDS User's Guide* are described in section 1.7.

Actually, it's possible that you can learn to use the TDS software with just a few bare-bones instructions. (The TDS source code is intended to be kind of self documenting, so you might be able to infer most of what you need to use TDS, just from perusing the C# and XML code in the TDS files.) The *TDS User's Guide* contains detailed instructions in section 4, the Tutorial, but, briefly, you can do the following to get going:

► [2] Start Visual Studio and import file TestMethodSnippet.snippet[3] into it.

► Add a new Visual C#, Windows Classic Desktop, Console App (or Application) Project to an existing Visual Studio Solution, call this new Project "TDS", and set it as the StartUp Project.

► Add existing item TDS.cs from TdsSource.zip to Project TDS, and delete existing item Program.cs from Project TDS.

► Edit the code in TDS.cs according to the instructions in the Task List.

► In Project TDS, set References to the namespaces of the working code that you wish to invoke.

► Use code snippet TdsTest to create the definition of a new TDS method[4] for each function member of your Solution that you wish to debug and/or test.

► Run the Solution, with suitable breakpoints set, to trace into the working code for debugging, or without breakpoints to perform tests and generate a test report. When an Assert exception occurs, uncheck the "Break when…" box and resume running.

► Ignore the other six files in TdsSource.zip, and skip reading the rest of this *TDS User's Guide* — you have most of what you need to create (almost) instant test methods.

If you would like some more detailed guidance, the rest of the *TDS User's Guide* provides explanations, instructions, and examples suggesting ways in which you may use TDS to help debug and test function members of your Visual Studio Solutions. For example, a somewhat more detailed version of the above steps is presented in section 2.

1.4 Background

Developing unit tests for software is an unattractive task for many developers. (I don't mind doing testing, but I can understand the feeling — it may seem that time spent on testing doesn't create any new functionality in the code being developed and therefore can seem wasted.) This *TDS User's Guide* and the accompanying software present a mechanism that I call "Test-Driven Scaffolding" (or "TDS"), an innovative use of NUnit and/or the testing facilities in Visual Studio, using a C# code snippet that you can use right away to help with

---

version of TDS, it will almost certainly include interpolated string expressions, since I think they're easier to read, but they do not appear here.

[2] The marker "► " identifies actions to be taken; see section 2.3.1.

[3] Snippets used by TDS are contained in the TestMethodSnippet.snippet file, included in the accompanying TdsSource.zip file. The file needs to be imported here only if it has not already been imported into your installation of Visual Studio.

[4] For example, insert it at Task "**`TODO: New TDS methods may be placed here`**" in file TDS.cs.

developing or modifying function members (metods, properties, indexers, etc.) belonging to the definitions of types in a program[5]. The working code does not need to be developed using C#; for example, I have also used TDS to exercise code written in Visual Basic[6], but C# is used for most of the examples in this *TDS User's Guide*. However, regardless of the language used, the working code does need to be able to be invoked from within a C# Project, since that Project is the means by which we shall add TDS functionality to a Visual Studio Solution.

I suppose I should mention that I haven't always considered testing to be as important as designing the project well and taking care in building it. If I never make a mistake, no bugs should ever appear in my code! Sadly, for most of us, mistakes are a part of life, and I claim that testing is a valuable adjunct to good design and documentation. Just as a cost/benefit analysis of the design may show that doing a mathematical proof of the code's correctness would be unreasonably expensive or difficult to accomplish, you may find that the same would be true of exhaustive testing. A suitable mix of some analysis and some testing, at a modest cost[7], would provide multiple means to attack/avoid malfunctions. Straightforward and well-documented design of your app can communicate to a user that it's reasonable to expect the app to work as advertised in real life. Tests can help do that as well, by demonstrating how the app performs in simulated realistic situations. From another viewpoint, re-checking the design and running tests before exposing the product to potential users can avoid an embarrassing failures. (For example, see the reference to failing the "happy path" in section 1.10.5.)

Since I suspect that thinking of testing as a burdensome task is likely to lead one to skip doing it, a major purpose of TDS is to try to make defining and running unit tests easy and inexpensive. Having added a TDS Proeject to a Solution, I can usually add a new TDS method to the TDS Project in a couple of minutes. The TDS method may not do much at first, but its presence is a reminder to attend to it and its corresponding working code, and adding test cases as needed should be easy to do. However, a possible danger in making it too easy to add TDS methods is to become complacent, thinking that a simple test is all that needs to be done, when it would actually be wise to add details, to give the working code some meaningful exercise. So… I suggest balancing convenience with effectiveness; do a reasonable amount of testing, but not so much as to unduly impact development of the working code.

In the examples in this *TDS User's Guide*, we shall use the TDS code not only for testing new or existing working code[8], as one would do using Test-Driven Development (TDD), but also for constructing drivers[9] that can be used to feed data to their function members for use in tracing execution.

When the working code has attained a sufficient level of functionality (for example, when there exists at least one set of inputs that enable the function member to return control to the caller), you can convert into a usable test method the TDS method that supplied data to it for tracing, as the examples will illustrate. All we need to do to change the TDS method to be a test method is to add some **`Assert`** statements, and the code snippet contains examples of those. The newly developed function member can remain untouched during this

---

[5] These type definitions are collectively referred to, in this document, as "working code". Most of the examples shown in this *TDS User's Guide* are expressed as C# source code.
[6] For an example of VB code run using TDS, please see section 5.4.1.
[7] I am assuming here that your software will not put human lives at risk nor be responsible for preventing major environmental damage, where your development budget should be essentially unlimited. Don't use TDS for that kind of project.
[8] I use the term "working code" to refer to the executable code that one might develop with the help of TDS methods, to distinguish it from code used to define the TDS methods themselves.
[9] These drivers are called "TDS methods" in this document.

conversion. In the examples provided in this *TDS User's Guide* and the accompanying code, we do leave the working code mostly unchanged, except to verify that its TDS test method correctly identifies failures.

The TDS package consists largely of C# source code that can be included as an added Project in a Visual Studio Solution containing your working code, and used either mostly unchanged or modified to suit your needs. You may find it useful to delete TDS features that you do not need, or to add new functionality to it that builds on the basic package. An examples of a feature that you might wish to delete could include the means for placing TDS methods in multiple source-code files, if you expect that you will always use only a single C# (*.cs) source-code file in a given Solution to contain that Solution's TDS methods. Details on how to extend TDS to add functionality that more precisely fits your needs are left as an exercise for the reader. ☺ However, the TDS source code does contain numerous comments that are intended to assist with doing this.

1.5 What is this "TDS" software supposed to do??

1.5.1 Quickly/easily define test methods for exercising working code

As you develop code for a C# type's function member, such as a method or an indexer, you must often also write some additional code that exists only to exercise that function member — which is the code you're really interested in. The purpose of the Test-Driven Scaffolding (TDS) code described in this *TDS User's Guide* is to help you quickly produce this less-interesting exercising or driving code, by providing a means to insert into your development projects some generic test-method code that you may use for this purpose, freely modifying it in any way you choose — but in many instances, only a modest amount of editing will be needed to make it work.

1.5.2 Support debugging and testing working code

TDS is intended to simplify two tasks:

1. (as drivers) providing a consistent means of specifying data for use in exercising or debugging code in your projects; this may be helpful if you are maintaining several function members
2. (as test methods) reporting on how well the observable results of running a function member match its intended behavior, once the function member's code is able to provide such results

1.5.3 Optionally generate simple test reports

Besides the test-method code snippet and example test methods, the accompanying TDS files in TdsSource.zip (see section 1.7 below) provide some basic infrastructure for generating test reports that can help identify code that needs further attention. However, TDS is also intended to save time and effort even if we do not use its test-report features.

1.5.4 Support test-first and/or code-first development

You may create a TDS method either before or after creating its corresponding function member, though the function member must be visible to the TDS method. If you start with defining the TDS test method, and letting VS generate a method stub from the calling expression, doing so may help with design by fixing in the code an idea of what the function member is expected to do.

On the other hand, even if much of the function-member code already exists by the time its TDS method is produced, the new TDS method is intended to be easy to add to the TDS Project and be customized for use with the existing function member. I do both at times; either way can work.

1.5.5 Easily remove TDS code

Like scaffolding on a building during construction or maintenance, the TDS code described in this *TDS User's Guide* can be added to a new or existing project and removed when it is no longer needed. (However, I suggest

keeping the TDS code on hand somewhere even after it's removed from the project, for reasons mentioned in section 1.10.3.7 below.)

1.6 About this document

1.6.1 Use the Tutorial as an introduction

This *TDS User's Guide* includes a Tutorial that illustrates how to add TDS to an existing development project and to use its facilities in debugging and (optionally) testing the function members in that project. It assumes that you are using Microsoft Visual Studio[10] to do your development work. However, I expect that one could achieve similar results using a bare-bones text editor and the command-line C# compiler, along with the .NET platform and a suitable operating system, as all of the needed infrastructure is included in the files (see section 1.7) in TdsSource.zip . (Actually doing so is left as an exercise for the reader. ☺ I suggest using Visual Studio, since a major benefit of using TDS is to save time and effort, a benefit that might be mostly lost without the help of VS.)

1.6.2 Don't try to read this document like a novel

Incidentally, you may have noticed that this *TDS User's Guide* is somewhat lengthy. (Sorry.) I suggest that you not try to read all of it in sequence, like a novel, unless you are perhaps having trouble sleeping. Although I recommend reading and following the Tutorial (in section 4) that way, so that you can become familiar with what TDS can do for you, even there you are welcome to skip over parts that you do not expect to use. I have tried to put the topics into a reasonably straightforward sequence, so I hope you can just read it without doing a lot of flipping back & forth, but I've also included a Subject Index identifying some topics that may be of interest.

1.6.3 TDS is written in C#

Since the TDS methods[11] are written in C#, it may help to have a copy of the *C# Language Specification* available. When you have installed VS onto your computer, a copy of the C# specification is likely to be a part of that installation. (For example, it might be located at "C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC#\Specifications\1033\CSharp Language Specification.docx" .) The C# specification is organized as a reference book, but it also contains numerous examples and tutorial information. Technical terms used in this *TDS User's Guide*, such as "type" and "indexer", are defined and described there.

1.6.4 Extra details are intentionally included in this manual

Some of the material in this *TDS User's Guide* may seem a bit elementary to you; I tried to aim this at someone who already uses C# but might be interested in looking at a different, and I think faster and easier, way to create test methods to help maintain a collection of working code.

So… I hope you're not bored by repetitious or too-obvious material; perhaps the Table of Contents can guide you to skip past those parts.

For example, if you have by now read as much as you want of this introductory material, which mostly presents reasons for using TDS, to help you decide if TDS is worth trying out, you can skip directly to the Tutorial in section 4 below.

---

[10] For brevity, "Visual Studio" is often abbreviated in this *TDS User's Guide* as "VS", though that's not its real name.
[11] The TDS methods are written in C#, but the working code may be written in some other language, as mentioned in section 1.2.

Incidentally, I hope that not much of this *TDS User's Guide* is actually erroneous! If you find anything in here that is false or misleading, please post a comment on the http://tds.codeplex.com Web site, so that others may be warned, until I have had time to correct the mistake.

1.7 Packing list

This *TDS User's Guide* includes instructions and examples for using the Test-Driven Scaffolding (TDS) system to construct basic unit tests for function members of C# types in a project.

The compressed file TdsSource.zip contains the following files that accompany the *TDS User's Guide*:

TDS.cs	Source code for a stand-alone basic unit-test system, plus example TDS test methods; this is used (with **`TDS_Ex01.cs`**) to build an executable application assembly (**`TDS.exe`**).
TDS_Ex01.cs	Example of a file containing additional TDS test methods.
Program.cs	Example program to be tested, source code for an executable application assembly (**`ConsoleApp1.exe`**).
Class1.cs	Example program to be tested, source code for a separate namespace (**`NewCodeNamespace`**) in the same assembly.
TestMethodSnippet.snippet	Visual Studio code snippets to generate templates for new TDS test methods.
CmdTds.bat	Microsoft Windows® Command Prompt script to run TDS tests.
PsTds.ps1	Microsoft Windows® PowerShell script to run TDS tests.
Sentence.xsd	XML schema used in example TDS test methods.

These files are intended to be used with the Microsoft .NET platform and Microsoft Visual Studio in a solution's "TDS" project to generate an application (**`TDS.exe`** file) that may be called

- from a command line or Windows® Explorer during debugging or unit testing, or
- by a unit-test system such as NUnit, Microsoft Visual Studio Test, or the built-in TDS platform.

1.8 Why is this called "Test-Driven Scaffolding"?

1.8.1 Test-Driven Development (TDD) is a well-known development methodology

The "Test-Driven" part of the TDS name is a reference to Test-Driven Development (see http://en.wikipedia.org/wiki/Test-driven_development), which involves the following steps:

- You develop a specification that defines a desired improvement or new functionality in a function member (a method or property, for example), identifying expected or required results that the new or changed function member will produce based on defined initial conditions, such as its parameters or the field values accessible to it.

- If no version of the function member yet exists, you can write a "stub" (= stubby subprogram, containing some comments and pseudo-code) of the working-code function member that you will later develop, a non-working, mostly empty version that might define the name and parameters, but that returns without actually doing anything useful, or that may not even compile.
- You write a driver (an initially failing test procedure) to run the function member or stub. This driver sets initial conditions by, for example, assigning values to parameters passed to the function member or to fields visible to it. The driver is intended to invoke the function member, which should eventually generate observable results, such as changes to variables (fields or parameters, for example) external to it. The function member could have other effects, such as sending a message, but to make an effect of this type testable, the TDD driver would need to have some means of monitoring such a message. The driver can then compare the observable results produced by the function member with the values that the specification requires it to produce, based on the given initial conditions.
- This TDD driver is expected to be complete before any production code is written and should be prepared to call the new/changed function member with a great enough variety of initial conditions that the results will show that all of the specification's requirements are met.
- You modify the function member into working code[12], which is invoked by the driver (the TDD test method) and returns results to the driver to be verified.
- You continue modifying the function member and testing it with the TDD test method until the function member returns results matching the expected results for all the given sets of input data, which we hope were comprehensive enough to show that it satisfies all the requirements of its specification.
- Whenever the specification changes, you first modify the TDD test method to account for the changes, and only after that do you revise its tested function member until this function member passes the updated TDD test method.
- If you refactor the code, for example to improve its performance, and its specification has *not* changed, re-running its TDD tests can help you determine if an unwanted functional change has accidentally occurred; this should be indicated by a failing TDD test. Therefore, I suggest retaining the test methods for as long as their corresponding function members are being maintained.

1.8.1.1 TDS is similar to TDD in some ways.

- Function code under development is invoked by a driver that is likely located in a separate namespace.
- The driver code is not intended to become part of the production system.
- The driver code may serve as (partial) documentation for the code under development, specifying in detail the desired behavior of that new (or newly revised) function member.
- The driver may continue to evolve (e.g., by adding test case data or by adding tests) as work proceeds on the code being developed.
- The test methods act as safeguards against failing to notice that accidental functional changes have occurred in the working code. They are easiest (and least costly) to correct if detected early.

[12] In this *TDS User's Guide*, "working code" refers to the executable code that the TDS methods can invoke.

1.8.1.2 TDS differs from TDD in some ways.

- Using TDS, you do not need to specify, in advance, exactly what the function code under development will do – only what data you will provide to it.
- You can use the TDS code snippet, and/or the examples of TDS driver code provided here, as a basis for your own drivers, instead of starting from scratch.[13]
- You can apply TDS to existing (legacy) code that you intend to modify.
- The function member under development and its corresponding TDS test methods may be changed concurrently. In contrast, under TDD, the test methods must be updated first, before the working code is updated, to match the changed requirements.

1.8.2 Scaffolding helps with construction.

The "scaffolding" part of the TDS name refers to its intended use as something that you can quickly build to assist in developing the working code, similar to the scaffolding used in constructing a building; this scaffolding can be removed without a trace once the building is erected.

1.8.3 Concurrent development

When you're ready, you may use the TDS driver as a test procedure (by adding **`Assert`** calls), but unlike in TDD you do not need to finish doing so before you start writing your working code. You can trace execution of the working code as it operates on the data supplied by the TDS method.

Allowing the TDS driver to evolve as the function member that it invokes is being developed allows the developer to take advantage of ongoing developments in technology. For example, suppose that a currently unfinished and unpublished method, **`A()`**, depends on existing method **`B()`**. While **`A()`** and its TDS method are being developed, suppose that a new release of **`B()`** is published, including some new capability that **`A()`** could use to enhance its own functionality. Under TDS, method **`A()`** and its corresponding TDS method can both be updated to take advantage of the change in **`B()`**. Similarly, if **`A()`** was not originally intended to call **`B()`**, but during development it becomes apparent that doing so would be useful, that is easy to do using TDS — the code in **`A()`** and the code in its TDS method can be updated together to account for this new use of **`B()`**.

1.8.4 Which is better?

I claim that either TDD or TDS can be useful. I offer TDS as a relaxed version of TDD, allowing some flexibility in the specifications, that can be helpful if nobody knows beforehand exactly what those specifications really are or should be. I claim that TDS may be able to support a team's nimble utilization of new technology better than a rigid adherence[14] to TDD rules can.

Of course, anyone who uses TDD is welcome to use the TDS code to set up TDD tests. If you do that, you will need to alter the steps shown below by defining all the tests (the **`Assert`** statements in your test methods, for example as illustrated in section 5.1.6.2.3) before creating the references (the "**`actual =`** ..." statements in these examples) to the function member under development. It's also possible to steer a middle course, beginning with a few **`Assert`** statements, developing some function-member code, and adding other **`Assert`**

[13] This may be a bit unfair to TDD; you may perhaps have a TDD environment that provides a similar infrastructure. The point here is that being able to use a convenient, standard environment, such as the TDS code snippet, for creating unit-test methods is a major benefit of using TDS.
[14] It's possible that not everyone who uses TDD is rigidly dogmatic about using it; perhaps formally adopting some version of TDS would allow developers in those environments to more easily avoid breaking methodology rules.

statements as the need for them becomes apparent. However, the discussion in this *TDS User's Guide* assumes that much of the testing is defined, and takes place, somewhat later than it should in a strict TDD environment.

1.9 Using the code and this document

This code is distributed exclusively as source code, rather than as a Wizard, as a Text Templating (*.T4) file, or in some other automated form, in order to allow you the maximum possible flexibility in incorporating it into coding projects. Starting with unadorned source code also helps you to verify that no malware is included, as you can visually inspect all of it.

Since I expect that you are already editing code while building and maintaining your working code, I felt that it would not impose a major additional hardship to ask you to also edit the corresponding TDS method code in the examples, as you explore its interactions with the working code. Ideally, for each of the function members that you are maintaining, you should have a corresponding test method (possibly more than one) that needs to be kept up to date with changes to that function member. Using TDS should make it easy to add such test methods to your VS Solution as you add VS Projects and function members to it.

The example source files used in the tutorial instructions contain function members[15] that are already written and working, so we don't spend much time developing new code. The development process illustrated in the Tutorial (section 4) illustrates more of a maintenance or debugging process (updating or refactoring existing working code to correct errors) than a development process (adding code to create new functionality). To make the examples more representative of the intended use of TDS, they do include a few intentional, simulated bugs or mistakes so that you can see (as we do in section 4.6) the results of correcting them. As shown below (in section 5.1, for example), you may create a TDS method to initially provide data to drive a method stub that you are just beginning to write; the TDS method and the new function member will ideally grow alongside each other. Others of the examples provided in this *TDS User's Guide* show a much later stage; the function members used in the examples are essentially complete, and they are tested by essentially complete TDS test methods.

In case some of the material seems to stray from strictly addressing adding TDS methods to your projects, well, yes it does. You may stop after completing the Tutorial in section 4. Most of the part of this discussion that I think is most relevant to testing is in the care and feeding of the **`testValues[]`** objects and the **`Assert`** statements, and this *TDS User's Guide* includes many examples of both. I try to mention alternate ways of handling some of these examples, to help you decide which (if any) of them you would like to use. There is also some background material (for example, the mathematical analysis in section 5.2.4, or the advice concerning XML comments in lots of places) that, if it doesn't interest you, I try to alert you to skip over. But, as I argue in various spots, I think it all has some bearing on attempting to produce trouble-free software, and since your copy of the *TDS User's Guide* is probably printed on virtual paper, I hope this extra stuff won't make it too darn heavy to carry. Good luck.

1.10 Takeaways

The code snippet file TestMethodSnippet.snippet[16] provides the means to insert a TDS test method template into your code, and most of the examples are based on this. Also, the example TDS methods provided in TDS.cs and TDS_Ex01.cs may be used as templates for your own TDS methods, customizable to your development project's needs.

---

[15] In the examples shown in this *TDS User's Guide*, most of the example function members are methods.
[16] This is in the TdsSource.zip file; see section 1.7, "Packing list".

The TDS infrastructure (located within file TDS.cs) provides unit-test-like feedback on the status of the TDS methods already defined and on the function members they test. (You may instead run TDS methods using other test platforms, as we demonstrate in section 4.5.)

The TDS test report identifies any mismatches between the set of TDS methods that you select to be run by the TDS system, and the set of test methods (having **`[TestMethod]`** Attributes) in your TDS Project that could be run by the NUnit (or VS, or TDS) unit-test system. It provides a mechanism for bypassing selected TDS methods during a test run, but, if you use it, a message listing the skipped TDS methods appears in the TDS test report.

The TDS infrastructure and the TDS methods jointly provide a mechanism for filtering the test data to allow only selected test cases to be run. Note that enabling such a filter generates a message in the test report stating that, because of the filter, some potentially failing tests may have been skipped, possibly giving a falsely rosy picture of the test run's results. This is a reminder to remove the filter when its purpose is accomplished.

1.10.1 Why should one use TDS?

1.10.2 Purpose of TDS

Why should one use TDS at all? TDS is intended to help take some of the repetitive nuisance out of setting up and running debugging harnesses or drivers, and later, if desired, unit tests. It is intended to give C# and .NET developers a code template for quickly building drivers for new or existing function members[17] of C# types (or of types in a similar procedural language). In this discussion, I shall usually use the name "working code" to refer to executable code in a function member that is under development, or that exists and is being modified to add functionality or to correct bugs. The name "working code" will serve as a concise way to distinguish code under development from the code in some TDS test methods that is intended to invoke it.

In contrast to the (relatively unconstrained) form that I assume working code has, the TDS code developed to invoke this working code has a more rigid, uniform structure[18]. It will always be a collection of methods belonging to the **`TDS.Test{}`** class, whose definitions may, if you wish, be distributed among several TDS source-code files (TDS.cs and others; see section 4.10).

Even after a function member is complete, tested, and apparently working correctly, you might be inclined to refactor it to (for example) make it easier to read, or more secure, or more accessible to a variety of users, but without changing its functioning in any undesired way. The test methods that were used while it was being developed can be used again to help verify that it is still working as expected. When you are satisfied that the refactoring caused no unwanted results, and since the test method is not part of the work that the working code is intended to do, you can again remove the test method from the development process until the next time it's needed.

1.10.3 Features of TDS

1.10.3.1 TDS method is usable as a driver and/or unit-test method

When a TDS method is newly added to a project, it can be used simply as a debugging harness or driver, a source of data for tracing through the working code it belongs to, to assist with debugging. New sets of test-

---

[17] In this *TDS User's Guide*, "function member" refers to a method, property, event, indexer, operator, constructor, or destructor belonging to some C# type. The working code to be exercised, however, may be developed using C#, VB, or some other language, so long as it can be called from a C# Project. See section 5.4 for a VB example.
[18] A list of these uniform contents may be found in section 1.10.3.2.

case values can be added (as illustrated in section 4.8.3.1 of the Tutorial) in a systematic way that makes them easy to organize, and easy to locate[19] and read after they are specified.

After the working code is complete enough to generate observable results, its TDS method can, if you wish, be used as a test method, providing a standardized way of examining the results (comparing values generated by the working code with the expected values) and summarizing the differences in a test report. Even after the TDS method has been modified to be a real test method, it can continue to be used as a driver for the working code, to assist with tracing and debugging.

Please note, however, that, while TDS can perform some basic unit-test functions, it is not able to perform some of the other functions of a comprehensive unit-test system, such as displaying a graphic summary of test results, performing load tests, or identifying and exercising race conditions. TDS is designed to make adding basic debugging and testing support available with a minimum of effort, and the result of this effort (the new TDS method[20]) is easily transferred to another platform whenever you wish to do so.

To assist in such a transition, the TDS test reports are presented in a format similar to the output from NUnit or VS Test, so comparing results from these platforms with the contents of TDS test reports should be straightforward. The information in the test reports generated using TDS methods with any of these three platforms is close to being identical regardless of which platform you use. (See section 4.5 for examples of these results.)

1.10.3.2 Uniform structure of TDS methods makes them easy to navigate

If you use TDS methods to exercise or test working code, it should be easy to give these methods a common structure based on that of the TDS test-method template that you use. For example, if you use the **`TdsTest`** code snippet (as we shall do in section 4.8.2.1) to define your TDS methods, then in each of these TDS methods,

- the method will have the Attribute "**`[TestMethod]`**"
- the method will be declared "**`public void`**"
- the method's name will suggest the name of the working code that it will call and will end with "Test"
- an optional test-case filter will appear at the beginning of the method body
- the input data and expected results will be located next, in an array of sets of test-case values called **`testValues[].`**
- invocation of the working code will follow those definitions
- tests to detect correctly thrown exceptions will follow the invocation code
- tests for exceptions that were not raised as expected follow those
- tests comparing actual results to expected results will appear near the end
- the final statement (until you remove it) will raise an exception indicating that this TDS method is not yet complete
- some Task List ("**`//TODO:`**") comments will appear (until you remove them) within the method body to identify code that needs to be customized to exercise the corresponding working code.

---

[19] Assuming all of your TDS methods are created using the same code snippet, these data are always located in the **`testValues[]`** array in a **`#region`** at the beginning of the method's code instead of being scattered throughout the method, so they should be easy to find.

[20] The process of constructing a new TDS method, once a TDS Project has been added to a VS Solution, is illustrated in section 4.8.2 in the Tutorial.

Just as using a consistent naming convention for your many test methods should make them easy to find, giving them a consistent structure such as this should make them easy to navigate, modify, and maintain, as you keep them consistent with changes in the working code.

Details of this uniform structure are under your control[21], but I suggest that if you change it, you should do the same for all of the TDS methods in your VS Solutions. Some of the examples in section 4.14 offer variations on this basic format, and if you find them suitable for your purposes, you are welcome to incorporate them into an updated version of the **`TdsTest`** code snippet, by editing your copy of the TestMethodSnippet.snippet file and re-importing it into VS (as described in section 4.4.4).

1.10.3.3 Test cases include an optional filtering mechanism

TDS test methods that contain more than one set of test data (in the **`testValues[]`** array of test cases) offer a standard means of filtering the test data sets to allow, for example, selecting just one of them to drill down to a specific part of the working code to allow tracing, or to set up an unusual condition to demonstrate that the working code generates the correct exception. Although this feature is of limited value for running tests of completed function members, it is intended to be helpful while debugging code, as illustrated in the examples (for example, in section 4.8.7 and, more extensively, in sections 5.3.17.4.3 and following).

1.10.3.4 Test case structure simplifies verifying coverage of input space

Within a TDS test method, the parameters or other data that you might use in calling your working code and tracing through it are organized into elements of the **`testValues[]`** array, one per test case. This is intended to make it easy to compare test-case data in order to, for example, verify that the working code can properly handle all of the input values that it might encounter. (As mentioned in section 1.10.3.2, this consistent structure helps make the TDS method code easy to read and understand. However, TDS does not provide any automatic mechanism to assess coverage of the domain of possible inputs.)

1.10.3.5 Drivers & testing code can be (somewhat) standardized

In code maintained by a team, the various TDS methods will, ideally, look similar enough to each other that any team member should be able to easily read, understand, and maintain[22] any of them. Since the TDS code snippet provided in file TestMethodSnippet.snippet (to be used as shown in section 4.8.2.1) is in the public domain, it may be freely modified to suit your, or your team's, specific needs. A TDS method might be used only as a driver for supplying debugging data to new function members in a standardized format. It could also, as illustrated in the present examples, be used unchanged[23] as a test method similar to one to be run via a unit-test facility (in these examples, NUnit or VS Test).

A separate unit-test facility (illustrated in section 4.5 of the Tutorial) is not strictly necessary for running tests — a TDS method, after it contains some testing code ("**`Assert`**" statements), can easily be used as a rudimentary form of a unit test, and, by default, the TDS infrastructure's test report provides summary information about tests passed or failed. Note that, even though I consider testing to be a vital part of development, a TDS method can also be useful merely as a driver, without ever using its testing functions.

---

[21] Better than that, since the code is in the public domain, *everything* in TDS is under your control.
[22] By "maintain", I mean that any changes in the working code can be reflected in its corresponding TDS test method(s), to keep them consistent with each other.
[23] The TDS method is upward-compatible to the alternate platforms, but not downward. For example, NUnit supports many overloads of the **`Assert`** statements that TDS does not support.

Of course, the details of different TDS methods will differ, depending on what kinds of function members they call, for example… but a uniform, predictable overall structure should make it easy to find the relevant parts of any TDS test method that need to be kept consistent with its corresponding working code.

1.10.3.6 TDS leaves the working code mostly untouched

Part of the purpose of using (somewhat) standardized TDS code in your projects instead of writing standalone driver code in an ad-hoc manner, is that you can leave your working code largely untouched. In the examples that follow, we do sometimes add the **`public`** field **`isInitialized`** to classes being developed, but only when no other examples of accessible **`static`** fields exist within the class. In some examples (as in section 4.8.8), we temporarily change the accessibility of some objects to make them visible for testing purposes.

1.10.3.7 TDS is easily removable

Except to support unit testing, a TDS method is not needed after the working code developed with its help is complete. The working code's TDS methods may, like the scaffolding on a building, be removed at that time, leaving little or no evidence of their having been used. Since the TDS code is located in a separate namespace and in a separate set of files, it can easily be kept separate enough to be deleted without harming your code.

1.10.3.8 TDS methods can serve as design documentation

Even after removing the TDS methods, I suggest keeping them available somewhere, in case the working code later needs to be changed, to help verify that its existing functionality has not been adversely affected by those later changes.

After the working code is no longer being actively maintained, its TDS methods can still serve as a form of detailed documentation. This could be helpful if the working code later needs to be updated or adapted to a new use.

As an example of such documentation, the TDS method might contain code revealing the exact set of values that some input variable could assume without causing failure of the function member, and the precise characterization of this set might be omitted from the user manual as being a detail of only minor importance to a user, but possibly valuable to a programmer who must maintain the code. Examining code in the TDS method could help another developer[24] understand undocumented details of how the working code was intended to be used.

1.10.3.9 Limitations in TDS

Despite what I think are substantial benefits to be realized by using TDS during software development, in some situations it doesn't help much and could become something of a nuisance. For example, I use it with working code that does calculations and returns results, or that can query a database. I do not expect it to be of much use in debugging or testing a Web page or a Form that supports extensive user interaction. See section 5.1.5.3.2 for a discussion. If it's not apparently helpful, don't use TDS; instead, use it only where it can be useful to you.

1.10.4 Steps for customizing/setting up

A bit of work is involved to customize the code in a new TDS method. Each new TDS method must, at the least, be given a name that is unique within the TDS namespace, and it must contain a reference to its working-code function member and, usually, at least one set of data values. These data are usually included in the first test case (the contents of **`testValues[0]`**), but some of them could instead be specified as local variables in the

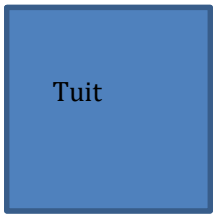
[24] Beware — this "other developer" might possibly be you, several months and several projects later.

TDS method. I occasionally do that to give names to fields that otherwise would be passed to the working code as constants.

1.10.5 "I'll take care of that pretty soon."

I talk at times with developers who do not dispute the desirability of using a Test-Driven Development (TDD) or similar unit-testing protocol, but who find that though it sounds great in theory, in practice it's not always so easy. Deadlines loom. Maybe someone is waiting for the method I'm working on to be available, or if not a fully polished version, at least a version that works well enough that it can begin to be used to check out other code that depends on it.

So… there's a hard-to-resist temptation to defer some of the testing until after the prototype version of the method is finished. (Or maybe to defer all of the testing — the method seems to be working for now and can be debugged later if something in it breaks.)



A square Tuit.

The kind I may need to get.

And… that being done, there's a temptation to work on that *other* method that someone is really eager to begin using, instead of doing this pro-forma test that may not be especially critical right now. (That first method is probably already in use by now, and if a bug shows up later, maybe I, or somebody else, can take care of it then.)

So… before I know it I've delivered a dozen new methods and not a single test. Maybe the tests never get written. (This was not how I'd planned to do it, six months earlier!)

However, once I've added a TDS Project to my VS Solution, much of the work needed to add new test methods is done. A few keystrokes should suffice to define a rudimentary test method that will at least demonstrate that the "happy path "[25] through the working code will pass. Once the TDS method is defined and active, only minimal additional work needs to be done to add sutiable test cases to it to cover other paths as needed. Also, pending that additional work, the TDS method is now listed on test reports as being unfinished, serving as a reminder to attend to it.

[25] See Page, Johnson, & Rollison, How We Test Software At Microsoft (2009), Microsoft Press, p. 69, for a description of testing the happy path — the default path through the working code. If even this simplest of all possible paths fails, it becomes embarrassingly apparent that somebody didn't even try to verify that the app was working.

2 Introduction to the Tutorial and example projects

2.1 Short-circuit the long explanation

Perhaps, like me, you approach a new toy with a view toward playing with it, without taking the time to patiently read the instructions. Although I have put significant effort into making the present instructions and examples clear and complete, I fully sympathize with such a feeling. Therefore, the Tutorial in section 4 gives a detailed introduction to the features included in TDS, which should provide everything you need to be able to use it. In the Tutorial, you will just copy some code into your Solution (simulating the writing of real code) and run the resulting program. You can, if you wish, play with this example code to see how it behaves under the control of TDS.

Following the Tutorial, some more detailed examples in section 5 illustrate using TDS in simplified (but I hope somewhat realistic) situations that build on what was done using the Tutorial.

2.1.1 Really quick overview

If you're a bit impatient and like to dive right into new stuff without fiddling around with details, just refer to section 1.1 for my idea of a lightning-fast summary[26], or to the somewhat more sedate version of it in section 3. These summaries briefly identify what needs to be done, but I suggest that you actually use them only after you've gone through the Tutorial (section 4) at least once, and then use the steps in section 3 to set up TDS for use with other Solutions, when you are more familiar with TDS's features. The list in section 1.1 could then be considered a checklist for setting up TDS.

If you've seen enough explanatory stuff, then I invite you to jump directly to the Tutorial in section 4 to start building. Otherwise, you may continue with section 2.2 for a meandering look at how I think TDS may help save time and effort in constructing and debugging working code.

2.2 What to expect

2.2.1 Results of running the Tutorial

Notice that I did just now give you a chance to skip the following verbiage. Since I see you're still with me, the following sections and those leading to the Tutorial (in section 4) give a rationale for using TDS as part of a development effort. If you follow the steps shown in the Tutorial, you should have a working (though rudimentary) Visual Studio Solution containing example function members with associated TDS test procedures. Although the example TDS methods provided in TDS.cs and the code generated by the **`TdsTest`** code snippet (see section 4.4.4) won't do anything useful by themselves, they are intended to be used as templates to help make developing your own code easier, by providing a systematic, simple framework for sending data to a function member that you are developing. At a suitable point in the development (you determine when that will be), you can, if you wish, begin to use the TDS code as a test method, to unit-test your code. From then on, you may continue to run that TDS method from within Visual Studio, or by clicking a Windows® Explorer icon, or from a Windows® command prompt. All of these means of running TDS provide test reports; for details, see section 4.11.2. You can instead run tests using NUnit or Visual Studio Test; doing so will give you a concise report on large numbers of tests, quickly identifying any that do not run as expected

[26] The summary in section 1.1 glosses over some details that are at least mentioned in the more detailed instructions in the Overview in section 3 or the Tutorial in section 4, including hints about how to do the tasks mentioned. Section 1.1 may be all you need if you have either already used TDS or you don't mind finding the details by reading the C# source code.

and that therefore need attention. Using a platform like NUnit can save time, for example, if you have several hundred such tests to track.

2.2.2 Using the Tutorial as a foundation

To avoid repetition in subsequent examples (section), those examples build onto the foundation created by the Tutorial without duplicating the steps needed to construct it, so if you wish to actually build the others you will need to begin by completing at least part of the Tutorial. The other examples assume that you have the TDS code snippet installed and that the TDS infrastructure is working. You will have reached that point by the end of section 4.4 in the Tutorial. In a real TDS-based project of your own, you could then follow the same steps to add TDS methods for any number of function members in your VS Solution.

2.2.3 Apology[27]

Although I have attempted to keep later examples somewhat independent of earlier ones (except that they all depend on the Tutorial), the illustrations in this *TDS User's Guide* reflect the assumption that you have completed all of them in order. If you skip some of the steps, you can probably complete the later example, but you may encounter error conditions while doing so. I expect that any such errors should be easy to handle, but if not, return to the Tutorial described in section 4 and build from there.

It's likely that, at times, the instructions in the Tutorial or examples will seem repetitious or unnecessary. Sometimes, when trying to follow instructions or some involved argument (such as the proof of a theorem or a detailed political discussion), it is frustrating to hit an assertion whose basis is not obvious, and no one is available to answer questions. ("Where did that statement come from??") I would like to avoid boring you with needless details, but I *really* want to avoid leaving out some detail that you, or someone else, might think is an essential part of the instructions that has been annoyingly omitted, thereby making it difficult to continue reading. (For that, I really would owe you an apology, in the sense that I would regret making it unclear.) Since I assume that, as you read this *TDS User's Guide*, there is nobody to ask for help if something about these instructions is confusing or unclear, I have tried to err slightly on the side of giving a bit more detail than the minimum needed. If you feel that there's too much detail in one of the steps, you may be able to skip to the next section without losing the thread of the discussion.

[27] This is an "apology" in the sense of an "explanation" or "justification", rather than a statement that I feel sorry about something.

2.3 Usage notes

To simplify reading and using the examples in this *TDS User's Guide*, it includes some navigational aids that I hope can save some time and effort.

Often, when I read a book like this, I first skim over it, to try to determine if it will be worth my time to actually follow all the directions. You are welcome to do that, if you wish; if so, you can find many of the intermediate results included in the text where they are described. However, if you think you might be able to use TDS in your own projects, I expect that you will have a better idea of how you can adapt TDS to your needs if you actually build some or all of these examples, and maybe play with them a bit, instead of only reading about them.

2.3.1 Essential steps

Throughout the examples in this *TDS User's Guide*, steps marked with "► " are suggested specific actions (as distinct from commentary, explanations, or summaries that may safely be ignored). If followed in order, performing only these marked steps should result in a working product. Be careful, however; marked actions are sometimes explained or modified in the following paragraph, so I suggest not blindly following them.

2.3.2 Typography

Most of the descriptive text in this *TDS User's Guide* is set in Times New Roman font. In contrast, C# code examples are displayed using **`Courier`** font and may appear either in line, **`like this`**, or in a separate, indented line,

```
    like this,
```

or as a set of indented paragraphs.

```
// If specified as a sequence of paragraphs, like this,
//   the code might extend over page breaks.
```

Output examples, the expected results of running the exercises, may also be displayed in this fashion.

Names of keyboard keys, such as <enter> or <space>, are enclosed in angle brackets. Press the specified key once for each of these. For the mode keys <shift>, <control>, and <alt>, hold the key down while you press the following key(s)[28].

When a block of code is in separate paragraphs, it may be color coded according to syntax (comments in green, C# key words in dark blue, etc.) similarly to how VS color codes it.

The purpose of using this alternate font is to make it easy to identify code that you may wish to copy or imitate as you follow the examples provided in section 4 below.

2.3.3 Copying code

Short code sequences (one or two lines) presented as examples in this *TDS User's Guide* may be quickest or easiest to enter by using the keyboard, and doing so is a better simulation of actual work than copying material from this *TDS User's Guide* would be. However, you will probably want to copy and paste longer code sequences by using the Windows® Clipboard. If you are using Adobe® Reader® to view this *TDS User's Guide*, you may be able to use the "Selection Tool" to activate the cursor that looks like either an arrow or an I-beam.

---

[28] In some apps, <alt> may be pressed and released before pressing other keys, similarly to the action of <caps lock>. This *TDS User's Guide* does not require it to be used in that way — in this document, the sequence <alt-space> means to hold <alt> down, press and release <space>, then release <alt>.5-7/[

It becomes an I-beam, I, when it's positioned over selectable text. Use that to select the code, then copy the code to the Clipboard, then paste the copied code into an editing window in VS.

Some of the example code extends over page breaks (which I have tried to avoid, but some of it is too long to fit onto one page). You may select and copy text that extends over page breaks to the Clipboard, but doing so may copy the page headers as well as the selected text, so you may have to edit out page header and footer lines after pasting. You may paste the copied code into an editing window in VS. (If VS asks you if you want to save the file as Unicode, click "No" — these projects do not need Unicode.)

In the example code, white space (blank lines or indentation) at the beginning of lines is usually included merely for legibility, but occasionally, especially in literal strings preceded by "`@`", it is significant. Code copied from Adobe® Acrobat Reader® is likely to be missing its white space at the beginning of each line, so I suggest that, after pasting into VS, you format the code using "Edit, Advanced, Format Document", then type the appropriate spaces, if you want the resulting output to look like the results shown in this *TDS User's Guide*. If you don't do this, the program should still compile and run, but the output might be formatted differently, and the code might be more difficult to read.

You might discover some wrapped lines while trying to compile or compare the copied example code. Please be aware that some lines that may appear to be improperly wrapped are actually parts of string literals beginning with "`@`", in which the line breaks are significant, so you should avoid unwrapping those. If you have trouble using the code, especially string literals, it may help to format your copy of it look like the code shown in this *TDS User's Guide*.

2.3.4 Navigational aids in this document

2.3.4.1 Traditional (text-based, inactive) mechanisms

The *TDS User's Guide* is organized like a book, with a table of contents and subject index, to help with navigating a printed copy or when using a reader that does not offer much automated support.

2.3.4.1.1 TABLE OF CONTENTS ("TOC")

This table is somewhat abbreviated, going down only three levels, as I expect you will use it only at first, to get an idea of the organization of the document.

The Table of Contents (TOC) can help in correlating page numbers (shown as page numbers in the page footers in the main document, and in the Subject Index) with the section numbers (shown in the Navigation Panes qof some readers). The TOC page numbers do not account for those occupied by the introductory material, such as the table of contents itself, and are thus about eight lower than those displayed by page-reading software such as Adobe® Acrobat Reader®.

2.3.4.1.2 GLOSSARY

Some terms that have specialized uses in this *TDS User's Guide* are listed in the Glossary in section 7, along with definitions or descriptions, and references to section numbers in text where they are introduced or defined.

2.3.4.1.3 SUBJECT INDEX

As you're probably well aware (maybe it's the second page you looked at), this Subject Index is in section 8, and I tried to include in it everything that I spent more than a sentence describing, including the "**Fibonacci Bunnies**". The "*See*" cross references are my attempt to avoid omitting stuff because I chose the wrong name; you might prefer different names.

The Index entry does not always match exactly the name on the cited section; the Index entry is often shorter, to avoid cluttering the Index. The names are usually somewhat similar, though.

You may use the cited page numbers in conjunction with the "Pages" panel, or use the Table of Contents to locate the corresponding section numbers. Sorry, the page numbers in the Index are not clickable links.

2.3.4.1.4 FOOTNOTES

Some explanatory material that may be safely skipped is shown as a footnote[29], which is displayed at the bottom of the current page.

2.3.4.2 Active content

Some active navigational aids are included in this *TDS User's Guide*, to take advantage of navigation features offered by many document-reading programs.

2.3.4.2.1 "FIND" FUNCTION

You know what this does, and there's nothing special here, but if the Index and Table of Contents aren't sufficient, you may be able to locate what you want by using menu item "Edit, Find" (usually, "<control>F"). This may call for you to do a bit of guesswork involving using synonyms for whatever it is you're really trying to locate.

2.3.4.2.2 LINKS IN TEXT

References in the text to sections other than the one you are reading are usually introduced by the word "section". (I would have called them "paragraphs" except that most of them include several paragraphs of the usual type.) For example, a reference to the section containing this sentence might appear as "see section 2.3.4.2.2 above".

If you are using Adobe® Acrobat Reader® to view this *TDS User's Guide*, you may follow any such link by moving the cursor to hover over it – so that the cursor changes to a hand-tool cursor[30] – and left-clicking on the link.

To return to the previous location after following a link, in either Microsoft® Office® Word or Adobe® Acrobat Reader®, use <alt><left arrow>.

Sorry, in Adobe® Acrobat Reader® the page numbers in the TOC seem not to be navigable that way, but you can find the listed pages by opening the "Pages" Navigation Panel (using menu "View, Navigation Panels, Pages"), and adding about eight to the number shown in Contents to account for the introductory pages.

The page numbers appearing in the page footers throughout the document correctly match the page numbers listed in the Table of Contents and the Subject Index.

2.3.4.2.3 FOOTNOTES

Text in a footnote, besides appearing at the bottom of the page, may in some readers appear in a pop-up text box when the mouse pointer hovers over the footnote's reference number in text.

2.3.4.2.4 BOOKMARKS NAVIGATION PANEL

You may also navigate via the links in the "Bookmarks" Navigation Panel in Adobe® Reader®. These correspond to the section headings, and all of the section headings are listed here; they are not limited to three levels, as the Table of Contents is. The Bookmarks panel may be navigated via the arrow keys or by using the mouse.

---

[29] Such as this example footnote.

[30] The Adobe® Acrobat Reader® hand-tool cursor looks like a hand,  , with only the index finger extended.

2.3.4.2.5 PAGES NAVIGATION PANEL

You may find the thumbnail versions of the pages shown in the "Page Thumbnails" Navigation Panel in Adobe® Reader® to be helpful, especially if you know approximately where in the document you're trying to go, and what the page looks like. Also, if you're willing to do a bit of arithmetic to account for the Table of Contents, you can use this to quickly find a page that is cited in the Subject Index at the end of the document.

2.3.4.3 Summary of active features

The active features are not consistently available in various tools I have used to read this *TDS User's Guide*. Although all of those shown here support "Edit, Find", not all of the other features listed here are included.

The readers listed (some, like Microsoft Office Word, may not be available free of charge) include these:

- Microsoft Office Word (see https://products.office.com/en-us/word), in Read-Mode View. A free-of-charge "Microsoft Word Viewer" may be downloaded from https://support.microsoft.com/en-us and used to read the document.
- WPS Office (see https://www.wps.com/office-free), in Print Layout View
- Adobe Acrobat DC (see https://acrobat.adobe.com), when reading a PDF version

Feature	Microsoft Office Word	WPS Office Writer	Adobe Acrobat DC
Link to section number in text and in Glossary	Yes; when hovering, the hand-tool cursor changes to a hand, , with only the index finger extended.	Yes; pop-up box appears with advice to hold down <control> and click.	Yes; when hovering, the hand-tool cursor changes to a hand, , with only the index finger extended.
Pop-up when hovering over footnote reference	Yes	Yes	No
Link to page number in Table of Contents	Yes (but page numbers are not displayed on the pages). Section numbers are not clickable.	No; TOC is displayed as text	No; TOC is displayed as text
Return to previous location (<alt><left arrow>) after following a link	Yes	No (possibly available via other keystrokes)	Yes
Navigation Pane (showing all section headings, and they are active links)	Yes, using View, Navigation Pane	Yes, using View, Document Map	Yes, using Bookmarks
Thumbnails Pane	No; may be simulated in Print Layout View by selecting Zoom, Multiple Pages, and zooming out	Yes, using View, Navigation Pane.	Yes, using Page Thumbnails

3 Overview

As mentioned previously, TDS can help with tracing and debugging working code (section 1.5), with testing it (section 1.10.3.1), and with documenting it (section 1.10.3.8). This section presents a summary of the steps needed to add TDS to a VS Solution containing some existing code. In contrast, the Tutorial in section 4 will illustrate tracing and testing working code in more detail, using a new Visual Studio ("VS") Solution (one having no existing code, and which we shall populate from scratch, along with the corresponding TDS methods). So if you do not have an existing Project that you wish to exercise using TDS, or if TDS is new to you, please go to the Tutorial in section 4 now.

Detailed instructions for installing TDS and exercising many of its features will be presented in the Tutorial (section 4). After you are familiar with the features of TDS, the brief steps beginning in section 3.1 should serve as reminders to enable you to use TDS with an existing[31] Visual Studio Solution. (An even briefer version was presented in section 1.1.) The instructions in this Overview assume that you have unpacked the contents of TdsSource.zip and imported into Visual Studio the code snippet file TestMethodSnippet.snippet, which needs to be done only once. (Importing it is discussed in the Tutorial, section 4.4.4.)

Having created one or more TDS test methods, you may wish to run them using either NUnit or the Microsoft Visual Studio Unit-Test platform, and instructions for doing both are included in the Tutorial (section 4.5). However, the instructions in most of the Tutorial assume that you will use the TDS platform for testing, since part of its purpose is to show how the features of TDS work.

In these abbreviated instructions, references to the corresponding (and usually more detailed) instructions in the Tutorial are provided.

3.1 Add a TDS Project to your Solution

► Unpack the contents of the accompanying file TdsSource.zip to a convenient location. (See section 1.7 for a description of the contents of this file.)

► Add a new Visual C# Windows® Classic Desktop Console App (or ConsoleApplication) Project to an existing VS Solution and name it "TDS". (See Tutorial section 4.4.1.1.)

► In VS's Solution Explorer, delete the TDS Project's Program.cs component and add existing item TDS.cs (copied from the contents of TdsSource.zip). (See Tutorial section 4.3.5.)

3.2 Set references

► In the Solution Explorer, in the TDS Project, in References (or by using menu "Project, Add Reference"), set a reference to each namespace containing function members to be debugged or tested. (See Tutorial section 4.4.1.2.)

► Set the TDS Project as the Startup Project. (See Tutorial section 4.4.3.1.)

3.3 Edit TDS.cs

This file requires some editing to make it work properly; the places where this needs to be done are identified by "**`TODO:`**" task comments (listed in the VS Task List window).

[31] In contrast, the Tutorial assumes that you are creating a new VS Solution, not adding TDS to an existing Solution.

3.3.1 Link to the namespaces of testable types

► Use the task "**`TODO: Usings`** ..." to navigate to the "**`using`**" statements and substitute correct names for those listed. These are listed near the beginning of TDS.cs . (Same as Tutorial section 4.8.2.2.)

3.3.2 Specify which static variables need to be activated

► Use the task "**`TODO: InitializeClasses(), static variables`**" to navigate to the references to static variables in your function members, and correct or delete them. (Same as Tutorial section 4.4.1.3.)

3.3.3 Identify active TDS source-code files

► Use the task "**`TODO: TestMethodsSourceFiles"`** to navigate to the list of TDS source-code files and delete the line containing "**`TDS_Ex01.cs"`**. That file is listed here because it is used in the Tutorial, but, at least initially, you will define only a few TDS test methods and, if you wish, you can place all of them within the TDS.cs file, so you will have no need for TDS_Ex01.cs or any other TDS source files. (See Tutorial section 4.9 for instructions on adding such a file.)

3.3.4 Delete unused TDS method definition

► In Task "**`TODO: TestableConsoleMethodTest() example -- Delete the contents of the following #region if`**...", follow the suggestion and delete the contents of the **`#region`**.

We do not expect to need to use the example TDS method **`TestableConsoleMethodTest()`** in this exercise. (We also delete this example TDS method in the example in section 5.4, but we use it in the Tutorial.)

3.3.5 Identify active TDS test methods

► Use the task "**`TODO: TestMethodsToBeRun`** ..." to navigate to the list of active test methods, and delete or comment out the names of all of the test methods listed there**`.`** (See Tutorial section 4.8.2.5.)

3.4 Do a TDS "smoke test"

► Run the Solution; if an AssertFailedException or AssertInconclusiveExeption pop-up message appears at any time while running the instructions in this section, clear the "Break when this exception type is thrown" checkbox, close the pop-up, and resume running. (See Tutorial section 4.4.2.)

If all goes well, we see near the bottom of the Console window this message:

```
Passed: 1  Failed: 0  Inconclusive: 0
```

3.5 Add a TDS test method

► Into VS, import the TestMethodSnippet.snippet file from TdsSource.zip if it hasn't yet been imported. (See Tutorial section 4.4.4.)

► Near the end of TDS.cs, at the "**`TODO: New TDS methods may be placed here:`**" Task, generate a new TDS test method template by typing "TdsTest<tab><tab>" and specifying the name of an accessible function member in the existing Projects. (See Tutorial section 4.8.2.1.)

► Add the name of the TDS test method to the list in **`TestMethodsToBeRun`** . (See Tutorial section 4.8.2.5.)

► Edit the TDS test method, in Task[32] "TODO: xxxTest() -- Define inputs and expected outputs", to specify input values for its function member.

[32] In contrast, the Tutorial assumes that you are creating a new VS Solution, not adding TDS to an existing Solution.

The name of the new TDS method will appear in its Task comments, to make them easy to find.

► Edit the TDS test method, in Task "**`TODO: xxxTest() -- Use a suitable default value`**" to declare and initialize a variable to receive results (eventually — we're not ready to use it yet).

► Edit the TDS test method, in Task "**`TODO: xxxTest() -- Provide a suitable calling expression`**" to invoke the function member. (See Tutorial section 4.8.2.3.)

3.6 Run TDS for tracing and debugging

► Set a breakpoint in the working code's function member that is to be debugged.

► Run TDS, breaking at the breakpoint. (See Tutorial section 4.8.2.5.)

► Trace execution of the function member using the inputs supplied by the TDS test method. Use the VS debugging tools to assist in examining and exercising the function member's code. (See Tutorial section 4.8.2.7.)

► As needed, stop debugging (<shift><F5>) and make corrections. (See Tutorial section 4.8.2.7.)

► Add test cases as needed, for example to trace various parts of the function member. Filter test methods and test cases as needed to reduce clutter while debugging. (See Tutorial section 4.8.7.)

During tracing or debugging, we do not expect the function member to return control to the TDS method; the purpose here is merely to follow the operation of the working code to verify that it is performing as expected.

If you are using TDS only as an aid in debugging, and have no need to run any tests, skip to step 3.8.

3.7 Run TDS for testing (optional)

► Edit the TDS test method to have it compare expected results with actual ones. (See Tutorial sections 4.8.3.1 and 4.8.3.3.)

► Clear breakpoints and run TDS (use <F5>).

Here we expect the working code to return results that we can compare with the expected values that we specify for each test case.

► As "unhandled exception" messages appear for Assert exceptions, disable them. (See Tutorial section 4.4.2.)

► Examine the TDS test report or save it as a record of test results; correct errors listed in the report.

► Continue debugging if unexpected results appear. (See Tutorial section 4.8.2.)

3.8 Hide TDS when done

► When TDS is no longer needed, follow the steps in section 4.12 to make TDS less visible.

4 Tutorial

4.1 Learning objectives[33]

This Tutorial is intended to introduce you to all of the intended[34] features of TDS, but without going into extensive detail on some of those. It describes most of what needs to be done to use TDS for tracing and unit testing and how to do so, but it touches only incidentally on some of the reasons for doing some of these steps. Why those steps are included is discussed more fully in section 5, which presents additional examples of developing working code with the help of TDS. The examples there provide some additional details, but they are not essential to enjoying the main benefits of TDS.

Summaries of these steps may be found in sections 1.1 and 3, but those summaries assume that you are adding TDS to an existing VS Solution. Setting up a VS Solution containing nothing but a TDS Project (for use in creating a new Solution with TDS pre-installed) is shown in section 4.14.7.

In this Tutorial, we construct a new VS Solution containing an example Project and add a TDS Project to it, making changes to the code in both Projects to simulate a debugging process and to illustrate how these editing changes affect the test report and other output.

When you have completed this tutorial, I expect that you will have done the following:

- constructed a Visual Studio Solution that includes function members (mostly methods) of a class, simulating code that you might develop or modify. One of these will read from the keyboard and write to the Console window, while others return the results of calculations.
- added working TDS methods to the VS Solution to call the function members that are being developed (contained in the "working code")
- used the TDS code as a driver to supply inputs to a method to assist in tracing its execution
- converted the TDS methods into unit-test methods that generate a test report displaying the results of running the tests
- observed the effects on the test reports of various types of program bugs, including missing, unexpected, or incorrect exceptions
- specified alternate inputs, as test cases in a `testValues[]` array, to a method or other function member, for use in tracing its execution or verifying results
- added new properties to the test cases, for use as parameters or to specify expected results
- temporarily filtered the set of test cases used in running a test
- observed the use of both anonymous-object and named-object specifications of test cases
- (optionally) run TDS unit tests using NUnit and/or the VS unit-test platform
- run some developed (and supposedly debugged) function members independently of NUnit or Visual Studio

4.2 Tutorial road map

In this tutorial, we shall begin by creating a simple Microsoft Visual Studio (often abbreviated here as "VS") Solution containing a single VS Project that will serve as an example of code that we might wish to debug or

---

[33] Yes, I know that you're not a kindergartener. I have put "learning objectives" at various places in this document to give you a chance to decide, upon reading them, if it will likely be worth your time to read the material and do the exercises. If not, you can skip over the uninteresting material.

[34] It's possible that bugs in the TDS software could lead to some *un*intended features, but those are not covered here.

test. These instructions make use of the files included in the TdsSource.zip file that accompanies this *TDS User's Guide*. (See section 1.7 for a list of the file's contents.)

In real life, I normally add the TDS code to a VS Solution that is already under construction (one that is perhaps working to some degree, but is unfinished). I copy the TDS source files from TdsSource.zip to a convenient location in the Windows® file system, such as near the folder that contains the existing VS Solution, and then I add to the Solution a new Project, which I name "TDS". The TDS Project facilitates updating and debugging the code in the Solution's existing Projects. (The source-code files Class1.cs and Program.cs in TdsSource.zip are included for use only with this tutorial as examples of working code, not for use with existing Solutions.)

If you woud prefer to use some other name for what this TDS User's Guide calls Project "TDS", then I suggest that you choose some other (short) name for your testing Project, rename the files within the TdsSource*.zip file whose names currently begin with "TDS" to match that new name, and edit those files to replace all occurrences of the string "TDS" in them with the new name that you have chosen.

The TDS Project that we shall add to this example VS Solution will define the **`TDS{}`** class, which will contain example test methods to exercise code in the existing Project(s) and will report the results using the Console. In its early stages, a TDS method is merely a debugging harness that sets up suitable conditions, such as parameter values, for calling a function member (for example, an indexer, property, or method) of some type, and the testing/reporting features of the TDS method are not used. As work proceeds on the function member, we can update the TDS method by adding tests that help to provide evidence that the function member is performing as expected.

We shall illustrate running the example test methods using not only the TDS platform, but also (optionally) the NUnit and VS Test platforms, to demonstrate that the test methods are compatible with those as well, and you may find that you prefer one of these over the TDS platform for most of your work. Regardless of which unit-test platform you choose to use, using the common TDS template to construct most of your unit-test methods should make it easy to navigate among them and to maintain them.

Using the TDS platform, we shall modify code in the testable Project to observe the effects on the TDS test report of the changes we make. We shall construct and run a new TDS method to exercise a testable function member. We shall demonstrate some optional features of a TDS method, for example these:

- filtering the test cases in the TDS method to exercise a specific path in the working code or to suppress unneeded details in the test reports
- refactoring the test cases to use named objects to make the data in the test cases easier to read and maintain, such as by allowing some of the parameters to be optionally omitted (taking their default values)

Most of this tutorial uses Microsoft Visual Studio to run the TDS tests. This is a realistic mechanism for much of the debugging and testing for which TDS is intended, as debugging usually calls for editing the source code (here we use VS to do the editing) as bugs are discovered, and a failed test would similarly call for locating a faulty component and correcting it. However, after the code in a function member has become fairly stable, you may wish to run the TDS tests independently of VS, so we shall also illustrate alternate ways doing so, for example using the provided script files (runnable via Windows® Command Prompt or Windows® PowerShell).

As you work through the examples, please bear in mind that the TDS code may be freely altered to meet the specific needs of your work (no copyright permission is needed), and it includes numerous comments to help you make any needed changes.

The time estimates shown are merely my guess as to how long it might take to accomplish an exercise. You may decide to skip some steps, or you might spend extra time playing with some feature, and your choices could affect the time needed to finish.

4.3 Set up Visual Studio and TDS files [20 minutes[35]]

4.3.1 Intended environment

The TDS code is intended for use with Microsoft® Visual Studio® and (if you wish) with the NUnit unit-test platform. It may also be useful in some other contexts, such as the .NET command-line C# compiler, but it has been tested only in the environments detailed in this Tutorial (section 4) and the extended examples (section 5).

Rather than try to identify in detail which specific environments will allow you to use the TDS software, I suggest simply trying to run the examples in this Tutorial. It should become amply obvious if something essential is missing, and it should also be obvious what to do about that. You have the means (the C# source code, an editor, and a C# compiler) and permission (public-domain code) to correct any faults that you encounter and make TDS work for you.

In this Tutorial, the steps are described as if you are using a recent version of Microsoft Visual Studio (sometimes referred to here as "VS"). If you are using some other editor, it may be difficult, though still possible, to follow the procedures as shown; for example, instead of using a code snippet to create new TDS methods, you will need to make a copy of one of the example TDS methods, such as `TestableNoConsoleMethodTest()`, to be pasted into your code as a template whenever you need to create a new TDS method.

4.3.2 Check that Visual Studio ("VS") is installed

- Microsoft Visual Studio Community 2015 (or similar version)
  See https://www.visualstudio.com/products/vs-2015-product-editions; this is free of charge to individuals, as well as to some groups. The TDS files should work with some other recent versions of Visual Studio as well, but details in the instructions in this Tutorial may not always match the behavior of those other versions. (For example, Microsoft Visual C# 2010 Express does not offer a built-in unit-test platform, used in section 4.5.2, but it supports most of the rest of this Tutorial.)

4.3.3 Create an empty file folder for your VS Solution

► Use File Explorer to create an empty file folder to contain your working code and the associated TDS Project.

A new File Explorer (also called "Windows Explorer") window may be opened by right-clicking on the Desktop in Windows 10 and choosing "New, Folder" from the pop-up menu, or opening "Windows System" on the list of apps in the Taskbar and choosing "File Explorer".

For this Tutorial, I suggest using "Demo", a short version of "Demonstration", as the name of this folder, since I assume that you will have no need to keep it after finishing the Tutorial. You may then erase that folder and its contents with no ill effects. After that, you can use the summary in section 3 or (the even shorter summary) in section 1.1 to add a TDS Project to a VS Solution. Having done that, you can quickly add TDS methods to it to help with tracing or unit testing your working code.

---

[35] All completion times mentioned here are rough estimates, and this one assumes that the necessary software, such as Visual Studio, has already been installed.

4.3.4 Extract the contents of the TdsSource.zip file

All of the files in TdsSource.zip (downloaded from http://tds.codeplex.com) may be inspected using a text editor such as Microsoft Notepad. To verify that these files contain no viruses or other malware that might harm your computer, you may wish to visually examine their contents before using them.

For the purpose of this Tutorial, I suggest copying file TdsSource.zip to an empty folder, which in this Tutorial shall be called "Demo\", but any valid folder name will work — none of the files in the Tutorial depend on this name. Perhaps you normally build VS Solutions in a different way, but using a single file folder, as I suggest here, for all of the files used in this Tutorial will allow you to dispose of them easily when you have finished playing with them.

► In Windows® Explorer, having copied TdsSource.zip to the Demo\ folder, right-click on the Zip file's name and click on "Extract All…". Change the new folder name if you desire (but I'll refer to it here as "TdsSource\", so its path will be something like …\Demo\TdsSource\ ).

For this Tutorial, we shall also build our Visual Studio Solution in subfolders of Demo\ and shall add the source files to the Visual Studio Solution as we build it. The purpose of doing this is to keep the source files close to the Solution files to make them easy to find as we run the Tutorial (and to erase afterward), but there is no need to do this in general. Just put them in a place where you can easily find them.

4.3.5 Configure Visual Studio

In this Tutorial, we shall simulate adding TDS to an existing Visual Studio (abbreviated "VS" in this Tutorial) Solution by beginning with an example VS Solution, a simple one that includes only one VS Project, which creates an assembly that implements an application (`*.exe`). We shall build and run this Solution, to demonstrate that it works. Having a working (but supposedly unfinished) VS Solution, we shall then add a new VS Project, to be called "TDS", to it to facilitate updating and debugging the working code in the Solution's existing Projects, and eventually testing that working code.

► Open Microsoft Visual Studio.

The version used in this Tutorial is Microsoft Visual Studio Community 2017, but any recent version should support most of the features used here. (A notable exception might be the VS "Test" facility, which is not available in some older Express versions of VS.) If you plan to use Visual Studio Community 2017, install it using the workload ".NET desktop development", or modify its current installation to add that workload..

► If you have already set the VS editing options, skip to section 4.3.6.1.

► Use VS menu "Tools, Options…" to open the "Options" window.

Some of the settings suggested here may not be your normal preference, but some examples in the Tutorial may depend on them. You may choose to use different settings from these as you run the Tutorial, but if so, its descriptions may not match your experience. (These instructions assume that you are starting with the default "Visual C#" collection of settings[36].)

► In the "Options" window, in "Text Editor, C#, General", under "Statement Completion", select "Auto list members" and "Parameter information" (these are the default settings). Under "Settings", select "Line numbers" (default setting).

---

[36] These default C# settings may be selected using menu "Tools, Import and Export Settings…, Reset all settings", then saving the current settings, then selecting the "Visual C#" collection and clicking "Finish", then "Close".

Some of the Tutorial instructions refer to IntelliSense pop-ups, and some refer to line numbers in the code.

Since some lines in the example code may be too long to display, I usually also select "Word wrap" and "Show visual glyphs for word wrap" to make the entire contents of each line visible, but this might make some code hard to read. You may prefer to leave this disabled, and use the scroll bar to view long lines of code.

► In "Text Editor, C#, Tabs", change the settings if you desire.

The default value is 4 spaces per tab, but to shorten the lines of code, especially in deeply nested expressions, the source code in this *TDS User's Guide* and the TDS files uses an indentation of 2 spaces per level. If you prefer to make your code's formatting take some value other than 2 (such as the default value of 4), set this value as you wish, and then reformat the example code when you open the files for editing. You may also wish to edit the contents of the code snippets (which we shall import in section 4.4.4) so that the code that they generate will match your formatting preferences.

► In "Text Editor, C#, Advanced", under "Outlining", set "Enter outlining mode when files open" (default setting).

This will make it easy to hide the XML comments, **`#region`**s, etc., that decorate the example code.

► In "Text Editor, C#, Advanced", under "Editor Help", set "Generate XML documentation comments for ///" (default setting).

Since I recommend using the optional XML documentation comments wherever appropriate in a C# program, this should save time when using them. I like to sprinkle these comments fairly liberally in my code, often using them as my primary means of documentation. Please see section 4.14.9 for examples and a discussion of XML comments.

► Click "OK" to close the "Options" window.

4.3.6 Set up simulated existing working code

The following steps create a VS Project that will simulate working code that we can invoke using TDS, to illustrate tracing and debugging the working code (as shown in section 4.8.2.7) or to illustrate unit testing of working code (as shown in section 4.4.3 and several other places).

4.3.6.1 Create a new Visual Studio Project

► In VS, create a new Project: On the Start Page, click on "Create new project …". Or instead, close the Start Page and use menu command "File, New, Project...".

► Choose a Visual C# Windows® Classic Desktop "Console App" (or "Console Application") project and keep its default name of "ConsoleApp1" (or, in older versions of VS, change the name to "ConsoleApp1").

► In the "New Project" window, click "Browse...", and browse to your new (and currently empty) folder, …\Demo\ .

► Uncheck "Create directory for solution" (if it's checked).

Not creating a new directory is optional, but following this instruction will generate a file structure that more closely follows the examples in this Tutorial.

► Click "OK" to set up the new Project.

► When the project has been created, open the VS Solution Explorer window.

If the Solution Explorer is not visible, use VS menu "View, Solution Explorer" to open it.

The project should now be created (in a few seconds).

(If you are setting up a new VS Solution based on the instructions in section 4.14.7.1.1, then return there now. To set up example working code for this Tutorial, continue doing so with section 4.3.6.2.)

4.3.6.2 Add example code to the Solution

► In the "ConsoleApp1" Project, delete the Program.cs file. In response to the warning "'Program.cs' will be deleted permanently.", click **OK**.

► In the VS Solution Explorer window, right-click on Project **ConsoleApp1** (not the Solution with that name) and choose menu item "**Add, Existing Item...**".

► In the **Add Existing Item** window, browse to the folder (such as Demo\TdsSource\) containing the TDS files extracted in section 4.3.3, select files Class1.cs and Program.cs, and click **Add**. Both files should now be included in Project **ConsoleApp1**.

► To verify the location of this Solution, use VS menu "File, Exit" to save the project and close VS. In the "Save changes to the following items?" window, click "Yes".

The newly populated Demo\ folder should now contain a subfolder named ConsoleApp1\ containing a file named "ConsoleApp1.sln". In Windows® Explorer, you may wish to create a shortcut to this file to make it easy to locate.

4.3.6.3 Restart the Solution in Visual Studio

► In Windows® Explorer, navigate to folder …Demo\ ConsoleApp1\ , which now contains file ConsoleApp1.sln .

► Double-click ConsoleApp1.sln to restart VS and open the Solution.

4.3.6.4 Try to run the example program (do a "smoke test")

You should now be ready to build a working program and run a "smoke test" on it.

Traditionally, in electronic hardware design, a smoke test involved applying power to the circuit laid out on a breadboard (which had been "borrowed" from the kitchen in pursuit of knowledge). If one could see or smell smoke from a too-hot component, it indicated a serious problem that would need to be corrected before continuing. Here, we don't expect any actual smoke, but we would like to see that nothing has been left undefined, misspelled, etc.

► Build and run the Solution, for example by using VS menu "Debug, Start Debugging" or by pressing <F5>, to observe output from two example methods.

These are run without invoking any TDS test-method code. A Console window should appear after a few seconds, containing the following text:

```
***** NewCode{} class's static constructor has been called.
***** StaticCode{} class's static constructor has been called.

Error: Program bug is detected; extermination is needed.
     False exception.

Press the <Enter> key to finish . . .
```


► Press <enter> to close the Console window.

This should show that the program, which for now does not involve any TDS code, is working (or, at least, that it's not crashing, which might result in an "unhandled exception" pop-up message).

4.4 Set up TDS

The example VS Project that we have just now created and run is intended to represent a development project to which we can add a new TDS Project. In this Tutorial, we shall refer to this "ConsoleApp1" Project as "working code", to distinguish it from the code in the TDS methods that will invoke or (eventually, if we so choose) test this working code.

It is also possible to begin a VS Solution with a default (mostly empty) Project, and adding a TDS Project to that, as illustrated in section 4.14.7. We could then add working code function members concurrently with adding TDS methods to invoke them.

4.4.1 Add TDS code to the Solution [6 minutes]

4.4.1.1 Copy TDS.cs into the new **TDS** Project and prepare for editing

► In the VS Solution Explorer window, right-click on "Solution 'ConsoleApp1' (1 project)" (not the Project with that name); choose "Add, New Project…" to add a new Project to the Solution.

► Choose Visual C#, Windows, Classic Desktop, Console App (or Console Application); change its Name from "ConsoleApp2" (or "ConsoleApplication2") to "TDS". Click OK.

The Demo\ folder should now also contain a folder named "TDS", in addition to "ConsoleApp1".

► Right-click on Project **TDS** in the Solution Explorer window and choose "Add, Existing Item...".

► In the "Add Existing Item – TDS" window, browse to the folder (such as Demo\TdsSource\) containing the TDS files extracted in section 4.3.3, and select files TDS.cs and TDS_Ex01.cs . Click **Add**.

Both of these files should now appear in the Solution Explorer, in Project **TDS**. (The "Ex" is an abbreviation of "Extension", as that file extends the contents of TDS.cs.)

File TDS_Ex01.cs is included in this Tutorial as an example of a place to house some TDS test methods; these TDS methods could instead all have been located in TDS.cs . You may prefer to distribute your own test methods among several files, perhaps to more closely match the structure of the code they exercise. In section 4.10 we shall set up another TDS file similar to file TDS_Ex01.cs .

Although file TDS_Ex01.cs is used here, in general it is not a necessary part of the TDS Project. For example, we will not use it in the example in section 5.4. Its name was also chosen, in part, as one that would be unlikely to to be used in a real project, so this example TDS_Ex01.cs file probably won't accidentally become part of someone's real TDS Project.

► In Solution Explorer, delete file Program.cs from Project TDS. In response to the warning "'Program.cs' will be deleted permanently.", click **OK**.

4.4.1.2 Set TDS references

► In Solution Explorer, in project TDS, right-click on its References item and click "**Add Reference...**".

Equivalently, use VS menu "Project, Add Reference…".

► In the Reference Manager window, in the Projects, Solution tab, add a reference to the VS Project **`ConsoleApp1`** .

► Click **`OK`**. The list of **`References`** for VS Project **`TDS`** should now include VS Project **`ConsoleApp1`**; check that it is listed there.

4.4.1.3 Call static constructors

► Use menu "View, Task List" to open the Task List window; double-click the Task "**`TODO: InitializeClasses(), static variables --`**" to navigate to the definition of the field **`callStaticConstructors`**.

No change is needed here if you are running this Tutorial using the example working-code files Class1.cs and Program.cs (added in section 4.3.6.2). If so, then the **`NewCode{}`**, **`StaticCode{}`**, and **`TimeRounded{}`** static constructors that are called by **`callStaticConstructors`** are defined in the example files, and no change is needed, so you may skip this step and continue with section 4.4.2.

► However, if you are adding TDS to an existing VS Solution, delete or comment out the expressions referring to these (undefined) types, and replace them with references to types that do need to be initialized, if any are present.

The statement defining **`callStaticConstructors`** allows you to call the static constructors of the types used in your working code, before any of the TDS methods are run. Its purpose is to allow you to avoid variability in your results that might arise from changes to the order in which static constructors are called in your Solution. If you think that there could be some side-effects to such changes, then include an expression including a **`static`** variable from each type in your working code that you wish to have **`InitializeClasses()`** open, before any of your TDS methods are called. See section 4.14.5 for a discussion.

4.4.2 Hide "unhandled exception" messages

4.4.2.1 Disable exceptions as they appear [3 minutes]

The **`Assert`** statements contained in the test methods raise exceptions that report on the performance of the tests. These are the possible exception types that may intentionally be raised in this Tutorial:

```
TDS.AssertInconclusiveException
TDS.AssertFailedException
NUnit.Framework.InconclusiveException
Microsoft.VisualStudio.TestTools.UnitTesting.AssertInconclusiveException
Microsoft.VisualStudio.TestTools.UnitTesting.AssertFailedException
```

Since these are unhandled[37] in the user code, they generate pop-up windows when they are raised, and we don't need to see these. You may either disable these pop-up windows as they appear, or identify them in advance in the Exception Settings window.

If you don't disable them in advance, then whenever one does appear, with a message like "An exception of type 'TDS.AssertInconclusiveException' occurred in TDS.dll but was not handled in user code", naming one of the exception types listed here, do this:

[37] They are not completely ignored; instead of in the user code, we intend to have the unit-test software handle them.

► Uncheck the "Break when this exception type is user-unhandled" check box, click the "x" in the upper-right corner to close the pop-up, and use VS menu "Debug, Continue" or <F5> to resume running.

If you choose to disable them as they appear, you need do nothing now — just keep in mind what to do as they pop up. Only the first two of these will appear while we are using the TDS platform; the pop-up exception window for the others will appear only after we have changed the platform to NUnit or to Visual Studio Test Tools, in section 4.5.

Warning: It is possible that other types of unhandled exceptions could be raised in the course of running tests, and those should *not* be routinely ignored. To avoid getting into the habit of routinely disabling exception messages as they appear, doing the extra work (in section 4.4.2.2) now to disable all five of those listed here should make accidentally disabling the wrong ones less likely.

Another alternative to using the Exception Settings window is that, after selecting a test platform (see section 4.5 for details), you might arrange to run one Failing and one Inconclusive test. Then one or both of the "Assert…Exception Occurred" dialog boxes will pop up immediately, and you can take care of both types of exception using the dialog boxes.

To skip the steps involved in setting the Exception Settings window, go to section 4.4.3.

4.4.2.2 Use the Exception Settings window [7 minutes]

The pop-up exception messages may be disabled via the Exception Settings window, which allows you to manage the reporting of the unit-test exceptions, instead of waiting to disable them as they appear.

► Use VS menu "Debug, Windows, Exception Settings" to display the Exception Settings window.

► Set the filter to "Show Only Enabled Exceptions".

Use the funnel-shaped icon at the upper-left corner of the window to do this.

► In the "Break When Thrown" list, do the following steps for each of the exception types listed in the table in section 4.4.2.1 above.

- Click on "Common Language Runtime Exceptions" (the group following the "C++ Exceptions").
- Click on the large "+" sign (for "Add an exception to the list") near the top of the Exception Settings window.
- In the "Exception Type" text box, enter (or paste) the name of the exception.
- Press <enter>.

Having added these exception types, then do this:

► In the Exception Settings window, uncheck the "Break When Thrown" check box by the name of each of the added exceptions, then right-click on its name and select the context menu option "Continue When Unhandled in User Code".

If the added exception does not appear in the list when you press <enter>, disable the "Show Only Enabled Exceptions" filter, then enter a string like "TDS" or "NUnit" or "UnitTesting" (it's case insensitive) into the Search window to display them (if they were indeed entered correctly).

Note: This context menu is a new feature in VS Community 2015; some of this step is not needed in earlier versions of VS.

A note indicating this should appear in the "Additional Actions" column for each of these added Exceptions.

► Close the Exception Settings window.

If you choose to allow the exceptions to generate pop-up windows as described in section 4.4.2.1, their names will be added to this list automatically. Making these settings in the Exception Settings window requires extra work as you set up the project, but it avoids the appearance of the pop-up dialog boxes at unexpected times during tests.

Caution: I suggest that ONLY these five types of exceptions be ignored this way; see section 4.14.7. for a discussion.

4.4.3 Run a "smoke test" of TDS [4 minutes]

You should now be ready to run a "smoke test" on the TDS code, as we did in section 4.3.6.4 on the simulated working code.

The TDS Project that we have added to the Solution provides support for test methods that can be used to debug and/or test new or existing function members in our VS Solution. We shall now illustrate how this may be done, using some TDS methods that are included as examples in files TDS.cs and TDS_Ex01.cs.

4.4.3.1 Change the startup project

► In Solution Explorer, set the startup Project to be "**TDS**" instead of "**ConsoleApp1**".

To do this, in Solution Explorer, right-click the **TDS** Project and select "**Set as StartUp Project**". The "TDS" Project name should now appear in bold face and the "ConsoleApp1" Project name in light face.

4.4.3.2 Begin running TDS

► Begin to run the Solution (use menu "Debug, Start Debugging" or press <F5>).

If you did not hide "unhandled exception" messages, this is where the first pop-up window with an exception message should appear, similar to this: "Exception User-Unhandled" or "An exception of type 'TDS.AssertInconclusiveException' occurred in TDS.exe but was not handled in user code". If this does appear, follow the directions in section 4.4.2.1.

4.4.3.3 Example test report

The report should look something like the one shown here; it shows that one of the three tests Passed, one Failed, and one was Inconclusive. (A fourth TDS test, TimeRoundedTest(), was not run because it was not listed in the list in the **TestMethodsToBeRun** field; see section 4.8.2.5 for details).

```
***** Test{} class's static constructor has been called.
***** InitializeClasses() has begun running.
***** NewCode{} class's static constructor has been called.
***** StaticCode{} class's static constructor has been called.
***** TimeRounded{} struct's static constructor has been called.
***** The following conditional compilation directive is
      included in TDS source-code file TDS.cs:
          #define TDS_platform

***** The following conditional compilation directive is
      included in TDS source-code file TDS_Ex01.cs:
          #define TDS_platform


***** TDS.Test.TestableConsoleMethodTest()
```



```
***** InitializeTestMethod() was called at 2017-01-30T09:28:23.4236534-06:00 .

---
Beginning test of case #A1 One line of input
" done
":
Finished line 1, case #A1 One line of input:
       Returned: "Returned: DONE"
       To Console: "To the console: DONE
"

---
Beginning test of case #A2 Test throwing exception
"I dislike
   gnats, bedbugs, and mosquitoes.
But none are here.
":
Finished line 1, case #A2 Test throwing exception:
       Returned: "Returned: I DISLIKE"
       To Console: "To the console: I DISLIKE
"
Finished line 3, case #A2 Test throwing exception:
       Returned: "Returned: BUT NONE ARE HERE."
       To Console: "To the console: BUT NONE ARE HERE.
"

---
Beginning test of case #B1 Multiple input lines
"  Say  hello
  score
":
Finished line 1, case #B1 Multiple input lines:
       Returned: "Returned: SAY  HELLO"
       To Console: "To the console: SAY  HELLO
"
Finished line 2, case #B1 Multiple input lines:
       Returned: "Returned: You're a winner!"
       To Console: "To the console: You're a winner!
"
***** CleanupTestMethod() is complete.
***** (End of test)


***** TDS.Test.TestableNoConsoleMethodTest()
***** InitializeTestMethod() was called at 2017-01-30T09:28:23.5237250-06:00 .
***** CleanupTestMethod() is complete.
***** (End of test)


***** TDS.Test.AllTestsAreToBeRunTest()
***** InitializeTestMethod() was called at 2017-01-30T09:28:23.5772694-06:00 .
***** CleanupTestMethod() is complete.
***** (End of test)


***** The final test was completed at 2017-01-30T09:28:23.5792704-06:00 .
```



```
***** CleanupTestSession() is complete.


_________________

*****  This was a test run.  The following results were generated. *****

_________________

Passed tests
  The following test method returned a status of Passed:
  - AllTestsAreToBeRunTest()

_________________

Failed tests
  The following test method returned a status of Failed:

  - TestableNoConsoleMethodTest()
      Exception message:
Assert.IsTrue failed.
TestableNoConsoleMethodTest(), test case 01 Out-of-bounds exception:
  The expected exception should start with "Whoop".
  This unexpected exception was thrown:
    "False exception."

_________________

Inconclusive tests
  The following test method returned a status of Inconclusive:

  - TestableConsoleMethodTest()
      Exception message:
Assert.Inconclusive was thrown. Verify the correctness of
TestableConsoleMethodTest().

_________________

The following TDS method has a [TestMethod] attribute
    but is not in the TestMethodsToBeRun list:
      TDS.Test.TimeRoundedTest()

All TDS methods that are in the TestMethodsToBeRun list
    have [TestMethod] attributes.

_________________
Passed: 1  Failed: 1  Inconclusive: 1

_________________

  The TestMethodsToBeRun list does not match the [TestMethod] methods.

***** (End of test summary)


Press the <Enter> key to finish . . .
```


4.4.3.4 Description of the test report

This TDS test report begins with some messages identifying the order in which some static constructors were run, and giving a record of which conditional-compilation directives were active at the time of the tests.

Following these appear messages related to each of the TDS methods that were run, including a time tag and the contents of any Console messages generated by the TDS method or by the function member (in the working code) that it invokes. Since a TDS method is likely to invoke its function member(s) more than once, any Console output generated by the function member may appear here multiple times. (The TDS method itself typically will not write anything to the Console for each test case, though you may choose to have it do so if you wish, as we do in the example TDS method **`TestableConsoleMethodTest()`** .)

A summary of the test results follows, such as the names of all TDS methods that Passed, that Failed, or that were Inconclusive. For any that Failed or was Inconclusive, a related message is also displayed.

A list of mismatches between the set of **`[TestMethod]`** methods and the list of TDS methods specified by **`TestMethodsToBeRun`**, if any, appears next, to help ensure that no TDS method is accidentally skipped. (This example report shows that the TDS method **`TimeRoundedTest()`** was not run and is not counted in the list of Passed, Failed, or Inconclusive tests.) The TDS method **`AllTestsAreToBeRunTest()`** is included by default in the set specified by **`TestMethodsToBeRun`** , even if **`TestMethodsToBeRun`** contains an empty or all-blanks string.

Next appears the summary of the number of TDS methods that were run that returned each of the three types of TDS status code, for example this line:

```
Passed: 1  Failed: 1  Inconclusive: 1
```

A message comparing the set of defined TDS methods with the the list of those to be run concludes the report.

4.4.3.5 Copy the report

If you wish to copy this report from the Console window to the Clipboard, click on the Console window to select it, then use the key sequence "<alt><space>ES<enter>" to copy the entire contents of the Console window. You may then paste the copied report to, for example, an open text file, where you can conveniently read it or copy parts of it.

This report shows one example of each type of outcome that TDS reports, so that you may observe how each is reported, and we'll take care of correcting their causes later.

► Select the Console window and press <enter> to close it.

► If you did not use the Exception Settings window to hide **`Assert`** exceptions (see section 4.4.2.2), then you may repeat the test, to verify that these exceptions are now being handled without halting processing because of unhandled exceptions, by pressing <F5> again. The same report should appear in the Console window, this time without any pauses for exception pop-ups.

► Again, select the Console window and press <enter> to close it.

4.4.4 Import the code snippet file [5 minutes]

It now appears that both the working code being tested and the TDS methods that invoke it are complete enough to allow us to modify, refactor, extend, trim, etc., both the working code and the TDS code. I usually do these concurrently, both editing the working code to accomplish its intended purpose, and editing its corresponding TDS method(s) to provide suitable data to the working code, and to report on ways, if any, in which the working code is not performing as expected. You may prefer instead to use the TDS method to

specify a comprehensive set of test cases, and after that to largely leave the TDS method untouched while you write and edit the working code to satisfy those tests.

► If the snippet file, TestMethodSnippet.snippet, has already been imported into VS, then skip past this section to section 4.4.5.

Since the main reason for using TDS is to make it easier to construct new test methods, which we shall do beginning in section 4.8.1, it should almost never be necessary to create one from scratch. A new test method may be created by copying one of the provided examples, such as **`TestableNoConsoleMethodTest()`**. However, if the function member happens to use the Console, then **`TestableConsoleMethodTest()`** might serve as a better starting point, since TDS also uses the Console to display its reports. For testing a function member with multiple overloads, TDS test method **`TimeRoundedTest()`** might serve as a suitable example; it uses a **`string`** property, **`OverloadSig`**, to determine which overload is to be used, and therefore how the parameters are to be interpreted. (An **`enum`** might be as effective as a **`string`** for this purpose.) If you choose to write and use a template TDS method, I suggest that you include in it some "//TODO:" comments to identify places in the code that will likely need editing; these comments will appear in the Task List and will therefore be easy to locate in the code of new TDS methods as you add them to the TDS Project.

However, for exercising and testing simple function members, instead of copying code from an existing TDS test method and editing the copied code, it is normally more convenient (and much faster) to use the provided VS code snippet to create a new TDS test method template. Once it's imported into VS, doing this requires only a few keystrokes.

File TestMethodSnippet.snippet (extracted in section 4.3.3 into a folder such as …\Demo\TdsSource\) defines a VS code snippet that will allow us to insert the definition of a new TDS test method wherever it is needed. It needs to be re-imported into VS only when you do things like changing the definition of the snippet or reinstalling VS. (If you should need to delete the imported snippets from your copy of VS, then see section 4.4.4.1.)

A code snippet file may contain more than one code snippet, and you may wish to add others of your own design later. File TestMethodSnippet.snippet also contains a snippet named "Tds Report Symbols", which we shall use in section 4.9.

If you have not already imported the contents of TestMethodSnippet.snippet into VS, then do the following:

► Use VS menu item "Tools, Code Snippets Manager…" to open the Code Snippets Manager. Set the Language to CSharp and select a suitable location, such as the "My Code Snippets" folder.

► Import the file …\TdsSource\TestMethodSnippet.snippet by clicking on "Import…", navigating to the …\Demo\TdsSource\ folder, and selecting file TestMethodSnippet.snippet .

► Select the desired folder (I suggest "My Code Snippets") and click "Finish" and "OK" to close the Code Snippets Manager.

Calling this code snippet will insert a template, "TDS Test Method", for a TDS method similar to **`TestableNoConsoleMethodTest()`**, which can serve as a basic framework on which to build your own test methods. The code that this code snippet inserts will always require some further editing, for example to specify its name, the statement(s) calling the working code, and some test-case data (for use as parameters), but much of the inserted code can be used unchanged.

4.4.4.1 Removing a Snippet

If you have already imported a snippet file with this name, it is possible that the new version will not replace the existing one as you desire. If you have trouble with this, then open the Code Snippets Manager window (via VS menu "Tools, Code Snippets Manager…") and in it select the (for example) "My Code Snippets" folder shown there, then use Windows Explorer to navigate to the location whose path is shown in the window. (For me, that path ends with "…\Visual Studio 2015\Code Snippets\Visual C#\My Code Snippets", but it may be different on your system.) If you delete the old version of the code snippet file (for example, by using Windows Explorer), you should then be able to import your new version to that same location.

4.4.4.2 Editing the snippet file (optional)

To edit a snippet contained in a snippet file, use VS to open the *.snippet file for editing (using VS menu "File, Open, File…"). VS will open it using the XML editor. To make your new versions of snippets available along with the existing ones, give them new, different **`<Shortcut>`** names. Having made the changes that you desire, save the file with a ".snippet" extension and import it into VS as shown in section 4.4.4.

4.4.5 TDS is ready to use

At this point, you have done all you need to do to make TDS usable with an existing Project (which here is the example ConsoleApp1 Project, which is intended to simulate the working code in a real VS Solution).

To clean up your Solution, removing some unnecessary code, you will likely wish to delete from TDS.cs the TDS methods that you do not intend to use, along with deleting the second TDS source file (Tds_Ex01.cs) and the example working code in the ConsoleApp1 Project. This clean-up process is illustrated in section 5.4.3. However, the rest of this Tutorial contains instructions on using features of TDS that have not been addressed yet, so I suggest leaving the two Projects in their current state (including the example code) and continuing with the following steps, if you have not already done so.

4.5 Use alternate unit-test platforms

TDS test methods are largely upward compatible to NUnit and/or Microsoft unit-test platforms[38]. (Note: if you're not using Visual Studio Community 2015, some of the instructions in this section may not work. For example, I have had trouble using NUnit with VS 2010.) See section 4.14.3 for comments on some differences between TDS and these other platforms.

You may wish to transform your TDS methods into NUnit or VS tests, leaving the existing behavior of the TDS method in place but also allowing it to run on the other test platform. As we shall show, the C# source code of the TDS methods may be left unchanged (they will become test methods in the other platform), but the TDS Project will need to be rebuilt (in section 4.5.1.3 or 4.5.2.1).

TDS is designed to be (somewhat) easy to set up and use, but it lacks many of the capabilities of a commercial unit-test system such as NUnit. You may find it helpful to use both: continuing to quickly set up new TDS methods for tracing through unfinished working code and doing some basic tests on it, but later running those same TDS methods using a more comprehensive framework. You might do this, when some working code is ready for testing, by moving its TDS methods to a VS Project that is compiled using references to the alternate platform, to be used for testing via that platform.

[38] In this document, a unit-test "platform" is synonymous with a unit-test "framework".

To demonstrate doing this[39], we shall now look at using two alternate unit-test platforms (instead of using only the basic TDS platform) to run these same TDS methods and report the test results, before returning to TDS to examine its facilities for defining and filtering test cases. We shall do this by reconfiguring the TDS Project for this demonstration, afterwards returning (in section 4.6) to a TDS-only configuration to explore other TDS features, such as test-case filtering.

4.5.1 NUnit demonstration [15 minutes]

► If you don't wish to use the NUnit unit-test platform, skip the following steps (skip to section 4.5.2 or 4.6).

The purpose of NUnit differs from that of TDS; TDS helps with construction and debugging, whereas NUnit or a similar unit-test system does more extensive testing and monitors large numbers of tests better than TDS can.

For information about the NUnit unit-test platform, see http://www.nunit.org. The TDS code works with various recent versions of NUnit (including version 3.6).

4.5.1.1 Check installation status

Note: Some of the following steps may be inaccurate, depending on changes/improvements that may be made to the NUnit software. If these instructions do not work as expected, please refer to the NUnit Web site.

To install NUnit into your VS Solution, do the following:

► Connect to the Internet.

► In VS, with your Solution active, use menu "Tools, NuGet Package Manager, Manage NuGet Packages for Solution…" to open the "NuGet – Solution" tab.

► If NUnit was not listed in the "Installed" tab, open the NuGet tab's "Browse" tab and type "NUnit" into the Search window.

4.5.1.2 Select the Project

► From the list of NUnit items, choose "NUnit" from the list.

► Make a note of which version you are installing. We shall use this version number in section 4.5.1.5.3.1 or 4.5.1.5.3.2.

This will install the framework that handles "Assert" commands.

► In the right-hand pane, check only project "TDS" (not "ConsoleApp1") and click "Install".

In Solution Explorer, the list of "References" in the TDS Project should now include a reference to "nunit.framework" .

► In the "Preview" window, click "OK".

► Repeat these steps for "NUnit.Console"

This will install the console runner that will generate NUnit test reports.

► Close the "NuGet - Solution" tab.

---

[39] Instead of running some TDS methods using TDS and others using another platform, in this exercise we shall temporarily substitute the other platform for all of TDS, and run all of the current TDS methods using that platform.

4.5.1.3 Configure TDS code to run using NUnit

Note: As of this writing, a graphical user interface (GUI) for NUnit 3.7.0 has not been published, so the following instructions assume that you are using a version of NUnit that does not possess a GUI.

- Near the beginning of file TDS.cs, locate the line (near line 46) containing "**`//#define NUnit_platform`**" and uncomment it.
- Comment out the nearby line (near line 58) "**`#define TDS_platform`**" (change it to "**`//#define TDS_platform`**").
- Do the same in file TDS_Ex01.cs (near lines 18 and 23).

If TDS_Ex01.cs is not open in an editing window and it is included in the TDS Project, double-click its name in the Solution Explorer to open it for editing.

- In Solution Explorer, check that project TDS is still set as the StartUp Project.
- Check that the Solution Configuration is still "Debug".
- Run the program (via <F5>) to configure the assembly for use with NUnit.

If the "NUnit.Framework.InconclusiveException was unhandled by user code" pop-up window appears, clear the "Break when this exception type is user-unhandled" check box, close the pop-up, then resume running (VS menu "Debug, Continue" or <F5>). Do the same for the "AssertFailedException was unhandled by user code" pop-up window if it appears. (This is the same procedure that is described in section 4.4.2.1.)

Note that, although the report in the command prompt window still looks like the one that TDS produced earlier, the **`Assert`** calls that do the testing are now being handled by NUnit instead of by TDS.

- Press <enter> to close the Console window.

4.5.1.4 Run tests using the NUnit Console Runner

The NUnit Console Runner may be run using either the Windows Command Prompt or Windows PowerShell. The commands are similar, and both are illustrated here. The commands in each of these are not case sensitive.

4.5.1.5 Open a command window for running scripts

- Open a command window for either the Windows Command Prompt or Windows PowerShell.

You may do this using the Windows taskbar. In Windows 10, in the Task Bar click on the Windows icon, then scroll down and click on either "Windows System, Command Prompt" or some item similar to "Windows PowerShell, Windows PowerShell".

In some earlier verions of Windows, you may accomplish this by clicking "Start" on the taskbar, "All Programs", "Accessories", then either "Command Prompt" or "Windows PowerShell".

At either prompt, use the command "**`Help`**" if you wish to see the general instructions.

4.5.1.5.1 USING A COMMAND PROMPT WINDOW

Instructions for both are included here.

4.5.1.5.1.1 USING A WINDOWS COMMAND PROMPT

If you do not intend to use the Command Prompt, skip the instructions that apply to it.

4.5.1.5.1.2 USING A POWERSHELL WINDOW

If you do not intend to use the Windows PowerShell Prompt, skip the instructions that apply to it.

4.5.1.5.1.2.1 Allow running Unsigned PowerShell scripts

We shall later (in section 4.11.2) run some scripts to automate the testing, but since we are setting up PowerShell here, we shall do a bit of housekeeping now, even though it may not be needed just yet.

For security reasons (such as protecting your system from malware), to be able to run unsigned PowerShell scripts, you may need to first open PowerShell as an Administrator. To do so, <right-click> on the "Windows PowerShell" menu item and click on "Run as Administrator" or (to open a Windows PowerShell Integrated Scripting Environment session) "Run ISE as Administrator".

Use a command such as

```
Set-ExecutionPolicy Unrestricted
```

A dialog similar to the following should appear:

```
PS C:\WINDOWS\system32> Set-ExecutionPolicy Unrestricted

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust.
Changing the execution policy might expose
you to the security risks described in the about_Execution_Policies help topic
at
http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the
execution policy?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default
is "N"): Y
```

Having done so, you should be able to run PowerShell scripts and commands as a Standard User.

When finished, you may return (again running PowerShell as an Administrator) to a restricted execution policy via this command (enter "Y" to confirm):

```
Set-ExecutionPolicy Restricted
```

4.5.1.5.1.2.2 Begin using PowerShell

► Open a Windows PoswerShell window if none is open.

In Windows 10, this may be done by clicking on the Start button, then scrolling down to "Windows PowerShell" and clicking on it. The first time you run this, you may need to <right-click> its name and choose the "Run As Administrator" option; use the "Update-Help" command to download the PowerShell documentation.

4.5.1.5.2 GO TO THE DEBUG DIRECTORY

► In either the Windows Command Prompt or the Windows PowerShell Window, go to the "Debug\" file folder containing the assemblies (such as TDS.exe) containing your TDS methods. This is two levels below the ...\Demo\TDS\ folder to which you copied the script files.

You may use a command (following the ">" command prompt), such as this:

```
CD "C:\Users\ ... \Demo\TDS\bin\Debug\"
```

"CD" = "change directory". For the ". . ." shown in this example, substitute the path to your "Demo\" folder.

To save effort in typing, you may copy the pathname from a Windows File Explorer window. Navigate to the Demo\TDS\bin\Debug\ folder, where the assemblies containing your TDS methods are located. <left-click> the pathname at the top of the File Explorer window, and use <control>C to copy the pathname to the Clipboard.

In the PowerShell or Windows command-prompt window, at the ">" prompt, type "CD " followed by a <double quotation mark> to begin the command[40], paste the copied pathname by typing "<alt-space>EP", then type a closing <double quotation mark> and press <enter>. On some systems (such as Windows 10), you may also be able to paste text into either of these command windows using <control>V, as is usual in other apps such as Windows Notepad.

4.5.1.5.3 SET A REFERENCE TO THE NUNIT DIRECTORY

4.5.1.5.3.1 USING THE COMMAND PROMPT:

- Set the PATH to include the NUnit console runner's location.

This will allow you to invoke the Nunit3-Console app without specify ing its full pathname.

Starting from your Demo\TDS\bin\Debug\ folder (where your TDS.exe program is located), use a command similar to the following. Use the correct path to the folder containing nunit3-console.exe ; it may differ slightly from this. (For example, use the correct version number; here, we assume it is NUnit version 3.7.0 .)

```
PATH  ..\..\..\packages\NUnit.ConsoleRunner.3.7.0\tools;%PATH%
```

4.5.1.5.3.2 USING POWERSHELL:

Set a reference to the pathname of the Nunit3-Console app:

```
Set-alias NUNIT3-CONSOLE
..\..\..\ConsoleApp1\packages\NUnit.ConsoleRunner.3.7.0\tools\nunit3-console
```

This will allow you to invoke it without specifying its full pathname.

4.5.1.5.4 RUNNING NUNIT

4.5.1.5.4.1 HELP INFORMATION

If you wish to see help information for NUunit3-Console, use the following command:

```
NUNIT3-CONSOLE
```

4.5.1.5.4.2 RUN ALL TDS TESTS

- To run all of your TDS tests using NUnit, use this command:

```
NUNIT3-CONSOLE TDS.exe
```

The displayed results should look like those shown in section 4.5.1.5.5 below, except that all four of the defined tests will have been run, rather than only the three that we shall select. Specifically, the NUnit test report should contain this summary:

```
  Test Count: 4, Passed: 2, Failed: 1, Warnings: 0, Inconclusive: 1, Skipped: 0
    Failed Tests - Failures: 1, Errors: 0, Invalid: 0
```

[40] The <double quotation mark> characters are not needed if there are no embedded spaces in the pathname, but they do not cause any harm, regardless of spaces.

4.5.1.5.4.3 RUN SELECTED TDS TESTS

► To run only a set of selected TDS tests using NUnit, invoke the Nunit3-Console program using a command similar (except all on one line, and with no space following "--") to this:

```
NUNIT3-CONSOLE TDS.exe --
test=TDS.Test.AllTestsAreToBeRunTest,TDS.Test.TestableConsoleMethodTest,TDS.Tes
t.TestableNoConsoleMethodTest
```

This command invokes NUnit to run the same three tests that are listed in the **TestMethodsToBeRun** list in file TDS.cs (see the "**TODO: TestMethodsToBeRun**" Task described in section 4.8.2.5 below).

Now the NUnit test report's summary should look like this, with only 3 tests run:

```
  Test Count: 3, Passed: 1, Failed: 1, Warnings: 0, Inconclusive: 1, Skipped: 0
    Failed Tests - Failures: 1, Errors: 0, Invalid: 0
```

In NUnit, instead of including the list of TDS methods in a "**--test=**" option in the command, as we did here, it could be kept in a text file named in a "**--testlist=**" option. You may use this by creating a text file, for example one named "TestMethodsToBeRun.txt", containing the names of the desired TDS methods, such as these:

```
TDS.Test.AllTestsAreToBeRunTest
TDS.Test.TestableConsoleMethodTest
TDS.Test.TestableNoConsoleMethodTest
```

Then the following command should produce the same results as if these names were listed in a "**--test=**" option:

```
NUNIT3-CONSOLE TDS.exe --testlist=TestMethodsToBeRun.txt
```

Note that the "TDS.Test." preceding each test name is required for NUnit; in TDS's **TestMethodsToBeRun** list it may be omitted. (All of the TDS methods must be located in the TDS.Test{} class, but NUnit does not require this, so the "TDS.Test." qualifier is required.)

4.5.1.5.4.4 TO CLOSE THE WINDOW

To close the Command Prompt or PowerShell window, use the following command or click on the X in the upper-right corner of the window:

```
EXIT
```

4.5.1.5.5 [EXAMPLE NUNIT TEST REPORT]

The reults from running NUnit on the three listed TDS tests is shown here (but I abbreviated the command line prompt here). The TDS test report for these same tests is shown in section 4.4.3.3 above; one test Passed, one Failed, and one was Inconclusive.

```
PS C:\Users\ … \Demo\TDS\bin\Debug> NUNIT3-CONSOLE TDS.exe --test=
TDS.Test.AllTestsAreToBeRunTest,TDS.Test.TestableConsoleMethodTest,TDS.Test.Tes
tableNoConsoleMethodTest
NUnit Console Runner 3.7.0
Copyright (C) 2017 Charlie Poole

Runtime Environment
   OS Version: Microsoft Windows NT 10.0.14393.0
  CLR Version: 4.0.30319.42000
```



```
Test Files
    TDS.exe

Test Filters
    Test: TDS.Test.AllTestsAreToBeRunTest
    Test: TDS.Test.TestableConsoleMethodTest
    Test: TDS.Test.TestableNoConsoleMethodTest

=> TDS.Test.AllTestsAreToBeRunTest
***** InitializeTestMethod() was called at 2017-01-30T10:14:01.4897830-06:00 .
***** CleanupTestMethod() is complete.
***** (End of test)

=> TDS.Test.TestableConsoleMethodTest
***** InitializeTestMethod() was called at 2017-01-30T10:14:01.5338199-06:00 .

---
Beginning test of case #A1 One line of input
" done
":
Finished line 1, case #A1 One line of input:
       Returned: "Returned: DONE"
       To Console: "To the console: DONE
"

---
Beginning test of case #A2 Test throwing exception
"I dislike
   gnats, bedbugs, and mosquitoes.
But none are here.
":
Finished line 1, case #A2 Test throwing exception:
       Returned: "Returned: I DISLIKE"
       To Console: "To the console: I DISLIKE
"
Finished line 3, case #A2 Test throwing exception:
       Returned: "Returned: BUT NONE ARE HERE."
       To Console: "To the console: BUT NONE ARE HERE.
"

---
Beginning test of case #B1 Multiple input lines
"  Say  hello
  score
":
Finished line 1, case #B1 Multiple input lines:
       Returned: "Returned: SAY  HELLO"
       To Console: "To the console: SAY  HELLO
"
Finished line 2, case #B1 Multiple input lines:
       Returned: "Returned: You're a winner!"
       To Console: "To the console: You're a winner!
"
***** CleanupTestMethod() is complete.
***** (End of test)

=> TDS.Test.TestableNoConsoleMethodTest
```



```
***** InitializeTestMethod() was called at 2017-01-30T10:14:01.5743811-06:00 .
***** CleanupTestMethod() is complete.
***** (End of test)

=> TDS.Test
***** Test{} class's static constructor has been called.
***** InitializeClasses() has begun running.
***** NewCode{} class's static constructor has been called.
***** StaticCode{} class's static constructor has been called.
***** TimeRounded{} struct's static constructor has been called.
***** The following conditional compilation directive is
      included in TDS source-code file TDS.cs:
          #define NUnit_platform

***** The following conditional compilation directive is
      included in TDS source-code file TDS_Ex01.cs:
          #define NUnit_platform


***** The final test was completed at 2017-01-30T10:14:01.5926902-06:00 .
***** CleanupTestSession() is complete.


Errors, Failures and Warnings

1) Failed : TDS.Test.TestableNoConsoleMethodTest
Assert.IsTrue failed.
TestableNoConsoleMethodTest(), test case 01 Out-of-bounds exception:
  The expected exception should start with "Whoop".
  This unexpected exception was thrown:
    "False exception."
at TDS.Assert.IsTrue(Boolean condition, String message) in
C:\Users\vjohn_413d\Documents\VRJ\TDS NUnit article\TDS docs
16F20\Demo\TDS\TDS.cs:line 359
at TDS.Test.TestableNoConsoleMethodTest() in
C:\Users\vjohn_413d\Documents\VRJ\TDS NUnit article\TDS docs 16F20\Demo\TDS
\TDS_Ex01.cs:line 205

Run Settings
    DisposeRunners: True
    WorkDirectory: C:\Users\vjohn_413d\Documents\VRJ\TDS NUnit article\TDS docs
16F20\Demo\TDS\bin\Debug
    ImageRuntimeVersion: 4.0.30319
    ImageTargetFrameworkName: .NETFramework,Version=v4.5.2
    ImageRequiresX86: True
    RunAsX86: True
    ImageRequiresDefaultAppDomainAssemblyResolver: False
    NumberOfTestWorkers: 4

Test Run Summary
  Overall result: Failed
  Test Count: 3, Passed: 1, Failed: 1, Warnings: 0, Inconclusive: 1, Skipped: 0
    Failed Tests - Failures: 1, Errors: 0, Invalid: 0
  Start time: 2017-01-30 16:13:58Z
    End time: 2017-01-30 16:14:01Z
    Duration: 2.870 seconds
```



```
Results (nunit3) saved as TestResult.xml
```

4.5.1.5.6 DIFFERENCES BETWEEN TDS AND NUNIT REPORTS

The output generated by NUnit for the three listed TDS methods appears, as does a summary of Passed and Failed TDS methods, similarly to what we saw in section 4.4.3.3, when we used TDS (instead of NUnit) to generate the test report.

The end of the NUnit report mentions that the results are also written to XML file TestResult.xml, which is a bit more comprehensive (and longer) than the displayed results. (This file should be in the same folder that contains file TDS.exe, the folder …\Demo\TDS\bin\Debug\,.) For example, it includes more timing information, a stack trace, and details on all the tests (not only the failures). You may examine that file in VS by using menu "File, Open, File…". VS's XML editor allows you to collapse XML elements that you do not wish to see.

Close the TestResult.xml editing window after examining the contents; we shall not need it again in this Tutorial.

4.5.2 Microsoft Unit Test platform demonstration [15 minutes]

These instructions illustrate using the Microsoft Unit Test platform to run these unit tests. (It is not available in some versions of VS, such as the Express version of VS 2010.)

If you don't wish to use the Microsoft unit-test platform, skip the following steps (continue at section 4.5.3).

4.5.2.1 Configure TDS code to run VS unit tests

► To use the Microsoft Unit Test platform to run these unit tests, in the TDS project in Solution Explorer add a Reference to **`Microsoft.VisualStudio.QualityTools.UnitTestFramework.dll`** .

This may be located in **`C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE\PublicAssemblies\`** . In the VS Reference Manager window, use Browse to navigate there.

If you've used this Reference in a previous VS project, it may be listed in the "Browse, Recent" tab.

Note: Instead of placing your unit-test methods in the TDS Project, you may choose instead to add a VS Unit Test Project, if it's available, to your VS Solution (via "Add…, New Project, Visual C#, Test"). That Project will contain a suitable Reference, as well as an example **`[TestClass]`** and an example **`[TestMethod]`**. You could build your TDS test methods in that class, preferably via the TDS code snippet that we have imported into VS (in section 4.4.4). However, for now, the instructions in this Tutorial assume that you are using the TDS platform to build the examples and that you are keeping your test methods in source files in the TDS Project.

► Near the beginning of TDS.cs, locate the line (near line 46) containing "**`#define NUnit_platform`**" and comment it out (if it is not already commented out).

► Also, if the nearby directive (near line 58) "**`#define TDS_platform`**" is not already commented out, comment it out now.

► Do the same in file TDS_Ex01.cs (lines 18 and 23).

► In Solution Explorer, if necessary, make project TDS be the StartUp Project, as we did in section 4.4.3.1.

► Check that the Solution Configuration is "Debug".

► Press <F5> or run "Debug, Start Debugging" to build a TDS.exe suitable for use with the Microsoft unit-test platform.

Note that, although the report in the command prompt window looks similar to the one that TDS produced earlier, the **`Assert`** calls that do the testing are now being handled by Microsoft's Unit Test platform instead of by TDS.

► If the " 'Microsoft.VisualStudio.TestTools.UnitTesting.AssertInconclusiveException' was unhandled by user code" pop-up window appears, clear the "Break when this exception type is user-unhandled" check box (possibly hidden inside an "Exception Settings" link), close the pop-up, and resume running (VS menu "Debug, Continue").

► Do the same for the " 'Microsoft.VisualStudio.TestTools.UnitTesting.AssertFailedException' was unhandled by user code" pop-up window, if it appears. (This is the same procedure as in section 4.4.2.1.)

If you wish to check that the exceptions are now being handled properly, press <F5> again. The tests should run without interruption this time.

► Press <enter> to close the Console window.

4.5.2.2 Run tests using the VS Test platform

► Use VS menu "Test, Windows, Test Explorer" to open the Test Explorer window.

► In Test Explorer, click the "Run All" tab to discover the TDS test methods using VS.

Wait a few seconds for VS to locate them; a list of four TDS methods (including "AllTestsAreToBeRunTest") should appear.

One way in which you can filter these tests is via a Playlist file.

► To select the same tests as those named in the list in **`TestMethodsToBeRun`** in TDS.cs, select their names in the list of TDS methods appearing in the Test Explorer window, using <control><left-click> and/or <shift><left-click>.

► <right-click> one of the selected TDS method names.

► <left-click> the pop-up menu item "Add to playlist, New playlist"

► Navigate to your **`...\Demo\TDS\`** (or other suitable place) folder, use a name like "MyTests" for the new Playlist file, and click "Save".

It will be given a file-type suffix of "playlist".

► To examine the list of tests in MyTests.playlist, select the Test Explorer tab "Playlist : MyTests".

► While the "Playlist: MyTests" list is visible, <left-click> the "Run All" tab to run the listed tests.

If you have the three tests visible that were listed in **`TestMethodsToBeRun`**, , you should be able to see output similar to what you saw in the TDS test report, though it is formatted slightly differently.

An icon by the name of each test ("✓" for "Passed Tests", "**×**"for "Failed Tests", or "**!**" for "Skipped Tests") indicates the result of its test. (Apparently, "Skipped" means what I would call "Inconclusive", and those that are really skipped, by being omitted from the Playlist, are called "Not Run Tests" and may not even appear in the list.)

As before, in the Test Explorer window, one test should be shown as a Failed Test, one should be a Skipped Test (= Inconclusive), and one should be a Passed Test.

► <left-click> on the name of a Failed Test (in this case, it is "TestableNoConsoleMethodTest") to see its failure message (stating that the expected Exception was not raised).

► To see details of a test, <left-click> its name in the Test Explorer window, then in the lower pane <left-click> on the "Output" link to see, in a new "Test Output" tab, the text messages generated by that test method.

► To copy the output to the Clipboard, <left-click> the "Copy All" link in the lower pane, from which you can paste it, for example, into a text file for analysis.

Alternatively, you can copy to the Clipboard a report of a test by right-clicking its name and using "<control>C", or "Copy" from the context menu. (The formats of these reports may differ from each other.)

The copied/pasted information (in VS Studio Community 2017) differs slightly from what is displayed in the "Test Output" tab (for example, it contains "Elapsed Time"), but the formats are similar.

► To navigate to the failing **`Assert()`** statement, click on a link in the "Stack Trace:" section of the lower pane.

► To navigate to the source code of the selected TDS method, <left-click> the link in the "Source:" section of the lower pane.

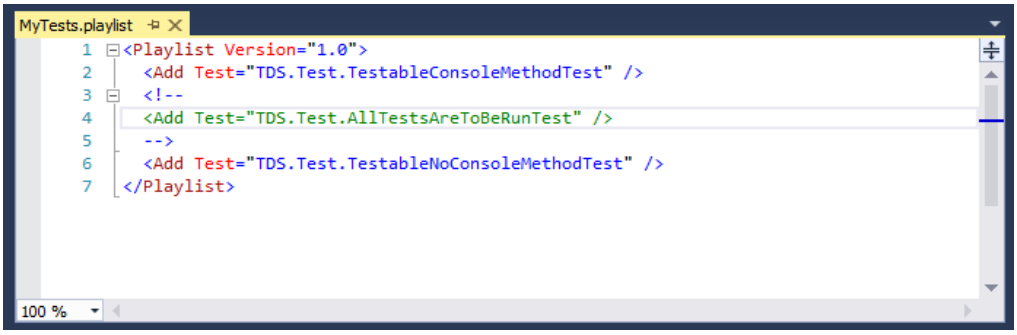
► To remove tests from the playlist, select it in the "Playlist :" tab, then use <control><left-click> to select one or more TDS test names, <right-click> on a selected name, and in the pop-up menu, <left-click> the "Remove from Current Playlist" item.

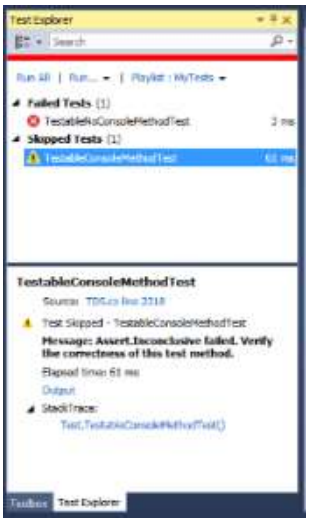
► To add tests to a playlist, for example to "MyTests", select the "Playlist: All Tests" tab, then use <control><left-click> to select one or more names, then <right-click> one of the names to bring up a pop-up menu. On the menu, <left-click> "Add to Playlist, MyTests".

If you add a TDS test name to a playlist that already contains that name, the name will appear only once in the Playlist. In contrast to this, in TDS a test listed twice in **`TestMethodsToBeRun`** is run twice. While VS Test supports running test methods in parallel, in TDS tests are run only sequence (as listed in **`TestMethodsToBeRun)`**, and not in parallel.

Note: Depending on the version of VS that you are using, you may have trouble editing the Playlist file using Test Explorer. If so, it may instead be edited by loading it into VS via menu "File, Open, File". (It's formatted as XML except without the usual "<?xml …" header.) Use menu "Edit, Advanced, Format Document" if necessary, to make it easier to read. After saving the edited version, use Test Explorer menu "Playlist : …, Open Playlist File" to reload the edited Playlist; it seems to work properly when edited that way.

In the example shown here, one of the tests (previously added to the "MyTests" Playlist) is commented out, and upon reloading this Playlist into Test Explorer (using menu "Playlist: MyTests, Open Playlist File"), that test is correctly omitted from the set of tests that are run.





► When you have finished examining the Test Explorer output, close the window by clicking the "x" on its tab.

4.5.3 Return to the TDS platform [3 minutes]

We shall use the TDS platform in the rest of this Tutorial, to illustrate its facilities for filtering test methods and the test cases that they contain. However, even if you choose to routinely use NUnit or VS Test instead of the TDS platform, you may find that using the TDS code snippet (or your own variation of it) can save time in generating test methods for your working code.

To return to using TDS, take the following steps:

► Near the beginning of TDS.cs, locate the line (near line 46) containing "**`#define NUnit_platform`**" and, if necessary, comment it out.

► Uncomment the nearby line (near line 58) containing "**`#define TDS_platform`**".

► Do the same in file TDS_Ex01.cs (near lines 18 and 23).

► Rebuild (menu "Debug, Start Debugging" or key <F5>) the Solution using TDS.

You may need to suppress the "AssertFailed" and "AssertInconclusive" exception messages again (as in section 4.4.2).

4.6 Exercise an existing TDS method [12 minutes]

At this point, you should have constructed a basic, working TDS-enabled VS Solution to which you can add TDS test methods to test function members of the types that you are developing and/or maintaining.

We shall now buggify[41] some of the working code, playing with it to see how it might affect the TDS test report. (You may choose instead to make these changes while using one of the non-TDS platforms described in section 4.5, to see the effects there, but these instructions assume you're using TDS, and some of the details may differ.)

4.6.1 Set up the Task List window

Much of the navigation in this Tutorial may be done with the help of the Task List[42] window.

► To be sure that that all of the tasks are visible in the Task List window, if necessary open all four source files (**`*.cs`**) for editing by double-clicking their names in the Solution Explorer window.

► Open the Task List window (using VS menu "View, Task List"). If necessary, adjust the "Priority" filter (upper-left corner, beside the word "Description") so that comments beginning with "**`HACK:`**" or "**`TODO:`**" appear in the list.

I am using "**`HACK:`**" Tasks to mark places in the (simulated) working code that are to be temporarily changed to illustrate how the test report is affected by the changes. The "**`TODO:`**" Tasks mark places in the TDS code that may need to be updated to make the TDS code function properly.

The Task List comments provide guides for customizing the TDS code. You may alphabetize the names of the Tasks, if you wish, by clicking on the "**`Description`**" column header in the Task List window.

4.6.2 Buggify: Raise the wrong exception

When we last ran the tests, **`TestableNoConsoleMethodTest()`** failed with a message including this:

```
TestableNoConsoleMethodTest(), test case 01 Out-of-bounds exception:
  The expected exception should start with "Whoop".
  This unexpected exception was thrown:
    "False exception."
```

This message shows that, though an exception was expected, one that does not match the expected one will generate a failure status.

(If you wish to review this message, run the tests again via <F5>, then close the Console window after examining it.)

► In the Task List window, find the Task List comment reading "HACK: TestableNoConsoleMethod() -- Remove this line, which raises the wrong Exception:". <double-click> on that Task List item to navigate to its comment.

► Delete (or comment out) these lines:

```
      //HACK: TestableNoConsoleMethod() -- Remove this line, which raises the
wrong Exception:
      throw new ApplicationException("False exception.");
```

4.6.3 Buggify: Ignore an illegal argument

► Run the tests again. (Press <F5>.)

---

[41] "Buggify" = temporarily introduce errors to verify that they are properly detected. Actually, these specific bugs were already added to the working code and labeled with "**`HACK:`**" Task comments, so this code is already buggified … but we shall illustrate adding bugs in section 4.6.6.

[42] For remarks on Task List comments, please see section 4.14.16.

The test should still fail, but this time the failure message should include this:

```
TestableNoConsoleMethodTest(), test case 01 Out-of-bounds exception:
  No Exception was raised in this test case,
  but Exception "Whoop" was expected.
```

This shows that the TDS test method correctly detects if an expected exception was not raised.

► Close the Console window. (Press <enter>.)

► In the Task List window, find the Task List comment reading "HACK: TestableNoConsoleMethod() -- Remove this line, which fails to raise an Exception:". Double-click on that Task List item to navigate to its comment.

► Delete these lines:

```
      //HACK: TestableNoConsoleMethod() -- Remove this line, which fails to
raise an Exception:
      return param1 + 1;
```

4.6.4 Buggify: Return a false calculated value

► Run the tests again. (Press <F5>.)

The test should still fail, but this time the failure message should include this:

```
Assert.AreEqual failed. Expected:
<4>. Actual:
<1004>.
TestableNoConsoleMethodTest(), test case "02 Sample test",
  Argument: 3
```

This shows that we are properly notified that the method returned a not-very-accurate 1004.

► Close the Console window. (Press <enter>.)

► In the Task List window, find the Task List comment reading " HACK: TestableNoConsoleMethod() -- Remove this line, which is intended to give a wrong answer:". Double-click on that Task List item to navigate to its comment.

► Delete the "**`//HACK:`**" comment, and the offending line containing "**`1000 +`**" (but not the entire statement):

```
            //HACK: TestableNoConsoleMethod() -- Remove this line, which is
intended to give a wrong answer:
            1000 +
```

4.6.5 Bugs are gone; signal "Inconclusive" result

► Run the tests again. (Press <F5>.)

Now none of the tests Failed; notice that the summary line now shows

```
Passed: 1 Failed: 0 Inconclusive: 2
```

since the formerly Failed test now is Inconclusive instead of Failing.

Why is it "Inconclusive"? That's the default status for unfinished TDS test methods, since we want to try to be sure that we're actually testing some functional code before we claim that it has "Passed" a test. Until then, we don't know if the working code is working, so the test result is "Inconclusive".

► Close the Console window.

4.6.6 Buggify: Cause an exception to fail to be raised

To observe failure reports for a method that uses the Console, we can make similar changes to **`TestableConsoleMethod()`** in file Class1.cs . The Console, which is being used by the working code, will also be used for displaying the test report.

► Navigate there using the Task List item "HACK: TestableConsoleMethod() -- Change string to "B UGS" to check test method".

The following lines are present:

```
      //HACK: TestableConsoleMethod() -- Change string to "B UGS" to check test
method
      if (nextLine.Contains("BUGS"))
```

► To verify that failures there are correctly identified, insert a space into the string "BUGS", as the comment suggests.

► Run the tests (press <F5>).

A message that the expected exception failed to be raised should appear in the test report:

```
TestableConsoleMethodTest(), test case A2 Test throwing exception:
  No Exception was raised in this test case,
  but Exception "Bugs are detected" was expected.
```

► Close the Console window.

► Remove the added space from the "B UGS" string.

However, don't delete this "**`//HACK:`**" comment just yet — we shall use it again in section 4.8.7.1.

If you wish to repeat these actions while using an alternate platform to do the testing (as we did in section 4.5), you may restore the code in Class1.cs and Program.cs to its original state by recopying these files from the …\Demo\TdsSource\ folder to the …\Demo\ConsoleApp1\ folder, replacing the versions that are already there, that we have been editing.

4.7 Run the working code without TDS

TDS is intended to be a temporary aid in detecting software bugs, and it is probably helpful to run the working code in its usual operating environment (that is, without being called by TDS). We'll illustrate doing that here.

► In the Solution Explorer, set ConsoleApp1 as the Startup Project.

To do this, in Solution Explorer, right-click the **`ConsoleApp1`**Project and select "**`Set as StartUp Project`**". The "ConsoleApp1" Project name should now appear in bold face and the "TDS" name in light face. (This is similar to what we did in section 4.4.3.1, but in reverse.)

► Run the Solution (use <F5>).

► Press <return> twice.

The (simulated) working code is running without any interference from a TDS method, and no test report appears in the Console window.

The following text should appear in the Console window:


```
***** NewCode{} class's static constructor has been called.
***** StaticCode{} class's static constructor has been called.

(This is a test of example method TestableNoConsoleMethod().)

  One plus 3 is 4.


(This is a test of example method TestableConsoleMethod().)

  Type any words except "bugs", followed by <Enter>.
    Do this twice.


To the console:

To the console:

Press the <Enter> key to finish . . .
```

- Close the Console window.
- Having observed this output, set Project "TDS" as the Startup Project, as we did in section 4.4.3.1.

4.8 Create a new TDS method

Now we shall illustrate how to construct new TDS methods to work with the code that we are developing or updating.

The following instructions guide you through additional examples of TDS code, including constructing new TDS methods, reporting exceptions, filtering test methods and the included test cases (to run only those you select), and running tests using command-line scripts.

4.8.1 Clean up the test report

To allow us to observe how the test report (in TDS and elsewhere) reflects unfinished code, we have left some tests in an Inconclusive state. The rest of this Tutorial does not require this, so from here onward we shall pretend that we have finished the TDS methods that are now returning Inconclusive.

4.8.1.1 Clean up "Inconclusive" code

- In the TDS methods containing the two **`Assert.Inconclusive()`** statements (**`TestableNoConsoleMethodTest()`** and **`TestableConsoleMethodTest()`**), delete or comment out these statements.

You may navigate to these via the Task List, for example via the "**`TODO: TestableNoConsoleMethodTest() -- Remove the Assert.Inconclusive()`**" Task.

In my projects, I normally merely comment out the **`Assert.Inconclusive()`** statement when I finish a TDS method instead of deleting the statement, because I might need it later while I update the TDS method. At that time, it would serve to remind me that the TDS method is unfinished. However, deleting it makes the code less cluttered and thus easier to read.

In real life, we should disable this statement whenever we decide that its TDS method has addressed all of the behavior of the working code that it needs to exercise and that it needs no further changes at this time. To save time in this Tutorial, however, we're just pretending to have perfected these TDS methods. A real one will

normally entail more analysis, to ensure that the behavior of the tested function member is more thoroughly explored.

See section 4.14.16.5 for a discussion of "**Inconclusive**" tasks.

► Run the tests (press <F5>).

Having deleted these **Assert.Inconclusive()** statements, we should see a report that ends with a summary similar to the following, a signal that, for now, no further work is needed on the three tests that we are running:

```
Passed: 3  Failed: 0  Inconclusive: 0
```

There is, of course, still a message in the report that we are not running some of the TDS methods in Project TDS, along with a list of those TDS methods that are not being run; we shall address those in section 4.8.2.5.

► Close the Console window.

4.8.2 Create and run a TDS test [22 minutes]

4.8.2.1 Set up a basic TDS method

Now we shall use an imported code snippet (imported into VS in section 4.4.4) to add a new TDS method to our VS Solution to modify and test an existing function member of a C# type. In this case, we shall test the indexer **NewCodeNamespace.BitArray[]** , which is defined in the **NewCodeNamespace.BitArray{}** class and currently has no associated TDS test method. We shall build its new test method in file TDS_Ex01.cs by using the "**TdsTest**" code snippet defined in the TestMethodSnippet.snippet file.

Following the "**TODO: New TDS methods may be placed here:**" Task in file TDS_Ex01.cs, but before the

```
}  // end:Test{}
```

line, we shall place our new TDS method. (A similar Task comment appears near the end of file TDS.cs, and new TDS methods may be placed there as well.)

We shall call the new TDS method "**BitArrayTest()**". If we already had several TDS methods defined here, I would place this new one in alphabetical order in the collection of TDS method definitions to make it easy to find, but the order is unimportant as far as the compiler is concerned.

► To insert a copy of the TDS method template, place the cursor on a blank line immediately following the Task "**TODO: New TDS methods may be placed here:**" in file TDS_Ex01.cs. (A similar Task is in file TDS.cs.) Type "**TdsTest**" (or select **TdsTest** from the pop-up list that should appear as you type), and press **<**tab**>** twice. Code for the new test method appears.

If you unintentionally <tab> past the field that you want to select, use <shift><tab> to reverse direction.

► Into the first highlighted field in the code snippet's XML comment, which initially contains "TestableFunctionMember", type "**BitArray**" (no parentheses, no brackets) and press <tab>.

The corresponding fields throughout the snippet are automatically updated to match it; for example, the name of the new TDS method becomes "**BitArrayTest()**". If you wish, change the contents of other fields in the snippet by tabbing to them and entering new names.

► Press <enter> when done.

For this example, we'll use the default values for the other highlighted fields.

4.8.2.2 Update usings

► Update the list of **`usings`** in TDS_Ex01.cs if necessary.

To do this, navigate to the "**`TODO: Usings`**" task in the Task List for the file where you are placing the new TDS method; the Task List may be filtered (click on the "File" header) to include only the Task comments in a selected file. Delete **`using`** statements that are not needed and, if necessary, add one for the namespace of the working code. You may also need to add a suitable Reference in the TDS Project to the working code. (The compiler will let you know if it can't find your working code.)

Updating the **`using`** statements is not necessary with the example files, but it may become so as you add TDS to existing working code. (If a suitable **`using`** statement were not already present in this file, the TDS method would need to refer to **`BitArray`** as **`NewCodeNamespace.BitArray`**.)

4.8.2.3 Customize the TDS method

At this point, depending on the type of expression needed to invoke the working code, the new **`BitArrayTest()`** TDS test method may compile, but it will do nothing useful until we customize it. (In its present state, it won't even compile, but we'll take care of that soon.)

Looking at the Task List, we find some tasks beginning with "**`TODO: BitArrayTest() --`**". (Open file TDS_Ex01.cs for editing if those tasks are not visible in the Task List.) See section 4.14.16.1 for suggestions on using the Task List to track unfinished work.

You may find Bookmarks (VS menu "Edit, Bookmarks, Toggle Bookmark") helpful as well for tracking unfinished work.

► Navigate to Task "**`TODO: BitArrayTest() -- Use a suitable default value.`**".

If it would make navigation easier, click the top of the "Description" column to alphabetize the Task names.

(Don't delete this "**`TODO:`**" Task comment yet; we'll use it in section 4.8.2.4.)

We could replace the

```
            var actual = 0;
```

statement following that "TODO:" with the following two lines of code:

```
            var bA1 = new BitArray(23);
            var actual = "";
```

The definition of **`bA1`** shown here would set up a fixed-length array. Although this could work, we may want to run tests on **`BitArray{}`** objects of different sizes… so it makes sense instead to set this up using a **`testValues[]`** property that we can change in different test cases.

4.8.2.4 Add a property to testValues[0]

We shall add properties to **`testValues[0]`** to set up a different object for each test case.

► Navigate to Task "**`TODO: BitArrayTest() -- Define inputs and expected outputs.`**" to go to the definition of **`testVslues[]`**, and replace the line beginning "**`Arg =`**" with the following two lines:

```
   Arg = new[] { 0, 2, 7, 22 },  //Elements to be set to true
   MyArray = new BitArray(23),  //BitArray instance to be copied and changed
```


Since **testValues[]** currently contains only one element, the order in which the properties appear within **testValues[0]** isn't important to the compiler, but placing it in the suggested location will make it easier to follow the examples in this discussion.

Although the program would run just as well without the comments that are included in these two lines as with them, I suggest always including some on each new property, as well as updating the comments on existing properties whenever we change how they are used or what they mean. For suggestions on naming properties added to **testValues[]** and commenting them, please see section 4.14.10.

The new property **MyArray** allows us to set up **BitArray** objects of various sizes. (Well, it's a fixed size, 23, in this first test case, but we can specify other sizes in other test cases.) We can now test bit arrays of various lengths by specifying their sizes in the test cases.

Besides specifying the length of the array, we shall want to be able to set the value of more than one bit in this array to 1 (which will be interpreted as **true**) for each test case, and we shall use the array elements in **Arg** to specify which bits (= array elements in the bit array) we want to change from 0 to 1.

▶ Navigate (again) to Task "**TODO: BitArrayTest() -- Use a suitable default value.**" and delete the "//TODO:" comment.

▶ Delete the line

```
    var actual = 0;
```

and replace it with these:

```
    var bA1 = tCase.MyArray;  //BitArray instance to be tested
    var actual = "";
```

The intent is to allow our test cases to specify **BitArray{}** object instances of various sizes that we can conveniently test.

Now the Task "**TODO: BitArrayTest() -- Provide a suitable calling expression**" needs some work.

▶ Replace this "**//TODO:**" comment and the "**actual =**" statement with these two lines:

```
    foreach (var i in tCase.Arg) bA1[i] = true;
    actual = bA1.ToString();
```

In the 23-element bit array **bA1[]** this is intended to change array elements 0, 2, 7, and 22 (as specified in the **tCase.Arg[]** array) to **true**.

You may also delete the comments following these lines, beginning with "**//Before any tests are added**".

Having done this, we again have a compilable program. (If you want to verify this claim, press <F5>, observe the Console window, then close it by pressing <enter>.)

4.8.2.5 Filter the TDS methods

At times, we may wish to run only one, or a small subset, of the TDS methods in a TDS project. This is one of those times; since, for now, we're developing and debugging the new code in **BitArray()**, we wish to focus on this one task without being distracted by other tests.

We can accomplish this by commenting out some of the names listed in **`TestMethodsToBeRun`**. We shall edit the list to run only our newly added TDS method and skip all the others. Note that, regardless of which tests are included in this list, the TDS test method **`TDS.Test.AllTestsAreToBeRunTest()`** is always run following the listed test(s); see section 4.8.7.1. Since it is run last, it does not create any side-effects that might interfere with our tracing into the **`BitArray()`** code.

► Navigate to **`TestMethodsToBeRun`** via the Task List; find "TODO: TestMethodsToBeRun -- " in file TDS.cs and double-click on that list item to navigate there.

You can instead navigate to this list of methods in any of several other ways; see section 4.14.17 for suggestions on navigation.

► Edit the list of TDS methods to run only the one that we just now added — temporarily comment out all the other test method names in **`TestMethodsToBeRun`**, for example by typing "//" at the beginning of each line, or by using VS menu "Edit, Advanced, Comment Selection".

► Add "**`BitArrayTest()`**" to the list; this name is case sensitive.

The edited list might now look like this, with skpped tests commented out:

```
//              TestableConsoleMethodTest
////            NonexistentTest  //Reported as a name that should be corrected
//              TDS.Test.TestableNoConsoleMethodTest()  //Includes qualified-
name prefix
////            TimeRoundedTest()  //Example not initially exercised in the
tutorial
////            TestableNoConsoleMethodTest  //Duplicate, would be run a 2nd
time
BitArrayTest()
```

Whenever this list excludes some TDS methods that are defined in Project TDS, a message appears near the end of the test report telling us which of the TDS tests are being skipped, as a reminder to include them again later.

4.8.2.6 Observe the list of filtered TDS methods

► In Solution Explorer, set TDS as the Startup Project if it is not set up that way, as we did in section 4.4.3.1.

► Run TDS (press <F5>).

We do this so that we can see that we are running only the TDS methods that we intend to run.

Near the end of the report, the following notice appears:

```
The following TDS methods have [TestMethod] attributes
    but are not in the TestMethodsToBeRun list:
      TDS.Test.TestableConsoleMethodTest()
      TDS.Test.TestableNoConsoleMethodTest()
      TDS.Test.TimeRoundedTest()
```

This indicates that the three tests listed here were not run, but they may be reinstated at any time.

Incidentally, the report also shows that **`BitArrayTest()`** was run and Failed, but for now that is not important, since what we want to do is to trace execution of the **`BitArray()`** constructor, not to test the results of running it.

► Close the Console window (press <enter>).

4.8.2.7 Begin debugging using the new TDS test method

We are ready to use TDS to help with debugging.

► In Class1.cs, navigate to the **`NewCodeNamespace.BitArray{}`** class's **`ToString()`** method, and set a breakpoint (VS menu "Debug, Toggle Breakpoint" or <F9>) on the **`return`** statement there.

► Run to the breakpoint (use <F5>).

Note that **`bA1.ToString()`** is not called in **`BitArrayTest()`** until after all four bits have been set,

Alternatively, to avoid having to look for the testable code, we could set a breakpoint on the calling statement ("**`actual = bA1.ToString();`**") that we set in the "**`TODO: BitArrayTest() -- Provide a suitable calling expression`**" Task and run (<F5>) to there. We could then remove that breakpoint, step into the **`BitArray.ToString()`** method (using <F11>), and set a breakpoint on its **`return`** statement.

► Examine field values in the Locals window.

The **`this.Length`** property should have a value of 23, **`this.BitsPerWord`** should be 5, and in the contents of the private **`bits[]`** array, **`bits[0]`** should have a value of 5 (= bits 0 and 2 of **`bits[0]`**), and the others should contain successive values of 4, 0, 0, and 4. Since these values are what we expected them to be, for now no changes to the code are needed. In real life, especially when an unexpected value is returned, this type of examination can reveal errors in the code.

► Use menu "Debug, Stop Debugging" (or <shift><F5>) when you have inspected the local values.

► Remove or disable the breakpoint when finished.

On a more complex type, you might want to write your own VS Custom Visualizer for that type, to help with debugging. (Sorry, how to do that is out of scope for this *TDS User's Guide*.)

4.8.3 Modify and test the working code [35 minutes]

4.8.3.1 Set up an AreEqual() test.

This working-code **`BitArray{}`** class is apparently ready to return testable values, so we can soon begin using this TDS method as a unit-test method. (For more on constructing unit-test methods, please see section 5.1.6.)

Since what we will compare are strings, we are changing the name of property **`ValueExp`** in **`testValues[0]`** to **`StringValueExp`**, as a reminder that its value will just be a string, rather than the complete value of the object.

► In the Task List, navigate to Task "**`TODO: BitArrayTest() -- Define inputs and expected outputs.`** ".

► Change the **`ValueExp`** line in **`testValues[0]`** to be

```
            StringValueExp = "",  // Expected returned value
```

► Give the **`Id`** line a more descriptive value, such as

```
            Id = "01 Multiple bits set, 23 elements", // Test case identifier
```

Now your **testValues[0]** definition will look something like this (maybe with different comments from those shown here):

```
        new {
          //TODO: BitArrayTest() -- Define inputs and expected outputs.
          Id = "01 Multiple bits set, 23 elements", // Test case identifier
          Arg = new[] { 0, 2, 7, 22 },  //Elements to be set to true
          MyArray = new BitArray(23),  //BitArray instance to be copied and
changed
          ExceptionExp = DefaultExceptionMessage,  // Expected exception
          StringValueExp = "",  // Expected returned value
        },
```

The example **Assert.AreEqual()** statement that we shall use in our test will compare the **ToString()** values of the given objects, and the test will pass iff[43] they match.

▶ In the Task List, navigate to Task "**TODO: BitArrayTest() -- Provide suitable non-exception tests here**".

▶ In the **Assert.AreEqual()** statement, change **tCase.ValueExp** to **tCase.StringValueExp**.

▶ Run TDS (use <F5>).

We expect this to fail, but the failure message will show us the returned value in the proper format. We see, in the failure message, these lines:

```
Assert.AreEqual failed. Expected:
<>. Actual:
<Hex. contents: 0x05, 0x04, 0x00, 0x00, 0x04>.
```

The actual value returned is shown between the angle brackets. We want to use this, if it's correct, as the "expected value", but we first need to examine it carefully for accuracy. A mistake here could be worse than doing nothing, as it would give us false confidence that the working code is correct.

4.8.3.2 Copy text from the Console into source code

In real life, we might do some desk-checking here, but for this Tutorial, let's assume that we have satisfied ourselves that this result is indeed correct. We shall copy the "Actual:" string, close the Console window, and paste the copied string into **testValues[0]**.

▶ Some versions of Command Prompt let you use the mouse to select and copy (using "<control>C") the text. If so, do that.

If that doesn't work, to copy this text from the Console window, select the window; use <alt><space>, **E**, **K**; use the arrow keys to move the cursor to the text to be copied; select the text using shift and arrow keys; press <enter> to copy the selected text to the Clipboard.

▶ Close the Console window, or stop debugging.

[43] "Iff" is an abbreviation for "if and only if".

► Use Task "**TODO: BitArrayTest() -- Define inputs and expected outputs.**" to navigate to **testValues[0]**, then paste the copied string into the

```
    StringValueExp = "",  // Expected ToString() value
```

line in **testValues[0]** to make it look like this:

```
    StringValueExp = "Hex. contents: 0x05, 0x04, 0x00, 0x00, 0x04",  //
Expected ToString() value
```

Having been updated, the TDS method should no longer fail (at least, not due to a mismatch in this value).

► Run TDS (use <F5>).

Near the end of the test report, the following text should appear:

```
_________________
Passed: 1  Failed: 0  Inconclusive: 1


_________________

  The TestMethodsToBeRun list does not match the [TestMethod] methods.
```

As explained in section 4.6.5, "Inconclusive" is the best result that we should properly expect for now, since we haven't finished fiddling with this TDS method.

The test shown as "Passed" in this report is **AllTestsAreToBeRunTest()**, which is always run following the other tests (see section 4.8.7.1).

The message that the **TestMethodsToBeRun** list does not match is a reminder to reactivate the TDS methods that we disabled earlier, in section 4.8.2.5.

► Close the Console window (use <enter>).

4.8.3.3 Set up an Assert.IsTrue() test

TDS also provides an alternate choice of testing methods, **Assert.IsTrue()**, which passes iff the given Boolean expression is true, and this offers greater freedom in defining tests than the one that compares strings. However, the failure message displayed by **Assert.AreEqual()** has a more informative default format than **Assert.IsTrue()** does; it displays both strings for visual comparison. (As mentioned in section 4.14.3, other test platforms offer a greater variety of **Assert{}** methods than TDS does.)

To illustrate the use of **Assert.IsTrue()**, we shall add a test (using the same test case that we just now ran) to compare some numeric values.

► In **testValues[0]** of **BitArrayTest()** (at its "**TODO: BitArrayTest() -- Define inputs and expected outputs.**" Task), add another property following the definition of **StringValueExp**:

```
        BinaryValueExp = 0X00400085,  //Expected packed numeric value
```

We could use the same trick as in section 4.8.3.2 above, and copy the returned value from the error message, but if we know the value already, we can simply state it here.

Incidentally, this number looks different from the value that we gave **testValues[0].StringValueExp** mostly because **StringValueExp** is packed 5 bits per word (displayed as two base-16 digits), and the value of **BinaryValueExp** is displayed using 4 bits per base-16 (hexadecimal) digit.

► To use the **`IsTrue()`** form, add the following statements immediately before the "**`#endregion Apply tests when no exception is raised`**" directive (below the Task comment "TODO: BitArrayTest() -- Provide suitable non-exception tests here"):

```
                    //actualBinary contains an integer in which
                    //  bit 2^n == 1 iff bA1[n] is true
                    //  and n < 32 (size of uint)
                    uint actualBinary = 0;

                    for (var i = 0;
                          i < bA1.Length
                            && i < 32;  //Copy only the low 32 bits
                          i++)
                        actualBinary |= ((bA1[i] ? (uint)1 : 0) << i);

                    Assert.IsTrue(
                          actualBinary == tCase.BinaryValueExp,
                            string.Format(@"
BitArrayTest() 2, test case ""{0}"",
         Expected: 0X{1:X8}
  Actual value is: 0X{2:X8}
  Bits are packed {3} bits per 32-bit word."
                              , tCase.Id  //{0}
                              , tCase.BinaryValueExp  //{1}
                              , actualBinary  //{2}
                              , bA1.BitsPerWord  //{3}
                            )
                          );
```

To use the **`Assert.IsTrue()`** method, we did some calculations, then called **`Assert.IsTrue()`** using the results.

To help identify which of these tests has failed, if any, I have included a " 2" following the "**`BitArrayTest()`**" name in this **`Assert.IsTrue()`** test.

► Similarly, in the error message of the existing **`Assert.AreEqual()`** test, immediately above the new statement, also insert a " 1" in the corresponding location, like this:

```
BitArrayTest() 1, test case ""{0}"",
```

Instead of sequence numbers like these, you might prefer to use descriptive names here, as we do in section 5.2.8.5.2.3. With either option, the purpose is to make it easy to get more information about a failed test, for example by setting a breakpoint at the indicated place in the TDS method code and examining variables there.

For the moment, I have left some of the **`//TODO:`** task comments active, to make the code they identify easy to find. They may be deleted whenever you feel that they are no longer needed. (Well, maybe not right now – this Tutorial still depends on some of them.) For example, while I am working in **`testValues[]`**, I leave its "**`//TODO: BitArrayTest() -- Define inputs and expected outputs.`**" comment in place. You may prefer using Bookmarks (VS menu "Edit, Bookmarks, Toggle Bookmark") to provide this navigation service.

To see the error message from the new test, you might change the initial value of **`actualBinary`** to 7, for example by inserting the following statement after its definition:

```
            actualBinary = 7;  //HACK: Cause failure.
```


(Some other initial values, such as 1 or 5, would be masked and thus not cause a failure.)

If you then run the program, the TDS test report would include the following failure message:

```
Assert.IsTrue failed.
BitArrayTest() 2, test case "01 Multiple bits set, 23 elements",
         Expected: 0X00400085
  Actual value is: 0X00000007
  Bits are packed 5 bits per 32-bit word.
```

Change the initial value back to 0 (delete the added statement) before continuing.

- Run the program (use <F5>) to verify that no error occurs.
- Close the Console window (use <enter>).

4.8.3.4 Run a test that raises an exception

We have defined some tests and have applied them to the first test case in **`testValues[]`**. Now we would like to verify that the indexer properly objects (by raising an exception) to improper inputs; we'll do this using a new test case.

Note that, from now on, any new test cases that we add to **`testValues[]`** must contain properties matching those of **`testValues[0]`** in name, order, and type. There will thus be a bit of extra work involved, from now on, in making changes to the structure of these test cases, to make sure that they all match.

We can give **`tCase.Arg`** a value that should cause failure, so that we can check that it raises the proper exception.

- Go to Task "**`TODO: BitArrayTest() -- Define inputs and expected outputs.`**".
- Copy the contents of **`testValues[0]`** to new element **`testValues[1]`** and change the line

```
    Arg = new[] { 0, 2, 7, 22 },
```

to

```
    Arg = new[] { 23 },
```

The other properties can be simplified, too, so that the value of **`testValues[1]`** will be

```
        new {
          Id = "02 Out-of-bounds exception, 23 elements",
          Arg = new[] { 23 },
          MyArray = new BitArray(23),
          ExceptionExp = "",
          StringValueExp = "",
          BinaryValueExp = 0X0,
        },
```

We could specify the real expected value of **`ExceptionExp`**, but it's probably easier to copy it from the failure message, so for now we leave it as an empty string.

We erase the comments copied from the first element, since they applied to the properties, not to any specific values, and these properties are the same as those in **`testValues[0]`**. (For a discussion of comments on **`testValues[]`** properties, please see section 4.14.10.2.) You may also delete the comments following, that begin with "**`//If more than one array element is defined here`**" and that serve as a suggestion on the use of comments in the **`testValues[]`** array.

The expected values we specified here for **`StringValueExp`**, etc., are of no importance, as we expect the working code to raise an exception instead of returning values. The compiler requires us to mention these properties, so that they will match those in **`testValues[0]`**, but we shall ignore their values in this test case.

- Run the program (use <F5>).

The failure report shows an error message of "Index was outside the bounds of the array." This isn't as informative as it might be, and we should probably improve it. (We shall do that soon, in section 4.8.3.5.)

We can repeat the trick we used earlier (with **`StringValueExp`** in section 4.8.3.2) of copying the beginning of the actual exception message from the Console window into this **`ExceptionExp`** property:

```
            ExceptionExp = "Index was outside",
```

- Run the program (use <F5>).

The TDS report should now show a status of

```
Passed: 1 Failed: 0 Inconclusive: 1
```

Translation: we won, since the indexer correctly raised the proper exception.

- Close the Console window (press <enter>).

4.8.3.5 Improve the working code

We may notice that the error message generated by an out-of-bounds index value is not very informative, and we would like to provide more details in the message. Since both of the accessors in the **`BitArray`** indexer, "**`public bool this[int index]{}`**", use the same test, and generate the same message, we can extract the method shown nearby from them. (Merely refactoring this code into a separate method, as we shall do now, will not improve the message. However, it will simplify the process of making the changes that we desire.)

After we extract the method that we shall use, whenever any changes are needed in this code, one change will suffice for both, making it easy to ensure that the generated messages will be consistent with each other. What we can do in this instance is to fancify the exception message, making it more specific and informative. We define the following new method, which either does nothing or raises an exception:

```
    /// <summary>
    /// Throw an "out-of-range" exception if needed
    /// </summary>
    /// <param name="index">Requested index value</param>
    /// <exception cref="IndexOutOfRangeException">
    /// Raise exception iff index is out of range.</exception>
    void CheckForIndexException(int index)
    {
      if (index < 0 || index >= Length)
        throw new IndexOutOfRangeException(
              string.Format(
                  @"Index was [{0}], which is outside "
                    + "the allowed range of 0 to {1}."
                  , index //{0}
                  , Length - 1 //{1}
                ));
    } // end: CheckForIndexException()
```


► Copy this code and paste it into a suitable place in the **`BitArray{}`** class in file Class1.cs , such as just before the

```
} // end: BitArray{}
```

line following the definition of its **`ToString()`** method. (This is *not* the closing brace that is at the end of the **`BitArray()`** constructor.)

Incidentally, the XML comments (C# single-line comments beginning with "///") are not a necessary part of the code, but I recommend using them; see section 4.14.9 for a discussion.

► In the code for the indexer's **`get`** and **`set`** accessors (in the definition of **`public bool this[int index]{}`**), replace each

```
        if (index < 0 || index >= Length)
          throw new IndexOutOfRangeException();
```

statement with the following line:

```
    CheckForIndexException(index);
```

4.8.3.6 Update the TDS test method to match

► Rerun this (using <F5>)

This time we see in the test report in the Console window a more helpful failure message, displayed because it didn't match the "**`Index was outside`**" message that we were previously looking for:

```
    "Index was [23], which is outside the allowed range of 0 to 22."
```

► Close the Console window.

► Back in **`BitArrayTest()`**, in the **`testValues[]`** element with Id = "02 …", to which we can navigate via Task "TODO: BitArrayTest() -- Define inputs and expected outputs.", change the value of the **`ExceptionExp`** property to match what we now expect to see:

```
    ExceptionExp = "Index was [23], which is outside",
```

► Run the test (<F5> again).

This time, no failure is reported; we should get a status of Inconclusive.

► Close the Console window.

4.8.3.7 Finish testing `BitArray[]`; remove its "Inconclusive" flag

In real life, we'd probably add more **`Assert()`** tests, such as code to check the value of each element before and after changing its value. Having done so, we would continue running the tests and use the report to help us look for anomalies. If necessary, we would continue tuning the working code (and maybe adding test cases) until we see no further unexpected results.

► However, pretending we're done for now, go to the "**`TODO: BitArrayTest() -- Remove the Assert.Inconclusive()`**" Task and either comment out the **`Assert.Inconclusive()`** statement or delete it completely.

► Also delete the other "**`TODO: BitArrayTest()`** …" Tasks that remain in **`BitArrayTest()`**.

For this Tutorial, we assume that the work they describe has been done. We can always add other task comments as needed.

- Run TDS; this time, we are rewarded with this status message:

```
Passed: 2 Failed: 0 Inconclusive: 0
```

- Close the Console window.

4.8.4 Resume testing elsewhere [4 minutes]

4.8.4.1 Reactivate the other tests

The report also contains a reminder to reactivate the tests that we put to sleep earlier, and we now do so.

- De-comment their names in **`TestMethodsToBeRun`**.

You can navigate there via the "**`TODO: TestMethodsToBeRun`** …" Task.

Selecting the names and using menu "Edit, Advanced, Uncomment Selection" should do it.

4.8.4.2 Test a type with multiple constructors

One test that we did not run earlier, but that is in the list of omitted tests, is **`TimeRoundedTest()`**. This TDS method shows an example of how one might test a type having several different constructors. Two such constructors are included in the **`TimeRounded{}`** type, and both are tested by **`TimeRoundedTest()`**.

This test method includes a feature that is not included in the test method generated by the **`TdsTest`** snippet (defined in the TestMethodSnippet.snippet file; see section 4.4.4): a **`switch`** statement conditioned on a property (**`OverloadSig`**) in **`testValues[]`** that selects a calling statement with a suitable parameter list. Some of the other **`testValues[]`** properties, such as **`ParamFloat`**, are overloaded, since they serve only to pass objects of compatible types to the function member that is being called, and they have no other meaning. Overloading these property names allows us to use fewer properties to call working-code methods with multiple signatures.

- The name of **`TimeRoundedTest()`** is included (but commented out) in **`TestMethodsToBeRun`**; de-comment it.

- Also, to see the effect of including a misspelled test name, de-comment **`NonexistentTest()`** in **`TestMethodsToBeRun.`**

If the name of **`TimeRoundedTest()`** were not already in the list, we could copy it from the list of omitted tests in the TDS report and paste it into **`TestMethodsToBeRun`** instead of retyping it. Note that, since all TDS tests are assumed to belong to the **`TDS.Test{}`** class, we may omit the "**`TDS.Test.`**" string from the beginning of its name.

- Run the tests (use <F5>).

This time, we see

```
Passed: 5 Failed: 0 Inconclusive: 0
```

along with a notice that the not-yet-defined TDS method "**`TDS.Test.NonexistentTest()`**" listed in **`TestMethodsToBeRun`** is nowhere to be found, and that the list of TDS methods in **`TestMethodsToBeRun`** does not match the collection of TDS methods in the TDS Project.

4.8.4.3 Clean up the `TestMethodsToBeRun` list

► Close the Console window and comment out or delete the "**`NonexistentTest`**" line in **`TestMethodsToBeRun`**.

This was included to illustrate the effect on the test report of listing a test that is not (yet) defined. As long as the list names a TDS method that cannot be found or omits one that is present, the test report will include the following line:

```
The TestMethodsToBeRun list does not match the [TestMethod] methods.
```

The specific unmatched TDS method names are listed near that line. Since we have now made the list match the set of TDS methods, let's see what the report looks like without this reminder.

► Run the tests.

Now, with the name **`NonexistentTest()`** gone, we see the following line near the end of the test report:

```
All listed TDS test methods passed.
```

This message appears only if

- all of the defined test methods in class **`TDS.Test{}`** are selected to be run,
- all of them return a status of "Passed", and
- none of the included test cases are bypassed using a "**`#define RunOnlySelectedTestData`**" directive (see section 4.8.7.1 below).

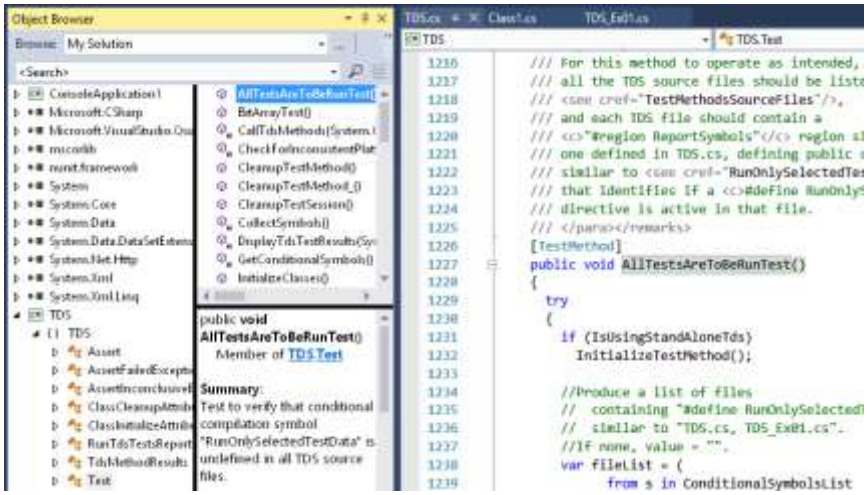
This is intended to serve as a safeguard against accidentally omitting tests that should be applied to the working code. If this message does not appear, an explanation should be in the summary that immediately precedes it in the test report.

► Close the Console window.

4.8.5 Find existing TDS methods [3 minutes]

► Use menu "View, Object Browser" to open the Object Browser window.

► In Object Browser, set the Object Browser Toolbar "Browse:" filter to "My Solution", then expand Project **`TDS`**, Namespace **`TDS`**, and class **`Test{}`**. All of the test methods (along with some other class members) should be listed in the upper-right pane of the Object Browser. Double-click on a class member's name, such as **`AllTestsAreToBeRunTest()`**, to navigate to its definition. IntelliSense information for the selected member appears in the lower-right pane of the Object Browser.



Class View and Resource View allow you to navigate to class members similarly to how Object Browser does, but they don't provide IntelliSense information[44] on these types. Please see section 4.14.17 for other suggestions on navigation within VS.

4.8.6 Use named objects in `testValues[]` [6 minutes]

▶ To illustrate the use of named objects in the **`testValues[]`** array of **`TestableConsoleMethodTest()`** instead of using anonymous objects, in TDS.cs (near line 67) uncomment the directive

```
//#define UseNamedObjectTypeInTestableConsoleMethodTest
```

(Either use menu "Edit, Advanced, Uncomment Selection", or just erase the "//" at the beginning of the line.)

This activates a refactoring that should have no effect on the contents of the test report but that can make the code easier to read and maintain. In return, a bit of work is needed to define a suitable class. Having defined a named type, you have access to IntelliSense documentation based on the type's XML comments and can use named and/or optional parameters in its constructors. Doing this is probably most helpful when each **`testValues[]`** element contains several properties. See section 5.2.9.6 for a discussion and examples; specifically, copying the comments from the properties of the anonymous version is illustrated in the example in section 5.2.9.6.3.2.

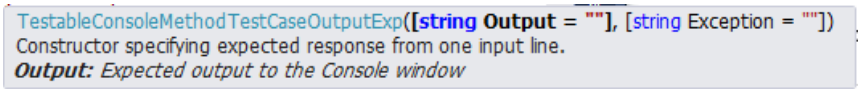
▶ To see an example of these IntelliSense pop-ups, navigate in TDS.cs to method **`TDS.Test.TestableConsoleMethodTest()`**.

I would use the Object Browser (as described in section 4.8.5 above) to locate this.

You should find two copies of the **`testValues[]`** array, only one of which is active.

▶ In one of the **`TestableConsoleMethodTestCase()`** constructors in its **`testValues[]`** array (the copy of **`testValues[]`** that is not grayed out), hover the mouse pointer over the **`OutputExp:`** parameter name, or over one of the parameters in the call to **`TestableConsoleMethodTestCaseOutputExp()`** within the **`Output:`** value.

A pop-up should appear describing the contents of this parameter, for example like this:



You may see similar information by clicking on a parameter or constructor name and selecting menu item "Edit, IntelliSense, Parameter Info". You may also see similar information by entering and erasing a comma in the parameter list of one of the constructor calls.

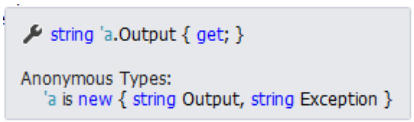
▶ To return to using anonymous objects in this **`testValues[]`** array, comment out the directive

```
#define UseNamedObjectTypeInTestableConsoleMethodTest
```

(near line 67 of TDS.cs).

[44] The IntelliSense information in the lower-right pane, and in pop-ups, is provided by the XML comments in the definitions of the objects. See section 4.14.9 for suggestions on the types of information that might be suitable there.

Having done so, if you look again at the IntelliSense pop-ups on the contents of **`testValues[]`** in **`TDS.Test.TestableConsoleMethodTest()`**, you will still get some information about the objects, but it will be noticeably less informative:



4.8.7 Filter test methods and test cases [15 minutes]

4.8.7.1 Enable filtering

At times, we may want to run only one of the test cases, or a small subset of them, so that we can debug a part of the testable code that would be ignored by the other test cases in the TDS test method. (For more on why one may find this kind of filtering useful, please see section 4.14.4.)

The filtering that we are about to do here will make some of the test report invalid while filtering is enabled, though the report will contain messages alerting us about that. However, while we are interested only in tracing and debugging the working code, it is unimportant that the test report is temporarily incomplete. (We shall restore the full report by the end of section 4.8.7.3.)

► To observe the effect of running only part of the test data specified in a test, in file TDS.cs uncomment the line

```
//#define RunOnlySelectedTestData
```

near the beginning of the file (near line 37), for example by erasing the "//" at the beginning of the line..

► Change the value of **`TestMethodsToBeRun`** to run only the TDS test method **`TestableConsoleMethodTest()`**, commenting out the others.

You may navigate there using Task "**`TODO: TestMethodsToBeRun -- List all TDS test methods to be run.`**".

As we did in section 4.8.2.5, we are doing this to avoid cluttering the test report and wasting time running TDS methods that we don't care about at the moment.

Even though we are now specifying only one TDS test to be run, TDS method **`AllTestsAreToBeRunTest()`** is automatically run as well. It should now fail, to remind us to turn off the filtering when we no longer need it.

► Run the tests, for example via pressing <F5>.

The summary should show the result

```
Passed: 1  Failed: 1  Inconclusive: 0
```

► Close the Console window.

Only the test cases identified in the **`testSelectionList`** string in each TDS method in file TDS.cs were run this time (because of "**`#define RunOnlySelectedTestData`**"), and only in **`TDS.Test.TestableConsoleMethodTest()`** (because of **`TestMethodsToBeRun`**).

Specifically, in **`TestableConsoleMethodTest()`**, the **`testSelectionList`** now allows only test cases whose names begin with "A3" or "B" to be run, so tests "A1" and "A2" are skipped.

Since the test case with **`Id = "A2 Test throwing exception"`** was *not* run this time, any unwanted behavior in the working code that this test case might have revealed was not included in the test report; the report will make it appear that that test case passed.

► To demonstrate this, repeat the change we made in section 4.6.6 to working-code method **`TestableConsoleMethod()`** to cause it to fail.

You may use the Task List to navigate to Task '**`HACK: TestableConsoleMethod() -- Change string to "B UGS" to check test method`**'.

► Having altered the "BUGS" string, run TDS (use <F5>) and observe the test report.

Notice that, according to the test report, **`TestableConsoleMethodTest()`** finishes with a status of "Passed" (which it would not do if the failing test case were included in the **`testSelectionList`**):

```
  The following test method returned a status of Passed:
  - TestableConsoleMethodTest()
```

To remind us that we may be skipping some test cases, TDS test method **`AllTestsAreToBeRunTest()`** fails:

```
  The following test method returned a status of Failed:

  - AllTestsAreToBeRunTest()
      Exception message:
Assert.IsTrue failed.
      *** Some test cases may have been skipped! ***

        To run all test cases in the test methods that are called,
        comment out or otherwise disable any
        "#define RunOnlySelectedTestData" directives
        in the following file(s):
        TDS.cs
```

This is a reminder that test-case filtering is enabled, as a safeguard against assuming that all cases passed, when some otherwise failing cases may simply have been skipped over.

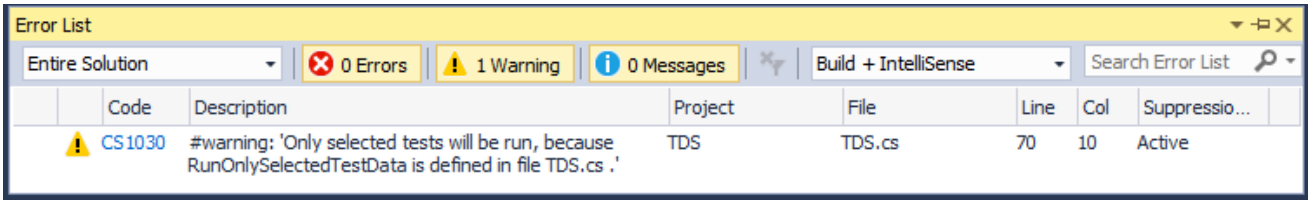
The beginning of the test report also identifies the file(s) in which we have defined the conditional compilation symbol **`RunOnlySelectedTestData`**:

```
***** The following conditional compilation directives are
      included in TDS source-code file TDS.cs:
          #define RunOnlySelectedTestData
          #define TDS_platform
```

Be aware that this "**`#define RunOnlySelectedTestData`**" directive affects all of the TDS methods defined in file TDS.cs. Whenever you enable **`RunOnlySelectedTestData`** filtering in a TDS source file, you should probably consider editing the **`TestMethodsToBeRun`** list to control the filtering more precisely, perhaps by selecting to be run only one or two of the TDS methods defined in that TDS source file.

► Close the Console window.

As another reminder that some failing test cases may be accidentally skipped, in the Error List window (VS menu "View, Error List") a Warning message should appear for each TDS source-code file that contains an active **`#define RunOnlySelectedTestData`** directive.



4.8.7.2 Use the test-case filter to help with tracing

In combination with the TDS method filter in the **`TestMethodsToBeRun`** list, the **`testSelectionList`** allows you to trace directly to a location of interest in the working code, by choosing only one, or a few, of the test cases in the **`testValues[]`** array.

To illustrate doing this, let's suppose that we wish to investigate the handling of the second line of input in test case "B1 Multiple input lines". This will be a line containing the string " **`score`** ".

► In file Class1.cs, in method **`NewCodeNamespace.NewCode.TestableConsoleMethod()`**, place a breakpoint on the line

```
        if (nextLine == "SCORE")
```

in the method body.

You may use the Object Browser to navigate to the definition of **`NewCodeNamespace.NewCode.TestableConsoleMethod()`**, or you may search for it in the Class1.cs editing window. See section 4.14.17 for suggestions on navigation.

► Run to the breakpoint. (Use <F5>.)

In the Locals window (VS menu "Debug, Windows, Locals"), local variable **`nextLine`** should have a value of "SAY HELLO".

► Since we are interested only in the second line of this test case, run (<F5>) to the breakpoint again.

Now **`nextLine`** should have a value of "SCORE", and we could, if we wish, step through the code to observe the processing of this value. VS provides several means of examining the contents of variables, including the Watch window(s), Immediate Window, and Locals window, and the IntelliSense pop-ups that appear (and may be pinned to the editor window) as you hover the mouse pointer over the names of variables.

As it happens, filtering the test cases this time avoided hitting the breakpoint with an uninteresting value only once (that would have been in the first test case, which we bypassed), but if we had had dozens of test cases to skip, then this filtering could have saved a significant amount of time. (No, we're not going to construct dozens of additional test cases in this example just to illustrate the wasted time. In real life, I hope that each of your test cases will serve some valid purpose.)

► When finished with tracing, stop debugging (via menu "Debug, Stop Debugging" or <shift><F5>) and remove the breakpoint.

If you run the tests now, you should again see that **`TestableConsoleMethodTest()`** finishes with a status of "Passed".

4.8.7.3 Disable the filters

► In file TDS.cs, comment out this line (near line 37):

```
#define RunOnlySelectedTestData
```

This allows us to resume running all of the test data specified in the test methods in file TDS.cs ,

- Run the tests.

Again, 1 test Passed, 1 test Failed — but not the same tests as before. This time, it was **`TestableConsoleMethodTest()`** that Failed because we mangled the working code that it was testing, so that the working code failed to raise the needed Exception:

```
  The following test method returned a status of Failed:

  - TestableConsoleMethodTest()
      Exception message:
Assert.IsTrue failed.
TestableConsoleMethodTest(), test case A2 Test throwing exception:
  No Exception was raised in this test case,
  but Exception "Bugs are detected" was expected.
```

- Close the Console window, and in **`TestableConsoleMethod()`**, in the line

```
            if (nextLine.Contains("BUGS"))
```

, change "B UGS" back to "BUGS".

As before, you may navigate there via Task 'HACK: TestableConsoleMethod() -- Change string to "B UGS" to check test method'.

This should restore the method to its correct state, and running the tests now should result in no Failed tests.

- In TDS.cs, de-comment the names of the other TDS methods in **`TestMethodsToBeRun.`**

De-comment only the names listed in the report as having [TestMethod] attributes, and not "NonexistentTest".

- Run the tests.

The summary should now show that all tests Passed:

```
  All listed TDS test methods passed.
```

- Close the Console window.

4.8.8 Test inaccessible function members (optional) [15 minutes]

To be testable using TDS, the working code that you wish to test will need to affect some publicly visible members, for example **`out`** or **`ref`** parameters or **`public`** fields or properties. If the working code is **`private`** or is otherwise not visible to the **`TDS.Test{}`** class, this step, which is optional,[45] illustrates a possible workaround. To skip this step, go to section 4.9.

This is a significant limitation of TDS; for example, making the temporary changes suggested here is not compatible with automated testing (as described in section 4.11). You may be able to perform tests using a **`public`** copy of the working code that you update whenever the real working code changes, so that you can keep the two versions consistent. Alternatively, you may be able to add a wrapper method (section 4.8.8.3) to the working code in its Debug configuration to enable testing, omitting the wrapper method from the Release configuration.

---

[45] This step is optional because in some projects it may not be suitable, or possible, to do what is suggested here. Also, no other part of this Tutorial depends on doing this.

In this step we illustrate using a wrapper method to gain temporary access to an inaccessible object.

4.8.8.1 Set up a private method to be tested

Build the following example by substituting it for the current contents of the **ConsoleApp1** Project's **NewCodeNamespace.Program{}** class.

► In the Program.cs file (the copy that we added to the **ConsoleApp1** Project), temporarily hide the definition of **NewCodeNamespace.Program.Main()** by changing the first line of its definition from

```
    static void Main(string[] args)
```

to

```
    static void Main_1(string[] args)  //HACK: AddSevenTest() -- Rename to Main
```

If the editing window for Program.cs is not open, in Solution Explorer, in the ConsoleApp1 Project, <double-click> on file Program.cs. We intend to remove the example code immediately after this demonstration and therefore will omit most of the usual comments. (Skipping the commenting is slightly risky, but I assume that we can keep track of the changes for the few minutes that the exercise will occupy.)

► Immediately following the

```
    }  // end:Main()
```

line at the end of the definition of the (renamed) **Main()** method, insert the following code:

```
    #region Main() calling private method
    //HACK: AddSevenTest() -- Remove this #region code when done

    private static void Main(string[] args)
    {
      Console.WriteLine("A dozen is " + AddSeven(5) + ".");
      Console.WriteLine(@"
Press the <Enter> key to finish . . .");
      Console.ReadKey(true);
    } // end: Main() substitute

    private static int AddSeven(int addend)
    {
      return addend + 7;
    } // end: AddSeven()

    //HACK: AddSevenTest() -- Add a wrapper method
    #endregion Main() calling private method
```

In this code, the **AddSeven()** method represents the inaccessible working code that we hope to test with the help of a wrapper method. The temporary version of **Main()** calls this **private** method, so that we can demonstrate that it is working. The code is enclosed in a **#region** region to allow us to easily delete it when finished. The included "**HACK:**" Tasks help with finding the added code, so that we can delete it easily.

4.8.8.2 Run it to verify that it works.

► In Solution Explorer, make ConsoleApp1 be the Startup Project.

To do this, in Solution Explorer, right-click the **ConsoleApp1** Project and select "**Set as StartUp Project**" (similarly to what we did in section 4.4.3.1). The ConsoleApp1 name should now appear in bold face.

▶ Run the program (for example, via <F5>).

You should see the following output in the Console window:

```
A dozen is 12.

Press the <Enter> key to finish . . .
```

4.8.8.3 Add a wrapper method to make it visible

▶ Close the Console window and set TDS as the StartUp Project.

Since **AddSeven()** is **private**, a TDS method has no means of gaining access to it directly. However, we could make **AddSeven()** testable using TDS by doing the following:

▶ In file Program.cs, change the line

```
internal class Program
```

to

```
public class Program  //HACK: AddSevenTest() -- Change to internal
```

This needs to be a temporary change, in effect only when TDS is being run. The "**HACK:**" Task is a reminder that the original version needs to be restored later.

▶ Navigate to the "**HACK: AddSevenTest() -- Add a wrapper method**" Task comment and paste the following method immediately following it:

```
    /// <summary>
    /// Public wrapper for private AddSeven() function.
    /// </summary>
    /// <param name="addend"></param>
    /// <returns></returns>
    public static int AddSevenWrapper(int addend)
    {
      return AddSeven(addend);
    }  // end: AddSevenWrapper()
```

The wrapper method, besides merely calling the function member being tested, as we illustrate here, might also provide to its TDS method some additional parameters. These could allow the wrapper method to read or set fields or properties used by the original function code (whose testing the wrapper method is simulating) that would otherwise be inaccessible to the TDS method.

4.8.8.4 Run a TDS test

▶ Generate a TDS test method for **AddSeven()**, as we did in section 4.8.2.

It's unimportant which TDS file we use; let's define it in TDS_Ex01.cs, and, to maintain alphabetical order among the TDS method definitions in that file, I would place this immediately before the definition of **BitArrayTest()**.

Typing "AddSeven" into the **TdsTest** snippet's "TestableFunctionMember" field will generate the new **AddSevenTest()** TDS method.

- Where the statement

```
actual = AddSeven(tCase.Arg);
```

appears, at Task "**TODO: AddSevenTest() -- Provide a suitable calling expression**", change it to something like this:

```
actual = Program.AddSevenWrapper(tCase.Arg);
```

The original definition of **AddSeven()** may remain **private**, and therefore will continue to be inaccessible to any TDS methods. However, a TDS test of the added method **AddSevenWrapper()** can report on the performance of **private** method **AddSeven()**.

The added wrapper method, and the code changing the accessibility of the **Program{}** class from **internal** to **public**, could be placed into conditional-compilation **#if** ... **#endif** regions, to avoid cluttering the namespace and avoid affecting the normal operation of the working code.

- If you wish to run this test, then set TDS as the StartUp Project, change the value of **testValues[0].ValueExp** from 4 to 10, and add **AddSevenTest()** to the list in **TestMethodsToBeRun**.

TDS method **AddSevenTest()** should run and return a status of "Inconclusive".

Alternatively, if you buggify **AddSeven()** to make it return a wrong answer, the test should Fail and display that wrong answer. You may navigate to **AddSeven()** via the Task "**HACK: AddSevenTest() -- Remove this #region code when done**"; the private **AddSeven()** method immediately follows the temporary **Main()** method definition.

4.8.8.5 Undo these changes

- In file Program.cs, in the definition of **class Program{}**, delete the contents of the

```
#region Main() calling private method
```

region to remove the added code. This #region directive immediately precedes the Task "**HACK: AddSevenTest() -- Remove this #region code when done**". Collapse this **#region** to make it easy to remove.

- At Task "**HACK: AddSevenTest() -- Rename to Main**", change "**Main_1**" back to "**Main**" and delete the "**//HACK:**" comment.

- At the "**HACK: AddSevenTest() -- change to internal**" Task, change **Program{}** from **public** to **internal**, and delete the "**//HACK:**" comment.

- (optional) To check that the changes are rolled back properly, set ConsoleApp1 as the StartUp Project and run it (VS menu "Debug, Start Debugging" or <F5>).

The original output should appear, and all of the "**HACK: AddSevenTest()**" Tasks should be gone.

- Erase the **TDS.Test.AddSevenTest()** TDS test definition from TDS_Ex01.cs and erase its name from **TestMethodsToBeRun.**

- In Solution Explorer, set TDS as the StartUp Project.

4.9 Add a #define symbol

In addition to **`#define`** directives, such as **`#define TDS_platform,`** that may already be part of the TDS method files, you may have reason to introduce other **`#define`** symbols and to have them reported in the test report just as the existing ones are.

► In file TDS_Ex01.cs, at the end of the file, after this line:

```
}  // end: Namespace TDS
```

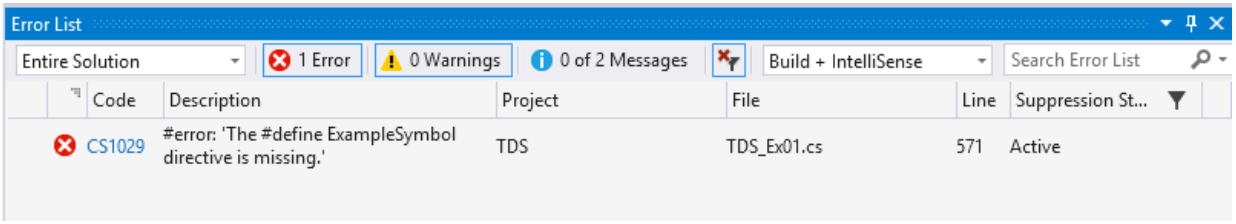
, insert these lines:

```
#if !ExampleSymbol
#error The #define ExampleSymbol directive is missing.
#endif
```

This is intended to create a very obvious effect (the program won't compile) of omitting the new conditional-compilation symbol "ExampleSymbol", if it is indeed omitted.

► Run TDS (press <F5>) to observe the compiler error message "**`There were build errors.`**"; click on "No".

The Error List window will display the description of the error:



► In file TDS_Ex01.cs, immediately following the

```
#region Conditional compilation symbols
```

directive (near line 5), insert this directive:

```
#define ExampleSymbol
```

The new symbol (it's case sensitive) could be defined anywhere else within this **`#region`**, as long as its definition appears in the source-code file before the symbol is used. The location we're using will make the added directive easy to find and remove later, but in this example we could have placed it anywhere before the last **`#if`** directive.

► Run TDS (press <F5>).

No compiler error occurs this time, and a TDS test report should appear that ends with

```
All listed TDS test methods passed.
```

► Close the Console window.

We have a working **`#define`** symbol, but we now want TDS to be aware of it, to include showing its setting in the TDS reports.

► Go to **`#region ReportSymbols.`**

You may navigate there via the Task "**TODO: Conditional-compilation symbols in TDS_Ex01.cs**" (near line 56).

► Navigate to the line immediately before the line

```
    #endregion ReportSymbols
```

(near line 108). You may be able to navigate there by placing your cursor on "**#region**" and pressing "<control>]".

► Enter the name "TdsSymbol" to insert the code snippet TdsSymbol.

Actually, this snippet could be placed pretty much anywhere in the definition of the **Test{}** partial class where a field could be defined, but this **#region** helps organize the definitions.

► Press <tab> twice to go to the first field in the TdsSymbol code snippet, which initially contains TDS_platform; enter the name of the symbol, in this case "ExampleSymbol".

► Press <tab> to go to the next field, containing TDS_Ex01, or click on it. If necessary, enter the name of this TDS source file (without the trailing ".cs").

Since we are editing file TDS_Ex01.cs, we would enter "TDS_Ex01", but since "TDS_Ex01" is the default value of this field, no change is necessary for this instance. If you prefer a different default value for the file name in this snippet, edit file TestMethodSnippet.snippet to change the default and re-import it (see section 4.4.4).

► Press <enter> to close the code snippet.

► Run TDS; examine the beginning of the test report.

A message akin to the following should appear there:

```
***** The following conditional compilation directives are
      included in TDS source-code file TDS_Ex01.cs:
          #define ExampleSymbol
          #define TDS_platform
```

If "**#define ExampleSymbol**" does not appear here, check the definition of **ExampleSymbol_TDS_Ex01**; the name is case sensitive.

Besides displaying this message in the TDS test reports, using this code snippet makes the value of field **TDS.Test.ExampleSymbol_TDS_Ex01** available for use in your TDS methods.

► Remove the

```
#if !ExampleSymbol
#error The #define ExampleSymbol directive is missing.
#endif
```

directives (at the end of file TDS_Ex01.cs) and the

```
#define ExampleSymbol
```

directive (near line 6) that you added earlier.

Running TDS now should produce the same results as before you added these; the **#define ExampleSymbol** line should no longer appear at the beginning of the TDS test report.

► Remove the definition of field **`TDS.Test.ExampleSymbol_TDS_Ex01.`**

We generated this at the end of the **`#region ReportSymbols`** region, near line 108, by using the TdsSymbol code snippet. Assuming you have no plans to use this symbol later, this will clean up unneeded code.

To find it later, this field is listed among the members of TDS.Test in the Object Browser, or you may also navigate to near its definition by using the "**`//TODO: Conditional-compilation symbols in TDS_Ex01.cs`**" Task comment.

4.10 Create a new TDS method file

4.10.1 Make a copy and customize it [15 minutes]

You may find it useful to use several source-code files to contain your TDS methods. They will need unique names, and I suggest using names that are easy to use. For example, you might choose file names that

- reflect the purpose of the working code that the included TDS methods will invoke, or
- reflect the organization of the working code, such as using a separate TDS file for each namespace or each class to be developed or tested, or
- reflect the file structure of the working code, such as using one TDS file for each file folder in the working code, or
- give each of several developers maintaining the TDS code a separate set of TDS files.

► To add another TDS method file, copy the unmodified file TDS_Ex01.cs from folder Demo\TdsSource\ to Demo\TDS\, confirm that you want to keep both copies of TDS_Ex01.cs, and change the name of the new copy to something meaningful.

For comments on this process, please see section 4.14.13.

Note that the file TDS.cs (but not the example file TDS_Ex01.cs) contains some code used by all of the TDS methods, so it should always be included in the TDS Project, even if all of your TDS methods are located in other files, and even if you are using some platform other than TDS to run the tests. For example, the field **`DefaultExceptionMessage`** is defined in TDS.cs and used by default in the TDS method code snippet.

(As with all of the TDS code, you may rename TDS.cs or modify any of its contents to suit your requirements.)

For this example, let's choose the name "MyTdsMethods.cs" for this copied file (a copy of TDS_Ex01.cs, which is now, perhaps, named "TDS_Ex01 (2).cs").

► Change the name of file "TDS_Ex01 (2).cs" to "MyTdsMethods.cs".

► Add this file to the TDS Project.

To do this, as we did in section 4.4.1.1, open the Solution Explorer, <right-click> the TDS Project, click on Add, Existing Item, and browse to "MyTdsMethods.cs". Click "Add".

The file MyTdsMethods.cs in folder …\Demo\Tds\ is now available for editing.

► Open MyTdsMethods.cs for editing in VS, using <double-click> on its name in Solution Explorer.

► Edit the file (via menu "Edit, Find and Replace, Quick Replace", or via "<control>H") to replace the string "TDS_Ex01" with the name of the file ("MyTdsMethods" in this example) everywhere in the current document, with the "Match whole word" option disabled.

There should be eight (8) occurrences; if you find fewer than eight, perhaps "Match whole word" was active. If so, repeat with "Match whole word" disabled.

► Save (via "<control>S") this updated file.

If it's a read-only file, when you try to save your edited version, VS will warn you that it is write protected and will allow you to select "Overwrite".

A

```
//#define RunOnlySelectedTestData
```

directive should exist, initially commented out, near the beginning of each TDS source-code file in the VS Solution, along with some related code: its "**#warning**…" directive and (see section 4.9) the definition of a field with a name similar to "**RunOnlySelectedTestData_TDS_Ex01**". Since you have copied and are updating a file that already contains these, they are present (with suitable new names) in this new file, and you do not need to do anything special concerning them.

► Delete all (originally two) definitions of TDS methods that appear in this file, including their XML documentation comments (comments beginning with "///") that precede them.

These two TDS methods follow the **#endregion ReportSymbols** directive (near line 107) within the **public partial class Test** in namespace **TDS**, and their names are **TestableNoConsoleMethodTest()** and **TimeRoundedTest()**. Collapsing them in Outlining view will make them easy to delete.

The file should now contain about 126 lines, and you may wish to save this edited version as a template for future use (see section 4.14.13 for comments).

► In TDS.cs, following the Task List comment "**TODO: TestMethodsSourceFiles"**, add the file's name, "MyTdsMethods.cs", to the list.

► Run TDS (via <F5>).

If the file name is misspelled in this list, or the ".cs" is missing, a message in likely to appear in the test report similar to the following:

```
***** Error -- The following TDS file has no matching
      "#region ReportSymbols" symbols defined: MisspelledFileName
      (and/or the "#region ReportSymbols" region is
      not up to date) *****
```

Alternatively, an exception is likely to be raised, falsely generating a message about unmatched platform names.

If all has gone well, the test report will look pretty much unchanged from the last good run, except that at the beginning there will appear a new message similar to this (but with the name you've chosen for the new file):

```
***** The following conditional compilation directive is
      included in TDS source-code file MyTdsMethods.cs:
          #define TDS_platform
```

► Close the Console window.

4.10.2 Add a TDS method [6 minutes]

In the new TDS source file, following the "**`TODO: New TDS methods may be placed here:`**" Task, we shall add to it (as we did in section 4.8.2.1) a TDS method to invoke or test a (not yet defined) method **`Abcde()`**.

► In the Task List, navigate to the Task "**`TODO: New TDS methods may be placed here:`**" in file MyTdsMethods.cs.

► On a blank line following that Task comment, type "TdsTest" to insert the TDS method code snippet, <tab> to the "TestableFunctionMember" field (the first field in the code snippet), type the name "Abcde", then press <enter> to close the snippet.

Simulating a TDD-style test-first strategy, this will generate a new TDS method, called "**`AbcdeTest()`**", that will exercise a to-be-defined working-code method, "**`Abcde()`**". We could next, assuming that we know what **`Abcde()`** is expected to do, add some **`Assert`** statements to verify its expected behavior, but for this exercise we shall postpone that task.

This TDS method will serve as an example of how to populate the MyTdsMethods.cs file with TDS methods. We assume that the working-code method to be defined, **`Abcde()`**, will become part of the existing working-code class **`NewCode{}`** in namespace **`NewCodeNamespace`**.

► Update the list of **`using`** statements near the beginning of the file, if necessary.

You may navigate there via the "**`TODO: Usings`**" Task in the Task List for MyTdsMethods.cs.

We would need to add

```
using NewCodeNamespace;
```

to the list of **`using`** statements. As it happens, this name is already present, but in general you would add a **`using`** statement here to specify the name of the namespace in your working code where you will place the function member that you are about to create, and that the new TDS method will exercise.

4.10.3 Add the working-code stub to be debugged/tested [20 minutes]

► Use the Task List to navigate to Task "**`TODO: AbcdeTest() -- Provide a suitable calling expression`**".

► To add a stub for the new method, we can type its class name, "**`NewCode.`**" in front of "**`Abcde(tcase.Arg);`**", to change the statement to

```
            actual = NewCode.Abcde(tCase.Arg);
```

(VS may help with typing the class name after the first few letters.)

► Click on, or hover over, the undefined name "**`Abcde`**". The pop-up menu offers the option "Generate method 'NewCode.Abcde' "; accept that choice.

You may navigate to the new definition using the Edit.GoToDefinition key (<F12>)[46]. The new method's definition should look like this:

```
    public static int Abcde(int arg)
    {
      throw new NotImplementedException();
    }
```

If you run TDS now without doing anything else, you should see near the end of the test report the message

```
The following TDS method has a [TestMethod] attribute
    but is not in the TestMethodsToBeRun list:
      TDS.Test.AbcdeTest()
```

► Add the new TDS method's name, **`AbcdeTest()`**, to the list in **`TestMethodsToBeRun`** and run TDS again.

You should see in the test report a summary similar to this (assuming you have several successful TDS tests):

```
Passed: 5  Failed: 1  Inconclusive: 0
```

where test method **`AbcdeTest()`** is the one that failed, as we expected, displaying this message:

```
AbcdeTest(), test case 01 Sample test:
  The expected exception should start with " No exception was thrown".
  This unexpected exception was thrown:
    "The method or operation is not implemented."
```

► Close the Console window.

Now the working-code method **`Abcde()`** and its TDS method **`AbcdeTest()`** may be developed together, or either one may be developed first and the other updated later (soon afterward, I suggest) to match it..

Since our TDS method template looks for a returned value of 4, if you replace the **`throw`** statement in **`NewCode.Abcde()`** with

```
      return 4;
```

and run TDS (using <F5>); the "Failed" status should become "Inconclusive".

► We no longer need **`Abcde()`**, so delete the definitions of **`Abcde()`** and **`AbcdeTest()`** (along with its XML comments).

► Remove the name of **`AbcdeTest()`** from the list in **`TestMethodsToBeRun.`**

Running TDS now should again generate the message "**`All listed TDS test methods passed.`**".

4.11 Automate the testing

Setting up automatic testing may be of use when the working code and its associated TDS methods have become fairly stable. For example, perhaps the working code inputs information from external files or databases, and it will be useful to learn if changes in those data have caused some unwanted effects in the

[46] See section 4.14.17 for other suggestions on navigation.

working code's behavior. You might want to run the TDS methods from time to time to check for these. I suggest that coding changes in the working code, such as apparently harmless refactorings, should be accompanied by updating and running the corresponding TDS methods at the time the refactorings are made, instead of postponing them, but if you do wish to run tests in the background, this section presents some ways to do that.

4.11.1 Set up script files [5 minutes]

TDS.exe may be run to exercise your testable code independently of any other test platform, if you used "**`#define TDS_platform`**" directives when you built TDS.exe, and if you specified Project TDS as the Startup Project of the VS Solution in the Solution Explorer window.

► Uncomment (activate) all the "**`#define TDS_platform`**" directives in your TDS source files, if any of them are commented out. Also comment out (deactivate) any active "**`#define NUnit_platform`**" directives.

The beginning of the test report should specify that each of the TDS files has an active "**`#define TDS_platform`**" directive.

If they don't match, your Solution may fail to build. Even if it does build, an exception will likely be raised when you try to run TDS. For example, the exception message might read, in part,

```
System.ApplicationException was unhandled
  Message=Only one of the following platforms should be used:
#define TDS_platform  (used in TDS.cs, TDS_Ex01.cs)
#define Microsoft platform  (used in MyTdsMethods.cs)
```

► Set Project TDS as the Startup Project of the VS Solution, if necessary.

To make it easy to do automated testing using TDS, sample command-line scripts are provided in the TdsSource.zip file. These script files don't actually do much that running TDS.exe by itself won't do, but they illustrate the use of exit codes to control program flow. For example, you might write a script to run TDS tests on an automatic schedule, save the reports, and send an email message containing the report whenever one of the tests fails.

Script file cmdTds.bat is intended to be used with the Windows® Command Prompt (Console) window, and file psTds.ps1 is for use with Windows® PowerShell (either in its command-prompt window or in its Integrated Scripting Environment). To allow the PowerShell script to run, set the execution policy if needed (see section 4.5.1.5.1.2.1).

► Use Windows® Explorer or a command-prompt command to copy "cmdTds.bat" and "psTds.ps1" from your ...\Demo\TdsSource\ folder into your ...\Demo\TDS\ folder .

4.11.2 Run tests independently of VS or NUnit

4.11.2.1 Run TDS.exe [6 minutes]

4.11.2.1.1 USE A WINDOWS COMMAND PROMPT

► Open a Windows Command Prompt window or a Windows PowerShell window (or both).

See section 4.5.1.5 for suggestions on opening one of these.

References in the following instructions to a "command-prompt window" apply to either one. Examples using scripts for both of them are given in section 4.11.2.2 below.

► Use a "**`cd`**" command (it's not case sensitive) to navigate to your ...\Demo\TDS\bin\Debug\ folder.

This is two levels below the ...\Demo\TDS\ folder to which you copied the script files. See section 4.5.1.5.2 for suggestions on entering the pathname.

- For instructions on running TDS.exe from the command-prompt window, use the command

```
.\TDS -?
```

(it's not case sensitive).

Called via this command, the TDS.exe program will display the following in either Command Prompt or PowerShell:

```
***** Test{} class's static constructor has been called.

TDS runs the current unit-test suite.

Syntax: Use
   TDS -?
to display this Help information.

Use
    TDS
to display the test results and wait for keyboard input.

Use
    TDS -nopause
to display the test report without pausing.

Use
    TDS <file>
  to write the entire test report to the specified <file>.
For example,
    TDS .\temp.txt
  will write the entire test report to file temp.txt .
Any text file name is suitable, but the ".txt" is required.
If the specified file exists, an error message is displayed.

In Windows PowerShell, also specify the path, as in ".\TDS".

Exit code values:
    0 = All tests passed.
    1 = Tests to be run did not match the defined test methods,
        or no test was run.
    2 = At least one test was Inconclusive.
    3 = At least one test Failed, or some other error occurred.


Press the <Enter> key to finish . . .
```

- As described in the instructions, use command

```
.\TDS -nopause
```

to run the tests and display results.

The displayed results will be essentially identical[47] to the report that you have already seen, ending with

```
  All listed TDS test methods passed.

***** (End of test summary)
```

4.11.2.1.2 USE WINDOWS EXPLORER

► Instead of typing the command, you may use Windows Explorer (also called File Explorer) to navigate to your Demo\TDS\bin\Debug folder, then double-click on TDS.exe .

A Windows Command Prompt (a Console) window will appear, containing the TDS test report. This window contains essentially the same report that you have already seen.

When the TDS test report appears in the window, you may examine it and optionally copy it to the Clipboard. To copy it there, select the window and type "<alt-space>ES<enter>". You may then paste the report into a text file for later analysis or comparison.

► Press <enter> to close the window when finished.

4.11.2.2 Use a script file [18 minutes]

You may run the current tests from a command prompt using a script file, using either the Windows Command Prompt or Windows PowerShell.

The supplied script files work similarly to each other, though psTds.ps1 (for Windows PowerShell) also provides for automatic generation of time-stamped report-file names, whereas cmdTds.bat (for Windows Command Prompt) does not do that. Either one may be used as an example of calling TDS.exe from a script file.

► If necessary (see section 4.5.1.5.1.2.1), set PowerShell's **`ExecutionPolicy`** to allow you to run unsigned scripts.

Here we shall demonstrate the use of cmdTds.bat and/or psTds.ps1 to produce a test report. You may run both at the same time, if you wish, to compare their behavior.

4.11.2.2.1 NAVIGATE TO THE TDS FOLDER

► Open a command-prompt window, or switch to one that is already open.

► Navigate (using a "**`CHDIR`**" or "**`CD`**" command; commands are not case sensitive) to your ...\Demo\TDS\ folder.

See section 4.5.1.5.2 for suggestions on using this command.

If the window is already open in the ...\Demo\TDS\bin\Debug\ folder (where we went in section 4.5.1.5.2), you may use this shorter command, using a relative pathname:

```
cd ..\..
```

4.11.2.2.2 LIST "HELP" INFORMATION FOR A SCRIPT

To see instructions for running a TDS script file, run the following command (not case sensitive):

► [Command Prompt]

---

[47] It's identical except for minor details like time stamps.


```
cmdTds
```

- [PowerShell]

```
.\psTds -?
```

4.11.2.2.3 VIEW A TEST REPORT

To view the current test report, use the following command:

- [Command Prompt]

```
cmdTds console
```

- [PowerShell]

```
.\psTds
```

4.11.2.2.4 WRITE THE TEST REPORT AS A TEXT FILE

To write the test report to a new text file, use a command that includes an identifying part of the name of the file to be written. For example, a tag like "Mon" or "Tue" could identify the day of the week on which the test was run. Use a command similar to these:

4.11.2.2.4.1 USING [COMMAND PROMPT]

- [Command Prompt]

```
cmdTds Mon
```

4.11.2.2.4.2 USING [POWERSHELL]

- [PowerShell] To write the test report to a new text file, use a command like

```
.\psTds Mon
```

4.11.2.2.4.3 USING EITHER

With either command, the report is written to file Tds_Mon_P.txt (if tag "Mon" is used) unless a file with that name is already present.

A message similar to this appears, naming the generated text file:

```
TDS test report has been written to file Tds_Mon_P.txt .

Exit code: 0 = Passed -- all tests passed.
```

The "_P" in the generated file name indicates that the report shows that all of the tests passed.

If a file with a name that conflicts with the chosen name is already present, a message similar to this will appear:

```
The name "Mon" may cause a conflict with
  one of the following files; please choose another name:

Tds_Mon_P.txt
```

4.11.2.2.4.4 WRITE THE REPORT WITH A TIME-STAMPED FILE NAME

(This feature is not available using the Command Prompt script cmdTds.bat.)

- [PowerShell] To write the test report to a new text file with a time-stamped name, use command

```
.\psTds x
```

The report is written to a new text file with a name[48] containing a sortable, numeric form of the date and time it wa s written. Sorting these files by name will automatically also sort them by the time they were written.

```
TDS test report has been written to file Tds_170204_084648_P.txt .

  Exit code: 0 = Passed -- All tests Passed.
```

4.11.2.2.5 CLOSE THE WINDOW

► [Both] To close the window, use the command "**exit**" or click on the "X" in the upper-right corner.

4.11.2.2.6 USE POWERSHELL FROM A WINDOWS® COMMAND PROMPT WINDOW [2 MINUTES]

► [Command Prompt] Assuming the PowerShell **ExecutionPolicy** permits it, you may use a command in the Windows Command Prompt window such as

```
PowerShell -command .\psTds x
```

to run your tests. This technique offers some additional options; use a

```
PowerShell -?
```

command for more information.

4.12 Hide TDS when done

► When development of a function member in your working code is sufficiently complete, comment out its TDS method's name in the **TestMethodsToBeRun** list in TDS.cs .

This will keep it from being included in the TDS test reports, except for being mentioned in the "mismatched" list at the end.

► If you wish to also remove the TDS method's name from the list following the heading

```
The following TDS method has a [TestMethod] attribute
    but is not in the TestMethodsToBeRun list:
```

near the end of the TDS test report, but you do not want to discard all of its TDS method code, also comment out the TDS method's "**[TestMethod]**" attribute.

► Whenever no TDS test methods in the Solution need to be run, set some other VS Project as the start-up Project and rebuild the Solution.

The "scaffolding" analogy is relevant here — the scaffolding for a building is removed after the walls are built, but it is kept available for use later, for when the building needs to be painted or repaired. Here, we want the code to be able to work properly on its own, without further involvement from TDS.

► When the TDS platform appears to be no longer needed, remove its Project from the VS Solution.

Section 5.1.2.12 contains a list of some ways in which working code having associated TDS methods may be run.

4.12.1 Consider keeping the TDS methods after the Solution is complete

Even after the new/revised working code appears to be complete and working well as part of your overall VS Solution, I claim that there still may be reason not to throw away its TDS methods, which you might be able to

[48] If you honestly want to generate a report file having the specific name "Tds_x_P.txt", either edit script file PsTds.ps1 to allow that name, or specify some other file name and rename it after it is written.

use in the future, as long as it remains possible for the working code's requirements to change or for new bugs to appear.

You will likely no longer need any of the TDS infrastructure that is in the TDS.cs file if you have migrated all of your testing from TDS to a different unit-test platform, such as VS Test or NUnit. Even then, the TdsTest code snippet, or your version of it, may help in setting up new test methods.

The tested function members really may be working perfectly right now (although proving that mathematically might be impractical), but it's possible that changed requirements may later call for changes or extensions to what they do. Also, an apparently harmless change to existing working code somewhere else in your solution (maybe in a place over which you have no control) could have some unexpected effects on your new function members' behavior, and those effects could affect other code that depends on them, or that uses objects that they touch. You can save time and trouble by having test methods available to check that at least the pre-existing behavior of these function members hasn't been corrupted.

Another possible benefit of maintaining the test code is that it provides detailed documentation that may not be available anywhere else. The expected results, as specified in the test cases, might be able to give a future developer (maybe you, two years hence!) details about exactly how the tested function member is expected to behave in special circumstances, if the XML comments or other documentation is not specific enough to clarify this.

4.13 What's next?

As illustrated in these examples, when your project outgrows the capabilities of the TDS unit-test platform, you should be able to use these same `[TestMethod]`s on a test platform that provides more extensive testing facilities.

All of the source-code files supplied in TdsSource.zip may be freely modified to suit your needs. For example, you may decide to use the example TDS test methods as a basis for your own test-method templates and create code snippet files to allow you to insert them into your test-method code. Adapt the script files to automate your tests.

What you do with any of these, of course, is at your own risk, sorry about that. But I have tried to keep the contents simple enough, including lots of comments, to make them easy to adapt. Have fun with them, and I hope you find them useful.

- The contents of the Demo\ folder that we built in section 4.3.3 are no longer needed, so you may delete it and its contents if you wish.

At this point, I expect that you are ready to use TDS with no further guidance. However, if you wish to see some examples of projects that could be developed with the help of TDS, please proceed to section 5.

Instead of deleting the VS Solution that you built during the Tutorial, you may instead use the VS Solution that you have built here for building the examples in section 5; just add a VS Project to contain the working code, and add the needed TDS methods to file TDS.cs .

4.14 Comments on the Tutorial

If the results you got from the Tutorial are not what you expected, perhaps some of the explanations in the following sections can help. You might think of it as an "FAQ" section (except that nobody has actually asked these questions, so I can't claim that there's anything "Frequent" about them).

4.14.1 Purpose

The instructions for running the Tutorial were intended to show how to use TDS, but some details on the rationale are omitted. These comments are an attempt to explain why some of those steps were included in the Tutorial, but are placed here so that they don't interrupt the flow of the instructions. (They are not a necessary part of the instructions, and as it's possible that you have no interest in what is discussed here, they are easily skipped.)

The application built using the Tutorial illustrates using TDS with methods under development that are essentially complete but may need to be modified in the future. For example, suppose that an existing method in working code, that has a corresponding, existing TDS test method, needs to be updated to meet new requirements, or someone has discovered a bug in it. Presented with such code, you would probably want to begin by compiling it, so that you can check that it is consistent and free of compile-time errors (as we had in section 4.3.6.3), though it might be wise not to actually try to run it before you know what's in it. You could also run its TDS method to see that it can, at least some of the time (for example, on its "happy path"), produce expected results. We begin our use of TDS by running some TDS methods against our simulated working code (section 4.4.3).

4.14.2 Requirements statement for code being developed

The design of the example working code was originally expressed as a set of requirements statements, which might in real life be expressed in an email message, or as notes from a conversation with a customer. In these examples, those requirements were copied into C# comments, then recast as C# XML comments, and those became the documentation for the working code. The original requirements statement was no longer needed.

For more discussion of project requirements, which are glossed over in the Tutorial, please see sections 4.14.9.1.1 (Content of XML comments; requirements statement) and 5.1.2.2 (State the purpose of the project).

Not all requirements are easily expressed as XML comments, but those that are expressed in that form are easily located and maintained, since they stay with the code.

4.14.3 Assert statements in TDS and other test platforms

The basic TDS unit-test platform provides only three forms of the `Assert` test:

- `Assert.AreEqual()` passes iff the `ToString()` values of the two objects match. (This is not the same as testing that the objects themselves have the same values.)
- `Assert.IsTrue()` passes iff the given Boolean expression is `true`.
- `Assert.Inconclusive()` always returns a status of "Inconclusive" (meaning "unfinished").

Other platforms provide many more choices; for example, the Microsoft Unit Test platform provides dozens of overloads of `Assert.AreEqual()`. Although I claim that it is possible to simulate much of that functionality using `Assert.IsTrue()`, and that you are welcome to add your own overloads or otherwise change these definitions, you may find that the versatility offered by the Microsoft Unit Test platform or NUnit or some other platform fits your needs better. Therefore, the TDS methods have been designed to make it easy to use them, unchanged, with other unit-test platforms, as well as with the basic TDS platform.

Even though the TDS tests have limitations, they possess something that the other platforms' `Assert` methods may not possess — the ability to be customized to suit your needs, such as by allowing you to change the format of the messages that they display in test reports. Also, by default, the TDS methods share a common, easily maintainable structure (as mentioned in section 1.10.3.2), so you may find them convenient even if used in conjunction with a different, full-featured testing platform, and the TDS environment provides some helpful functions such as a common filtering system for test cases (section 4.14.4).

4.14.4 Filtering test cases in TDS

The purpose of this feature is to help select specific test cases for use in debugging. For example, you may wish to use a set of values that leads to the execution of a rarely used branch in a function member's code, especially if several branches are present, and you don't want to have to wade through other test cases that have nothing to do with the path that interests you. You could do this by setting breakpoints, but it may also be useful to be able to specify some specific set of data (from one of the test cases in the TDS method) that will create the conditions that interest you. This may be done via test-case filtering, as illustrated briefly in section 4.8.7.2 in the Tutorial and in more detail in section 5.2.6.11.

Except when you have a specific need to suppress some of the test cases, and especially during testing later in the project, quietly omitting some of the test cases might give a false indication that some test cases passed when they were not even run. This effect is demonstrated in section 4.8.7.1, in which a function member containing a (known) program bug falsely appears to pass its test. Therefore, test-case filtering should be used only when it is helpful, and not on a routine basis. Therefore, TDS provides some warnings that are active whenever a test-case filter is enabled (as shown in section 4.8.7.1).

Running a TDS method with all of the test cases enabled gives the same result as running it with **`RunOnlySelectedTestData`** undefined, except for the compiler warning and the failing **`AllTestsAreToBeRunTest`** method.

4.14.5 Calling static constructors

The purpose of updating the expression in the definition of `callStaticConstructors` at the beginning of `InitializeClasses()` is to ensure that the static constructor of each type referenced there has been called at a predictable time, in the order that you specify, so that the TDS test results will be consistent. Doing this might be of value if some static constructors have global side-effects, such as changing the value of a publicly accessible field.

Note that, after TDS is removed from your system, any protection offered by this mechanism will be gone, so it may be appropriate for you to either not use this feature at all, or to intentionally vary the order in which the constructors are called by this statement. (In the Tutorial, we bypass this feature in section 4.4.1.3.)

In the present examples, the static constructor of each of the referenced types has an obvious side-effect — it sends a message to the Console, and, if TDS is running, that message also appears in the TDS test report. This makes it easy to observe the results. All of these constructors are run before any of the TDS methods is run, and (at leat in Debug mode) in the order listed in `callStaticConstructors`. This is somewhat similar to the way in which TDS methods are run in order according to the contents of `TestMethodsToBeRun`, as we do in section 4.8.2.5, except that a static constructor is run only once.

Of course, the static constructors of all types used in the program will be run eventually, but this feature helps ensure that the order is consistent, in case some of the side-effects might affect the results of tests or of each other. To observe the effect, run TDS with one or more of the expressions commented out, or reordered, and compare the resulting TDS report with the unmodified version.

It is possible that you wish to omit some types in this list, for example so that you can trace into a static constructor from a TDS method. If so, include in the `callStaticConstructors` expression only those types whose static constructors you would like to have run at the beginning of each TDS test session, before any of the TDS methods are run.

4.14.6 Running NUnit

If you run the NUnit project a second time, your output in NUnit's Text Output window will differ – you will no longer see the messages that the three static constructors have been called. NUnit apparently considers them to have been built, so it doesn't rebuild them. However, if you run "File, Reload Tests" and rerun the tests, the static constructors will again be called (once).

If it seems surprising that NUnit may not run tests in a predictable order while TDS does, remember that TDS is not intended to run in a multithreaded environment, but NUnit is. With multiple threads active, the order of events is likely to be unpredictable. You can still use TDS for assistance in exercising new or changed function members, but such actions as profiling, detecting race conditions in multiple threads, stress testing, etc., are beyond the scope of this *TDS User's Guide* (and of TDS).

4.14.7 Setting up a stand-alone TDS Project

Many of the examples shown in this *TDS User's Guide* begin with a VS Solution that already contains working-code Projects, and we add a TDS Project to that Solution. Those examples usually involve some code that is intended to illustrate a feature of TDS, but which is not a necessary part of a Solution.

However, you may instead choose to begin by constructing a VS Solution containing nothing but a TDS Project, with no associated working code, to which you can add TDS methods that will (TDD-style) invoke the to-be-created function members of types that will eventually comprise, or at least be a part of, the Solution's working code. You would add other Projects to the Solution to contain the working code to be developed, and add TDS methods (to the TDS Project) and the TDS methods' corresponding working-code function members (to the working-code Projects) to build the Solution.

These instructions may instead be used to add a TDS Project to an existing VS Solution, if the Solution contains accessible (for example, **`public`**) function members that you wish to trace into or test.

The following steps show how to…

- add a TDS Project to an existing Solution without adding any example code, or
- set up a VS Solution containing nothing but a TDS Project (and some disposable, fake working code).

If you are adding this TDS Project to an existing VS Solution, then these steps are similar to those in the Tutorial, beginning at section 4.4, but omitting the examples and some of the details.

4.14.7.1 Add a TDS Project to a new or existing Solution

► Set up Visual Studio as described in sections 4.3.1-4.3.5 of the Tutorial.

Briefly, do this:

- Check that Visual Studio ("VS") is installed (section 4.3.2)
- Create an empty file folder for your VS Solution (or use an existing folder that contains some working code) (section 4.3.3). In the description of this example we'll call the folder "Demo\", but existing code will likely occupy a folder with some other name.
- Extract the contents of the TdsSource.zip file (section 4.3.4)
- Configure Visual Studio to edit C# code (section 4.3.5)

► If you are using an existing VS Solution, and it already contains a project called "TDS" that is unrelated to the programs described in this *TDS User's Guide*, then you may need to delete the "TDS" Project from the Solution, or rename it. Also delete the Demo\TDS\ folder and its contents from your file system; we shall replace it soon.

Following this instruction might be impractical for you, if you have an extensive Project called "TDS" that can't easily be renamed; the best workaround I might suggest would be to change the string "TDS" to something else, in all of the files in the TDS distribution set[49].

4.14.7.1.1 CREATE AN EXAMPLE WORKING-CODE PROJECT IF NEEDED

If you are creating a new VS Solution, instead of adding TDS to a Solution containing some existing working code, follow the instructions in section 4.3.6.1, "Create a new Visual Studio Project". (Return here upon

[49] I regret any inconvenience that this may cause; it is the result of my trying to keep the structure of the files as simple as possible while also making them easy to use.

opening the Solution Explorer window.) This will set up a VS Solution containing a mostly empty VS Project[50], to which we can add a new Project to be called "TDS".

The VS Project that we set up using section 4.3.6.1 to contain our simulated working code is a Console App, but the TDS methods are intended to be usable with any of a variety of Project types. For example, section 5.4 illustrates using TDS with a Visual Basic Project.

We use the Console App type in this example to keep the output easy to read and easy to compare with the printed version shown in these instructions.

4.14.7.1.2 ADD A TDS PROJECT TO THE SOLUTION.

► With the VS Solution Explorer window open, right-click on the Solution (not the similarly-named Project), then in the pop-up menu select Add, New Project. Choose a Visual C#, Windows Classic Desktop, Console App (or "Application") Project. Replace its default name (such as "ConsoleApplication1" or "ConsoleApp1" or "ConsoleApp2") with "TDS" (as we did in section 4.4 in the Tutorial).

This "TDS" Project will house your TDS methods and some infrastructure code that supports them.

► Set the Project's "Location:" to be the file folder (called "Demo\" in the Tutorial examples, and which by now likely contains a folder named "ConsoleApp1") that is to house the new Solution, perhaps using the "Browse..." button, as illustrated in section 4.3.6.1.

► Click "OK" to set up the new TDS Project.

The Demo\ folder should now contain a folder named "TDS", along with the "ConsoleApp1" folder, both of which should be visible in the Windows File Explorer window.

► When the project has been created, open the VS Solution Explorer window.

If the Solution Explorer is not visible, use menu "View, Solution Explorer" to open it. The Solution should now contain two Projects, "ConsoleApp1" and "TDS".

► In the "TDS" Project, delete the Program.cs file. In response to the warning "'Program.cs' will be deleted permanently.", click **OK**.

► In the VS "Solution Explorer" window, right-click on Project **TDS**; choose menu "Add, Existing Item… "

► In the "Add Existing Item – TDS" window, browse to the folder (such as Demo\TdsSource\) containing the TDS files extracted in section 4.3.3, and select file TDS.cs . Click **Add**.

This file should now appear in Project TDS in Solution Explorer, and the Demo\TDS\ folder should now contain a copy of file TDS.cs.

Do not add example file TDS_Ex01.cs to the TDS Project. It is used in the Tutorial, in section 4.4.1.1, but it is not needed for other purposes, such as using TDS with real working code or with the examples in section 5. Accidentally adding it would not be harmful, but it would be unnecessary.

4.14.7.2 Delete the TestableConsoleMethodTest() method

► Use menu "View, Task List" to open the Task List window.

---

[50] In the Tutorial, we populated this Project with some example code to be exercised by the TDS methods during the Tutorial, but here, we add nothing to the default main program until after we set up the TDS Project.

► In the "**`TODO: TestableConsoleMethodTest() example -- Delete the contents of the following #region if`**..." Task, follow the suggestion and delete the contents of the **`#region`**, along with this Task List comment.

To locate this, open the Task List window (via VS menu "View, Task List") and double-click on the list item "**`TODO: TestableConsoleMethodTest() example -- Delete the contents of the following #region if`**".

Collapse the **`#region`** and delete its contents.

We use the example TDS method **`TestableConsoleMethodTest()`** in the Tutorial, but it is usually not needed. Like the contents of the TDS_Ex01.cs file, it is included in TDS.cs only to support the Tutorial, so removing it does no harm. Anyway, whatever you do with it can be reversed, since you are editing only a copy of TDS.cs, not the original.

A bit of housekeeping is needed; most of the following changes are located at Task List comments.

► At the "**`TODO: InitializeClasses(), static variables`**" Task, comment out or delete the example Boolean expressions in the statement.

The comments near there provide guidance on adding other expressions as needed.

► At the "**`TODO: Usings`**" Task, delete the "**`using NewCodeNamespace;`**" statement.

You will likely need to replace it with a reference to the namespace of some working code, so leave this Task List comment in place for now.

4.14.7.3 Delete the reference to TDS_Ex01.cs

► At the "**`TODO: TestMethodsSourceFiles –`**" Task, delete the line containing "TDS_Ex01.cs".

If you do not plan to use any additional TDS files, you may also delete this Task List comment.

► At the "**`TODO: TestMethodsToBeRun`**" Task, delete or comment out the names of TDS methods in the **`TestMethodsToBeRun`** list, but leave the blank-filled string, and its "////TODO:" comment, in place.

As you create new TDS methods, you will need to insert their names here to be able to run them using TDS.

However, if you use a different unit-test system (as we do in section 4.5 of the Tutorial), the **`TestMethodsToBeRun`** field will not be needed, and it may remain a blank-filled string.

Having edited TDS.cs to delete the unneeded example code, you may find it useful to save this edited version of TDS.cs for use in future projects.

4.14.7.4 Check for errors via a "smoke test"

► Set Project TDS as the Startup Project..

In the Solution Explorer window, right-click on the "TDS" Project and select "Set as StartUp Project", as we did in section 4.4.3.1. The "TDS" Project name changes to bold-face type.

► Optionally, hide "**`Assert`**" exception messages, as we did in section 4.4.2, 'Hide "unhandled exception" messages'.

► Run the Solution (using <F5>).

The purpose of doing this is to reveal any typographical mistakes or other errors that may have cropped up during editing.

A(n empty) TDS test report should appear in a Console window, looking similar to the following:

```
***** Test{} class's static constructor has been called.
***** InitializeClasses() has begun running.
***** The following conditional compilation directive is
      included in TDS source-code file TDS.cs:
          #define TDS_platform


***** TDS.Test.AllTestsAreToBeRunTest()
***** InitializeTestMethod() was called at 2017-02-27T15:42:31.4589532-06:00 .
***** CleanupTestMethod() is complete.
***** (End of test)


***** The final test was completed at 2017-02-27T15:42:31.5279789-06:00 .
***** CleanupTestSession() is complete.



_________________

*****  This was a test run.  The following results were generated. *****


_________________

Passed tests
  The following test method returned a status of Passed:
  - AllTestsAreToBeRunTest()


_________________

No called test method returned a status of Failed.


_________________

No called test method returned a status of Inconclusive.


_________________

All TDS methods that have [TestMethod] attributes
    are in the TestMethodsToBeRun list.

All TDS methods that are in the TestMethodsToBeRun list
    have [TestMethod] attributes.


_________________
Passed: 1  Failed: 0  Inconclusive: 0


_________________

  All listed TDS test methods passed.
```



```
***** (End of test summary)


Press the <Enter> key to finish . . .
```

The **`AllTestsAreToBeRunTest()`** method is always included in a TDS test report. (See section 4.8.7.1.)

OK, it's not totally empty, but this report tells you nothing useful except that TDS is ready to be used. If a report like this does *not* appear now, check the Error List window (VS menu "View, Error List") for messages.

► Close the Console window. (Click on it, then press <enter>.)

4.14.7.5 Add working code

At this point, TDS is ready to be used, although without any working code for it to call, it won't do anything useful.

If your VS Solution does not yet contain any working code, the next step is to add new Projects to contain the working code (as we did in section 4.3.6.1, "Create a new Visual Studio Project"), and create new TDS methods to exercise that working code, perhaps as shown in section 4.8.2.1, section 5.1.5.1.1, section 5.2.6.1.1, section 5.3.6.1.1, or section 5.4.3.

4.14.8 Ignoring unhandled exceptions

Caution: I suggest that *only* the five types of exceptions specifically mentioned in section 4.4.2.1 be ignored as described in section 4.4.2.2.

I expect that if you encounter any other types of unhandled exceptions that generate pop-up windows, those might indicate serious problems that need attention and should therefore *not* be routinely hidden. This is why (never mind that VS requires some extra work to do this) I suggest that you go to the trouble of using the Exception Settings window to hide only the five specified types. Otherwise, it may become too easy to assume that any pop-up box that appears is merely being raised by another unit-test exception that should be ignored, and you may miss seeing an important one.

Incidentally, I consider it good coding practice to use exceptions (aside from those used to report unit-test results) *only* to report unusual conditions that need special attention. For example, one usually doesn't intentionally try to divide by zero (you already know what the result of that would be, so there's no need to actually do it), so such an attempt is clearly an accident, and it makes sense to respond to it by raising an exception. It makes less sense to use exceptions for ordinary processing, such as signaling the end of a sentence while processing text, which would be a frequent event. Such signaling can be done by using parameters or returned function values.

The unit-test platforms' use of exceptions to signal the results of tests falls somewhere in the middle. With a new test method, "Failed" or "Inconclusive" results are pretty routine. Later, most of the results should be "Passed" (= no exception raised by **`Assert`** methods), so by then a non-Passed status would indeed be exceptional, and thus worthy of special attention. At any rate, I'm following the example of the masters in this use of exceptions to signal unit-test results.

4.14.9 Documentation (XML) comments

4.14.9.1 What to include in comments

4.14.9.1.1 CONTENT OF XML COMMENTS; REQUIREMENTS STATEMENT

(This section describes the types of information you may want to place into the XML comments. For a discussion of using them once they are in place, please see section 4.14.9.3.)

Often, a new project will begin with a specification or statement (perhaps brief and informal, or existing only in your mind) of its purpose, a description of whatever you intend it to be able to do when it is complete. You know what you want the proposed function member, field, or code that is to be updated to accomplish, or someone (your customer or boss) has told you what the code needs to do.

It is probably helpful to express this statement in written form, both to assist your own memory as you develop the code, and (if it's not your own project) to help you and your customer agree on what needs to be done. The statement doesn't have to be fancy, but it should express the results in some testable or observable form, allowing you (or someone) to have a way to distinguish when it is working as expected from when it is not.

I suggest stating the purpose in operational terms (= identifying what it is expected to be able to do, and under what conditions) in some kind of design document[51], stored in a convenient location. For example, I sometimes start with a simple text file, or a compiler source file containing only comments, that describes the expected inputs and desired results. This might be adequate by itself, or it could be supplemented with some non-verbal information[52]. For the present examples, plain text will be good enough.

In these instructions, we assume that all of the design information (the statement of requirements) can be expressed in the form of text strings. If this is possible in your project, and that text is located within the code to be developed or updated, it should be not very difficult to keep the code and its documentation consistent with each other. Anyone with access to the code would also automatically have immediate access to the statement of requirements, making it easy to make the code meet changes in the requirements, such as customizing the handling of special cases, or needing to throw a new exception.

We can think of the XML comments on a field or function member as a record of its requirements. In some development projects, these XML comments might be the only current documentation describing the interfaces between the working code and the outside world. Ideally, we can make the XML comments grow in parallel with the working code and the corresponding TDS methods, keeping them consistent. If we do that faithfully, then

1) as the code evolves to correct errors or to satisfy new or changed requirements, then
2) its XML comments will continue to describe what it does currently, to make it easy to use, and
3) its TDS methods will continue to check for errors in the updated code.

As you might infer from the example code in TDS.cs, I usually put a fairly large amount of detail, including examples, into my XML comments, largely because I like having the descriptions in a place where I can easily update them while I am working on the code — comments are less helpful, maybe even misleading, if they are not kept consistent with the executable code.

---

[51] For a small project, my "design document" often consists of comments in the project's source code.

[52] Examples of non-verbal documentation that might be relevant could include tables of values, drawings, equations, videos, databases, flowcharts, CAD models, geometric diagrams, or bitmap images that would be impractical to include in program source code, though the code's comments could make reference to such documents.

For remarks relating to copying text from other documents into XML comment code and formatting it for the Object Browser, plase see section 5.1.5.1.4.

4.14.9.1.2 USING ORDINARY C# COMMENTS WHERE XML COMMENTS ARE INADEQUATE

However, if some of your design information is not suited to C# XML comments — it might be in the form of non-verbal expressions, or be too lengthy to be included in the code — you may wish to maintain these in a separate document associated with the code files, and refer to that document in the XML comments. For example, the mathematical derivation shown for the Fibonacci sequence example in section 5.2.4 below, "Analyze the problem mathematically", should probably be in a document separate from the code, since much of it is unlikely to change, and a programmer will normally need to use only the concluding result, not the (detailed) reasoning leading to it.

In these instructions, we assume that all of the design information (statement of requirements) can be stated as text. If this is possible in your project, and that text is located within the code (for example, as an extended block of ordinary C# comments) to be developed or updated, it should be easy to keep the code and its documentation consistent with each other. Anyone with access to the code would also have immediate access to the statement of requirements, making it easy to make the code meet new requirements, such as customizing the handling of special cases, or needing to throw a new exception.

Note: In XML comments, only a limited number of HTML or XML tags are effective; for example, tables are not supported for display in VS's Object Browser. However, links are supported and may be used to refer to associated documentation.

4.14.9.1.3 USING ORDINARY C# COMMENTS WHERE XML COMMENTS ARE NOT SUPPORTED

XML comments are not always suitable as a means of recording specifications. For example, local variables of a function member are not visible outside the function member, and XML comments don't help to document them (and are not supported by the VS editor). Typing "///" for a local variable generates only an ordinary single-line comment, and VS doesn't create a template for it.

Even there, one might use comments to add information (if it isn't obvious from context, such as the member's name) to describe uses or limitations of a local variable.

I think that using ordinary C# comments formatted to mimic the contents of C# XML comments is especially helpful in the definition of a local variable whose value is a **`delegate`**, such as a **`Func()`** or **`Action()`**. You may see an example of such comments in the definition of local variable **`RunTest`**, which is an **`Action`**, in the code in section 5.2.8.5.2; the code using **`RunTest`** calls it as if it were an ordinary method. Other examples that come to mind are XML-valued strings (containing named elements) or regular-expression (RegEx) patterns (possibly containing named substrings that might benefit from descriptive comments)[53]. The string format specifications (used in calls to String.Format() that do not utilize the new interpolated format) in these examples also contain ordinary C# comments, both to connect the parameters with their placeholders in the format via comments such as "//{2}" and, sometimes, to describe complex expressions.

In the definition of a **`Func()`** or **`Action()`**, when what that object does can be a bit complex, there may be parameters and/or return values to document, and I may wish to refactor the it later into a method with XML comments containing similar information. For a **`Func()`** or **`Action()`**, I normally both give it comments that mimic the XML comments that I would use on a method definition, and encapsulate the comments and the definition using a **`#region`**. With comments like these, I don't get the advantages of IntelliSense support,

[53] Both XML code and RegEx patterns also provide mechanisms that allow internal (non-C#) comments.

but the format is familiar, and since the definition is located close to where I need to use it, it's easy to find the comments when I need them.

Keeping the body of the function member reasonably short can keep the definitions of local variables, and their associated (ordinary, non-XML) comments, physically close to where we need to use them so that we can find them easily.

In my programs, by the time one of my methods occupies more than about the amount of code I can see at once on the screen, I begin to consider refactoring some of it into a separate method, or at least hiding some of it using a **`#region`**, to help keep the code legible. Also, if I see that the same, or similar, code appears in more than one place in either my working code or my TDS methods, I consider refactoring it using an expression common to all of the places where it occurs, as we did in section 4.8.3.5, to possibly make maintenance of the common code easier, and sometimes to discover bugs in the process of refactoring the code.

4.14.9.2 How to update XML comments

We could add new XML comments to a field or a function member even before writing much code for it, as soon as we know how we intend to use it, or what changes we want to make to it. Given this information, we could use the XML comments as a sort of mini-specification of what the code is expected to do or how it will be used.

For example, suppose we want to create and test new method "**`Abcde()`**", as we did in section 4.10.3. The method stub that we had VS generate there, based on an expression invoking it in a new TDS method, looked like this:

```
public static int Abcde(int arg)
{
  throw new NotImplementedException();`
}
```

Even without using the VS Quick Action "Generate method 'NewCode.Abcde' ", we could instead have directly entered this, or similar, code into our working code. (The new stub could be the beginning of any kind of function member, not necessarily a method.)

Having established a name for the new function member, we can begin to add XML comments to it. For example, if we type "**`///`**" on the line immediately above this definition of new method "**`Abcde()`**", VS generates a template for its XML comments.

```
/// <summary>
///
/// </summary>
/// <param name="arg"></param>
/// <returns></returns>
```

This template includes (empty) XML elements for the method's parameters and return value, into which we can enter descriptive comments. Using the template will help avoid opportunities for misspelling the parameter names or the XML tags. (However, as with the method template, you are free to supply your own XML comments, instead of using the generated template.)

Into the **<summary>** or **<remarks>** elements' text we could insert the text-friendly part of our specifications; see section 5.1.5.1.4 for a discussion of how to reformat it as XML, escaping the special characters "&", "<", and ">".[54]

If we wish to use a TDD-style development approach (as described in section 1.8.1), we could next create a TDS method to invoke the yet-to-be-defined working code, including test cases to address the specifications, and include code in the TDS method to invoke the new function member.

If we wish to include in the XML comments some specific examples of content (for a field) or output (from a function member), we may be able to avoid some work by first writing or modifying some of the working code, then executing some part of it and copying the resulting values of fields or expressions into a text editor, from which (edited for legibility) we can paste them into the XML comments and/or into the TDS method's test cases[55] as examples of correct output or content. Once there, the examples can easily be compared wi.th the output from the working code. If a difference appears, the code can be corrected or the example updated to match the improved version. (See section 5.1.5.1.3 for an example of how the XML comments might be updated.)

$$$Merge this paragraph with the preceding one:

Wherever I use XML comments, I want them to be useful without spending much time on maintaining them. If I plan to include examples of the displayed text in the XML comments, I usually first take care to edit the code to produce nicely formatted (and correct) updated text, putting most of my effort there. When I'm happy with that, I run it to display the output in a text box or on the Console, then copy the text from the Console window (see section 4.8.3.2) to a text-editor file, where I reformat it (see section 5.1.5.1.4) for use as XML and paste it into the XML comments. Besides accuracy, qualities such as style and visual appeal are relevant to the content and format of the output, so that's where I begin working on needed changes. In contrast, I want the XML comments to reflect only what the code actually does, and copying the output accomplishes that.

4.14.9.3 When and how to view XML comments

4.14.9.3.1 AS A SEPARATE DOCUMENT

Outside of VS, the C# compiler can generate an XML document from the XML comments; compile with the /doc option to process the XML documentation comments to a file. (To keep this document file current, you would need to repeat this process whenever the XML comments are changed.) To create documentation based on this compiler-generated file, you can create a custom tool, or use a tool such as Sandcastle (see http://sandcastle.codeplex.com/).

To generate this file from within VS, in the Solution Explorer, open the TDS Project's properties window, and on the Build tab, in the "Output" area, check "XML Documentation File".

---

[54] If any part of the XML comments does not conform to valid XML syntax, the likely result is that none of the contents will appear either in the Object Browser or in the IntelliSense pop-ups. Please see section 5.3.15.3.2.1 for an example. Apparently, no other harm results, but debugging the faulty XML may not be easy, as the compiler provides very little assistance. I usually debug a faulty XML comment by selectively erasing parts of it until the remainder again appears in the Object Browser, then I re-insert the parts that I removed. For me, the culprit is often an unescaped special XML character.

[55] Copying output from the Console window to XML comments or to test cases in **testValues[]** is illustrated in section 4.8.3.2.

4.14.9.3.2 WHILE EDITING CODE

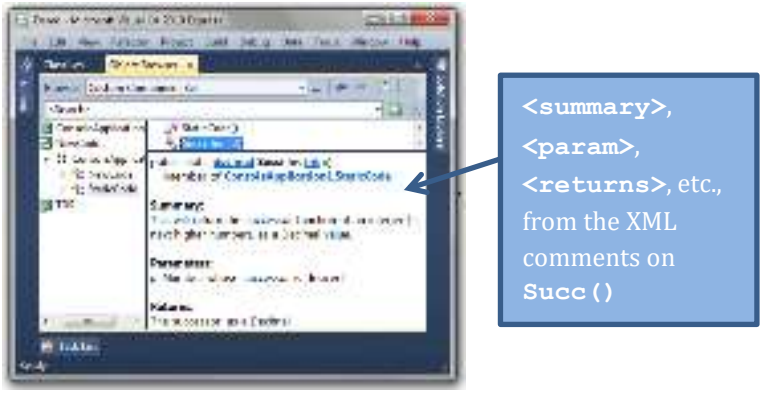
Even better than enabling us to automatically generate documentation, and more usefully to us during development, whenever we are editing the code using the VS editor, the material in an identifier's XML comments is readily visible as we use the identifier. The contents of XML comments' **<summary>** XML elements are visible in the Object Browser window and in the IntelliSense popups that appear, for example, when you hover the mouse pointer over an identifier, or while you are typing an expression (as illustrated in the nearby figures). Even within nested types, which may not appear in the Object Browser window (depending on their accessibility), the XML comments still appear in IntelliSense pop-ups (for example, when specifying parameters for a constructor).

While we are writing or editing code in VS we can also see the comments in the IntelliSense pop-ups

- as we type the method name,

- or when we hover the mouse pointer over the method name,
- or as we specify a parameter.

The comments, including the contents of the **<remarks>** element, also appear in VS's Object Browser window, which displays them in an attractive, legible format. I sometimes use the Object Browser to examine the XML comments to verify that they are formatted properly, for example by not containing any improper unbalanced "**<**" characters. (VS's somewhat similar "Class View" window, in contrast, apparently does not display any XML comment information.)

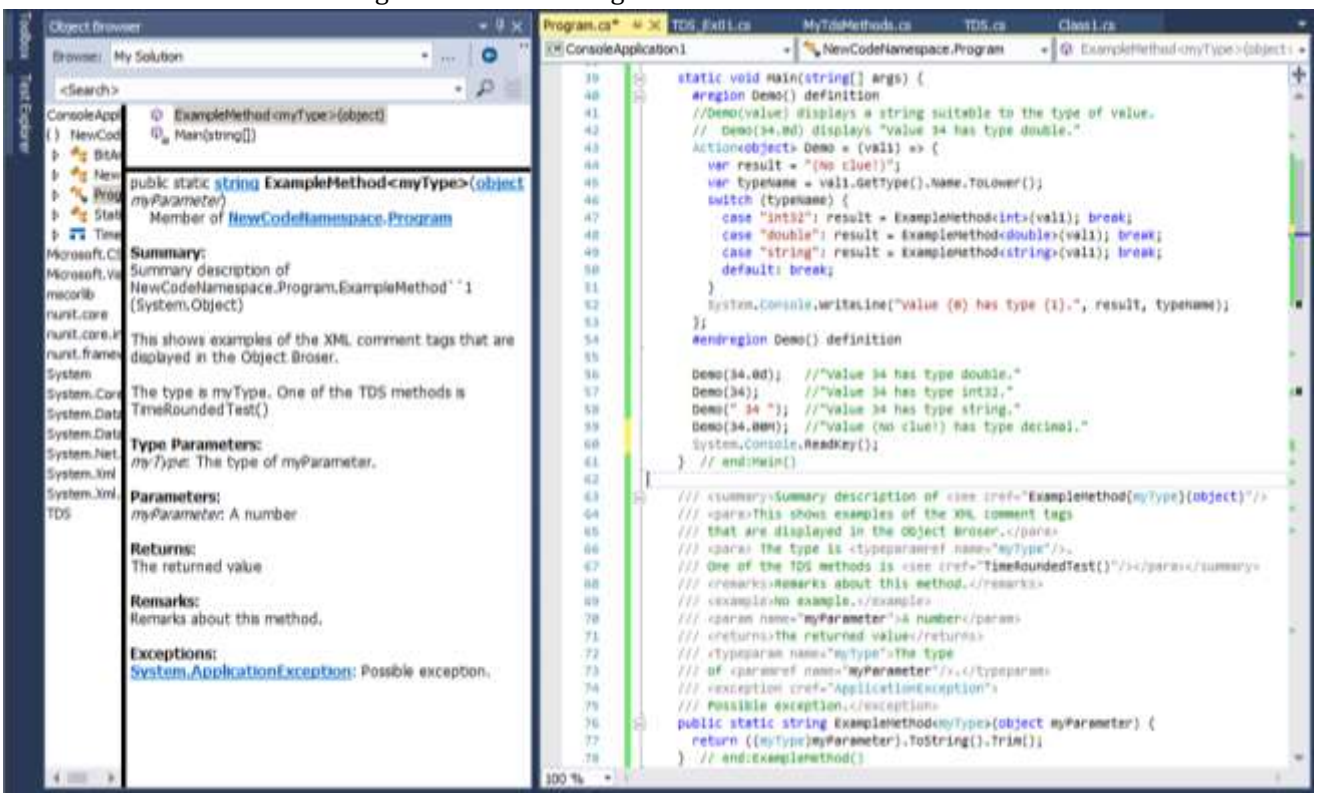


Comments in **<example>** elements do not seem to be reflected in the Object Browser and not in IntelliSense, so I usually put information about examples into the **<remarks>** element so that I can see them there, or into the **<summary>** element if I also want to see them in the IntelliSense popups. (Be careful about putting too much information into the **<summary>**; you don't want to unduly clutter the IntelliSense pop-ups. Include only enough to remind you of what the object does and how to use it. More space is available in **<remarks>**.)

Within the **<summary></summary>** element, most tags, such as **<list>** or **<item>** or **<code>**, are ignored by Object Browser and IntelliSense, but the included text is displayed. Similarly, **<paramref>**, **<typeparamref>**, **<see>**, and **<seealso>** seem to have no effect on the Object Browser display. However, if I do an "Edit, Refactor, Rename" operation in the VS editor to rename an identifier, the editor recognizes the names in the **@name** or **@cref** attributes of these elements as instances to be renamed.

In the **<summary>** section, **<para></para>** tags can be used to force new lines.

Here is a screen shot illustrating several of these tags.



If you wish to reproduce this example so that you can experiment with it, the source code displayed in this screen shot follows:

```
    static void Main(string[] args) {
      #region Demo() definition
      //Demo(value) displays a string suitable to the type of value.
      //  Demo(34.0d) displays "Value 34 has type double."
      Action<object> Demo = (val1) => {
        var result = "(No clue!)";
        var typeName = val1.GetType().Name.ToLower();
        switch (typeName) {
          case "int32": result = ExampleMethod<int>(val1); break;
          case "double": result = ExampleMethod<double>(val1); break;
          case "string": result = ExampleMethod<string>(val1); break;
          default: break;
        }
        System.Console.WriteLine("Value {0} has type {1}.", result, typeName);
      };
      #endregion Demo() definition

      Demo(34.0d);    //"Value 34 has type double."
      Demo(34);       //"Value 34 has type int32."
      Demo(" 34 ");   //"Value 34 has type string."
      Demo(34.00M);   //"Value (No clue!) has type decimal."
      System.Console.ReadKey();
    }  // end:Main()

    /// <summary>Summary description of <see
cref="ExampleMethod{myType}(object)"/>
    /// <para>This shows examples of the XML comment tags
    /// that are displayed in the Object Broser.</para>
    /// <para> The type is <typeparamref name="myType"/>.
    /// One of the TDS methods is <see
cref="TimeRoundedTest()"/></para></summary>
    /// <remarks>Remarks about this method.</remarks>
    /// <example>No example.</example>
    /// <param name="myParameter">A number</param>
    /// <returns>The returned value</returns>
    /// <typeparam name="myType">The type
    /// of <paramref name="myParameter"/>.</typeparam>
    /// <exception cref="ApplicationException">
    /// Possible exception.</exception>
    public static string ExampleMethod<myType>(object myParameter) {
      return ((myType)myParameter).ToString().Trim();
    }  // end:ExampleMethod()
```

Contrary to what the instructions seem to state, the contents of the following elements in XML comments are displayed in the Object Browser: **<param>**, **<returns>**, **<remarks>**, **<typeparam>**, and **<exception>**.

4.14.9.4 When to update XML comments

As with any documentation, we will need to keep the XML comments consistent with any changes in the code, so that they don't become misleading and thereby hinder development, instead of helping it. Whenever the visible behavior of a function member changes, we need to check its XML comments to verify that they still

reflect the behavior of the function member, and, if necessary, to update the comments to reflect the changes. (It might be better to have no comments at all than to leave false or misleading ones in the code!)

Internal details such as the names of local variables are not visible outside the function member, so there is normally no need to mention them in the XML comments, and the editor does not support XML comments on these[56]. Even so, it is possible that information on some aspect of the coding that is not normally visible, such as advice on efficient usage, might be worth including in a **`<remarks>`** element of the XML comments.

[56] Any comments on these that begin with "**`///`**" are treated as ordinary C# comments.

4.14.10 Adding properties to testValues[]

You may want to adopt some conventions for naming and organizing the properties in the elements of the **`testValues[]`** array, since much of the activity of updating the TDS method will involve these elements. Some suggestions for property names and comments follow; choose an order and naming convention that are easy for you to use. These suggestions apply especially to anonymous elements of **`testValues[]`**; converting these elements to objects of a named type can give you more freedom[57] in specifying their values. Plese see sections 4.8.6 and 5.2.9.6 for suggestions on specifying and using a named type for these elements.

Even though (I claim) a property's name should suggest what its purpose is, I try to keep the name short enough to be easy to type, read, and remember, whereas the comment on that property can be somewhat longer and may be used to add some helpful detail; please see section 4.14.10.2 below for suggestions concerning the comments.

The order in which the properties in this initializer are listed is unimportant, except that if you have more than one anonymous-type element of **`testValues[]`** defined, the same properties (same names, same types) must appear in the same order in each element, so any changes made to one must be made to all of them[58].

4.14.10.1 Names of `testValues[]` properties

So long as **`testValues[]`** contains only a single element, it is trivially easy to add new properties or to rearrange their order. The outer braces enclose a tiny little namespace in which even duplicates of identifiers appearing elsewhere in your code (though not any C# reserved words) are permitted to be used as property names.

Since you may eventually have dozens of properties in **`testValues[0]`**, it may help to organize them as you add them, for example by listing them alphabetically, or by listing inputs before expected outputs.

In the examples in this *TDS User's Guide*, I list the **`Id`** property first, to make it easy to find. Following the "**`Id`**" property, I usually list all the input values (alphabetically), followed by the expected values of outputs (alphabetically). I use the "**`Exp`**" suffix in property names like "**`ValueExp`**" to identify them as expected values rather than inputs, but that suffix has no other significance.

4.14.10.2 Comments on `testValues[]` properties

I usually keep the property names short, to make it easy to type them and to read them in code, but a short name often does not provide much insight into such qualities as how the property is expected to be used, what it means, or what its extreme values might be. There may eventually be many properties in the **`testValues[]`** objects, and it may not be obvious to me in six months (or to someone else now) how each property was intended to be used. Also, although most of the properties used in examples in the Tutorial have simple types (such as **`int`**), the new function member may use or generate more complex objects (such as XML), and they might be difficult to understand at a glance. (We'll illustrate this in section 5.3.6.2, "Generate a value for the input parameter".) Therefore, I almost always add a descriptive comment for each property in **`testValues[0]`** at the time I define that property, while its intended usage is still obvious to me. (Please see the table in section 5.2.9.6.3.2 for a discussion of various types of comments in **`testValues[].`**)

---

[57] For example, we could then omit some of their names or alter the order in which they are specified.
[58] This doesn't apply if you use a named type in the **`testValues[]`** array; see section 4.8.6 for a Tutorial example and section 4.14.11 for a discussion.

I sometimes include temporary "**`//TODO:`**" Task List comments in **`testValues[0]`** suggesting special values that might be useful in debugging or testing, if I happen to think of such values when I don't have the time to add **`testValues[]`** test cases that use them, or to otherwise properly account for them. (Please see section 4.14.16 for suggestions on using Task List comments.)

4.14.11 Using named types in testValues[]

By default, the objects in **`testValues[]`** have an anonymous type, but you may instead give them a named type. The Tutorial, in section 4.8.6 , shows a completed conversion of the anonymous objects in **`testValues[]`** to a named type.

Using an anonymous type for the elements of **`testValues[]`** is convenient when **`testValues[]`** contains only one or two elements, as when we set up a new TDS test method. For example, while **`testValues[0]`** is the only element present, its properties can be added, modified, or deleted with no more effort than doing those operations on local variable declaration statements. Even with two to four instances of the anonymous type defined,

- all of the defining information is visible in one place,
- fewer lines of code may need to be entered than with a named type (assuming that we count the named type's definition, which must be provided somewhere else in the code), and
- the descriptive information (expressed as comments) for each property is visible at or near the code where the instance's property is given its value.

However, a named type may work better for you if you have several instances, or if they have many properties. I usually convert the elements of **`testValues[]`** to a named type after adding maybe three or four elements (each specifying a test case) to the set.

Converting the elements to a named type allows more flexibility in defining data for the test cases. For example, you can assign default values to the object's properties or omit some of the property names in the constructors. You can also apply XML comments to the properties in a named class, making them visible via IntelliSense in VS. While you are using the anonymous type, all of the elements of **`testValues[]`** must specify the same property names in the same order, with no default values and little[59] IntelliSense help.

The example in section 4.8.6 shows the results of converting the anonymous objects in **`testValues[]`** to named-type objects but does not provide much detail about the process. An example illustrating how one might do this conversion is shown in section 5.2.9.6.

[59] Only the name and type are shown in the IntelliSense for an anonymous-type property; no comments appear.

4.14.12 Using Debug mode

All of the exercises in the Tutorial are expected to be run in "Debug" mode, not "Release" mode. This suppresses the code optimization that the compiler would normally apply, so that the generated code can closely track the source code and make tracing easy to follow, and so that we can set and use breakpoints. Anyway, any reduced efficiency due to the use of unoptimized code is likely to be immaterial if we are stopping to examine variables from time to time.

4.14.13 Multiple TDS source files

Using two separate TDS source files, TDS.cs and TDS_Ex01.cs (along with file MyTestMethods.cs, added in section 4.10.1), in these examples was intentional. It was meant to illustrate distributing the TDS methods among several files to facilitate maintenance by multiple developers, perhaps using a source-code control system. On a small project, you may find it more convenient to put all of your TDS code into the TDS.cs file, adding other TDS source files (as in section 4.10.1) only when the original becomes unwieldy.

Having added a new TDS source file and removed the example TDS methods and their XML comments from it, you may choose to save it as a template for creating other TDS source files. Using the template will avoid the work of removing the example TDS methods, but the rest of the process will be about the same:

- each copy of the simplified file template will still need to be given a unique name,
- the old name will need to be replaced with the new one eight times in the copy,
- the new file name will need to be added to the **`TDS.Test.TestMethodsSourceFiles`** string in file TDS.cs, and
- the new file will need to be added to the TDS Project in VS.

You may find that you prefer to entirely eliminate TDS_Ex01.cs from your TDS project, using only copies of your updated TDS file template instead, and if you did that, no harm would result. I used the "TDS_Ex01" name partly to try to make it look quite different from anything that might already exist in your projects, while still associating it with TDS.

4.14.14 Example project name

The name of the VS project, "Demo", and the original name of the folder containing the working files for these exercises (short for "Demonstration"), is intended not to conflict with the names of any production code you might be working with. Using this name makes it easy, in the discussion, to refer to the demonstration folder that we create to house the example project. In section 4.3.3 you are explicitly invited to rename the folder, so that you can avoid conflicts with names in your own projects. If the "Demo" name on the VS Solution gives you problems as well, you may use VS's Solution Explorer to rename it.

Of course, in your own work, you may choose just about any name you wish to use. Even in the TDS code, most of the names used may be changed. (Exceptions include things like the name of the "`[TestMethod]`" Attribute, which is the same name used by NUnit and VS Test to identify test methods.)

4.14.15 Customizing TDS

You will probably want to modify the contents of the public-domain TDS.cs file to suit your own needs, such as removing unneeded comments and the two example TDS test methods, but the rest of this *TDS User's Guide* (section 5 and following) doesn't contain anything essential to using TDS in your development projects; it consists mostly of examples and illustrations, along with some propaganda in favor of writing good software documentation, unit testing your working code, and communicating with your customer.

4.14.16 Task List (`//TODO:`) comments

4.14.16.1 Tracking unfinished work

To save time or to avoid interrupting our flow of thought as we develop working code and its TDS methods, we may freely add "**`//TODO:`**" Task List comments to identify code that is not yet written or is unfinished. This might include planned **`Assert`** statements or changes to the properties in **`testValues[]`** in a TDS method, and we can replace those comments with real code when convenient. All of the "**`//TODO:`**" and "**`//HACK:`**" comments in the VS Solution's source code can be found in VS's "Task List" window[60] and, being comments, they have no effect on the compiled program.

Whenever I add a Task List comment, I usually begin it with the name of the function member to which it belongs, to help me remember where it's located in the code, and to help keep related Task List items together. I follow the name with a brief description of what needs to be done. In the code, I sometimes follow that "**`//TODO:`**" comment with ordinary comments that provide more details.

In a new TDS method constructed via the "**`TdsTest`**" code snippet (as we did in section 4.8.2.1) are several comments beginning with some version of "**`//TODO:`**", which initially serve to identify C# code that needs to be customized. These are intended to be removed when the indicated changes are made, though I sometimes leave some of them in place as bookmarks (as suggested in section 4.8.1.1) until after I have finished working in those areas.

4.14.16.2 Types of Task List comments

Like "**`//TODO:`**" or "**`//HACK:`**" comments, "**`//UNDONE:`**" comments may also be listed in VS's "Task List" window. You may also add other types of Task List Tokens of your choosing via VS's menu "Tools, Options, Environment, Task List". The Task List may be sorted or filtered by priority (low, normal, or high), by file or line number, by project, or by name. We can navigate to a specific task comment in the code by double-clicking on its line in the Task List.

4.14.16.3 Removing Task List comments

When the work done to satisfy a "**`//TODO:`**" or other Task List comment is complete, we can delete the comment (or its "**`TODO:`**" label) to remove it from the Task List.

4.14.16.4 Using Task List comments as place markers

Some of these comments in the TDS files may serve as place markers that you may wish to keep active, though you might consider using some more suitable name than "TODO" for their Task List Token if you make a habit of making them permanent. For example, since I have a continuing need to make changes to the **`TestMethodsToBeRun`** field, I leave its Task List comment in place after changing its contents.

4.14.16.5 "Inconclusive" Task List tasks

I sometimes merely comment out the **`Assert.Inconclusive()`** statement at the end of a TDS method that I think may need to be updated soon, as a reminder to activate it again at that time. For example, it is possible that the behavior of the working code will need to be changed to correct bugs in the code or to conform to new requirements. The working code's corresponding TDS method will likely need to be updated to match, and while it is being modified, an **`Assert.Inconclusive()`** statement would be useful as a reminder that it is

[60] The Task List window may be opened via VS's menu "View, Task List".

being updated. I think of this statement as similar to yellow "**DO NOT CROSS THIS LINE**" tape at a construction site that warns passersby of unfinished work.

Note that there is no need to add an **`Assert.Inconclusive()`** statement if the working code is merely refactored without any intended effect on its behavior, since the original, unchanged TDS method should still work as before. This TDS method can be used to help verify that the refactoring was in fact harmless, as it was intended to be.

4.14.17 Navigating in Visual Studio[61]

Besides using menu "Edit, Find and Replace, Quick Find", and selecting "Current Document" or "Entire Solution" in the drop-down list to locate a desired method or field (such as the **`TDS.Test.TestMethodsToBeRun`** string), VS offers a variety of other navigation mechanisms.

To navigate to existing test methods, I suggest opening the Object Browser window (via VS menu "View, Object Browser"), the Solution Explorer ("View, Solution Explorer"), the Class View window ("View, Class View"), or the Resource View window ("View, Other Windows, Resource View"). Double-click on a TDS method's name to navigate to its definition. Of the windows mentioned here, apparently only the Object Browser displays IntelliSense information on these types (the contents of the XML comments in the definitions).

If you know which file contains the object you seek, open that file in an editing window and select the desired object using the drop-down lists at the top of the editing window.

You may set Bookmarks via menu "Edit, Bookmarks, Toggle Bookmark", then navigate to them via actions such as "Edit, Bookmarks, Next Bookmark". However, it's easy to define so many bookmarks that finding the one you want among all the others can become difficult.

In the Task List, besides the "//TODO:" and "//HACK:" comments used in the TDS source code, you may define others using menu "Tools, Options, Environment, Task List" and adding your own new Task List Token using the Name box. The text you include in these comments can make them easy to identify, and they can be alphabetized. I begin most of those in the TDS source code with a short name describing their location.

To locate the definition of a desired object, you may use VS's Object Browser window or Class View window, though they don't help much with local variables. If a reference to the object is visible in an editing window, you may navigate to its definition by using the Edit.GoToDefinition key (<F12>), or by right-clicking on its name and choosing that pop-up menu option.

[61] This section deals with navigating within a VS Solution. For navigating within this *TDS User's Guide* , please see section 2.3.4.

4.14.18 Use of "////" in comments

Some of the comments in TDS.cs begin with "////", and as far as the compiler is concerned, these are merely ordinary comments. (In contrast, "///" can be special in some places, as it introduces a line in a C# XML comment.)

They do have a purpose, however. These comments begin with "**////**" instead of "**//**" so that, if you use VS menu "Edit, Advanced, Uncomment Selection" or a similar action (once), these lines will remain comments, for example changing

```
      ////TODO: SuccTest() -- Remove the Assert.Inconclusive()
      ////  statement after this [TestMethod] is working:
//      Assert.Inconclusive(
//@"Verify the correctness of AbcTest() .");
```

to

```
      //TODO: SuccTest() -- Remove the Assert.Inconclusive()
      //  statement after this [TestMethod] is working:
      Assert.Inconclusive(
@"Verify the correctness of AbcTest() .");
```

If you allow them to retain their "**////**" prefix, this will help to distinguish them from the nearby code that you want to have active at times and inactive at others, such as the names of TDS methods in **TestMethodsToBeRun**, or the **#define** directives near the beginning of TDS.cs . If you accidentally remove too many copies of "**//**", then the (former) comments are interpreted as (probably erroneous) C# code.

5 Examples – long, picky details

The preceding Tutorial (section 4) was intended to touch on all of the intentional[62] features of TDS. The present section goes into a bit of additional detail on the nits & grits of adding TDS methods to a Visual Studio Solution, using a variety of examples. You can use TDS effectively without looking at any of these, but these are intended to elaborate a bit on some of its features, providing a rationale for utilizing those features as you develop/debug working code.

In the Tutorial, we assumed that any needed components were already complete, without mentioning the decisions that went into developing them. In the examples here in section 5, we'll simulate more of the development process a bit, including some justification for making some of the choices involved. You still won't need to write any actual code; everything you will need is provided, but you will be able to play with the code if you wish, to see the effects of changes that you make to it.

Although the examples in section 5 build on the results of the Tutorial (section 4) and assume some familiarity with it, they are intended not to depend on any of the other examples in this section, so if you are interested in only one or two of them, you may go directly there. The following examples are included here; click on the name of any of them to go to its description:

- Example: Adding a new method, Succ(), to a new Solution

  In this one, we define a new method, then trace into it or run it using TDS.

- Example: Adding a new method, Fib(), to a new Solution

  Here we use multiple calculations to check each other's results. Optionally, we view a mathematical analysis of the algorithms used.

- Example: Modifying an XElement via a new method

  We define and run tests using complex values (XML-valued objects).

- Example: Testing a Visual Basic Project

  We trace and test code in a non-C# method using TDS.

As in the Tutorial, these examples assume that you are using a recent version of Microsoft Visual Studio (sometimes referred to here as "VS"); see section 4.3.1.

In an attempt to keep the documentation as simple as possible, each of these examples assumes that you are creating the code for the example as a new VS Solution, instead of adding it to an existing Solution that already contains a TDS Project. If you still have a usable VS Solution containing a TDS Project, then just use that for these examples and skip over the setup instructions. Otherwise, follow the instructions as shown here to construct the example; if you do this, the output that your Solution generates should match the illustrations shown in these instructions (for example, no other TDS methods will be mentioned in the generated test report).

Each of these examples will begin with setting up a VS Solution containing nothing but a TDS Proejct, created according to the instructions in section 4.14.7, never mind that that would not normally be a realistic way to

[62] Any *un*intentional features are probably either bugs that need to be corrected, or else features to be documented. There are no known "Easter eggs" in the TDS software.

proceed. You may then add code, as directed, and run the Solution to see the results of applying a TDS method to some simulated working code.

5.1 Example: Adding a new method, Succ(), to a new Solution

The following description is a bit long winded; its purpose is to provide an explanation of the reasons behind some of the steps in the Tutorial (section 4).

We shall construct here a simulated VS Solution containing a TDS Project. We shall add to it a TDS method along with a corresponding new working-code method (to be called "Succ()"), both of which we shall develop concurrently, then remove the TDS method when the working code is complete, a few minutes later. (This is somewhat accelerated over a usual development schedule.)

Many of the ideas mentioned here are likely obvious to you (or you might disagree with some of them), so you may prefer just to skip ahead to section 5.1.4, where we actually begin to build the example code, and refer to the discussion here only if the reason for some step is not obvious.

5.1.1 What we shall do in this example

The steps in this example are pretty elementary, but they are intended to illustrate the use of TDS as an aid in development, before (or without) doing any automated testing. As I mention elsewhere, I think that one should actually do some testing, too, but this example is intended to show that it is possible to benefit from using TDS without running any tests.

In this example, we shall build a very short working-code method, illustrating some of the steps involved in building it as we also build its corresponding TDS method.

$$$ We need to add a working-code Project here without changing its name

We could add our new method to the "ConsoleApp1" Solution that we built in section 4.3, but to make it easy to follow this example in case you don't have that Solution available to you, we'll use the procedure in section 4.14.7, "Setting up a stand-alone TDS Project", to construct a new VS Solution. We shall add to this Solution a Project to contain our working code, including the new method. As we add functional code to the new method, we shall use its TDS method to provide data for use in debugging the added code. These data, along with breakpoints that we can set, will help us examine variables whose values are changed by the new code, to help us verify that the processing is being performed properly.

Having used the TDS method as a means to trace into the working-code method without running any tests, when our new function member is complete enough to begin generating output, we can convert it into a test method if we wish (as illustrated in section 4.8.3). We shall do this by customizing the calls to the example **`Assert{}`** methods such as **`Assert.AreEqual()`** or **`Assert.IsTrue()`** that are already present in the TDS method, and perhaps by adding new **`Assert`** method calls to it as well.

5.1.2 Overview of this example

5.1.2.1 Summary

In this example, we shall do the following:

- Record the new method's requirements/purpose (section 5.1.2.2)
- Name the method "**`Succ()`**" (section 5.1.2.3)
- Add its TDS method, **`SuccTest()`**, to the Solution (section 5.1.2.4)
- Edit **`SuccTest()`** to call **`Succ()`** with suitable inputs (section 5.1.2.4.3)
- Add a stub for **`Succ()`** to the working code (section 5.1.2.5)

- Add functional code to **`Succ()`** (still in section 5.1.2.5)
- Run **`SuccTest()`** with breakpoints to trace into the code in **`Succ()`** (section 5.1.2.8)
- Add comments to **`Succ()`** to document its current behavior/purpose (section 5.1.2.9)

If we have no need to test, this completes the example.

- However, if we wish to have TDS do some testing, too, we can do so (section 5.1.2.10), perhaps specifying tests before (or instead of) using tracing and/or breakpoints to examine the code's behavior.
- When **`Succ()`** is complete, working properly so far as anyone knows, we can deactivate or remove **`SuccTest()`** (section 5.1.2.11)
- We run the completed **`Succ()`** method as part of its VS Project (section 5.1.2.12)

This completes the example; details of these steps are described in the following sections.

5.1.2.2 State the purpose of the project

Please see the comments on requirements statements in section 4.14.9.1.1.

The specification for **`Succ()`** in this example is stated in section 5.1.4.1.

5.1.2.3 Choose a name for the function member, if appropriate

This section applies only to new code, as existing code usually already has a name. In this example, it is new, and we'll call it "**`Succ()`**".

The names of user-defined types and members should be chosen carefully[63], to suggest or help one remember what these types or members are intended to do, so that the program will be easy to understand and easy to change later, if necessary. Even so, it is possible that too much detail in the name itself can make it become so long that it interferes with easy reading. The C# compiler will not help with this — it allows us to define grotesquely long names. Much of the burden of describing what the name means can be off-loaded to the XML comments located at its definition, to provide specific information about how to use it and allow the name to look like a word or two in a sentence, instead of like an entire paragraph. (For more about viewing the XML comments, see section 4.14.9.3.2.)

In this example, we shall develop a function member (this one will be an extension method) that performs some observable action (it will return a value) based on some input variable(s) accessible to it. We begin by passing the input as the value of a parameter passed to the method, and later as the value of an object of which the method is to be an extension method.

5.1.2.4 Construct a TDS method for the to-be-defined function member

5.1.2.4.1 WHY DO THIS?

Some of the value of TDS is that you can use it, even if you never do any testing with it, as a standardized[64] way — customized to your, or your team's, needs — to supply input values to exercise working code in your projects, making it easier to locate and update these inputs when dealing with numerous function members.

---

[63] An example of a naming convention for methods that you may find usable is one used in Windows PowerShell, where many command names consist of a verb, a hyphen, and a noun, as in using "Get-Help" to display "help" information.

[64] The usual structure of a TDS method is sumarized in section 1.10.3.2 .

To enable this standardization to be as effective as possible, I suggest that, if the original TDS code snippet (the one we imported in section 4.4.4) doesn't quite meet your needs, you and your team should modify it early in a project so that, as work proceeds, everyone involved can try to always use the same pattern (generated by the snippet) for developing all of the project's TDS methods, giving them similar, legible structures that will be easy for everyone involved to read, understand, and update as needed.

For example, having a standard structure in place should make your TDS methods easy to navigate and modify. When you are familiar with the usual organization of your TDS methods, it will be obvious where to look within any of them for values assigned to the input variables, for function-member invocations, and for unit-test (**`Assert`**) statements. If many or all of your TDS methods are based on your standard version of the TDS templates, they will have similar structures that should be easy to read and navigate. Within the TDS framework, it will also be easy to set up filtering conditions, as shown in section 4.8.7, to help with tracing through working code.

The working code may take various forms, not always invocable via a simple method call, but the TDS method itself is always expressed as an instance method with **`public`** accessibility and with no parameters and no returned value. (Unexpected results are returned by raising exceptions. A TDS method that returns normally, raising no exception, is reported as having "Passed".)

5.1.2.4.2 CONSTRUCT A TDS METHOD

We shall create this in section 5.1.5.1.1, using the name of the function member (in this example, the name to be used is "Succ") as part of the name of the new TDS method. This TDS method (in this example, it will be called "SuccTest()") will be located in TDS.cs or a similar TDS source file, and it will be part of the **`TDS.Test{}`** class.

Let's assume we have a TDS method and a statement of requirements[65], but no working code. If you were about to modify existing working code, you would skip those of the following steps that involve, for example, creating function-member stubs.

5.1.2.4.3 USE THE TDS METHOD TO INVOKE THE NEW FUNCTION MEMBER

We shall update the new TDS method to do this in section 5.1.5.1.1.4.

Identify sources of information (such as **`static`** fields in the calling class) that the working code will need to access. (A partial list will be good enough at this point; references to additional resources can be added as needed.)

As we did in section 4.8.2.4, in **`testValues[0]`** in the TDS method, we add properties to specify example values for input parameters or any accessible fields or properties that the TDS method can use to communicate with the working code.

5.1.2.5 Construct a function-member stub

Now that we have defined a TDS method with the resources that allow it to realistically call a new function member, we need to set up a method stub that we can call, as we did in section 4.10.3. We shall do this in section 5.1.5.1.1.4.

[65] The requirements statement we shall use for this example is shown in section 5.1.4.1 .

In this example, the working code is to be a method that will perform a calculation and return the result of the calculation. We shall develop this new function member by adding statements, setting breakpoints and Watch expressions, and adding or revising comments.

As we add functional code to the method stub to convert it into a real method, we shall use the TDS method to provide data for use in tracing through and debugging the added code. These data will serve as inputs to the calculations, and will also guide control flow through various paths in the working code. Among other goals, we want to ensure that all possible paths are utilized during tests. (Arguably, any code or path that is never used is apparently not needed and should be removed.)

We shall set breakpoints, step through code, and examine the values of variables transformed by the working code to help us verify that the processing is being performed as we expect.

5.1.2.6 Relationship of development to testing

If you are following orthodox TDD procedure (see section 1.8.1), you should begin by defining some (at first, always failing) tests, based on the requirements, that will attempt to demonstrate that the working code is satisfying all of those requirements. Add **`Assert`** statements to the TDS method definition to match the requirements and run the TDS method whenever you make a change to the new function member. The TDS test should continue to fail until the working code is complete, and the failure messages should help guide the development of the working code.

I usually use a more relaxed procedure than strict TDD, on the basis that it's not always easy to determine in advance all of the nuances of the process. Consequently, as I add code, I also add corresponding tests, examining both to verify that at all times both the working code and its tests are moving toward satisfying the requirements.

An even more relaxed approach, and one that may be forced on you by circumstances (for example, having some mostly finished code handed to you to be debugged and tested), is to write the bulk of (or all) the tests after the code is in place. (A more extreme option would be to throw away all the existing code and start over, TDD-style, based on the specifications. You might want to avoid letting the original developers know that you are doing this.)

5.1.2.7 Developing in a changing environment

In this example, we'll be following the middle-of-the-road path I mentioned, of developing some code, then testing it, then adding some more code or refactoring what is there, testing the results, etc. I claim that this facilitates adapting to changes either in the environment or in the development tools, though you are welcome to differ with me on this.

The reason I suggest not doing any tests[66] at first is that you may discover in the process of implementing the working code that you need to change its actions, such as the results it returns or the exceptions that it might raise under various conditions. You would thus also need to add or change some **`Assert`** statements to match the changes, so you could save some effort by making the bulk of the changes before adding the tests. I usually make liberal use of "**`//TODO:`**" comments at this stage as reminders of the types of tests that I intend to apply, instead of specifying the tests themselves.

If you find that the requirements were stated ambiguously or are somehow inconsistent with what seems to be really needed, it's not a bad idea to stay in touch with your customer. In the early stages of development, the

[66] That is, using **`Assert`** statements similar to the examples that are included in the TDS method template.

working code's connection to its neighbors or to the outside world may be somewhat flexible, so the statement of requirements might need to change to match the customer's changing needs. (In later stages, that might still occur, but by then it will take more effort to correct any mismatches, or it might become necessary to scrap the project. Oog. It's been known to happen. Let's try to avoid that by planning ahead.) If you're really lucky, you may discover that you have already satisfied all the requirements and can finish early and celebrate!

Another possibility is that, even with unchanging requirements, you may come across new tools (perhaps a new compiler release?) that you might use. As you refactor the working code (or the testing system) to take advantage of those, you will want to verify that nothing that is already working has broken, and that you have taken care of any damage. New tests may be called for in such events.

As the requirements or the working code evolve over time, the result could be that some **`Assert`** statements written too early might need to be redone, though that is not a major hazard. (Much worse would be allowing a bug to remain undetected for too long by including too few **`Assert`** statements.)

It is possible that you may develop (or be handed) a set of requirements for a new function member that are so specific that all of its external couplings are completely obvious. When that is the case, TDD is probably superior to TDS; go ahead and define **`Assert`** statements (in the function member's TDS method) corresponding to all the requirements before writing any code.

However, even if you expect never to have any need to modify any of your working code's interfaces with its environment, you may still find it useful to define a TDS method to generate test cases for the function member. Doing this could be especially helpful if you also use the TDS code snippet in your project for developing new function members that are not as well defined. By giving all of your TDS methods a consistent structure, you can make both your finished unit-test methods and your unfinished TDS methods easier to read and navigate.

5.1.2.8 Run the working code, observing variables

Run the project, calling the TDS method to exercise the working code, for example by setting breakpoints and Watch expressions, and single-stepping through the code to observe (for example, as we did in secton 4.8.7.2) that the variables are being given the correct values.

5.1.2.9 Add or update comments documenting changes

As you make changes or additions to the working code, update its XML comments (both on fields and on function members, such as methods or properties) to reflect changes to its intended behavior. See section 4.14.9 for remarks on maintaining XML comments.

5.1.2.10 Modify the TDS code to unit-test the method

At a suitable point, when you can see that the function member is working sufficiently well on its initial set of data, you can remove the

```
throw new NotImplementedException();
```

statement at the end, freeing the TDS method to begin applying its **`Assert`** statements and returning a status of "Passed" if successful. We can update these **`Assert`** statements, and add new ones, to compare outputs from the new working code with the expected results.

We can add properties[67] to **`testValues[0]`** to specify the expected values of any outputs of the new function member, as we did in section 4.8.3.3. For example, the function member may make, or be intended to make, changes to objects passed to it via parameters or to **`static`** properties or fields of the calling object, or it may be a method whose returned value you can predict. Looking ahead, you can use such properties to identify the expected values of these variables, for use in later testing.

Up to here, TDS has done all that we expect of it, short of running tests. However, adding tests to this TDS method, if we wish to do so, does not require much extra effort.

As discussed in section 4.8.3.1, when our new function member is complete enough to begin generating output, we can convert its TDS method into a test method by adding calls to "**`Assert`**" methods such as **`Assert.AreEqual()`** or **`Assert.IsTrue()`**.

We can add alternate sets of inputs in the form of additional elements of **`testValues[]`**, to support testing using different input variables. See section 4.8.3.4 for an example.

5.1.2.11 Remove TDS code

We can now remove the TDS Project from the working code.

In a normally completed project, all of the TDS methods should have been converted into test methods, all of the test methods should have run successfully, and the test methods should have no further value after that unless some change in the working code becomes needed. The need for such a change might arise when, for example, the project's requirements change, or a bug apparently related to working code invoked by one of the TDS methods appears, or some environmental change in the system occurs (such as that the contents of some field accessible to the code are assigned a new meaning).

5.1.2.12 Run the working code outside of VS.

When we are finished, we shall be able to run the newly developed working code in various ways (as illustrated in the Tutorial):

- Run independently of Visual Studio:
  - called directly from an operational project (in this example run as part of the executable program ConsoleApp1.exe), bypassing all of the TDS methods, as it might be in normal[68] use
  - called via its TDS methods as part of the executable program TDS.exe, which produces a test report on the Console or in a text file
- Called from within VS:
  - run as part of a project not involving TDS; in this example (project ConsoleApp1) it produces some Console output, but in general such output is not necessary
  - called via its TDS methods, using the TDS platform, which generates a TDS test report

---

[67] As noted in section 4.8.3.4, it is easiest to add and/or modify properties while **`testValues[]`** contains only one element.

[68] This example would probably be a simulation of "normal" use, initiated from a user interface such as Windows Explorer, the Windows Command Prompt, Windows PowerShell, or from a script operating under one of these. In real life, most tested working code is likely deeply embedded in a much larger operational system, and the unit testing is a small part of a more comprehensive testing protocol. If we called it without involving any TDS methods, the working code might not generate any visible output, though the example we shall build does send some output to the Console.

- called via its TDS methods, and possibly also via some non-TDS methods having **`[TestMethod]`** attributes[69], using the Visual Studio Test platform or another test platform such as NUnit to report the results

5.1.3 Learning objectives

When you complete this example, you will have…

- added a working TDS method to the Solution
- added a new function member to the working code called by the TDS method
- used the TDS method to help trace execution through the working code to locate a bug
- used TDS to test the new function member and generate reports on the results of the test
- run the function member separately from the TDS code

We have already done all of these, briefly, in the Tutorial, but this example goes into more detail.

5.1.4 Requirements for "Successor" calculation

5.1.4.1 Requirements statement

Suppose that we want to develop …

```
a C# extension method, to be called Succ(), a shortened form of "Successor".
It is to return a Decimal object whose value is the successor function (= next
higher whole number) of a given int (= 32-bit signed integer) argument.
```

We might need to add more detail later, but for now this statement is specific enough to let us begin.

I am aware that many people claim that one must do some more detailed design before writing any code. I suggest that it is often possible, and maybe even helpful, to express the design in the form of code (also called "pseudo-code" at this stage), at first mostly as comments, to keep the documentation of the ideas physically close (such as in the same editing window) to where one is going to be doing most of the work that uses them, and easily visible. If it becomes apparent that a change to the code is necessary, for example to accommodate a requested design change or to raise a new exception, comments documenting that change can be included in the source file, along with the updated code. Since the code to implement the change will be located only a few lines away from the comments, it should be easy to keep the code consistent with the comments.

For now, hold onto this requirements statement — it will soon be copied into some C# comments.

We shall write an expression in a new TDS method, and a similar expression in **`NewCodeNamespace.NewCode.RunWithoutTds()`** (which has no connection with any TDS methods and is called by **`NewCodeNamespace.Program.Main()`**), to invoke the method to be developed. We shall generate a method stub that can be invoked using either expression.

5.1.4.2 Recording the statement

► Record the requirements statement in a convenient location.

We did not do this for the Tutorial, as the expected behavior was already documented in the code's XML comments, but for new code it would be needed.

---

[69] In these examples, all of the **`[TestMethod]`** methods can be run using the TDS platform.

If the statement already exists in a suitable location that future developers will always be able to reach, there is no need to do anything special; maybe just put a shortcut to its file into the Demo\ folder, so that you can find it when you need it.

We could copy the statement from section 5.1.4.1 above to a text (*.txt) file in the Demo\ folder, but we will soon (in section 5.1.5.1.3 below) have a place for it in the source code, so for this examples we shall copy it there instead.

This record (text file or whatever we have chosen to use), for now, will serve as our design document. It is possible that the design will need to be changed later, but if so, we can revise the statement at that time.

This statement describes what result we want but says little about the means to provide that result. It might be thought of as a contract that promises that the new working-code function member, which in this case is to be the method **`Succ()`**, will perform the specific action described here whenever it is invoked.

We shall add the proposed **`Succ()`** method to our existing "ConsoleApp1" Solution.

5.1.5 Set up a new function member and its TDS method

5.1.5.1 Create a TDS method to exercise **`Succ()`** .

5.1.5.1.1 SET UP A NEW TDS METHOD.

5.1.5.1.1.1 CREATE AN EMPTY VS SOLUTION

Follow the steps in section 4.14.7, "Setting up a stand-alone TDS Project", to construct a new VS Solution containing only a TDS Project.

(The following steps are similar to what we did in section 4.4.1.1.)

► In the VS Solution Explorer window, right-click on "Solution 'TDS' (1 project)" (not the Project with that name); choose "Add, New Project…" to add a new Project to the Solution.

► For this example, choose Visual C#, Windows, Classic Desktop, Console App (or Console Application); set its Name to be "ConsoleApp1". Click OK.

This Project is to contain our simulated working code.

Here we could have chosen any Project type that can be called from a TDS method. We use the Console App type in this example to keep the output easy to read and to compare with the printed version shown in these instructions.

5.1.5.1.1.2 ADD A NEW TDS METHOD

Similarly to what one might do following Test-Driven Development (TDD) rules, we'll begin by inserting into our TDS Project a new TDS method, which will call a method that does not yet exist, as we did in section 4.10.3.

► Within the **`TDS.Test{}`** class in file TDS.cs, somewhere after the "**`TODO: New TDS methods may be placed here:`**" Task comment near the end of the file, use the TdsTest code snippet to generate a TDS method for to-be-defined method **`Succ()`**, as we did in section 4.8.2.1.

► Type the name "Succ" into its "TestableFunctionMember" field.

► Press <enter> to accept all of the default names and close the snippet.

If we already had several TDS methods defined here, I would define this new one in alphabetical order in the collection, to make it easy to find, but the order is immaterial to the compiler.

If we wanted **Succ()** to accept and return only **(int)** values (32-bit signed integers), we would have no need to modify the lines in **SuccTest()** that specify these types. Until we change it, in the **testValues[0]** definition, which follows the "**TODO: SuccTest() -- Define inputs and expected outputs.**" Task List comment, the line

```
Arg = 3,  // Input value
```

specifies that **Arg** is an **(int)** variable, which may be used unchanged as an input parameter, though we may want to change the value.

However, following the Task List comment "//TODO: SuccTest() -- Use a suitable default value.", the line

```
var actual = 0;
```

would specify that the returned value is also to be an **(int)**, which is not what we want it to be, according to our Requirements statement (section 5.1.4.1).

5.1.5.1.1.3 SPECIFY THE TYPE OF THE RETURNED VALUE

In the present case, the type of **actual** is wrong; we will need (as we did in section 4.8.2.3) to modify the TDS method to specify that the returned value, **actual**, be of type **Decimal** (28-digit signed decimal value), since our method is required to return a **Decimal** value.

▶ Following the "**TODO: SuccTest() -- Use a suitable default value.**" Task, edit the statement

```
var actual = 0;
```

to read something like

```
var actual = 0M;
```

or, to accomplish the same result but with a slightly longer line of code, to be

```
decimal actual = 0;
```

To navigate there, double-click its entry in the Task List window.

▶ Delete this statement's "**//TODO: SuccTest() -- Use a suitable default value.**" Task List comment after specifying a suitable type and value for variable **actual**.

5.1.5.1.1.4 CREATE A METHOD STUB FOR THE WORKING CODE

Depending on the kind of function member we are developing, we will need to call it in a suitable way. For example, if it be an indexer, we would need to use an indexer-access expression to invoke it. In this case, since we are developing an extension method, we shall eventually call it using an instance-method expression. However, right now we want VS to create a method stub, so we'll start by letting VS generate a static-method call stub and change it after that to be an extension method.

▶ In VS Project "ConsoleApp1", open file Program.cs for editing.

▶ Just before the closing brace of

```
namespace ConsoleApp1
{
…
}
```


, insert the following code:

```
/// <summary>
/// Extension methods, etc.
/// </summary>
static class StaticCode
{
}  // end: StaticCode{}
```

This gives us a suitable place in which to define the new method **Succ()**.

(In a real project, it would probably be better programming style to place this static class into a separate source file, but for this example this is a convenient location.)

► Similarly to what we did in section 4.4.1.2, in Project TDS add a Reference to Project ConsoleApp1.

► In the "**////TODO: Usings -- Include "using" statements for the namespaces of the code**" Task in file TDS.cs, add a **using** statement if it's not already present:

```
using ConsoleApp1;
```

► Change the invocation statement following the Task "**TODO: SuccTest() -- Provide a suitable calling expression**" to include the static-class name **StaticCode**, to make it look like this:

```
    actual = StaticCode.Succ(tCase.Arg);
```

We shall change this statement soon to call it as an extension method. If we did that now, VS would have difficulty determining where to define it, since no class would be specified for the method, and VS would wrongly assume that we wished to use the current class, **TDS.Test{}**. We want VS to automatically generate a suitable method stub in the correct location, which should be somewhere in our working code.

The identifier **Succ** should have a wiggly red underline flagging it as undefined.

► Hover the mouse pointer over **Succ**, then choose the drop-down menu item "Generate method 'StaticCode.Succ' ".

It will be created with a parameter of "arg", but you may feel that some other name would be more suitable. If so, before creating the new method definition, you could rename **tCase.Arg** or, if you prefer, you could rename the parameter after the code is generated, before doing any other work there. (In this example, we shall keep the parameter name "arg".)

► Navigate to the definition of newly created method **Succ()** (for example, use <F12>) and edit its parameter to change it from **(int arg)** to **(this int arg)**.

The automatically generated method stub (with "**this**" that we added) should now look like this:

```
    public static decimal Succ(this int arg)
    {
      throw new NotImplementedException();
    }
```

Now **Succ()** will be able to be called either as a static method,

```
actual = StaticCode.Succ(tCase.Arg);
```


, or as an extension method,

```
actual = tCase.Arg.Succ();
```

, whichever is convenient.

We shall add some comments to its definition presently, along with some functional code.

5.1.5.1.1.5 UPDATE THE CALLING CODE

► Back in **`SuccTest()`**, at the "**`TODO: SuccTest() -- Provide a suitable calling expression`**" Task, change the invocation statement

```
    actual = StaticCode.Succ(tCase.Arg);
```

to this:

```
    actual = tCase.Arg.Succ();
```

VS's AutoComplete feature, this time, does suggest **`Succ()`** as a potential member for **`tCase.Arg`** .

► Delete its Task List comment:

```
//TODO: SuccTest() -- Provide a suitable calling expression
```

5.1.5.1.1.6 RUN A SMOKE TEST

► As shown in section 4.8.2.5, add the name of **`SuccTest()`** to the list in the "**`TODO: TestMethodsToBeRun -- List all TDS test methods to be run.`**" Task.

► If project TDS is not the Startup Project, make it be so (as we did in section 4.4.3.1).

We have now done enough work to allow us to run a "smoke test" (as in section 4.3.6.4), to allow us to check for major mistakes, such as compiler syntax errors.

► Run the program, for example using VS menu "Debug, Start Debugging" or <F5>.

If an exception message pop-up window should appear, deal with it as described in section 4.4.2.

A TDS test report should appear, showing that the test of **`Succ()`** fails, generating a message (among many others) stating, in part,

```
SuccTest(), test case 01 Sample test:
  The expected exception should start with " No exception was thrown".
  This u+nexpected exception was thrown:
    "The method or operation is not implemented."
```

(Yes, I remember that we weren't going to run any tests until later, or maybe not at all in this example. TDS calls the report that it generates a "test report", which is what it will become by the time we get to section 5.1.5.2.3 or so. For now, it merely contains some basic information like the time of day, and a record of the exception that stopped execution.)

► Close the Console window.

5.1.5.1.2 COMMENT THE CLOSING BRACE (OPTIONAL)

Although the following is optional, I usually apply a comment to the closing brace of a long block, mostly to help match the braces while I edit code, but also to help match them visually as I read the code.

► I would edit the last line of the definition of the new method **`Succ()`** to make it look like this:


```
    }  // end: Succ()
```

5.1.5.1.3 ADD XML COMMENTS

Applying XML comments to your definitions of methods, fields, etc., will, among other benefits, allow the comments to appear in the IntelliSense pop-ups (see a discussion and examples in section 4.14.9).

► In file Program.cs, in **`StaticCode{}`**, on a new, blank line preceding the **`Succ()`** method definition, type "**`///`**" to generate an XML comment template for the method, including tags for its parameter and its returned value. (If VS did not produce any blank XML comment tags, use VS menu "Tools, Options"; then, in the tab "Text Editor, C#, Advanced", select the option for "Generate XML documentation comments for ///".)

For this method, XML comments similar to the following should appear:

```
    /// <summary>
    ///
    /// </summary>
    /// <param name="arg"></param>
    /// <returns></returns>
```

I usually insert into the XML comments the design document, or at least the essence of it, to identify what the new function member is intended to do and what resources it needs to use. It may need to be reformatted to escape special HTML characters (see section 5.1.5.1.4).

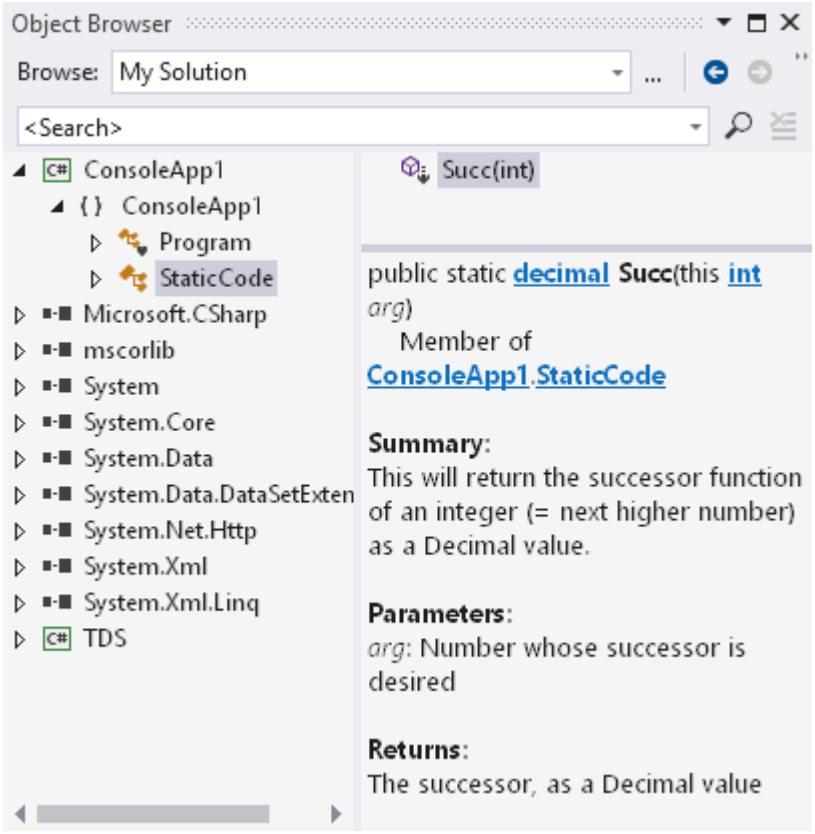
► Copy the design statement from section 5.1.4.1 above into the **`<summary>`** tag, and briefly describe the parameter and the returned value in their tags.

For this exercise, you may simply copy the following example, replacing the empty XML comments that we just now generated. (See section 2.3.3 for a note on copying code from this *TDS User's Guide*.)

The results for this example might look like the following code; I changed the wording of the original statement slightly in this **`<summary>`** to better describe the method:

```
    /// <summary>
    /// This will return the successor function
    /// of an integer (= next higher number)
    /// as a Decimal value.
    /// </summary>
    /// <param name="arg">
    /// Number whose successor is desired
    /// </param>
    /// <returns>
    /// The successor, as a Decimal value
    /// </returns>
```


In the VS Object Browser window, these comments generate the following display:



5.1.5.1.4 NOTE ON REFORMATTING XML TEXT (ESCAPING HTML CHARACTERS)

Since these comments include XML code, take care[70] to escape the special HTML characters "&" and "<" when copying and pasting text into XML comments. If I suspect that any of these characters may be present, I usually do the following:

► Create a temporary editing file via, for example, using VS menu "File, New, File…, General, XML File", then clicking "Open".

Opening a new file in a text editor such as Notepad would also work, but the XML editor in VS can help identify unescaped special characters.

► Paste the (at this point, unescaped) text into the empty editor window.

---

[70] Not escaping these characters won't cause your project to fail to compile, nor even to produce a warning message, but it will likely cause the entire XML comment not to appear in the Object Browser and IntelliSense. Try to escape all of them, because finding the one "<" that you missed, among a forest of valid ones, may not be easy, and VS will not generate a helpful error message. For example, it will give no clue as to which line contains the faulty XML code.

If you're using the XML editor, paste the text into the lines following the first line, which contains this:

```
<?xml version="1.0" encoding="utf-8"?>
```

- In the pasted text, first replace all "**&**" characters with "**&**".
- Replace all "**<**" characters with "**<**" and (optionally) all "**>**" characters with "**>**".

For example, the text in the preceding paragraphs might look like this after these characters are escaped:

```
In the pasted text, first replace all "&amp;" characters with "&amp;amp;".

Replace all "&lt;" characters with "&amp;lt;" and (optionally) all "&gt;"
characters with "&amp;gt;".
```

The order in which we replace these characters is important, as the 2nd and 3rd changes introduce "**&**" characters that should not be escaped. It's not really necessary to replace the ">" characters, but I usually do that, too, to make the matching pairs look consistent in the C# source code.

Also remove any nested commenting strings "//", "/*", or "*/". They are not needed, as the XML comment is already a C# comment, and they may interfere with displaying the comment's text.

You may wish to enclose the inserted text with **<summary></summary>** tags, to suppress XML editor error messages. You may also format the document (via VS menu "Edit, Advanced, Format Document" to make it more legible, for example by combining short lines or breaking long ones.

- Copy the escaped text and paste it into the appropriate **<summary>, <remarks>**, etc., elements of the XML comments.
- Discard the *.xml or *.txt file that you used to edit the text.

You may use **<para></para>** elements to help format the comments, so long as you keep them properly nested. You may use VS's Object Browser to check the contents and formatting of your XML comments. If the comments are syntactically correct, they will appear in the Object Browser and IntelliSense pop-ups; otherwise, only the name and membership information will appear (no "**Summary:**", etc.), as if none of the XML comments were present.

For formatting, if you want to use **
** to start new lines, you may be disappointed — it seems to be ignored in Object Browser and IntelliSense. But **<para>** and **</para>** do work, so I use them instead, even though they must be paired and, I think, can look confusingly similar to **<param>** and **</param>**.

5.1.5.1.5 VIEW THE XML COMMENTS

For details and examples of viewing the comments, see section 4.14.9.3.2.

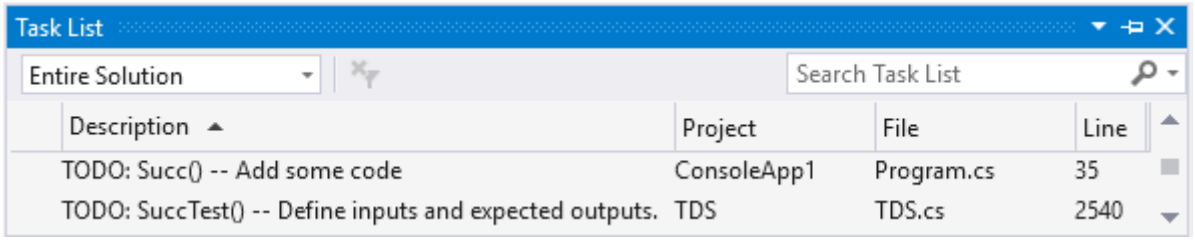
5.1.5.1.6 NOTE ON //TODO: COMMENTS

Add reminders of unfinished work using temporary **//TODO:** comments, which will appear in the Task List. (See the discussion in section 4.14.16.)

For example, immediately before the **throw** statement in **Succ()**, one might add the comment

```
//TODO: Succ() -- Add some code.
```


Of course, this isn't really needed, since you probably remember where **`Succ()`** is and that it's unfinished, but this comment would place a memo into the Task List that could make navigating to the comment's location easier. For example, part of the Task List might now, temporarily, look like this:



5.1.5.2 Do a manual/visual test

5.1.5.2.1 BEGIN TRACING

Now suppose that we wish to verify that the method is being invoked properly,

► In the definition of the **`Succ()`** method, on its **`throw`** statement, set a breakpoint (perhaps using VS menu "Debug, Toggle Breakpoint" or by clicking in the breakpoint column).

If you wish to use a different input value, then at the "**`TODO: SuccTest() -- Define inputs and expected outputs.`**" Task, in the line

```
    Arg = 3,  // Input value
```

, you could change the value 3 to some other value or type suitable to your new method. For this example, 3 works, and we'll leave it unchanged.

► Use VS menu "Debug, Start Debugging" or press <F5> to begin running the program until it reaches the breakpoint.

► Examine the value of the parameter (for example, in the VS Locals window, or by hovering the mouse pointer on the declaration of **`arg`** in the parameter list in the editing window), to observe that it has the value of 3 passed by the calling TDS method.

We're not actually using this value yet, but we now see that it is available for use and has the expected value.

► Use VS menu "Debug, Stop Debugging", or <shift><F5>, to resume editing the code.

5.1.5.2.2 ADD SOME CODE TO `SUCC()`

Now we shall add some code to **`Succ()`**, as an attempt at doing the desired calculation. In real life, we might add several statements; here, we'll simply add one short one.

► In Class1.cs, copy or type the following code[71] into the definition of **`Succ()`**, immediately before the **`throw`** statement on which we placed the breakpoint:

```
    decimal result = arg++;
```

Now that we have added some working code to the new function member, we can call it using some example data, and trace its execution to try to ascertain that nothing unexpected is happening. (Please try to ignore that

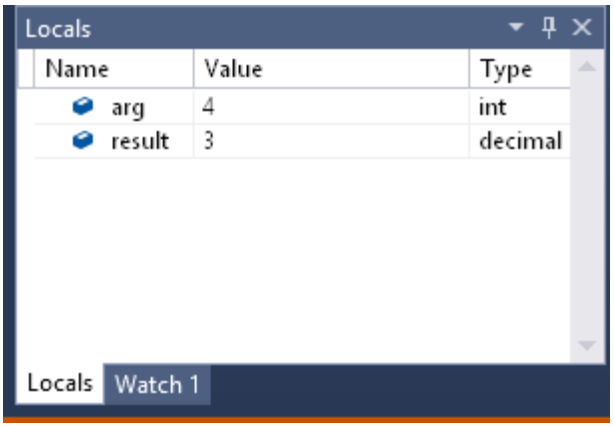
[71] For the purpose of illustration, this code intentionally contains a mistake, which we shall correct presently.

the given example is so simple-minded that nobody should be confused about what's happening; the purpose is to illustrate a technique.)

The **`throw`** statement protects us from accidentally running tests using code that is known to be incomplete or faulty. We can insert more C# statements above it, and if we miss hitting a breakpoint or accidentally resume running after stopping for a breakpoint, the new working code will immediately, and we hope harmlessly, raise an exception that will be reflected in the test report.

- Run to the breakpoint (use <F5>); we notice that the value of **`result`** is not the 4 that we wanted.

At a breakpoint, we can observe variable values in VS in any of several ways — hovering the mouse pointer over a reference to a variable such as **`result`**, looking at a variable's value in the "Locals" window, entering an expression such as "**`result`**" into the "Watch" window, or executing a command such as "**`?result`**" in the "Immediate" window. The "Locals" window should look something like this:



Oops – although **`arg`** has been updated to have the correct final value, 4, we see that **`result`** didn't get the memo; we see that our working-code method updated the values in the wrong order.

- Use VS menu "Debug, Stop Debugging" or <shift><F5>, to permit editing the code.
- Change **`arg++`** to **`++arg`** or to **`arg + 1`** to correct the problem that we noticed.
- Run the test again.

At the breakpoint (which is still on the **`throw`** statement), we see that **`result`** has the correct value this time.

- Remove the breakpoint, which we no longer need. Press <shift><F5> to return to editing.
- Replace the **`throw`** statement with the following:

```
    return result;
```

Since the method now returns a value, we no longer need the previously included **`throw`** statement. This new code may still be buggy, but it won't generate irrelevant exceptions any longer.

You may also delete the "**`//TODO:`**" Task comment if you don't expect to need it further.

We could have replaced both statements with this:

```
      return (decimal)++arg;
```

, but by intentionally giving the name **`result`** to this expression, even though we use it only once, we can see its value in, for example, the Locals window while stopped at a breakpoint. This is a matter of style; the optimizing C# compiler probably generates the same code with either version. Adding too many unnecessary variable names might clutter the namespace, but making the code too compact, in the case of complex expressions, may make the code unnecessarily difficult to read, understand, and debug.

In real life, a new function member might contain dozens of statements, rather than just one or two, as this one does. It could also include calls to other unfinished function members, which in turn would likely need their own TDS methods, so we may have several interacting function members under development at one time. The **`throw`** statements can help to keep track of which of these are not yet ready to be called by other code in the project. Exactly when it is best to remove the **`throw`** statement is a matter of judgment — too early might increase the area that we must search for the causes of bugs, too late could delay the testing of code that depends on the new function member.

5.1.5.2.3 REPEAT THIS PROCESS AS NEEDED

We can do some types of editing while debugging, or we can stop the debugger (perhaps via VS menu "Debug, Stop Debugging", or <shift><F5>), modify the code and possibly move the breakpoint, and then run again to the breakpoint to see the results of running the corrected code.

As we develop the new function member, we can modify the TDS method that calls it by changing the properties of its first test case (the one in **`testValues[0]`**, located at the "**`TODO: xxxTest() -- Define inputs and expected outputs.`**" Task) and/or adding new properties. (Later, when we are ready to run tests of the new function member , we can add new test cases to the TDS method's **`testValues[]`** array, as we did in section 4.8.3.4, to invoke the function member with alternate sets of inputs.)

Continue tracing and editing until the code appears to be[72] operating correctly. Also, as needed, update the XML comments to document anything that might be of interest to users of this function member, such as special handling of exceptional cases or significant changes in any of its outputs.

- ► Run TDS (press <F5>).

The test report in the Console window should show that our new TDS method, **`SuccTest()`**, returns a status of **`Inconclusive`**, while the other test returns a status of **`Passed`**.

Our TDS method is not yet ready to do any automatic testing; we are using it for now only as a source of inputs for tracing, so its **`Inconclusive`** status is correct at this time.

- ► Close the Console window.

5.1.5.3 Call `Succ()` from non-TDS code

5.1.5.3.1 INSERT CODE TO CALL THE NEW METHOD

Here we can demonstrate that TDS is not an essential part of our Solution (and therefore can easily be removed at any time).

We shall add code to **`Main()`** to display on the Console the result of a call to **`Succ()`** that bypasses all the TDS code.

---

[72] I said "appears to be" instead of "is" because a mathematical proof of correctness is usually impractical. It could take a long time to do and document a proof (see section 5.2.4.2 for an example), and so could showing that there are no errors in the proof itself. Therefore, I feel that one should also do some unit testing.

In a real project, we would probably put the invocation of **`Succ()`** into a place in the code where we would need to use it, maybe accompanied by a "**`//TODO:`**" Task List comment that identifies it as unfinished for now and needing further attention.

- In file Program.cs, into method **`ConsoleApp1.Program.Main()`**, place a copy of the following lines:

```
//TODO: Main() -- Call Succ() without using TDS
#region SuccDemo()
for (var n = -2; n <= 2; n++)
{
  Console.WriteLine("(" + n + ").Succ() = " + n.Succ());
}
Console.WriteLine("\n(Please press <enter>.)");
Console.ReadKey();  //Wait for a response
#endregion SuccDemo()
```

The **`#region`** and **`#endregion`** C# directives are not necessary, but I use them to help organize the code, hiding it (via VS menu "Edit, Outlining, Toggle Outlining Expansion") when the code is not of immediate interest. Visually, when this **`#region`** is collapsed, the result looks as if we had replaced it with a short, parameterless method call:

```
//TODO: Main() -- Call Succ() without using TDS
SuccDemo()
```

5.1.5.3.2 TEST THE NON-TDS VERSION (NO TDS CODE)

- In the Solution Explorer window, set **`ConsoleApp1`** as the Startup Project (as we did in section 4.7).
- Run this (using <F5>).

The following Console output should appear:

```
(-2).Succ() = -1
(-1).Succ() = 0
(0).Succ() = 1
(1).Succ() = 2
(2).Succ() = 3

(Please press <enter>.)
```

If this new code had any lasting value, it would be appropriate to add some XML comments to **`Main()`** to document the visible changes to the output. A description summarizing these displayed lines might be placed into the **`<summary>`** or **`<remarks>`** element. However, in this instance, since this new code will be removed shortly, we'll omit updating the comments this time.

- Close the Console window.

5.1.5.4 For tracing purposes, the TDS method `SuccTest()` is complete.

Returning to the TDS method, the only input to the **`Succ()`** method is its parameter, **`arg`**, to which we have already given a value by using the **`Arg`** property of the **`testValues[0]`** object, so this is about as far as the pre-testing phase of TDS needs to go in this example.

If we have no intention of doing any unit testing[73] of **`Succ()`**, the TDS method **`SuccTest()`** has accomplished everything it needs to accomplish — providing values for the parameters or other inputs and calling the function member, while providing a framework that might later be used for automated testing.

For a method as simple as **`Succ()`**, you might even be justified in not converting its TDS method into a test at this time; you would be able to notice and correct the bug just by tracing execution of the code or by observing the results.

If this be true for you, you can stop using TDS now for this function member. If its "Inconclusive" messages annoy you, you might also comment out or delete the entry "**`SuccTest`**" that you added to **`TestMethodsToBeRun`** and remove the **`[TestMethod]`** Attribute from the **`SuccTest()`** code. If you do so, **`SuccTest()`** will no longer be called by a unit-test platform, and the results of running **`Succ()`** will no longer be reflected in the test report generated by the TDS Project.

Since this TDS method doesn't actually do any testing yet (you're doing some of that manually), its **`Assert.Inconclusive()`** statement should be left active for now, to avoid giving the false impression that its function member is being tested automatically and is passing all its tests.

5.1.6 Convert TDS method to a test procedure

5.1.6.1 Rationale for unit testing

(This section is a more detailed version of section 4.8.3 in the Tutorial.)

Suppose that we have done some tracing of our new function member, the method **`Succ()`**, and have determined that its local variables are behaving as we expect. For the remainder of this example, since our new function member is ready to return at least one value that can be analyzed, let's assume that we have now mostly finished refining its code and are ready to begin unit-testing the results.

In developing **`Succ()`**, as we are doing here, there is only one input, its parameter, and we shall run the method multiple times so that we can analyze its behavior with various values of that one input. However, instead of using the Watch or Locals window in VS to observe the values, now we would like to be notified only if the behavior is somehow different from what we expect. Doing this could save us time and effort, assuming that its TDS method is already invoking it properly.

The rest of this example is intended to show that adding tests to a TDS method will involve only a small amount of additional work, and to make it more apparent why some of the features of the TDS methods are included, such as using properties of objects in **`testValues[]`** to specify the values of input variables, instead of using literal values to do that. For example, in **`SuccTest()`** we can use an expression similar to

```
    actual = tCase.Arg.Succ();
```

instead of

```
    actual = 3.Succ();
```

to send a value of 3 to the new method. With only one test case, these statements have identical effects, and the second one appears to be shorter and simpler; its problem is that it doesn't support multiple test cases. (We'll begin using multiple test cases in section 5.2.6.4 below.)

---

[73] However, in the following sections we assume that we *do* intend to use this TDS method for testing, so don't lose it yet.

The following steps illustrate setting up the means to do some simple automated testing of **`Succ()`**.

5.1.6.2 Reconfigure from the pre-testing phase to early testing

5.1.6.2.1 IN `SUCCTEST()`, UPDATE THE `VALUEEXP` PROPERTY IN `TESTVALUES[0]`.

Since the method we're developing, **`Succ()`**, now returns values that can be analyzed, we can transform our TDS code into a unit-test method, similarly to what we did in section 4.8.3.1 of the Tutorial.

► In file TDS.cs, in TDS method **`SuccTest()`**, in **`testValues[0]`**, change the value of the **`ValueExp`** property to be the value we expect **`Succ()`** to return. Make it look like this:

```
ValueExp = (decimal)4,  // Expected returned value
```

or, a bit more concisely, make it look like this:

```
ValueExp = 4M,  // Expected returned value
```

Either expression would serve to specify that the type of property **`ValueExp`**. is **`decimal`**.

This "expected value" is intended to match the **`decimal`** value expected to be returned from the method whenever the value of the parameter sent to the method is **`(int)3`**.

The new code in **`testValues[0]`** might look like this[74]:

```
      var testValues = new[] {

        new {
          //TODO: SuccTest() -- Define inputs and expected outputs.
          Id = "01 Sample test", // Test case identifier (required),
              //  consisting of a unique 2- or 3-character tag, a space,
              //  and a short description of the test case.
          Arg = 3,  // Input value
          ExceptionExp = DefaultExceptionMessage,  // Expected exception
              //  This specifies a string that the beginning
              //  of the exception message, if any, is expected to match.
              //  "" is treated as "No exception is expected".
          ValueExp = 4M,  // Expected returned value
        },
```

The "**`DefaultExceptionMessage`**" is an indication that we expect no exception to be thrown from this set of inputs.

[74] In this example, the input values are followed by the expected output values, but so long as you have only one element defined in **`testValues[]`**, you may use any order you wish.

Although in this example we have placed a comment on the same line as each property definition, you may prefer to place them into a block of comments preceding the definition of **testValues[0]**, perhaps like this:

```
        var testValues = new[] {
            //TODO: SuccTest() -- Define inputs and expected outputs.

            /*
            Id = Test case identifier (required),
                  consisting of a unique 2- or 3-character tag, a space,
                  and a short description of the test case.
            Arg = Input value
            ExceptionExp = Expected exception
                  This specifies a string that the beginning
                  of the exception message, if any, is expected to match.
                  "" is treated as "No exception is expected".
            ValueExp = Expected returned value
            */

          new {
            Id = "01 Sample test",
            Arg = 3,
            ExceptionExp = DefaultExceptionMessage,
            ValueExp = 4M,
          },
…
```

Placing the comments describing the properties into a separate block of delimited comments, as shown here, would allow all the following array element definitions to look alike. This would make it easy to construct **testValues[1]**, etc., by copying the definition of **testValues[0]** and modifying the values, without having to delete redundant comments from the copy. Also, to comment on a noteworthy quality of the value of a property in **testValues[0]**, you could place the comment about that value directly on the line defining the property, without interference from the comment describing the property itself.

On the other hand, using single-line comments in **testValues[0]**, as we do in these examples, makes it easy to keep them with their corresponding property definitions if we wish to rearrange their order in the initializer.

Your naming convention and how you handle comments could be affected if you convert your anonymous-type **testValues[]** elements to a named-object type, such as **TestableConsoleMethodTestCase**, as shown in the Tutorial, section 4.8.6, and in section 5.2.9.6 below. Using a named type will allow more freedom in specifying test cases, so you might keep that in mind as you organize the elements of **testValues[]**.

Please see section 4.14.10 for suggestions on naming properties in **testValues[0]** and adding comments to them, and see the table in section 5.2.9.6.3.2 for a comparison of various types of comments in **testValues[].**

5.1.6.2.2 IN **SUCCTEST()**, ADD AN **ASSERT** STATEMENT.

We can use the example **Assert** statements in the "**#region Apply tests when no exception is raised**" region as patterns for testing now. The first statement checks that the new function member was not expected to raise an exception of any type and did not in fact do so. You can probably use this in exactly its current form.

You would normally need to make some changes to the **Assert** statement following the "**//TODO: SuccTest() -- Provide suitable non-exception tests here**" Task List comment, but in this case we have already specified an input value by using the property **testValues[0].Arg** and the expected output value by using the property **testValues[0].ValueExp**, both of which are already defined, so the second statement will also work correctly in its current form.

We have now added (or are preparing to use existing) code to check that , using the specified test-case properties,

- no exception has been thrown by the method, and
- the returned value matches the expected value.

The messages that the **Assert** statements emit identify the test (in this example, it's the literal string "**SuccTest()**") and the test case (using the value of **tCase.Id**), but you may want to edit them to provide additional, or less, information. For example, the "**SuccTest()**" name in these **Assert** statements is redundant and could be omitted if you wish. You might want to display further details about the returned value, for example the values of selected properties of an object that has several properties. (An example with additional details is shown in section 5.2.8.5.2 below.)

5.1.6.2.3 ADD MORE **ASSERT** STATEMENTS

We probably have included enough tests (**Assert** statements) in the TDS method to handle this example, but if the function member that we're testing were to produce several outputs, or any outputs with complex values, we might need to add other **Assert** statements to it to help verify that the function member is working properly.

If we do have several **Assert** statements, we might also include in the message of each one a label identifying it, as we did in the example in section 4.8.3.3, to make more obvious which one of the **Assert** statements failed.

5.1.7 Test the new method

Since the testing infrastructure was already in place in the TDS method from the time we created it, not much remained to be done to convert it into an actual test — adding the **Assert** statements was enough. It's still a pretty rudimentary test, but at least it verifies that the newly defined method can (maybe depending on the input values) return a value without crashing.

► In the Solution Explorer window, set project TDS as the StartUp Project.

► Run the test, for example using VS menu "Debug, Start Debugging" or <F5>.

The test report in the Console window now shows that **SuccTest()** has a status of "Inconclusive".

► Press <enter> to close the Console window.

► In **SuccTest()**, immediately before the end of its definition, delete the "**Assert.Inconclusive()**" statement.

► Also delete this statement's "**////TODO:**" comment, as there's nothing more to be done here.

In **SuccTest()**, since we are now using **Assert** statements to do some actual testing, and all of the test code appears to be working, the **Assert.Inconclusive()** statement that we have removed is no longer needed.

► Run the test again using TDS. (Use "Start Debugging" or <F5>.)

Now the summary near the end of the output in the Console window shows that "all" (both) tests passed.

5.1.8 Refine the new function member and its TDS method

Initially, the purpose of using the TDS method is to help you feed known values to the new function member to make it easy to examine how those values are used and transformed in your code. Any obvious mistakes, such as misspelled names, are easy to correct early, when there is almost no cost to making the correction and the program flow is fresh in your memory.

However, as you add code to your new function member, perhaps to account for unusual circumstances, you may also notice opportunities for additional testing, such as when you add an **`if`** or **`select`** statement, a conditional expression, or a **`where`** clause. You will probably want, at least, to be sure that test cases are present that ensure that every possible branch is exercised by at least one of your test cases. (If some branch never gets used, there's no need to include any code for it.) You will probably also want to specify test cases that provide a variety of extreme or disallowed values, to be sure that your code fails when, and only as, expected. Added code that refers to external fields or other data will call for corresponding properties to be added to the **`testValues[]`** elements to set their values before invoking the function member or checking their values after it returns. You may find it helpful to add other **`Assert`** statements as well.

This is the end of the example involving new method **`Succ()`**. You may delete the program files (in folder …\Demo\ in the example) when you have finished playing with them.

5.2 Example: Adding a new method, Fib(), to a new Solution

Although the new method may be added to an existing VS Project, it is depicted here instead as being placed into a new VS Solution, as we did in the example in section 5.1. If you add it to an existing Solution, it should perform largely as shown here, except that other test methods might appear in the TDS test report. To keep this documentation as simple as possible, what is shown in this example is the output from an initially empty VS Solution.

5.2.1 Overview of this example

This example will illustrate using TDS to help build a new method, using multiple techniques to perform the same calculation. We shall add the new method to the **`Working_Code.NewCode{}`** Class. We shall gloss over most of the details of construction addressed in other examples.

5.2.1.1 Importance of testing

In this example, I attempt to illustrate, based on the easy-to-understand (and well known) Fibonacci sequence, that the testing of software is not a complete solution to all quality-control problems. It may be apparent by now that I believe that testing and commenting code are important parts of development that are often neglected, perhaps because they are not directly reflected in the generated code or because they seem tedious (which is why I share this TDS stuff with you, in an effort to make the process a bit easier).

Poor or absent testing might expose a programmer to ridicule and cause unnecessary hardship to the users. No sane, honest person would intentionally deliver a computer program that claims to perform a calculation but in reality does nothing but raise exceptions or hang in an endless loop, due to some stupid mistake in the code. Any attempt to use such a program would immediately fail. (Obvious mistakes can happen, but if they do, we definitely want to be able to notice them, and as early as possible! I am an accomplished creator of stupid mistakes, but I also try to do a pretty thorough job of removing them.)

5.2.1.2 Importance of analysis

Testing has value, but I claim that analysis is important, too. I have not been able to imagine any testing scheme that would generate the code (see section 5.2.6.3.2) that we shall use in this example. It was derived entirely from analysis, not from testing, so I claim that analysis of a problem can be essential, too, just as testing can be. In this somewhat extreme example, the final, publishable version of this method uses, for some values of its argument, far less (by several orders of magnitude) time and memory than the more intuitive, recursively calculated version does.

In contrast to such obvious, unfortunate outcomes, poor analysis could lead to subtler problems, such as returning unrealistic results that appear to be correct. A classic example of that is results misleadingly expressed to 12 significant digits that are accurate only to zero or one significant digit. In the example that we are building here, the main benefit of the analysis will be avoiding a (major) waste of resources, but the results should be identical in all cases within the restricted domains of input values that we will allow. You might find that in real life the outcome isn't always as clear, so I recommend both good analysis *and* adequate testing, maybe even along with some good communication with your customer. (The communication part of such projects is out of scope in this *TDS User's Guide*.)

In this example (particularly in section 5.2.4), I admit to going slightly over the top in analytical detail; the purpose is to give an impression of the type of thinking that you might find helpful in designing code. If you wish, just have fun skimming over those zigzaggy "Σ" signs, be happy you aren't taking a test on the material, and continue with the following section, 5.2.5, "Requirements for Fibonacci sequence calculation", that resumes playing with C# code.

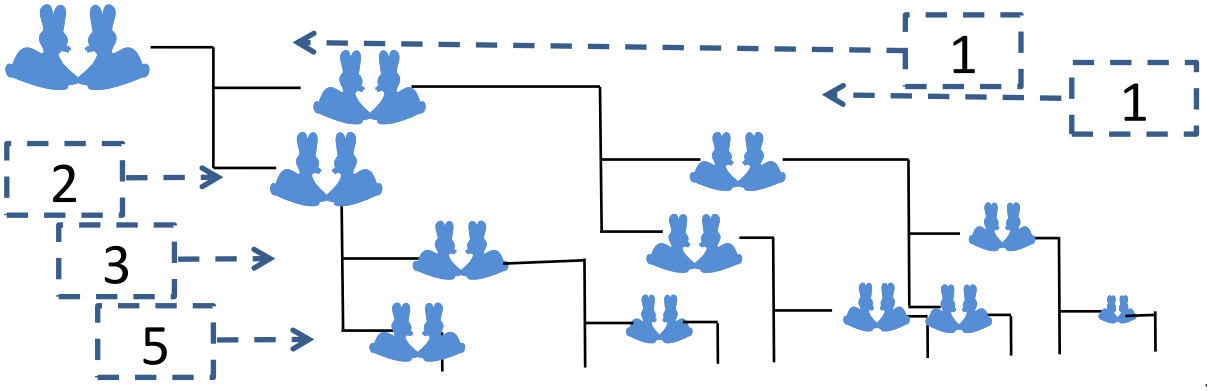
5.2.2 Learning objectives

When you complete this example, you will have done the following;

- (optionally) examin d a mathematical derivation and analysis of the algorithm to be used in a new method
- specified alternate inputs, as test cases in `testValues[]`, to a method for use in tracing its execution
- temporarily filtered the test cases used in running a test
- added new properties to the test cases
- used test-case properties to specify a range of test values
- converted anonymous-object initializers to named-object constructors
- compared the use of initializers with that of constructors for named-type test cases
- determined that the tested method correctly raised some exceptions

5.2.3 Statement of purpose of the code in this example

The end product of this exercise will be a method that calculates a specified element of the Fibonacci sequence, defined as a sequence of integers beginning with (0, 1) and in which each element after those is the sum of the previous two. For example, the first few elements of the sequence are (0, 1, 1, 2, 3, 5, 8, 13…).



Fibonacci Bunnies: In the above family tree, the first couple shown mates and has a (small) litter of one bunny, who grows up and starts his own family. At the same time, let's call it a month, the original parents welcome a second bunny into the world, who does the same thing. (This family is probably not typical of real bunnies, but even with really small, spaced-out litters, it grows impressively.) Each set of parents gives birth to two offspring, a generation apart. The number of new bunny families in successive months is shown.

5.2.4 Analyze the problem mathematically

This section contains mathematical material that you may safely skip over; to do so, skip to section 5.2.5. This discussion provides a rationale for the content of the code that we shall use in the example method that we shall build soon, but the actual C# code will be provided later, in section 5.2.6.3.2. The example code presents several versions of the definition of the Fibonacci sequence, and I claim that they are mathematically identical to each other (for some elements of the sequence), but they are wildly different in the way they do the computation and the resources they consume. Also, if you haven't seen it before, you might consider it

surprising that the various versions of the method that does the computation are really guaranteed to give identical results, and might not see any obvious similarities in the way they work.

5.2.4.1 Derivation using generating functions

The following mathematical discussion is presented as an example of how some analysis of a problem can make a computer program run more efficiently than a direct translation of the definition into code might run. The discussion is intended to show that the calculation to be done by our new method **`Fib(n)`** really does yield the Fibonacci numbers, and how to derive that expression from the recursive definition given to us.

I got the idea of using generating functions in connection with the Fibonacci sequence from Donald Knuth's monumental (and maybe still unfinished) multi-volume book *The Art of Computer Programming*, which includes an example roughly identical to what you see here. I may show a different number of steps in my refactorings of the original expressions, but my intention was to make each step simple enough that each expression would pretty obviously be equivalent to the previous one.

Suppose that you have written a method to calculate the nth element of the Fibonacci sequence, using code similar to its recursive definition. (C# code for a recursive definition may be found in the **`FibTestRecursiveCalc()`** method in section 5.2.8.3.1.2.) Having done so, when you try to run it, you notice that, although your method gives correct answers for small arguments, running it tends to crash your system (it takes a long time and then throws an "Out of memory" exception). This is not very satisfactory, so you look for another way to do the calculations. (Let's suppose that another obvious solution, storing pre-calculated values in an array, is also less than satisfactory, perhaps because you don't have a suitable location in which to store the array.)

Aha! It occurs to you that the definition might lend itself to analysis via a generating function. (Exactly how ideas of this ilk might occur to you is beyond the scope of this discussion, but it probably helps to have seen similar examples in the past.)

Let $fib(n)$ be a function that returns the n^{th} Fibonacci number based on the definition at the beginning of section 5.2.3: $fib(0) = 0$, $fib(1) = 1$, and $n > 1 \Rightarrow fib(n) = fib(n-2) + fib(n-1)$.

(For example, $fib(2) = fib(2-2) + fib(2-1) = fib(0) + fib(1) = 0 + 1 = 1$.)

Let's define a function $f()$ (which will be our generating function) such that

$$f(x) \stackrel{\text{def}}{=} \sum_{n=0}^{\infty} (fib(n))x^n$$

With this definition, the function *f*() can be represented by a power series in which the coefficients are successive elements of the Fibonacci sequence. The first few terms of the function will look something like this:

$f(x) = 0 + 1x + 1x^2 + 2x^3 + 3x^4 + 5x^5 + 8x^6 + \cdots$

We can shift them about by multiplying by *x* and by x^2:

$x \cdot f(x) = 0x + 1x^2 + 1x^3 + 2x^4 + 3x^5 + 5x^6 + \cdots$

$x^2 \cdot f(x) = \quad 0x^2 + 1x^3 + 1x^4 + 2x^5 + 3x^6 + \cdots$

Adding these two expressions, we get

$x^2 \cdot f(x) + x \cdot f(x) = (0+1)x^2 + (1+1)x^3 + (1+2)x^4 + (2+3)x^5 + (3+5)x^6 + \cdots$

, which is pretty close to $f(x)$, except for the missing $(1x)$ at the beginning.

If we add a copy of (x) to this, we get exactly $f(x)$, as shown here:

$$f(x) = x^2 \cdot f(x) + x \cdot f(x) + x$$

$$= x^2 \sum_{n=0}^{\infty} \big(fib(n)\big)x^n + x \sum_{n=0}^{\infty} \big(fib(n)\big)x^n + x$$

$$= \sum_{n=0}^{\infty} \big(fib(n)\big)x^{n+2} + \sum_{n=0}^{\infty} \big(fib(n)\big)x^{n+1} + x$$

$$= \sum_{n=2}^{\infty} \big(fib(n-2)\big)x^n + \sum_{n=1}^{\infty} \big(fib(n-1)\big)x^n + x$$

$$= \sum_{n=2}^{\infty} \big(fib(n-2) + fib(n-1)\big)x^n + \big(fib(1-1)\big)x^1 + x$$

$$= \sum_{n=2}^{\infty} fib(n)x^n + \big(fib(0)\big)x^1 + \big(fib(1)\big)x^1 + \big(fib(0)\big)x^0$$

$$= \sum_{n=2}^{\infty} fib(n)x^n + (0)x^1 + \big(fib(1)\big)x^1 + \big(fib(0)\big)x^0$$

$$= \sum_{n=0}^{\infty} fib(n)x^n$$

$= f(x)$.

Therefore,

$$f(x) = x^2 \cdot f(x) + x \cdot f(x) + x$$

$$f(x) - (x^2 + x) \cdot f(x) = x$$

$$f(x) \cdot (1 - (x^2 + x)) = x$$

$$f(x) = \frac{x}{1-(x^2+x)}$$

$$f(x) = \frac{-x}{(x^2+x-1)}$$

To check this, we can calculate f(0.01) and see that its value is 0.01010203050813…, which seems to match what we know of the beginning of the Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, …) . This is what we expected, so it appears[75] that we are on the right track.

Now we want to express this as a power series, so that we can get expressions for the coefficients of the terms in *f*(); these coefficients, if we can determine them, will also be the Fibonacci sequence terms.

Factor the denominator:

$$f(x) = \frac{-4x}{\left(2x+1-\sqrt{5}\right)\left(2x+1+\sqrt{5}\right)}$$

Express this as a sum of partial fractions (I skipped a few steps here, but you can check this result by adding the two terms to get the previous expression):

$$f(x) = -\frac{\sqrt{5}-1}{2\sqrt{5}x+\sqrt{5}-5} - \frac{\sqrt{5}+1}{2\sqrt{5}x+\sqrt{5}+5}$$

Since (again, I won't derive them here, but it's easy to prove that they are correct — multiply the right-hand side by the expression in parentheses on the left, to see that their product is 1)

$$\left(2\sqrt{5}x+\sqrt{5}-5\right)^{-1} = \sum_{i=1}^{\infty}\left(\left(\sqrt{5}-5\right)^{-i}\left(-2\sqrt{5}x\right)^{i-1}\right)$$

and

$$\left(2\sqrt{5}x+\sqrt{5}+5\right)^{-1} = \sum_{i=1}^{\infty}\left(\left(\sqrt{5}+5\right)^{-i}\left(-2\sqrt{5}x\right)^{i-1}\right)$$

, we get

$$f(x) = \left(-\sqrt{5}+1\right)\sum_{i=1}^{\infty}\left(\left(\sqrt{5}-5\right)^{-i}\left(-2\sqrt{5}x\right)^{i-1}\right) + \left(-\sqrt{5}-1\right)\sum_{i=1}^{\infty}\left(\left(\sqrt{5}+5\right)^{-i}\left(-2\sqrt{5}x\right)^{i-1}\right)$$

$$= \sum_{i=1}^{\infty}\left(-\sqrt{5}-1\right)\left(\sqrt{5}+5\right)^{-i}\left(-2\sqrt{5}x\right)^{i-1} + \sum_{i=1}^{\infty}\left(-\sqrt{5}+1\right)\left(\sqrt{5}-5\right)^{-i}\left(-2\sqrt{5}x\right)^{i-1}$$

$$= \sum_{i=1}^{\infty}\left(\left(-\sqrt{5}-1\right)\left(\sqrt{5}+5\right)^{-i} + \left(-\sqrt{5}+1\right)\left(\sqrt{5}-5\right)^{-i}\right)\left(-2\sqrt{5}x\right)^{i-1}$$

$$= \sum_{i=0}^{\infty}(-1)^i\left(\left(-\sqrt{5}-1\right)\left(\sqrt{5}+5\right)^{-i-1} + \left(-\sqrt{5}+1\right)\left(\sqrt{5}-5\right)^{-i-1}\right)\left(2\sqrt{5}\right)^i x^i$$

[75] But appearances can be deceiving, so we'll prove it in section 5.2.4.2 in addition to deriving it.

$$= \sum_{i=0}^{\infty} (-1)^i \left((2\sqrt{5})^i (-\sqrt{5}-1)(\sqrt{5}+5)^{-i-1} + (2\sqrt{5})^i (-\sqrt{5}+1)(\sqrt{5}-5)^{-i-1} \right) x^i$$

$$= \sum_{i=0}^{\infty} (-1)^i \left((2\sqrt{5})^i \left(\frac{-1}{\sqrt{5}}\right)(\sqrt{5}+5)(\sqrt{5}+5)^{-i-1} + (2\sqrt{5})^i \left(\frac{1}{\sqrt{5}}\right)(\sqrt{5}-5)(\sqrt{5}-5)^{-i-1} \right) x^i$$

$$= \sum_{i=0}^{\infty} (-1)^i \left((2\sqrt{5})^i \left(\frac{-1}{\sqrt{5}}\right)(\sqrt{5}+5)^{-i} + (2\sqrt{5})^i \left(\frac{1}{\sqrt{5}}\right)(-1)^{-i}(-\sqrt{5}+5)^{-i} \right) x^i$$

$$= \sum_{i=0}^{\infty} \left((2\sqrt{5})^i \left(\frac{1}{\sqrt{5}}\right)(5-\sqrt{5})^{-i} + (-1)^i (2\sqrt{5})^i \left(\frac{-1}{\sqrt{5}}\right)(\sqrt{5}+5)^{-i} \right) x^i$$

$$= \sum_{i=0}^{\infty} \left(\frac{1}{\sqrt{5}} \left(\frac{2\sqrt{5}}{5-\sqrt{5}}\right)^i + (-1)^i \left(\frac{-1}{\sqrt{5}}\right)\left(\frac{2\sqrt{5}}{\sqrt{5}+5}\right)^i \right) x^i$$

$$= \sum_{i=0}^{\infty} \left(\frac{1}{\sqrt{5}} \left(\frac{2}{\sqrt{5}-1}\right)^i + \left(\frac{-1}{\sqrt{5}}\right)\left(\frac{-2}{\sqrt{5}+1}\right)^i \right) x^i$$

$$= \sum_{n=0}^{\infty} \left(\frac{1}{\sqrt{5}} \left(\frac{\sqrt{5}+1}{2}\right)^n - \frac{1}{\sqrt{5}} \left(\frac{-2}{\sqrt{5}+1}\right)^n \right) x^n$$

, and we now have it in the form of a power series where the coefficient of x^n is $fib(n)$, the n^{th} Fibonacci number.

5.2.4.2 Inductive proof of correctness

Since we've derived this expression, we know it works, so there's no need to prove that (assuming we made no mistakes). However, the derived expression doesn't exactly *look* like the recursive definition, so a bit of skepticism might be understandable. If you came across it somewhere else, or it was revealed to you in a dream, or you derived it numerically and had doubts about its validity, you could use an inductive proof like the following one, to be sure that this expression will always give the exact correct answer:

We want to prove that what we just now derived is correct, that

$$fib(n) = \frac{1}{\sqrt{5}} \left(\frac{\sqrt{5}+1}{2}\right)^n - \frac{1}{\sqrt{5}} \left(\frac{-2}{\sqrt{5}+1}\right)^n$$

We can calculate the first 2 values of this supposedly correct definition:

$$fib(0) = \frac{1}{\sqrt{5}} \left(\frac{\sqrt{5}+1}{2}\right)^0 - \frac{1}{\sqrt{5}} \left(\frac{-2}{\sqrt{5}+1}\right)^0$$

$$= \frac{1}{\sqrt{5}} - \frac{1}{\sqrt{5}}$$

$= 0$ (which is what we expected).

$$fib(1) = \frac{1}{\sqrt{5}} \left(\frac{\sqrt{5}+1}{2}\right)^1 - \frac{1}{\sqrt{5}} \left(\frac{-2}{\sqrt{5}+1}\right)^1$$

$$= \frac{1}{\sqrt{5}}\left(\frac{\sqrt{5}+1}{2} - \frac{-2}{\sqrt{5}+1}\right)$$

$$= \frac{1}{\sqrt{5}}\left(\frac{(\sqrt{5}+1)(\sqrt{5}+1)+4}{2(\sqrt{5}+1)}\right)$$

$$= \frac{1}{\sqrt{5}}\left(\frac{(5+2\sqrt{5}+1)+4}{2(\sqrt{5}+1)}\right)$$

$$= \frac{\sqrt{5}+1}{\sqrt{5}+1}$$

$= 1$ (which is also what we expected).

These are both correct, and to complete the induction, all we have left to show is that,
for all $n > 1, fib(n) = fib(n-2) + fib(n-1)$.

Substituting the expression that we would like to show is correct for the terms on the right-hand side, we get

$$fib(n-2) + fib(n-1) = \frac{1}{\sqrt{5}}\left(\frac{\sqrt{5}+1}{2}\right)^{n-2} - \frac{1}{\sqrt{5}}\left(\frac{-2}{\sqrt{5}+1}\right)^{n-2} + \frac{1}{\sqrt{5}}\left(\frac{\sqrt{5}+1}{2}\right)^{n-1} - \frac{1}{\sqrt{5}}\left(\frac{-2}{\sqrt{5}+1}\right)^{n-1}$$

$$= \frac{1}{\sqrt{5}}\left(\left(\frac{2}{\sqrt{5}+1}\right)^2\left(\frac{\sqrt{5}+1}{2}\right)^n - \left(\frac{\sqrt{5}+1}{-2}\right)^2\left(\frac{-2}{\sqrt{5}+1}\right)^n + \left(\frac{2}{\sqrt{5}+1}\right)\left(\frac{\sqrt{5}+1}{2}\right)^n - \left(\frac{\sqrt{5}+1}{-2}\right)\left(\frac{-2}{\sqrt{5}+1}\right)^n\right)$$

$$= \frac{1}{\sqrt{5}}\left(\left(\left(\frac{2}{\sqrt{5}+1}\right)^2 + \left(\frac{2}{\sqrt{5}+1}\right)\right)\left(\frac{\sqrt{5}+1}{2}\right)^n - \left(\left(\frac{\sqrt{5}+1}{-2}\right)^2 + \left(\frac{\sqrt{5}+1}{-2}\right)\right)\left(\frac{-2}{\sqrt{5}+1}\right)^n\right)$$

$$= \frac{1}{\sqrt{5}}\left(\left(\frac{2}{3+\sqrt{5}} + \frac{2}{\sqrt{5}+1}\right)\left(\frac{\sqrt{5}+1}{2}\right)^n - \left(\frac{3+\sqrt{5}}{2} + \frac{\sqrt{5}+1}{-2}\right)\left(\frac{-2}{\sqrt{5}+1}\right)^n\right)$$

$$= \frac{1}{\sqrt{5}}\left(\frac{2(2\sqrt{5}+4)}{4\sqrt{5}+8}\left(\frac{\sqrt{5}+1}{2}\right)^n - \frac{2}{2}\left(\frac{-2}{\sqrt{5}+1}\right)^n\right)$$

$$= \frac{1}{\sqrt{5}}\left(\left(\frac{\sqrt{5}+1}{2}\right)^n - \left(\frac{-2}{\sqrt{5}+1}\right)^n\right)$$

$$= \frac{1}{\sqrt{5}}\left(\frac{\sqrt{5}+1}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{-2}{\sqrt{5}+1}\right)^n$$

$= fib(n)$ ■

Woo hoo![76]

[76] 100 years ago, people would have said "QED" here, for "quod erat deonstrandum" (= "which was to be demonstrated"), and nowadays it's usually just a simple end-of-proof "■" sign, but I think "Woo hoo!" more accurately conveys the emotional impact.

Although this proof does nothing to show where the expression giving our definition came from, as the previous derivation showed, it does demonstrate that the given expression always gives an exactly correct calculation of $fib(n)$.

5.2.4.3 Simplified version, using the Golden Ratio

We can express this result slightly more concisely, for the purpose of calculation in our program. Suppose we define *phi*, also written as "φ", also called the "Golden Ratio", to be

$$phi \stackrel{\text{def}}{=} \frac{\sqrt{5}+1}{2}$$

With this definition, the n^{th} Fibonacci number, which we have determined is (exactly)

$$\frac{1}{\sqrt{5}}\left(\frac{\sqrt{5}+1}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{-2}{\sqrt{5}+1}\right)^n$$

can be simplified, using *phi*, to

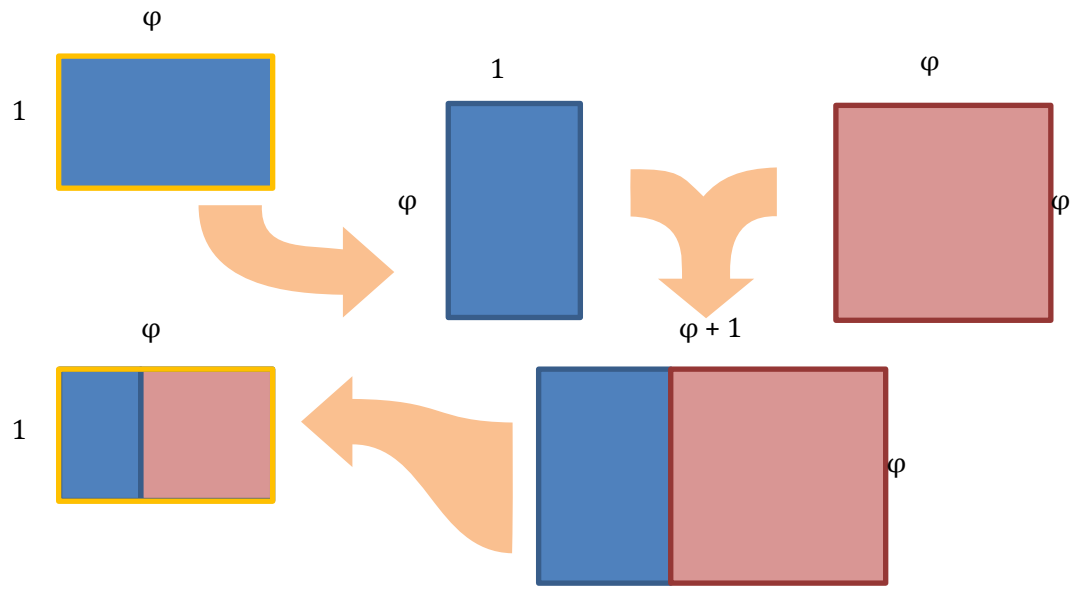
$$\frac{(phi)^n}{\sqrt{5}} - \frac{(1-phi)^n}{\sqrt{5}}$$

Since the 2nd term, $(-\frac{(1-phi)^n}{\sqrt{5}})$, always has an absolute value < 0.5 if n ≥ 0, this expression can be written even more simply (by rounding the nearest integer) as

$$fib(n) = floor\left(\frac{(phi)^n}{\sqrt{5}} + .5\right)$$

(The "+ .5" part of the expression gives us the nearest integer.)

This expression, for large *n*, should be (much) faster to calculate than by using the recursive definition of $fib(n)$ shown in section 5.2.8.3.1.2.



Golden Rectangle: There is only one shape of rectangle which, if you attach a square to its longer side, will form a rectangle of the original shape.

This is the expression that we shall use to calculate the value to be returned by our new method, **`Fib(n)`**.

5.2.5 Requirements for Fibonacci sequence calculation

We want to write a method, let's call it **`Fib()`**, that, given an element number in the Fibonacci sequence, will return its value. For example, we want **`Fib(7)`** to return a value of 13. If too high an argument is passed, an exception is to be raised. Besides having it return the correct value, we also wish to minimize the amount of storage and the amount of processing time that the method consumes.

For the purposes of this calculation, we'll call the first "1" element number 1, so element number 2 will equal 0 + 1, or 1. The sequence that we will calculate should thus begin with element 0 = 0, followed by (1, 1, 2, 3, 5, 8, 13, 21, ...). We shall consider "too high" to be whatever argument would return a result that will not fit into a C# **`(uint)`** field (32-bit unsigned integer).

This statement describes what result we want but says little about the means to provide that result. It might be thought of as a contract that promises that the function member (in this case, the new method **`Fib()`**) will perform the specific action described here, in this case returning a correct value, whenever it is invoked.

5.2.6 Set up a new function-member stub and its TDS method

(We shall develop the next example, in section 5.3.6, similarly to this one.)

5.2.6.1 Create a TDS method to exercise `Fib()`

5.2.6.1.1 SET UP A PROJECT WITH TDS CODE

If you already have a VS Solution with a TDS Project to which you want to add this example code, open that Solution and skip to section 5.2.6.1.2.

- Follow the steps in section 4.14.7, "Setting up a stand-alone TDS Project", to construct a new VS Solution.

The VS Solution that you have just now constructed should contain only a (mostly empty) "ConsoleApp1" Project and the "TDS" Project that we shall use to exercise the **`Fib()`** method, which will be our working code in this example.

5.2.6.1.2 ADD A TDS METHOD

In file TDS.cs, we shall create a new TDS method, **`FibTest()`**, to call our about-to-be-defined working-code method.

► Within the **`TDS.Test{}`** class in file TDS.cs, after the "**`TODO: New TDS methods may be placed here:`**" Task List comment near the end of the file, use the **`TdsTest`** code snippet to generate a TDS method for the to-be-defined method **`Fib()`**, as we did in sections 4.8.2.1 and 5.1.5.1.1.2 . Type the name "Fib" into its "TestableFunctionMember" field and press <enter>[77].

5.2.6.1.3 CREATE AN EXAMPLE WORKING-CODE NAMESPACE

You may place your working code into an existing namespace or create a new one. These instructions assume you are creating a new one, using the names shown here, but using an existing namespace and class (as we did in section 4.3.6.2) will work, too, if they are accessible from the TDS methods.

Let's assume that our new function member, **`Fib()`**, will be located in new namespace **`Working_Code`** and class **`Working_Code.NewCode{}`**.

► Define these by placing the following code at the end of file Program.cs in the ConsoleApp1 Project, following all the existing code in that file (that is, following the closing brace, "**`}`**", of namespace **`ConsoleApp1`**):

```
/// <summary>
/// Simulated working code to be exercised by TDS methods
/// </summary>
namespace Working_Code
{
  /// <summary>
  /// Class containing methods to be developed
  /// with the help of TDS
  /// </summary>
  public class NewCode
  {
  }  // end: NewCode{}
}  // end: Working_Code namespace
```

(The XML comments on the **`Working_Code`** namespace are ignored by the Object Browser, but they are harmless.)

► In the Solution Explorer, in VS Project **`TDS`**, set a Reference to VS Project **`ConsoleApp1`**. (See section 4.4.1.2.)

► Following the Task comment "**`TODO: Usings -- Include "using" statements for the namespaces of the code`**" in file TDS.cs, insert the statement

---

[77] After pressing <enter> to complete the snippet, I would also change "Fib" in the XML comments to "Fib()" to indicate that the function member being called is a method, but this is just a matter of style and is completely optional.+


```
using Working_Code;
```

At first, this will be grayed out in the VS editor, since this namespace is not being used yet.

Now we're ready to start using the **Working_Code** namespace to add some simulated working code.

5.2.6.1.4 GENERATE A NEW METHOD BASED ON ITS INVOCATION

▶ To link the new TDS method with the not-yet-defined method, **Fib()** that it is to call, in the Task List window go to the "**TODO: FibTest() -- Provide a suitable calling expression**" Task and change the

```
    actual = Fib(tCase.Arg);
```

statement in it to invoke the new method, now to read

```
    actual = NewCode.Fib(tCase.Arg);
```

The VS editor, now that the **using** statement and reference are set, will help by showing the class name, "**NewCode**", as a member of a pop-up menu as you begin typing its name before the space before "**Fib(**". It also flags "**Fib**" with a wiggly underline, since we still haven't defined Fib().

▶ At the "**TODO: FibTest() -- Use a suitable default value.**" Task, change

```
    var actual = 0;
```

to

```
    var actual = 0U;
```

to declare it as a **(uint)**, since we will be comparing it with the unsigned-integer **(uint)** values to be returned by **Fib()**.

5.2.6.1.5 ADD THE TDS METHOD'S NAME TO TESTMETHODSTOBERUN

▶ In TDS.cs, into the literal string following the "**TODO: TestMethodsToBeRun -- List all TDS test methods to be run.**" Task comment, enter the name of the TDS method that we have just now defined, "**FibTest()**".

This name is case sensitive, but the parentheses are optional.

▶ Since we want to focus on this new TDS method, temporarily comment out or erase any other tests listed in **TestMethodsToBeRun**, as we did in section 4.8.2.5.

5.2.6.2 Create a method stub for Fib() .

5.2.6.2.1 GENERATE AND CUSTOMIZE THE STUB

▶ At the "**TODO: FibTest() -- Provide a suitable calling expression**" Task, in the call to **NewCode.Fib()**, right-click on **Fib()** and select "Quick Actions, Generate method 'NewCode.Fib' ".

Equivalently, hover the mouse pointer on the name "**Fib**", click on the menu that appears, and choose this "Generate method" option.

▶ To find the definition of the stub, right-click on its name and select "Go To Definition" (or press <F12>).

The **throw** statement that it contains we will replace soon, so leave the statement unchanged. Since we will replace it almost immediately, don't bother adding the message we might normally apply to a **throw** statement, such as this:


```
        throw new NotImplementedException("Unnecessary exception message");
```

► In **NewCode.Fib()**, change the parameter name from **arg** to **n**, to match the specification statement in our documentation.

Yes, you could call it anything you wish, but let's not make it any trickier than necessary to match the code with the description, and this method's description uses "**n**" for the element number.

► Optionally, add a matching comment to the closing brace.

The result might look something like this:

```
    public static uint Fib(int n)
    {
      throw new NotImplementedException();
    }  // end: Fib()
```

► If you wish, move this new definition of **Fib()** into alphabetical order (or whatever location makes sense to you) within the **NewCode{}** class.

Since this class was empty, this definition of **Fib()** is already in order, but otherwise now would be a convenient time to move it to wherever it belongs.

5.2.6.2.2 ADD XML COMMENTS

5.2.6.2.2.1 EMPTY TEMPLATE

For this method, we could type "**///**" on a blank line immediately before the definition of **Fib()** to generate the following XML comments (but don't do it yet; I'll soon provide a more complete version that you may simply copy):

```
    /// <summary>
    ///
    /// </summary>
    /// <param name="n"></param>
    /// <returns></returns>
```

As you set up your own function members, this template will save a bit of time in identifying key information about them, and it will help avoid misspelling the parameter names.

5.2.6.2.2.2 BASIC VERSION

We could add a little to these XML comments to give a basic idea of what the method is expected to do, maybe including a "TODO:" if we don't have all the details at hand right now.

```
    /// <summary>
    /// Return the nth Fibonacci number
    /// </summary>
    /// <remarks>
    /// TODO: Fib() -- Explain what the Fibonacci sequence is
    /// </remarks>
    /// <param name="n">Index into the Fibonacci sequence</param>
    /// <returns>The nth Fibonacci number</returns>
```

However, since in this example we already have usable specifications for this new or to-be-updated function member, we can copy those specifications, or an abbreviated version, into the function member's XML comments. For this method, we include some additional details about the definition of the sequence, the limiting values, and exceptions that might be raised; such information might help us design tests.

Note that, since these comments still need some more detail, in this version of the XML comments we have inserted into the **<remarks>** element a Task List comment containing the method's name, **Fib()**, as a reminder to add relevant details.

When you insert text into XML comments, remember to escape any special HTML characters that the pasted or typed material might contain, as described in section 5.1.5.1.4 above.

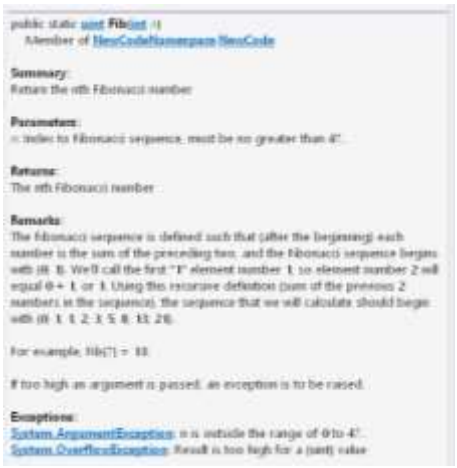
I am usually somewhat liberal with the contents of my XML comments, putting into them roughly everything that I think might be helpful to someone who needs to use the code, especially into the **<remarks>** section. (I feel that it should not be necessary to read a function member's source code to understand how it's intended to be used.) Even I have limits, though, in the amount of detail I feel is needed, and in this example, the mathematical analysis in section 5.2.4 is lengthy enough that I would put it where it won't clutter the IntelliSense pop-ups, such as into an **<example>** section of the XML comment, or into an ordinary (non-XML) comment, or into a separate document to which I could refer in the comments.

5.2.6.2.2.3 MORE COMPLETE XML COMMENTS

- Copy the following XML comments to the line before the definition of **Fib()**, as shown here:

```
/// <summary>
/// Return the nth Fibonacci number
/// </summary>
/// <remarks>
///The Fibonacci sequence is defined such that
/// (after the beginning) each number is
/// the sum of the preceding two,
/// and the Fibonacci sequence begins with (0, 1).
/// We'll call the first "1" element number 1,
/// so element number 2 will equal 0 + 1, or 1.
/// Using this recursive definition
/// (sum of the previous 2 numbers in the sequence),
/// the sequence that we will calculate
/// should begin with (0, 1, 1, 2, 3, 5, 8, 13, 21).
/// <para>For example, Fib(7) = 13.</para>
/// <para>If too high an argument is passed,
/// an exception is to be raised.</para>
/// </remarks>
/// <param name="n">Index to Fibonacci sequence,
/// must be no greater than 47.</param>
/// <returns>The nth Fibonacci number</returns>
/// <exception cref="ArgumentException">
/// n is outside the range of 0 to 47.</exception>
/// <exception cref="OverflowException">Result
/// is too high for a (uint) value</exception>
```


This version replaces the Task List item in **<remarks>** with a description, and adds details to the **<param>** and **<exception>** elements. The C# syntax makes this a bit tricky to read in the source code, but it looks better in the Object Browser window:



How much detail to include in XML comments is a matter of style, but this version is close to what I would use in the code in my personal projects. Of course, in this type of matter, I would definitely defer to the wishes of my boss or my customers! However, if they don't care, I prefer to save the info with the code, in case I need to use the code in the future, or to update it.

5.2.6.3 Do a first manual test

5.2.6.3.1 BEGIN TRACING

To verify that the method is being invoked properly,

► In the definition of the **Fib()** method, on its **throw** statement, set a breakpoint (perhaps via <F9>).

► Use VS menu "Debug, Start debugging" or <F5> to begin running the program until it reaches the breakpoint, demonstrating that the **Fib()** method is being called (and that there are no C# syntax errors).

For example, if I misspelled "FibTest" as "Fibtest" in **TestMethodsToBeRun**, I'd likely not stop at a breakpoint, and I would instead see a test report indicating that **FibTest()** had not been run at all.

► Examine the value of the parameter.

For example, in the VS Locals window, or by hovering the mouse pointer over the declaration of **n** in the parameter list, observe that it has the value of 3 passed by the calling TDS method.

► Cancel running the program, for example via VS menu "Debug, Stop Debugging" or <shift><F5>.

5.2.6.3.2 ADD SOME CODE TO **FIB()**

Since we have determined that it's being called correctly, we shall now add some code to **Fib()**, as an attempt at doing the desired calculation.

► Copy the following code into the definition of **Fib()**, before the **throw** statement. This code implements the alternate definition that we derived in section 5.2.4.3 above, "Simplified version, using the Golden Ratio",

using *phi*, and it also raises an exception if the value of the argument is too great to allow returning an unsigned 32-bit **(uint)** value.

```
      //HACK: Fib() -- Change to 48 to observe overflow failure
      const int upperLimit = 47;
      var phi = (Math.Sqrt(5.0) + 1.0) / 2.0;
      if (n < 0 || n > upperLimit)
        throw new ArgumentException(string.Format(
@"n must be between 0 and {0}, inclusive.
    The value specified for n was {1}."
                , upperLimit  //{0}
                , n  //{1}
              ));
      //HACK: Fib() -- Temporarily remove "checked" keyword
      //  or change it to "unchecked" to see false results
      return checked((uint)Math.Floor(Math.Pow(phi, n)
            / Math.Sqrt(5.0) + .5));
```

In this code, **Math.Pow(phi,n)** returns (**phi**) raised to the power **n**, **Math.Floor(x)** returns the greatest integer that is no higher than **x**, and **Math.Sqrt(x)** returns the square root of **x**.

If we were to set the value of **upperLimit** too high, we might see that an overflow exception is raised when the calculation gets into trouble. Using **upperLimit** lets us avoid wasting resources by skipping the calculations if the result is doomed to failure (in this case, by raising an overflow exception).

This code also includes some "**HACK:**" Tasks that we shall use soon to demonstrate that the exceptional conditions are being handled properly.

- Since the method can now return a value, delete the previously included

```
      throw new NotImplementedException();
```

statement.

Doing this also cancels the breakpoint that we set earlier.

5.2.6.3.3 TRACE INTO THE NEW DEFINITION

Now that we have added some working code to the new function member, we can call it using some example data and trace its execution.

- In the definition of **Fib()**, place a breakpoint on the **if()** statement.

- Use VS menu "Debug, Start debugging" (or <F5>) to begin running the program until it reaches the breakpoint. We can now observe, for example in VS's "Locals" window, that **n** has a value of 3, **phi** has a value slightly greater than 1.618, and **upperLimit** has a value of 47.

- Use VS menu "Debug, Stop Debugging" (or <shift><F5>) to cancel running the program.

- Remove the breakpoint.

5.2.6.4 Specify an additional set of input values

At this point, we have enough code in our TDS method, **FibTest()**, to be able to feed any value to **Fib()** that we wish to use in tracing its execution, through its (only) parameter. In developing **Fib()**, we might have it

incorporate additional inputs, such as new parameters or settable fields or properties that are also accessible[78] to **`FibTest()`**. If we do so, then we can also add properties to the **`testValues[0]`** object to use in assignment statements to set those variables before invoking the new method, **`Fib()`**. (Depending on what our tests change, we might also need to consider including code near the end of the TDS method, or perhaps in **`TDS.Test.CleanupTestSession()`**, to restore any changed data members, disk files, etc., to their previous states.)

Having set up an initial test case in **`testValues[0]`** that defines all the properties we expect to need, we can add a second element to the **`testValues[]`** array to specify different property values. Let's suppose that we want to run our new method using a different set of inputs, checking that the results are what we expect. Currently, there is only such input — the method's parameter, **`n`**. Let's try using another value besides 3. Let's use 48, which we expect will raise an exception.

We can create a new **`testValues[]`** element by copying an existing one, deleting the copy's comments, and specifying new values for the properties. We shall copy **`testValues[0]`** to create the new **`testValues[1]`**.

- Navigate to the definition of **`testValues[0]`** in **`TDS.Test.FibTest()`.**

You may do this via the Task List, by double-clicking the "**`TODO: FibTest() -- Define inputs and expected outputs.`**" Task.

To save time, instead of copying **`testValues[0]`** and editing it as described here, for now we can replace it (and its //TODO: Task comment) with the example code shown below. (However, the actual editing is usually not difficult to do.)

[78] If the working code uses as input or output some fields or properties that are not accessible to its TDS method, it might be necessary to use a wrapper method that makes them accessible; see section 4.8.8.3.

► Replace **testValues[0]** and its //TODO: Task comment with the following code for **testValues[0]** and **testValues[1]**:

```
//TODO: FibTest() -- Define inputs and expected outputs.
new {
  Id = "01 Low, valid input", // Test case identifier
  Arg = 3,  // Index to Fibonacci sequence
  ExceptionExp = DefaultExceptionMessage,  // Expected exception
  ValueExp = 2U,  // Selected element of Fibonacci sequence
},

new {
  Id = "10 This should raise an exception",
  Arg = 48,  //This should cause overflow
  ExceptionExp = "n must be",
  ValueExp = 4U,
},
```

In this revised, expanded version, I trimmed some of the comments in **testValues[0]** and specified that **ValueExp** be a **(uint)**, since this is the type that will be returned. I changed ValueExp from 4 to 2, since 2 is obviously the desired value of **Fib(3)**. The value "**4U**" value specified for **testValues[1].ValueExp** is unimportant; it is ignored because an exception is (or should be) thrown.

I moved the "**TODO: FibTest()**" Task comment to just before **testValues[0]**, but left it active so that it could continue to be listed in the Task List, to make it easy to find **testValues[]** as we update the test cases. I'll probably remove it about the same time that I remove the **Assert.Inconclusive()** statement at the end of the TDS method, when I've largely finished updating it and the working code that it tests.

5.2.6.5 Include comments on the test case, if needed

The comments on the properties of the first test case, **testValues[0]**, were intended to describe the properties, rather than to identify anything special about the values being used in that specific test case. After the first test case, there is no need to repeat these comments, as each of the test cases contains the same properties. (For more on the suggested location of comments in elements of **testValues[]**, please see section 4.14.10.2.)

In test cases after the first one, I sometimes apply a comment to a property to describe some special quality of that value. For example, on the

```
  Arg = 48,  //This should cause overflow
```

line in the new test case, I have placed a comment indicating that 48 is expected to produce unsigned-integer overflow.

Comments that properly apply to an entire test case, rather than to a property or a particular value, may be included in the value of that test case's **Id** property. The value of this **Id** property contains comment-like information specific to this entire test case, which in this instance is "**This should raise an exception**". Including this note in the **Id** value allows the information to be used in a message in the test report relating to a failed test case.

The value we give the **ExceptionExp** property here is taken from the beginning of the string we specified in the new **throw** statement in **Fib()**. We include enough of that message to distinguish it from any other

exception that we expect to pop up during the execution of **`Fib()`**. We can include a fuller explanation in the **`Assert`** statement that is thrown if the test ever fails.

Note that, if we were to use a named type for the test cases (as we did in section 4.8.6 of the Tutorial and shall do soon, in section 5.2.9.6 of this example), the comments that apply to one of the properties rather than to a specific value of the property would be located in the property's definition within the definition of the named type, rather than on that property's line in **`testValues[0]`**. Using an anonymous type, as we're doing here, limits our choices of places to add comments.

▶ Run TDS. (Press <F5>.)

5.2.6.6 Handle exceptions, if necessary

In case you did not disable the "**`Assert`**" exception messages as suggested in section 4.14.7.4, and as described in section 4.4.2, you will likely get an "Exception User-Unhandled" pop-up message.

▶ If it's a **`TDS.AssertInconclusiveException`** exception, uncheck the "Break when this exception type is user-unhandled" box (in the "Exception Settings" menu) and press <F5> again to finish processing.

The test returns a result of "Inconclusive", since no failure occurred but our TDS method is not yet complete.

However, if we see an "Exception Thrown" pop-up window for any other type of exception, it is probably a real exception, not the result of an **`Assert`** statement, and we don't want to ignore it. Don't uncheck its "Break when this exception type is thrown" box. An unexpected exception normally indicates that something's gone badly wrong, such as a typographical error that was not caught by the compiler. We should examine the exception message to determine the cause of the exception, then stop execution (via <shift><F5>) and correct the faulty code. We may rerun the program after making corrections.

5.2.6.7 Observe overflow failure

To observe the results of not checking for overflow in the working code, we can temporarily buggify the code in **`Fib()`** and test it.

▶ At Task '**`HACK: Fib() -- Temporarily remove "checked" keyword`**', change "**`checked`**" to "**`unchecked`**".

▶ At Task "**`HACK: Fib() -- Change to 48 to observe overflow failure`**", change 47 to 48 or some higher number.

▶ Run TDS.

If an "Exception User-Unhandled" pop-up message appears for a **`TDS.AssertFailedException`** exception, uncheck the "Break when this exception type is user-unhandled" box (in the "Exception Settings" menu) and press <F5> to resume processing.

The test report contains the following message:

```
  The following test method returned a status of Failed:

  - FibTest()
      Exception message:
Assert.IsTrue failed.
FibTest(), test case 10 This should raise an exception:
  No Exception was raised in this test case,
  but Exception "n must be" was expected.
```


So the test failed because the working code did not raise any exception when it should have raised one.

Although it's not obvious from this report, **`Fib()`** also returned a wrong value of 512,559,680, as you may observe by setting a suitable breakpoint. If you happened to examine it, you might notice that this number is far too low, but the program, absent a suitable exception, would otherwise merrily continue as if nothing were amiss (i.e., would not exhibit ideal computer behavior).

5.2.6.8 Restore "checked" behavior

- At Task '**`HACK: Fib() -- Temporarily remove "checked" keyword`**', change the word "**`unchecked`**" immediately following "**`return`**" to "**`checked`**".

This will allow the program to again check for overflow.

- Run TDS.

The test fails again, but the message is slightly different:

```
  The following test method returned a status of Failed:

  - FibTest()
      Exception message:
Assert.IsTrue failed.
FibTest(), test case 10 This should raise an exception:
  The expected exception should start with "n must be".
  This unexpected exception was thrown:
    "Arithmetic operation resulted in an overflow."
```

An exception was raised this time, just not the expected one.

So we see that with an upper limit of 48 we encountered overflow; we can short-circuit that failed calculation by lowering the value of the constant **`upperLimit`** in **`Fib()`** to 47.

- Close the Console window.
- At Task "**`HACK: Fib() -- Change to 48 to observe overflow failure`**", change 48 back to 47.
- Run TDS.

This again gives us the results we saw earlier; as before, the result is "Inconclusive".

- Close the Console window.
- In **`Fib()`**, remove both of the "**`HACK:`**" Task comments.

5.2.6.9 Add a test case for high input value

- Add another test case to **`FibTest()`**:

```
        new {
          Id = "11 Highest valid value",
          Arg = 47,  //This should return a valid (uint)
          ExceptionExp = "",
          ValueExp = 4U,
        },
```

We expect that **`Fib()`** will not raise an exception from this, but the test should fail because **`ValueExp`** will not match the returned value.

► Run TDS to determine the (apparently) correct value.

The test report shows us this:

```
Assert.AreEqual failed. Expected:
<4>. Actual:
<2971215073>.
```

► In **testValues[2]**, which has **Id** tag "11", change the value of **ValueExp** to **2971215073U,** close the Console window (via <enter>), and run TDS again (via <F5>).

This number may be copied from the error message in the Console window (see section 4.8.3.2).

The test should return a status of "Inconclusive".

Actually, in this specific case (but not the other test cases), the **"U"** integer-type suffix is not necessary, since the number is too high to be mistaken for an **(int)**. The program will compile without it, but I included it for consistency with the other test cases, thus making the program a bit easier to read.

The test passes now, and the **upperLimit** value accurately filters out improper values of **n**, at least at the high end. We haven't checked for negative values of the parameter and, actually, will leave that as an exercise for the reader[79]. In real life, such a test would be necessary, unless we chose to circumvent it by making the parameter a **(uint)** as well. (But that's not always possible, and besides that, the original specification called for passing an **(int)** to the method, so that kind of change it would require a conversation with the customer.)

5.2.6.10 Check the results

Now, we have kind of cavalierly accepted the value returned with an index of 47 as being correct. (If you, in a skeptical mood, already checked that 2,971,215,073 is the 47th Fibonacci number, or if you happen to already know that, good for you! For this example, I'm assuming that, to save time and effort, we are letting the computer do some of the work.)

Merely accepting the function member's results unquestioningly is a bit sloppy. As mentioned in section 4.8.3.1, we should take extra care in claiming that an expected value is accurate. Fear not — we are going to get some independent confirmation of this and some other values by writing a bit of additional code (to be revealed presently). That will allow us to compare the values returned by **Fib()** with those generated by some alternative means.

In some other situations, we might determine values like this by checking reference works or using high-precision utility packages or doing detailed calculations by hand. The point here is that it is probably wise to have an objective basis for choosing criteria for our tests, such as information from these reference works, etc.

5.2.6.11 Filter the test cases

5.2.6.11.1 RATIONALE

As mentioned in section 4.14.4, suppose we want to run only one or two selected test cases, so that we can trace into our working code without wasting time on test cases that do not use the code that interests us.

Although the code in this "**Fib()**" example does not have a complex structure, let's assume that what we want to do is to examine in some detail the method's behavior when it encounters an extreme value — like the value

[79] The method should raise an exception for negative parameter values.

47 that we just fed it. For the moment, let's assume that we are not interested in any other values, so we want to use only the test case with the label "10 This should raise an exception".

5.2.6.11.2 ENABLE FILTERING

- As we did in section 4.8.7.1, enable test-case filtering in the file in which **`FibTest()`** is defined, file TDS.cs,

To do so, uncomment the line "**`//#define RunOnlySelectedTestData`**" near the beginning of the file (near line 37).

If you think it might help, you could also add a Task List comment such as "**`//TODO: FibTest() -- Remove the test-case filter`**", making the code look like this:

```
//TODO: FibTest() -- Remove the test-case filter
#define RunOnlySelectedTestData
```

Doing this would place this Task near the others in the Task List for this test case, so it woud be easy to find and remember, but I usually don't do it because the code is already easy to find (being near the beginning of its file) and to remember (since the test report displays a reminder, and also because the compiler issues a Warning). For a description of using "**`//TODO:`**" comments as a navigation aid, please see section 5.1.5.1.6 above.

Instead of adding a Task, you might prefer to set a bookmark (menu "Edit, Bookmark, Toggle Bookmark") on the **`#define`** line.

Since we have already (in section 5.2.6.1.5) suppressed running any other defined TDS methods, filtering the test cases will affect only **`FibTest()`**.

5.2.6.11.3 SPECIFY THE TEST-CASE FILTER

- In the definition of **`TDS.Test.FibTest()`** , near the beginning of the "**`#region testValues[]`**" region, change the statement

```
const string testSelectionList = @"01";
```

to be

```
const string testSelectionList = @"10";
```

To navigate there via the Task List, go to Task "**`TODO: FibTest() -- Define inputs and expected outputs.`**" and go up about three lines. Otherwise, you may look for the first statement within **`FibTest()`**.

This specifies that we want to run only the test cases whose **`Id`** tags start with "10" — a set that currently includes only the test case in **`testValues[1]`**. This test case has the label "10 This should raise an exception", which begins with the tag "10" that we just now specified in **`testSelectionList`**.

If we run TDS now, the test report shows that our watchdog test, **`AllTestsAreToBeRunTest()`**, Failed (because specifying a filter may cause us to skip running some of the test cases) and the status of **`FibTest()`** is Inconclusive (because the only test case that we did run, case 10, did not Fail). Any other TDS methods defined in our Solution would be listed on the report as not being run (because they are not listed in **`TestMethodsToBeRun`**).

We could list the tags of other cases here as well, separated by spaces. Regardless of the order in which we list them here, or the number of times we list any of them in this statement, the test cases will be run in the order in which they appear in **`testValues[]`**, and only once each.

Although the comment on **`testValues[0].Id`** says that the identifying tag consists of "two or three" characters, that's only a suggestion. The tags may be of any length, as long as they do not contain blanks. If we specified "**`1`**" instead of "**`10`**", all the test cases with **`Id`** values beginning with "**`1`**" would be run, including both **`"10 This should raise an exception"`** and **`"11 Highest valid value"`**.

Since the tag at the beginning of each **`Id`** value is used for filtering test cases, I suggest that you make them unique. If you specify duplicate **`Id`** tags in **`testValues[]`** elements in the same TDS method, nothing terrible will happen (no error message will appear), and all of the selected test cases will be run. However, there will be no way to run them separately using **`testSelectionList`**, and **`Assert`** error messages identifying specific test cases may be ambiguous.

If you have many test cases to manage (say, 50 or 100 cases), you might find it helpful to specify a naming scheme in which, for example, all those test cases intended to raise an exception might have **`Id`** values that begin with "**`X`**" and all others might begin with "**`A`**" or "**`B`**".

If you set up that naming convention and then wished to test only those cases that should *not* raise exceptions, you could use

```
const string testSelectionList = @"A B";
```

to allow you to run or omit test cases by group instead of individually. The "**`A`**" and "**`B`**" groups of test cases would be run, but not the "**`X`**" group (the exception-raising cases).

5.2.6.11.4 NOTES ON THE TEST-CASE FILTERING SCHEME

Note that the TDS code does not support using an expression involving only "**`X`**" that would have the effect of excluding all the "**`X`**" test cases. Any test case with an **`Id`** property whose tag begins with at least one of the substrings in this list will be run. Choose your naming scheme accordingly.

In case you're wondering, this filtering scheme was chosen to make it easy to run a single test case, or a small set of related test cases, if you want to track down a specific problem. It wasn't obvious what importance a general subset of cases would have, short of the full set, so the current version of TDS makes no special provision for general subsets.

However, even though TDS provides no automatic specification of general subsets of test cases, there is a fairly easy way to achieve a similar result if you need it. You could list in **`testSelectionList`** the tag of every test specified in **`testValues[]`**, for example generated via statements like these:

```
#region Generate tag list
Console.Clear();
Console.WriteLine(
      @"const string testSelectionList = @""{0}"";"
      , (from element in testValues
         select element.Id.Split(new[] { ' ' },
           StringSplitOptions.RemoveEmptyEntries)[0])
        .OrderBy(t => t)
        .Aggregate((list, tag) => list + " " + tag)  //{0}
      );
return;  //Set a breakpoint here
#endregion Generate tag list
```

This statement assumes that each **`Id`** value begins with a tag that is followed by a space.

You could use this code by performing the following steps:

- copy these lines and paste them immediately after the

```
#endregion testValues
```

line in a TDS method

- place a breakpoint on the **`return;`** statement
- run to the breakpoint
- copy the result from the Console window (as we did in section 4.8.3.2) as a statement listing all the current **`testValues[].Id`** tags.
- stop debugging (<shift><F5>)
- paste this copied statement in place of the existing **`testSelectionList`** definition statement
- delete the "**`#region Generate tag list`**" region and its contents

In the current example, the result might look like this, after the copied line is pasted into the TDS method's code in VS:

```
const string testSelectionList = @"01 10 11";
```

This statement is allowed to occupy multiple lines, so you may want to improve legibility by breaking it, for example to look like this:

```
      const string testSelectionList = @"
  01
   10 11
";
```

(Sorry, TDS makes no provision for commenting out lines or individual values within this string.)

With all the existing **`Id`** tags conveniently listed in this one statement, you could then selectively erase the tags of all of these tests that you wish to skip, allowing you to select any subset of them[80]. This would be effective only when the test cases are filtered, since, regardless of any such commenting-out, all of the test cases in **`testValues[]`** are run whenever the "**`#define RunOnlySelectedTestData`**" directive at the beginning of the source file is disabled.

5.2.6.11.5 REORDER THE TEST CASES

Unless some of your test cases make changes that are intended to affect the code as it runs later test cases (which I recommend *avoiding*, unless any such dependencies are well documented), you may wish to reorder the elements of **`testValues[]`** to group together test cases having similar functions. This might make it easier to read the list, helping you to avoid needlessly repeating existing tests, or to ensure that all members of some set of options in your code are properly addressed. The test cases will be run in the order in which they appear in **`testValues[]`**.

There's not much need to reorder the test cases in this example, as we have only a few of them defined.

5.2.6.11.6 TURN OFF FILTERING

► Assuming that we have finished looking at the test case that we needed to look at in detail, we disable filtering by commenting out the

[80] If you think that someone might have reason to do this frequently, you could put a statement like this into a method or into a code snippet, to be called as needed. However, since I could not imagine any obvious need for such a feature, I have not included it in TDS.cs .


```
#define RunOnlySelectedTestData
```

directive near the beginning of TDS.cs, to make it again look like this:

```
//#define RunOnlySelectedTestData
```

► Also, if you placed a "**TODO: FibTest() -- remove the test-case filter**" Task comment there, delete that Task.

In general, except when we need to filter out some of the test cases in a TDS method, and especially during later testing, quietly omitting some of the test cases could give the false impression that some test cases Passed that would have Failed had they been run (as we saw in section 4.8.7.1), so this directive should be used only when it is helpful, and not on a routine basis.

5.2.6.12 For tracing, TDS is complete.

As in the previous example, if you planned to use this TDS test method only in support of tracing execution in **Fib()** (which is what we did in section 5.2.6.3.3) and have no intention of doing any testing with it beyond what we've already done, it has accomplished everything it needs to accomplish.

5.2.7 Overview of using alternate calculations

Up to here, we have specified and run a few tests of individual values. For the remainder of this example, we shall look at calculating the same results in several different ways, to give us confidence that the results are valid. In this example, the **Fib()** method, there are only 47 possible choices for the returned value, so this example is kind of trivial, but in real life there could be thousands, and testing them could involve considerable detail. This example is intended to suggest ways to do that.

5.2.8 Convert the TDS method to a test procedure

5.2.8.1 Specify a range of input values

5.2.8.1.1 HIDE THE **TESTVALUES[]** ELEMENTS AFTER THE FIRST

While we are setting up our first automated test, we will make some changes to the **testValues[]** elements, and it will be easier to do that only once, before we have a whole flock of them to modify with each new property. (This would be less of a problem if we were using a named type for our **testValues[]** elements; see section 5.2.9.6 below.) We'll begin by looking only at the simplest values, and later address more unusual ones, such as those that should raise exceptions.

► In **FibTest()**, comment out **testValues[1]**, t and **testValues[2]**, the test cases whose Id values contain tags of "**10** " and "**11**".

We comment them out here, instead of deleting them, because their contents will be able to inform some of the test cases that we intend to add.

5.2.8.1.2 DEFINE A CONSTANT NEEDED BY THIS EXAMPLE

We'll want to check the behavior of **Fib(n)** near the limits of its operation, among other places, so we'll define a new constant in **TDS.Test{}** matching the constant **upperLimit** used in **Fib(n)**.

► Copy the following code into **TDS.Test{}**, outside the definition of **FibTest()**. The exact order is not critical, but I would place it immediately before the XML comments preceding the definition of **FibTest()**, to keep the members related to **FibTest()** near each other in the program.

```
    /// <summary>
    /// Maximum index permitted for calculation of a Fibonacci number,
```



```
    /// set to avoid numeric overflow.
    /// <para>Current value is 47,
    /// the highest value not causing (uint) overflow.</para>
    /// </summary>
    //HACK: FibTestLimit -- Change to 48 or higher to see overflow
    const int FibTestLimit = 47;
```

Instead of a constant, we might have defined a read-only property such as this:

```
    static int FibTestLimit { get { return 47; } }
```

(using these same XML comments), except that we shall later need to use this name as a compile-time constant instead of a property, for use as the default value of a parameter or the length of an array.

We shall remove that "**HACK:**" Task List comment presently.

5.2.8.1.3 DEFINE PROPERTIES TO SPECIFY RANGE LIMITS

We could specify additional test values for **n** by adding elements to the **testValues[]** array, for example using their **Arg** property to specify discrete values besides **n=3**, such as **n=8** and **n=22**. We could do this by including an array-valued property in **testValues[]** to generate each of these values. We did something similar in section 4.8.2.4, specifying an array of values to be used in a **for()** loop.

However, for our TDS method in this example, we will define some properties to provide a range of values to send to **Fib(n)**, so that we can examine the results automatically, via unit-test procedures (using **Assert**), rather than via visual observation of values during tracing.

It is easy to add properties to be used in our tests, as long as **testValues[]** contains only one element. The definition of the anonymous object in **testValues[0]** includes property names that may be chosen fairly freely (see section 4.14.10.1).

- In **FibTest()**, in the definition of **testValues[0]**, insert these lines following the line containing "**Arg = 3,**":

```
    N0Low = 0, // Lower value of argument used in this test
    N1High = FibTestLimit, // Upper value of argument used in this test
```

You may navigate there via the Task "**TODO: FibTest() -- Define inputs and expected outputs.**".

The "**N0**" and "**N1**" in these names allow us to list them in alphabetical order while letting the property with the lower value be listed first; there's no other purpose to the numbers. The ordering of the property names has no effect on the code; choose whatever order you wish (but it must be the same in every element of **testValues[]** that we add later).

We'll soon replace the references to **Arg** in the code.

5.2.8.1.4 DEFINE A LOOP TO UTILIZE MULTIPLE VALUES

We want to use all the values in the specified range, so we enclose our calling code in a **for()** loop.

- In **FibTest()**, in the **foreach(var tCase …)** loop, immediately before the line containing

```
    #region Invoke testable function members
```

, place a copy of the following code:

```
for (var nValue = tCase.N0Low; nValue <= tCase.N1High; nValue++)
{
```

- Place a matching closing brace, "}", immediately after the line containing

```
    #endregion Apply tests when no exception is raised
```

and before the brace ending the **foreach(var tCase…)** loop, perhaps with an identifying comment, like this:

```
    }  // end: for (var nValue =...
```

- Change the line containing

```
    actual = NewCode.Fib(tCase.Arg);
```

to look like this:

```
    actual = NewCode.Fib(nValue);
```

This is located at the "**TODO: FibTest() -- Provide a suitable calling expression**" Task.

- In the "**TODO: FibTests() -- Provide suitable non-exception tests here**" Task , change the line in the **Assert.AreEqual()** statement containing

```
    , tCase.Arg  //{1}
```

to

```
    , nValue  //{1}
```

We're testing a range of values, but only one at a time, and **nValue** is that one.

5.2.8.1.5 DELETE AN UNUSED PROPERTY FROM **TESTVALUES[0]**

- In the definition of **testValues[0]** (in the "**TODO: FibTest() -- Define inputs and expected outputs.**" Task), delete the line containing **Arg**, as we no longer need it.

5.2.8.2 Rationale for unit tests

We now want to do some simple automated testing of **Fib()**. The following steps illustrate setting up the means to do that.

We shall use the same test case several times with different input values, and we can expect different results from those, which we need to be able to compare with the values we expect them to have.

We could do that by adding an array-valued property to **testValues[0]**, for example by changing the definition of **ValueExp** to look like this:

```
ValueExp = new[]{  //Array of the first few expected results
    0, 1, 1, 2, 3, 5, 8
  },
```

If we were to do this, we might calculate these values by hand or have some other means of knowing that they are accurate and suitable for testing the new code. For **Fib()**, however, we know of some other ways to get the same results, and we can compare those results to what **Fib()** returns. What we shall do instead of listing a few specific test values is to add some methods to our test to perform those alternate calculations.

5.2.8.3 Add code for alternate calculations

5.2.8.3.1 INSERT METHOD DEFINITIONS

5.2.8.3.1.1 NOTES ON THE METHODS

We wish to compare the output of **`Fib()`** with values of the Fibonacci sequence calculated by some other means that, perhaps, are easier to verify as correct but might have other defects, such as consuming resources that are not guaranteed to be available in the production environment. We'll do that by writing some ancillary methods that use these other means, then we shall compare the results of calculations of **`Fib(n)`** performed by these methods.

To help associate the following two ancillary methods with the main TDS method (**`FibTest()`**) that uses them, since they are used only to support that main method, I give each of them a name beginning with "FibTest", followed by a modest amount of descriptive text. More detailed descriptions go into their XML comments.

5.2.8.3.1.2 RECURSIVE DEFINITION

Only for testing purposes, we shall define a method, **`FibTestRecursiveCalc()`**, that will compute the desired value based on a naïve translation of our original definition. Of all the choices of ways to do this that we shall use, this one most obviously follows the definition of "starting with (0, 1), each member of the sequence is the sum of the previous two members." Unfortunately, as we shall see, although this gives correct answers, it works slowly and gluttonously. To get some insight into the extent of resources used by the recursive version, we have added to it a parameter, **`numCalls`**, whose value we can observe, but this parameter is not used in calculating the result; it's to be used only as a monitoring device.

This method, **`FibTestRecursiveCalc()`**, does have the advantage of being easy to understand, especially if we ignore the references to **`numCalls`**. After taking care of the special cases 0 and 1, it merely returns the sum of the previous two elements in the sequence. Its problem is that it recalculates both of those each time, instead of simply remembering them. As we shall see (section 5.2.9.5.1 below, "Mathematical side note"), the time and space it hogs are apparently O(exp(n))[81], a good reason to seek out some kind of alternative algorithm. For some values of its argument, this method gobbles several orders of magnitude more time and memory than our final version does. We include it in our test method because it obviously matches the definition, even though it should not be used for real work.

+[81] The expression "O(exp(n))", or "order of (exp(n))", implies that as n increases, the quantity (in this case, the time and space used) grows by a roughly constant percentage, As we shall see, it's a substantial percentage in this example.

Here is its definition, and I would place it immediately after that of **FibTest()**; its name is chosen to let its position in alphabetical order put it close to the TDS method that uses it.

```
    /// <summary>
    /// Calculate the nth element of the Fibonacci sequence
    /// by using the following recursive definition:
    /// <para>
    /// f(0)=0; f(1)=1; if n&gt;1, then f(n)=f(n-2)+f(n-1) .
    /// </para><para>If n&lt;0 or n&gt;FibTestLimit,
    /// then a value of 0 is returned instead of
    /// the nth Fibonacci number.</para>
    /// </summary>
    /// <remarks>The code in this method is
    /// grossly inefficient to compute,
    /// but it is easier to relate to the given
    /// recursive definition than the code used in
    /// <see cref="NewCode.Fib"/>() , with whose
    /// output the returned value of this method
    /// is to be compared.</remarks>
    /// <param name="n">Index to the Fibonacci sequence.
    /// <para>If this is out of bounds,
    /// the returned value is null.</para></param>
    /// <param name="numCalls">Number of times
    /// this recursive method has been called.</param>
    /// <returns>The nth element of the sequence,
    /// or (null) for an improper argument.</returns>
    static uint FibTestRecursiveCalc(int n, ref int numCalls)
    {
      numCalls++;
      return
            n < 0 || n > FibTestLimit
              ? 0
              : n < 2
                ? (uint)n
                : checked(FibTestRecursiveCalc(n - 2, ref numCalls)
                  + FibTestRecursiveCalc(n - 1, ref numCalls));
    }  // end:FibTestRecursiveCalc()
```

- Paste a copy of this code into a suitable place wihin **TDS.Test()**.

Except for the **numCalls** references, in English this code says "If the index, n, is out of bounds, return 0. If it's 0 or 1, return its value. Otherwise, add the previous two elements of the sequence and return the sum." If the given index is one that won't cause overflow, this is quite similar to the statement in section 5.2.3. (To me, the similarity of the code in **Fib()**, involving phi, to the original statement is not nearly as obvious as this is.)

5.2.8.3.1.3 ITERATIVE DEFINITION

Well, OK, the recursive definition makes sense, but let's try to avoid recalculating the same numbers multiple times, by saving our results as we calculate them.

We shall define another method, **FibTestIterativeCalc()**, that will calculate the desired value in an iterative fashion, building later values based on already-calculated earlier values; this one is much faster (it's

O(n)[82]. This is still slightly slower than what we use in **`Fib()`**, which is probably O(1)[83], since, depending on your processor, functions such as **`Floor()`**, **`Pow()`**, and **`Sqrt()`** are calculated by single machine-language instructions and are thus probably also O(1). This method's definition is slightly trickier to understand than that of **`FibTestRecursiveCalc()`**, since, after accounting for the special cases, it has to set up and maintain a couple of local variables to keep track of the preceding two values in the sequence.

Here is its definition, and I would place it (perhaps in alphabetical order) near that of **`FibTest()`** .

```
/// <summary>
/// Calculate the nth element of the Fibonacci sequence
/// by adding successive elements until reaching element n
/// <para>
/// f(0)=0; f(1)=1; if n&gt;1, then f(n)=f(n-2)+f(n-1) .
/// </para><para>Do this by iterating over the sequence,
/// building it from the initial values of (0, 1).</para>
/// <para>If n&lt;0 or n&gt;FibTestLimit,
/// then a value of -1 is returned instead of
/// the nth Fibonacci number.</para>
/// </summary>
/// <param name="n">Index to the Fibonacci sequence.
/// <para>If this is outside the specified range
/// the returned value is 0.</para></param>
/// <returns>The nth element of the sequence,
/// or 0 for an improper argument.</returns>
static uint FibTestIterativeCalc(int n)
{
  if (n < 0 || n > FibTestLimit) return 0;
  if (n < 2) return (uint)n;
  uint firstBack = 0;
  uint currentSum = 1;
  for (int i = 2; i <= n; i++)
   {
    var secondBack = firstBack;
    firstBack = currentSum;
    currentSum += checked(secondBack);
  }
  return currentSum;
}  // end:FibTestIterativeCalc()
```

- Paste a copy of this code into a suitable place wihin **`TDS.Test()`**.

5.2.8.3.1.4 ARRAY LOOKUP

In addition to using these two methods to generate values for comparison, we shall also calculate, just once, an array of values that we can use to look up the expected result. After the values are calculated, the array lookup can be plenty fast (O(ln(n)) or better[84], depending on your code), but in this case it requires populating an array-valued variable dedicated to use by the **`FibTest()`** TDS method, which could be impractical if you have many thousands or millions of values to maintain. This might be the best way to do the calculation if we did not have the algorithm used in **`Fib()`** available to us, and assuming that we can place a firm upper limit on the number of elements we need to store. In the specifications we are using, we have set such an upper limit, by

[82] O(n) is roughly proportional to n.
[83] O(1) impies that the calculation takes about the same amount of time regardless of the argument.
[84] O(ln(n)) implies that, whenever we double n, the calculation takes only roughly a fixed amount of additional time.

allowing only **(uint)** values for the result, thus limiting the size of the array to only a few dozen elements, but your projects may not always allow that.

We need to set up some additional resources for these methods.

- Insert the following code into **FibTest()**, immediately following the statement

```
if (IsUsingStandAloneTds)
  InitializeTestMethod();
```

that appears a few lines below the "**#endregion testValues[]**" directive:

```
#region Set up fibValues[]
//Values of the Fibonacci sequence
// for comparison with those computed by
// alternate test methods.
var fibValues = new uint[FibTestLimit + 1];

//Calculate enough elements of the Fibonacci sequence to cover
//  all those that the test methods might be asked to calculate.
fibValues[0] = 0;
fibValues[1] = 1;
for (int i = 2; i < fibValues.Count(); i++)
  //HACK: FibTest -- Change to unchecked to see effects
  fibValues[i] = checked(fibValues[i - 2]
        + fibValues[i - 1]);
#endregion Set up fibValues[]
```

This code will run only once for a **FibTest()** test, not once for each test case.

5.2.8.3.1.5 COMPARISON OF ALTERNATE METHODS

Normally you will not have a variety of means to verify the workings of your working code, so in that respect this example is unrealistic. It probably won't be worth your time to try to develop multiple alternatives. They are included here to illustrate some shortcomings of each. The recursive method (section 5.2.8.3.1.2) clearly expresses the essence of the original definition, but it wastes time and storage space. The array lookup, along with the code to populate the array (section 5.2.8.3.1.4), makes efficient use of time (though maybe not as much so as our working-code version, **Fib()**), but it does require some storage space, so a decision must be made as to how much space is needed. The iterative method (section 5.2.8.3.1.3) uses very little space, but it requires calculating the result each time it is called, at the expense of a varying amount of processing time.

5.2.8.3.2 RATIONALE FOR NOT TESTING THE ALTERNATE CODE

We do not intend to test any of this new code directly, but we hope that any mistakes in it will become apparent as we compare results. If a difference between actual and expected values does appear anywhere, the test report will display both. Examination of the values should reveal which is wrong and will thus help to identify the location of the mistake.

This can be an effective technique, but it is not foolproof. Alternate methods might generate consistently wrong, but matching, results[85], perhaps as a result of someone's misunderstanding the requirements. The test method, not noticing any difference, would falsely claim that the test passed. It's probably a good idea to involve your customer, to some extent, in developing your test methods to be sure that this never happens.

[85] See section 5.2.6.7 for an example of this, in which the code may return a wrong value as if it were correct.

5.2.8.3.3 RATIONALE FOR TESTING IN GENERAL

Testing should be a supplement to careful analysis, not a substitute for it; both are important. Mistakes in coding are almost inevitable (for me, at least), and unit testing can detect them early, when they are easy and inexpensive (and non-embarrassing) to correct.

As an example of the value of testing, a program I once was modifying displayed text that included some highlighted fields. Some of the text was dark with a light background, some light with a dark background. Text was sent to the display as a string of characters to be displayed, along with some control characters that changed the format (light or dark) of the following text. What I originally did was, knowing where the highlighted fields began and ended, to insert the proper control characters into the displayed string. My initial, cursory testing showed that the fields were being correctly displayed. As it turned out, the users (my co-workers) were not happy! Each mode-changing character changed the entire rest of the displayed text, and the entire process took maybe two seconds, so every time the text was updated, much annoying flickering ensued in the lower part of the display. (Inserting the control characters beginning with the end of the string instead of at the beginning solved the problem, and I quickly took care of it.) Mathematically, the original code had looked OK; either order of inserting control characters produced the correct result. Detailed testing (in this case done, unfortunately, by my users) revealed an unacceptable side-effect that was not apparent from the analysis.

In defense of detailed analysis, and in contrast to testing, I find it difficult to imagine how any amount of testing could enable one to derive the code, involving **`Math.Pow(),`** that we are using in **`Fib()`**.

In the present example, I have tried to make the definitions (and the XML comments) of the methods we just now added be so easy to understand that there will be little question of their correctness. If some part is unclear, we could simplify the code and/or judiciously add comments, to make its operation obvious to anyone who might have to visit it later. (The "anyone" might possibly include ourselves, six months in the future). The efficiency or cleverness of the design is not an important criterion in a TDS method, as this code will not be part of the "Release" configuration. (OK, efficiency is somewhat material in this example, as one of our methods used for comparison is so horribly *in*efficient that we will not even try calling it for some of our test cases… but normally we shouldn't have to worry about that.)

5.2.8.4 Update the description of `FibTest()`.

Since the test performed by **`FibTest()`** is now becoming more complex than it was at first, this may be a good time to update its XML comments, from

```
/// <summary>
/// TDS Test of Fib .
/// </summary>
```

to

```
/// <summary>
/// A test for <see cref="NewCode.Fib()"/>,
/// comparing its returned values
/// with those returned by
/// <see cref="FibTestIterativeCalc()"/>
/// and <see cref="FibTestRecursiveCalc()"/>,
/// and those stored in <see cref="fibValues[]"/>.
/// </summary>
```


5.2.8.5 Add some testing code

5.2.8.5.1 PREPARE TO ADD `Assert` STATEMENTS

Now that we can calculate values for comparison with the results of **`Fib()`**, let's add some code to **`FibTest()`** to do the actual tests.

► We're about to add real **`Assert`** statements now, so delete the **`Assert.Inconclusive`** statement at the end of the **`FibTest()`** method, along with its "**`TODO: FibTest() -- Remove the Assert.Inconclusive()`**" Task comment.

5.2.8.5.2 REGION INCLUDING ADDED "`Assert`" STATEMENTS

► Following the "**`TODO: FibTest() -- Provide suitable non-exception tests here:`**" Task comment in **`FibTest()`**, delete the **`Assert.AreEqual()`** statement that begins with

```
            Assert.AreEqual(
                  tCase.ValueExp,
```

We shall replace this **`Assert`** statement, an example statement provided as part of the original TDS method template, with some other tests.

In place of the **Assert** statement that we just now deleted, include the following code:

```
            #region RunTest() definition
            //Compare value returned by Fib() with the value
            //  calculated by an alternate means
            Action<uint?, string, string> RunTest = (
                alternateValue  //Value calculated by other means than Fib(),
                // where null implies an improper argument
                , identifier  //Tag to distinguish Assert statements
                , adverb  //Description of method used
              ) =>
              Assert.AreEqual(
                    alternateValue,
                    actual,
                    string.Format(@"
      FibTest()_{0}, test case {1}:
        Value calculated {2} was {3};
        value returned by NewCode.Fib({4}) was {5}."
                      , identifier  //{0}
                      , tCase.Id  //{1}
                      , adverb  //{2}
                      , alternateValue  //{3}
                      , nValue  //{4}
                      , actual  //{5}
                    )
                  );
            #endregion RunTest() definition

            //Compare with the stored value
            var fibStored = nValue < 0 || nValue > FibTestLimit
                  ? 0
                  : fibValues[nValue];
            RunTest(fibStored, "Stored", "via stored values");

            //Compare with an iteratively calculated value
            RunTest(
                  FibTestIterativeCalc(nValue),
                  "Iterative", "iteratively");

            //Number of times the method called recursively is called
            var numRecursiveCalls = 0;
            //Compare with (inefficiently) recursively calculated value
            //Do this test only if n isn't too large
            if (nValue <= 20)
              RunTest(
                FibTestRecursiveCalc(nValue,
                  ref numRecursiveCalls),
                "Recursive", "recursively");
```

5.2.8.5.2.1 BENEFITS OF USING AN ACTION (OR FUNC) INSTEAD OF A SEPARATE METHOD

Since the three tests are quite similar to each other, I have also defined an **Action**, called **RunTest**, to perform each of the tests. This allows me, if I wish to change the error messages that they generate, to do that in one place and have it applied consistently within those three tests. Also, if I should discover a bug in the **Assert** statement, I can correct it in one place instead of several.

We have done some slightly tricky stuff here, replacing some ordinary, but repetitive, code with an **`Action`**. Such trickiness could hinder maintenance of our code. Since we don't want to make the code unduly difficult to read or understand, comments clarifying what we've done may be in order, along with using (as usual) concise but suggestive names for the variables and parameters; I included a short comment at the beginning of the definition of **`RunTest()`**, as if it were a method definition. I also placed comments on its parameters, such as **`alternateValue`**, similar to the comments that would go into the **`<param></param>`** element of a method's XML comments.

The benefit of factoring out common code like this is not great if we have only two or three instances, as we do here, but this is merely an illustration — suppose you have a battery of a dozen or more tests that are similar to each other. You could use this technique to standardize your code and reduce the number of lines that someone would have to read to verify that, for example, the tests are accurate, do not duplicate other tests, and cover all the relevant situations.

Why did I use an **`Action`** instead of pulling the code all the way out to a separate method, as I did with **`FibTestIterativeCalc()`**, etc.? Certainly I could have done that, and you may prefer to do so in your own projects, but defining it as we did in this example can be useful because it

- keeps the definition physically closer to the invocation – just a few lines away – where it can easily be found,
- gives the definition access to local **`FibTest()`** variables such as **`tCase.Id`** and **`actual`**, which otherwise would need to be passed to it as additional parameters, and
- avoids cluttering the namespace with extra method names.

Since the name "**`RunTest`**" belongs to a local variable, it doesn't conflict with anything outside the body of **`FibTest()`**, and I could have made the code even more compact by calling this **`Action`** something even shorter, such as "**`RT`**", without having to worry too much that the name is not suggestive enough of its function — its definition is physically quite close to all of its uses and is therefore fairly easy to find when needed. Sorry, its IntelliSense is not of much help, but the VS editor does allow you to split the screen (to do so, click on the  symbol in the upper-right corner of the VS editing window) so that you can see both the call and the definition at the same time.

However, there may be some disadvantages to using an **`Action`**-valued local variable as we did here, instead of a method definition, such as these:

- not being able to apply XML comments to it or its parameters, to support IntelliSense, and
- not being able to see or use it anywhere outside the block where its definition is located.

Since we can't use it outside its block, we

would not be able to test it directly via a TDS test (assuming we'd even want to do such a thing).

5.2.8.5.2.2 PLACEMENT OF EXCEPTION TESTS

In a TDS method, we place the tests concerning exceptions first because, if an exception is thrown, there are no useful returned values to be compared, so we won't need to do any further checking — we'll already know that the test case either has failed or has raised an expected exception.

In **`FibTest()`**, we have by now added code to help with testing, to check that the value returned by the working code matches the value that

- we stored in the **`fibValues[]`** array,

- was returned by the method we added, that does an iterative calculation, and
- was returned by the recursive method that we added.

Not wishing to waste lots of processor time, as we're trying to test **`Fib()`** rather than **`FibTestRecursiveCalc()`**, we use "**`if (nValue <= 20)`**" before the call to **`FibTestRecursiveCalc()`** to skip over some of the lengthier test cases for the recursive-method test.

Here we are using several alternative methods of calculating what we expect to be the same values and use several **`Assert`** statements to verify that they are indeed the same. In a similar situation in your own testing, you might also need to add more **`Assert`** statements to the TDS method if the new function member produces several outputs, or outputs with complex values (as we shall do in section 5.3), to verify that the function member is working properly. If you have large numbers of possible values for some variables, an exhaustive test will be impractical, so you will need to test using some subset of the possible values.

5.2.8.5.2.3 IDENTIFYING THE ASSERT STATEMENTS

I included in the error messages some identifying information (in the **`String.Format()`** part of the **`Assert`** statement). Suppose you have defined, say, 45 of these **`Assert`** statements, run a test, and found that one of them failed. How can you quickly determine which one that was? As we have done in the definition of **`RunTest()`**, by changing the line

```
FibTest(), test case {1}:
```

to read something like

```
FibTest()_Recursive, test case {1}:
```

it is made apparent which **`Assert`** statement led to the failure and which test case was involved. You might also consider preceding the identifier tag with a number, as we did in section 4.8.3.3, to make its failing **`Assert`** statement easy to find within a large collection of other **`Assert`** statements in the source code.

Similarly, if the processing of a test case involves a loop, we will likely want the **`Assert`** message to identify which iteration of the loop caused the failure; see section 5.2.9.6.3.9 for a discussion and example.

5.2.9 Test the new method

5.2.9.1 Begin testing

We are about to compare the value of **`NewCode.Fib(nValue)`** with various calculated values, for example that of **`FibTestIterativeCalc(nValue)`**, rather than with the original **`tCase.ValueExp`**.

► Remove any active breakpoints and run TDS (<F5>).

Ideally, we see that **`FibTest()`** Passed. (If not, examine the **`error`** messages and correct the problems.)

► Close the Console window.

Part of the reason for success is our checking for conditions like numeric overflow.

► To see some possible results of *not* checking, go to Task "**`//HACK: FibTestLimit -- Change to 48 or higher to see overflow`**" and, in the **`const`** line, change the 47 to, say, 50.

► Run TDS.

Oops — now TDS crashes with a **`System.OverflowException`** as we are trying to set up our **`fibValues[]`** array.

We see in the Locals window that the value of local variable **i**, the array index[86], is 48, so evidently 48 is a value we shouldn't use.

Do not disable this unhandled exception message (in contrast to what we did in section 4.4.2 to hide the exceptions raised by **Assert** statements)[87]. We want to know immediately of any real exceptions in the test-method processing, such as this one. Ignoring an overflow exception might merely give us a wrong answer and continue processing with no indication of any error, as happened in section 5.2.6.7.

► Use VS menu "Debug, Stop debugging" (or <shift><F5>) to cancel the test.

If you were to continue running (using <F5>) after encountering the exception, instead of stopping, the test report would include the following message, instead of the normal TDS test report of an exception:

```
  - FibTest()
      Exception message:
Arithmetic operation resulted in an overflow.
```

This terse response is generated because this unhandled exception occurred within TDS (which normally should not happen) rather than in the working code being tested.

5.2.9.2 Demonstration of "unchecked" operation

As an illustration of the danger posed by not checking for overflow, do the following (otherwise, skip to section 5.2.9.3):

► Go to the Task "**HACK: FibTest -- Change to unchecked to see effects**" and in the following statement change "**checked**" to "**unchecked**".

► On the **foreach()** statement following that statement, or following the directive

```
#endregion Set up fibValues[]
```

, set a breakpoint.

► Run TDS, stopping at the breakpoint.

► In the Locals window, examine the value of **fibValues[48]**.

Note that it is not correct, and neither are any subsequent elements of **fibValues[]** that you may have calculated.

► Use VS menu "Debug, Stop debugging" (or <shift><F5>) to cancel the test.

► Remove the breakpoint.

► At Task "**HACK: FibTest -- Change to unchecked to see effects**", in the following statement change "**unchecked**" to "**checked**", and delete the "**HACK:**" Task comment.

---

[86] This value of i also matches the value of the argument **nValue** in the "**actual = NewCode.Fib(nValue);**" statement that gives us the value which we plan soon to compare with the value stored in this array element, **fibValues[nValue].**

[87] If you accidentally do so, then open menu "Debug, Windows, Exception Settings". In Common Language Runtime Exceptions find "System.OverflowException", and check its "Break When Thrown" box.

► In the statement following "**HACK: FibTestLimit -- Change to 48 or higher to see overflow**", change the value back to 47 and remove the "**HACK:**" Task comment.

► Run TDS.

Everything should Pass (and correctly, this time).

► Close the Console window.

5.2.9.3 Test the ancillary test methods

To demonstrate that wrong values are properly detected, and that the error report is accurate and properly formatted, we could temporarily alter the code in the methods we are using to check the results of calling **Fib()**. Here we shall intentionally buggify them to introduce false outputs so that we can see that the tests report the false results. By analogy, consider that a burglar alarm that never generates a false alarm may be too insensitive to possible trouble to be useful, or pehaps it doesn't work at all.

(There's no need to actually do that here; this is just an illustration.)

5.2.9.3.1 BUGGIFY FIBVALUES

For example, immediately following

```
            //TODO: FibTest() -- Provide suitable non-exception tests here:
```

we could insert lines such as

```
            //HACK: FibTest(): Remove this buggifying line:
            fibValues[17] = 1234;
```

and run the tests. The **//HACK:** ... comment will appear in VS's Task List window, making it easy to find and remove.

If we now run TDS, then in the test report we should see this message:

```
Assert.AreEqual failed. Expected:
<1234>. Actual:
<1597>.
      FibTest()_Stored, test case 01 Low, valid input:
        Value calculated via stored values was 1234;
        value returned by NewCode.Fib(17) was 1597.
```

Having observed this, we remove the added line of code and its "**HACK:**" Task comment.

5.2.9.3.2 BUGGIFY AN ANCILLARY METHOD

Similarly, to check **FibTestIterativeCalc()** or **FibTestRecursiveCalc(),** we could insert lines similar to following ones immediately before the first statement in its method body:

```
      //HACK: FibTestIterativeCalc(): Remove this buggifying line:
      if (n == 17) return 4321;
```

Running TDS should then produce in the error report either this:

```
Assert.AreEqual failed. Expected:
<4321>. Actual:
<1597>.
      FibTest()_Iterative, test case 01 Low, valid input:
        Value calculated iteratively was 4321;
        value returned by NewCode.Fib(17) was 1597.
```


or this:

```
Assert.AreEqual failed. Expected:
<4321>. Actual:
<1597>.
      FibTest()_Recursive, test case 01 Low, valid input:
        Value calculated recursively was 4321;
        value returned by NewCode.Fib(17) was 1597.
```

5.2.9.3.3 RESTORE PROPER OPERATION

Of course, in each of these cases, it would be the "Expected" value that is wrong, not the "Actual" one.

Having observed this, we remove the added line of code and its "**`HACK:`**" Task comment.

One could test this TDS test method by creating another TDS test method to test it … but there's a (low) limit on how many levels of TDS tests testing other TDS tests I would consider to be a good idea. You might notice that we have not done any of that in these examples, though we could have done so, for example with **`FibTestIterativeCalc()`**, and I have occasionally done it in my own projects. Testing the test methods can raise the question "*quis custodiet ipsos custodes*?"[88]; at some point there is no good substitute for clearly thinking about what the code ought to be doing. Factoring out common elements into **`Func`** or **`Action`** variables can, if done well, help the thought process by reducing the number of parts, standardizing their use, and making their organization as clear as is reasonably possible.

Concerning testing the **`Action`** we defined in the "**`TODO: FibTest() -- Provide suitable non-exception tests here`**" Task, **`RunTest`**, it contains few branches, and the results of running it are immediately visible. So are the results of running the erroneous versions of the methods called by it, so there seemed to be little need to create a separate test method to check its behavior. (A bad experience with an undetected bug related to something as simple as **`RunTest`** would likely change my mind about testing it, however.)

5.2.9.4 Add more test criteria

5.2.9.4.1 ADD A PROPERTY FOR NUMBER OF RECURSIVE CALLS

Suppose it occurs to us to want to identify the expected number of recursive method calls, not so much because we're interested in the value itself as to detect if that number changes in the future, indicating an unexpected change in the processing. We'll add a property to **`testValues[0]`** to track this. Our ancillary method **`FibTestRecursiveCalc()`** already tracks this, so it doesn't need to be changed.

We'll use a value of -1 to allow us to bypass the test, in cases where it should not apply. For example, in the first test case, we use many different values for **`n`**, so the numbers of expected calls will differ.

- In **`FibTest()`**, in **`testValues[0]`**, add the following lines after the line defining **`ExceptionExp`**:

```
            NumCallsExp = -1, // Expected # of invocations
                              // of the recursive function,
                              // or -1 if we are not checking it
```

You may navigate there via the "**`TODO: FibTest() -- Define inputs and expected outputs.`**" Task.

---

[88] = "Who shall guard the guards themselves?", from Juvenal's *Satires*; see https://en.wikipedia.org/wiki/Quis_custodiet_ipsos_custodes%3F .

- In **testValues[0]**, delete the line containing **ValueExp**.

The ancillary methods that we added calculate the expected returned values, so we have no need for this property.

5.2.9.4.2 COMPARE EDITING RESULTS

The contents of **testValues[0]** should now look similar to this:

```
        new {
          Id = "01 Low, valid input", // Test case identifier
          N0Low = 0, // Lower value of argument used in this test
          N1High = FibTestLimit, // Upper value of argument used in this test
          ExceptionExp = DefaultExceptionMessage,  // Expected exception
          NumCallsExp = -1, // Expected # of invocations
                            // of the recursive function,
                            // or -1 if we are not checking it
        },
```

5.2.9.4.3 CHECK NUMBER OF CALLS

- Immediately before the

```
      #endregion Apply tests when no exception is raised
```

directive near the end of **FibTest()**, add this code:

```
            //Check number of calls to the recursive version
            if (tCase.NumCallsExp >= 0)
              Assert.IsTrue(
                    numRecursiveCalls == tCase.NumCallsExp,
                    String.Format(
@"FibTest()_NumCalls, test case {0}:
    # of recursive calls was{1,6};
              expected # was{2,6}."
                      , tCase.Id //{0}
                      , numRecursiveCalls //{1}
                      , tCase.NumCallsExp //{2}
                    )
              );
```

We have put this at the end because, if the returned value (checked in the previous **RunTest** statement) is wrong, the number of iterations will be of no interest to us.

Running the test, using VS menu "Debug, Start Debugging", should produce the same "**Passed: 2**" message as before, since the first test case has "**NumCallsExp = -1**", to cause it to skip this new test.

5.2.9.5 Add a test case using the new property

The (only) existing test case exercises **Fib()** using a set of simple values.

We want to add a test case that will allow us to examine the number of recursive calls returned by **FibTestRecursiveCalc()**., using the properties we just now added.

► Immediately before the closing brace in FibTest() of the definition of **`testValues[]`**, which is on the line preceding the **`#endregion testValues`** directive, include this code:

```
        new {
          Id = "02 High number (18) sent to slow version",
          N0Low = 18,
          N1High = 18,
          ExceptionExp = "",
          NumCallsExp = 0,
        },
```

Be sure that these properties appear in the same order as in **`testValues[0]`**.

In this code, which is the new **`testValues[1]`**, we have omitted any comments describing what these properties mean, have given **`Id`** a value beginning with a different tag ("02"), and have given **`NumCallsExp`** a value (0) that we know will cause the test to fail, but will also reveal the expected result.

► Test the code, for example using VS menu "Debug, Start Debugging" or <F5>.

Output in the Console window indicates that a test failed, and the following message is included:

```
  The following test method returned a status of Failed:

  - FibTest()
      Exception message:
Assert.IsTrue failed. FibTest()_NumCalls, test case 02 High number (18) sent to
slow version:
    # of recursive calls was  8361;
              expected # was     0.
```

We can check this "8361" number to determine if it is accurate, or (more quickly) just assume it's accurate and set the expected value to match what we found. We're looking for stability here. If later changes to the method disturb this value, so that it no longer matches, this test will fail and our attention will be attracted, and we can then determine what happened. Until then, we'll assume that we don't need to think about it again. (If this thinking is too sloppy for your taste, some better analysis appears in section 5.2.9.5.1 below.)

► Close the window.

► Change the value of **`NumCallsExp`** in this test case from 0 to 8361, and add another similar test case, where (I claim) we have determined the expected number in a similar way:

```
        new {
          Id = "03 High number (20) sent to slow version",
          N0Low = 20,
          N1High = 20,
          ExceptionExp = "",
          NumCallsExp = 21891,
        },
```

► Test the code again; close the window after viewing the output.

The test report should show that **`FibTest()`** passed. We now know that the results calculated in four ways (our **`Fib()`** method, the contents of the **`fibValues[]`** array , and the two methods that we introduced for comparison) are all consistent with each other. Of course, we hope that they are all *correct* as well, but that is a separate matter.

5.2.9.5.1 MATHEMATICAL SIDE NOTE

(This section isn't essential; please feel free to skip ahead to section 5.2.9.6.) We are using the number of calls as a rough indicator of resources used by the recursive method, as well as a signal (if it unexpectedly changes) to alert us that something might have happened to affect the calculation, so its actual value is not of great importance to us.

However, if you wished to analyze it further, you might note that in each call to **`FibTestRecursiveCalc()`** its value is increased by 1 plus the sum of the two calls with parameters of **`(n-1)`** and **`(n-2)`**. It could be characterized, similarly to what we did with **`Fib(n)`**, as a function **`NumCalls(n)`** defined by

$$NumCalls(0) \stackrel{\text{def}}{=} 1 ,$$

$$NumCalls(1) \stackrel{\text{def}}{=} 1 ,$$

and

$$n > 1 \Rightarrow NumCalls(n) \stackrel{\text{def}}{=} NumCalls(n-2) + NumCalls(n-1) + 1 .$$

The first few values are (1, 1, 3, 5, 9, 15, 25), and it can be expressed in terms of $Fib(n)$ as

$$NumCalls(n) = \big(2 * Fib(n+1)\big) - 1$$

So we see that the resources (processing time and memory) that it gobbles grow about as fast as the values it calculates. ☹

5.2.9.6 Convert **`testValues[]`** elements to a named type

5.2.9.6.1 HOW WOULD USING A NAMED TYPE HELP?

See a discussion in section 4.14.10 for a comparison of using named objects in **`testValues[]`** with using anonymous objects. We'll define a new class that we can use to replace the anonymous elements that **`testValues[]`** currently contains. It may be that the anonymous objects will do everything we need, so that naming them will be unnecessary, but for this exercise we'll assume that we expect to need them.

In the following sections we shall step through the process of constructing these named objects.

5.2.9.6.2 NAME THE CLASS

Let's assume that we have made most of the changes we need to make in the properties of the **`testValues[]`** elements to run our tests. If we decide to change them to named objects in this TDS method, it will help to do that before we have defined too many anonymous test cases, which will need to be reformatted into constructors, but after we have done enough work on the tests that we have included in the test cases most of the properties that we might need. (However, don't worry too much about possibly needing to add more properties later — they can be made optional, allowing any existing test cases to be left unchanged even after new, optional parameters are added to the constructor.)

Note: Some versions of Visual Studio can refactor these automatically, though I think that the results are a bit skeletal — the comments tend to become lost. You might want to make a copy of the existing elements in **`testValues[]`** and comment out the copy or save it in another document, before using that refactoring feature.

In this example, we shall assume that we want to preserve the comments, so we shall refactor the test cases manually, in stages.

One way to set up a named class (though we won't illustrate doing so in the current example) would be to use as a model the **`TestableConsoleMethodTestCase{}`** class defined in file TDS.cs when we de-comment, in TDS.cs, the directive

```
//#define UseNamedObjectTypeInTestableConsoleMethodTest
```

(We did this in section 4.8.6 of the Tutorial.)

For example, we could

- copy the definition of **`TestableConsoleMethodTestCase{}`** to a suitable location near our test method;
- replace the name of the class, **`TestableConsoleMethodTestCase`**, where it appears in the copy with a name such as **`FibTestCase`**;
- remove the unnecessary parts;
- copy the properties from the anonymous objects (but, in the named-object definition, the order in which they are defined is no longer important); and
- copy the comments from **`testValues[0]`** into the new XML comments (escaping special characters such as "**<**", as described in section 5.1.5.1.3 above).

However, instead of copying and modifying **`TestableConsoleMethodTestCase{}`**, it will likely be easier simply to build the new class directly, which we shall now illustrate.

Since each TDS method needs at most one of these named classes for use in its **`testValues[]`** array, I usually name this class by appending "**`Case`**" to the TDS method name, in the present example forming the class name "**`FibTestCase`**". You could perhaps abbreviate it by omitting the "**`Test`**" part. It is probably unwise to plan to use this class anywhere else, since doing so would hamper making changes to it to accommodate new properties/parameters in the **`testValues[]`** elements. Using a name closely resembling the TDS method's name will help make its definition easy to find.

We shall first set up (based on our current contents of **`testValues[0]`**) an initializer for the as-yet-undefined class **`FibTestCase{}`**, and have VS generate the class from that.

5.2.9.6.3 INSERT CODE DEFINING THE NAMED TYPE

*5.2.9.6.3.1 CONVERT **`TESTVALUES[0]`** TO BE A NAMED-TYPE INITIALIZER*

► For the moment, in **`TDS.Test.FibTest()`**, comment out all the members of **`testValues[]`** except for **`testValues[0]`**, which should still look like the code shown in section 5.2.9.4.2 above.

We can navigate there via the "**`TODO: FibTest() -- Define inputs and expected outputs`**" Task.

Don't delete these other members yet, as we may be able to use their contents soon.

► Change the first line of **`testValues[0]`** (the line prceding the one beginning with '**`Id = "01 Low, valid input" `**') from

```
        new {
```

to

```
        new FibTestCase {
```

The name "**`FibTestCase`**", being undefined, should have a wiggly underline.

► Move the mouse pointer to the "**FibTestCase**" name (about to become a class name) that we just now added; it should be underlined to flag an apparent error.

A pop-up icon should appear; clicking on the icon will give you the option of having VS generate the corresponding constructor.

► Click on "Generate nested class 'FibTestCase' ".

► Navigate to its definition by right-clicking its name and choosing "Go To Definition", or via placing the cursor on its name and pressing <F12>.

This definition is likely to be near the end of the **Test{}** class definition.

► Change the accessibility of **Test.FibTestCase{}** from **private** to **internal**.

This will allow its XML comments to be reflected in the Object Browser, in this case, listed in the TDS namespace below the class **Test{}** with the name **Test.FibTestCase{}**. If it were to remain **private**, its XML comments would be reflected in the IntelliSense pop-ups on constructor parameters and instance properties but would not be visible in the Object Browser.

► Although it's not necessary, I would also add a comment, to change the line containing the closing brace to look like this:

```
    }  // end: FibTestCase{}
```

This can be especially helpful whenever I edit the source code outside the VS source-code editor.

You could, instead of making this a nested class, generatite this as a class at the same level as **TDS.Test{}**, but I recommend not doing so for this class. That could be done by clicking on "Generate class 'FibTestCase' " instead of "Generate nested class…". If you do this, then within its definition, references to members of **TDS.Test{}** will be limited; for example, **private** members such as **TDS.Test.FibTestLimit** will be inaccessible. References to accessible members will need to explicitly mention **Test**, as in "**Test.DefaultExceptionMessage**" instead of simply calling it "**DefaultExceptionMessage**". Also, making it accessible outside **TDS.Test{}** will cancel its ability to be freely modified it in response to changed requirements within the **FibTest()**, since function members outside of **TDS.Test{}** could come to depend on it, and changes to it could affect those other function members.

Now that the new class **Test.FibTestCase{}** is defined, including some properties matching those of the former anonymous class, it should look somewhat like this:

```
    internal class FibTestCase
    {
      public string ExceptionExp { get; set; }
      public string Id { get; set; }
      public int N0Low { get; set; }
      public int N1High { get; set; }
      public int NumCallsExp { get; set; }
    }  // end: FibTestCase{}
```

To assist in navigation between **testValues[0]** and the new class definition, I usually set bookmarks (menu "Edit, Bookmarks, Toggle Bookmark") in both places while I'm setting up the new class, or I split the editing window (by double-clicking the  button in the upper-right corner of the VS editing window, as suggested in section 5.2.8.5.2.1) to display both areas.

Notice that the properties in the new class are listed alphabetically, so that the **`Id`** property is no longer first. If this annoys you, then you might…

- Move the property's definition to its rightful place (somewhere it can be easily found) in the new class definition.
- Rename the property to something like "**`Aardvark_Id`**" when you create a new TDS test, to automatically put if first alphabetically in the list of property definitions. (This is the same reason "N1High" contains a "1"; see section 5.2.8.1.3.)
- Change the **`TdsTest`** snippet to change the property's default name.
- Do nothing to it (my choice, and what we'll do in this example).

5.2.9.6.3.2 ADD XML COMMENTS

Using this named class, **`Test.FibTestCase{}`**, we gain the ability to apply XML comments to the properties and to omit some explicit member initializers[89] from the object constructors in the test cases in **`testValues[].`** For example, if we were to erase the line containing **`N1High`** in **`testValues[0]`** while it is still present in **`testValues[1]`**, that would not cause a syntax error, as it would if we were using an anonymous type.

NOTE: In this example, many of the following instructions call for doing the same thing to each of the five properties in the class. For this example, I suggest that you actually do so with only enough representative samples to be familiar with the process and to verify that it works on your system as described here, for example by modifying only two of them instead of all five. Completed examples will be shown at various places throughout the example, allowing you to observe the results without having to do the editing.

► Move the comments from the properties in **`testValues[0]`** to XML comments on the corresponding properties in the **`FibTestCase{}`** class definition.

As you paste the copied text into XML comments, take care to escape any special HTML characters "&" and "<", as described in section 5.1.5.1.4. Though there aren't any in the present example, in general any such characters would need to be escaped.

Splitting the editing window (via its  button) can make it easier to move or copy this code, as both the source and the destination can be visible at the same time.

[89] Don't worry; they will all be initialized, but we are now able to do so with some of them by using default initialization values.

Much of the action in a TDS method takes place in its **`testValues[]`** array, and a comment appearing there may apply to an unusual individual value, to one of the included items (a test case), or to one of the properties in the test cases. Suggested locations of these differ depending on whether you use an anonymous-object initializer or a named-type constructor, as shown here:

Location of comment	Anonymous type	Named type
Comment about a specific value	Use an in-line comment where that value appears, but in **`testValues[0]`** do something to distinguish it from any in-line comments that may be present about a property.[90]	In-line comment where that value appears in **`testValues[]`**
Comment about a test case	As part of the "**`Id`**" property, following the tag	As part of the "**`Id`**" property, following the tag
Comment about a property	In-line comment on that property in **`testValues[0]`**, or in a block of comments preceding **`testValues[]`**.	XML comments on the property or field and on constructor parameters related to the property or field.

- Add an XML comment at the beginning of the class with contents similar to the following:

```
/// <summary>
/// This defines a test case for <see cref="FibTest()"/>.
/// </summary>
```

In these lines, the "<" and ">" should not be escaped, as they are legitimate parts of the XML code.

[90] You can avoid an ambiguity associated with an in-line comment that applies only to a specific value in an anonymous initializer, by putting only boring, run-of-the-mill values into **`testValues[0]`** when you are using anonymous initializers, so that none of that first test case's property values will have any need of special comments.

After adding these comments, the class definition should look something like this:

```
    /// <summary>
    /// This defines a test case for <see cref="FibTest()"/>.
    /// </summary>
    internal class FibTestCase
    {
      /// <summary>
      ///  Expected exception
      /// </summary>
      public string ExceptionExp { get; set; }

      /// <summary>
      /// Test case identifier
      /// </summary>
      public string Id { get; set; }

      /// <summary>
      /// Lower value of argument used in this test
      /// </summary>
      public int N0Low { get; set; }

      /// <summary>
      /// Upper value of argument used in this test
      /// </summary>
      public int N1High { get; set; }

      /// <summary>
      /// Expected # of invocations
      /// of the recursive function,
      /// or -1 if we are not checking it
      /// </summary>
      public int NumCallsExp { get; set; }
    }  // end: FibTestCase{}
```

At this point, the definition gives us IntelliSense support and allows us to easily add new properties or omit existing ones as we add test cases to **`testValues[]`**. In your projects, these auto-implemented property definitions seen here may be good enough for what you need, since we expect that this class will not be used outside its TDS test method, and you may not need to expend the effort to define the components as read-only properties.

However, unlike in the anonymous objects, these properties are now changeable, and we can gain some protection from accidental changes by doing a bit more editing, which we shall illustrate next. As a frequent victim of self-imposed mistakes, I prefer making all the properties in these objects be read only, trying to take advantage of all the automatic protections the system affords me, so most of the examples in this *TDS User's Guide* use the **`{get; private set;}`** style.

*5.2.9.6.3.3 CONVERT **`TESTVALUES[0]`** TO BE AN INSTANCE CONSTRUCTOR OF THE NAMED TYPE.*

We shall replace the named-type object initializer that we just now created in **`testValues[0]`** with a constructor call for that same type, to allow us to specify the values and types of the read-only[91] properties

[91] Actually, they're still read-write properties, but we'll soon be able to convert any of them to be a read-only property, via code such as "**`public string Id { get; private set; }`**", if we replace the present object initializers with constructor calls, as we are about to do in this example.

that we plan to create in **`FibTestCase{}`**, similarly to specifying the read-only properties of the anonymous object initializers.

We'll bring the other **`testValues[]`** elements back soon. We need to edit **`testValues[0]`** a bit, without much help from VS, to construct an instance constructor call for the class we just now defined. Having done that, we'll ask VS to construct a matching instance constructor definition from this call, and we'll edit the constructor definition to give it qualities (such as default values) that we want it to have. After the constructor is complete, we can use it to provide IntelliSense help in editing the other existing members of **`testValues[]`**, which are currently commented out, and (even more helpfully) to provide IntelliSense help as we add new members to **`testValues[]`**.

In **`testValues[0]`**, do the following:

► Change the outer braces around **`testValues[0]`** (beginning with the line containing "**`new FibTestCase`**") to parentheses: "{" to "(" and "}" to ")".

There's only one of each. Other braces that may appear between them (for example, appearing in expressions giving the properties their values) should not be changed. In the present example, of course, there aren't any of those.

► Change the equal signs to colons.

Replace the first "=" sign after each property name with ":". There should be one of these following each property name. Be careful — as with braces, there may also be "=" signs embedded in the expressions that provide some of the values, so don't blindly do a find-and-replace operation. That would, however, work in this example — here there are five "**=**" signs, one for each property (each of which is about to become a parameter).

► Remove the comma following the last member declarator of this object initializer.

The comma following the last member declarator ("**`NumCallsExp : -1,`**" in this example), which I usually leave in place to make rearranging the members easier, is not permitted in a constructor call — it confuses the compiler into thinking that a parameter is missing.

5.2.9.6.3.4 GENERATE THE CONSTRUCTOR DEFINITION FROM THE CONSTRUCTOR CALL

Now the compiler notices that the **`Id`** parameter is unexpected and has flagged it with a wiggly underline.

► Move the mouse cursor to the **`Id`** parameter, click on the pop-up, and click on "Generate constructor in 'FebTestCase' ".

A constructor definition appears at the beginning of the **`FibTestCase{}`** class definition; you may navigate there by right-clicking on the name "FibTestCase" and selecting "Go To Definition".

The beginning of the class definition might now look like this:

```
    internal class FibTestCase
    {
      public FibTestCase(string Id, int N0Low, int N1High, string ExceptionExp,
int NumCallsExp)
      {
        this.Id = Id;
        this.N0Low = N0Low;
        this.N1High = N1High;
        this.ExceptionExp = ExceptionExp;
        this.NumCallsExp = NumCallsExp;
      }
…
```

With this definition, we can now do more, such as to comment the parameters and give them default values, as well as make the properties be read only.

Let's begin with the properties.

► In the definition of **`FibTestCase{}`**, in the definition of each of its five properties, change "**`{ get; set; }`**" to "**`{ get; private set; }`**". For example, change this:

```
    public string ExceptionExp { get; set; }
```

to this:

```
    public string ExceptionExp { get; private set; }
```

I might do this by selecting these five lines, using menu "Edit, Find and Replace, Quick Replace" to replace each "**`{ get; set; }`**", limiting the replacement to the selection, and replacing all five occurrences. With only five to modify, however, it may be easier just to paste "**`private `**" (or type "p<tab>") in front of each "**`set;`**".

Doing this makes each property read only (except in the constructor) and gives it an invisible backing field.

► To help us match braces, we might add a comment on the closing brace of the constructor, to look something like this:

```
      }  // end: FibTestCase()
```

On such comments, I usually leave the parentheses empty unless the definition is overloaded, in which event I include only enough information to distinguish among the overloads.

► Add XML comments to the constructor.

To do this, insert a blank line above its first line (beginning "**`public FibTestCase(…`**") and type "///" there. Tags for the parameters will appear.

VS should generate the following XML comments for this contructor:

```
      /// <summary>
      ///
      /// </summary>
      /// <param name="Id"></param>
      /// <param name="N0Low"></param>
      /// <param name="N1High"></param>
      /// <param name="ExceptionExp"></param>
      /// <param name="NumCallsExp"></param>
```


I usually copy the XML comments from the new class's properties to its constructor parameters, since the IntelliSense pop-ups use both (the parameter comments as you edit a constructor call, the property comments as you use the class's properties in expressions). To facilitate such copying in my code, I sometimes break the **<param>** and **</param>** tags for each parameter onto separate lines, to allow easily inserting the copied comments into the lines between them. For example, see the treatment of the]\ **<param name="NumCallsExp">** comment in this example. I usually use a split editing window (via the  button in the editor) to facilitate copying.

With text from the XML comments on the properties copied to these **<param>** elements, along with something relevant in the **<summary>** element, the constructor definition's XML comments might look like this:

```
        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="Id">
        /// Test case identifier
        /// </param>
        /// <param name="N0Low">
        /// Lower value of argument used in this test
        /// </param>
        /// <param name="N1High">
        /// Upper value of argument used in this test
        /// </param>
        /// <param name="ExceptionExp">
        /// Expected exception
        /// </param>
        /// <param name="NumCallsExp">
        /// Expected # of invocations
        /// of the recursive function,
        /// or -1 if we are not checking it
        /// </param>
```

These XML comments are kind of skimpy, but in this context, probably all we need are reminders. XML comments on objects used in more distant locations are likely to be more helpful if they contain enough detail to make it unnecessary to examine the source code when using the objects.

All that remains to be done here is specifying the default values, if we wish to do so.

▶ From **testValues[0]**, copy the values of the parameters, except for that of **Id**, and set those as default values in the constructor definition.

We do this here because it's convenient, but in your own projects, you may find that other default values would be more suitable than whatever your **testValues[0]** happens to contain.

I usually break up the first line of the constructor definition, inserting a line break before each comma and the closing parenthesis, to put the value on the same line as the name, making it easier to add default values,

reorder them, and read them. Having added the default values, the constructor's identifier and formal parameter list (inside the **`FibTestCase{}`** class's definition) might look like this:

```
public FibTestCase(string Id
  , int N0Low = 0
  , int N1High = FibTestLimit
  , string ExceptionExp = DefaultExceptionMessage
  , int NumCallsExp = -1
  )
```

I intentionally did not give **`Id`** a default value, because the **`Id`** property is intended to give each test case a unique name, and I wanted that name to be required and to be listed first in each constructor call. If we accidentally omit or misspell the "**`Id`**" parameter from a test-case constructor call, we will want to get an immediate compile-time error message warning us to correct that.

For the other properties, having default values makes them optional, so they may be either specified or omitted in a constructor call, and any that are omitted are given their default values. Using a default value (by omitting a parameter in the constructor call) is most helpful if that value is likely to be used frequently and is somewhat easy to remember. Some of the values that we used earlier in the anonymous constructors may be good candidates for default values. The IntelliSense pop-ups will display the default values of parameters even if you choose not to specify any XML comments, but using easily remembered names for properties and easily remembered default values for parameters helps make the code easier to read and understand.

For each parameter, it is probably most helpful if you choose a default value that

- can be used in several of your test cases, or
- is otherwise easy to remember (XML comments in the constructor can help with this).

Effectively defining and using default values in your constructor calls in **`testValues[]`** can simplify them, for example allowing you to shorten a test case by specifying only the values that apply specially to that case. It is possible that, as with **`Id`**, you may want some of the parameters to be required because they should not have the same value in more than one invocation of the constructor.

5.2.9.6.3.5 NOTE ON XML COMMENTS

What we have done with these XML comments may seem a bit excessive, considering that they apply to an **`internal`** type that is used only in one TDS method and is not delivered to the customer. But consider also that as you develop the TDS method (along with the function member it calls), you may wind up defining many dozens of instances of the type (as additional **`testValues[]`** test cases) and may place dozens of references to them into **`Assert`** statements. Both the definitions and the references need to be accurate, and these XML comments (reflected in IntelliSense as you use the test cases while editing code) can do much to help ensure that they are accurate and consistent, with almost no effort required once the named-type definition is complete.

In case your effort spent on writing comments seems wasted, consider that, if your project goes well, this test code will be used only rarely, for example when a bug appears months in the future, and you will not have seen this code for a while. Don't spend too much time on comments — copy when you can — but try to make them useful. Helpful comments now can make this by-then-unfamiliar test code far easier at a future time to understand, modify, and use. (I have dealt with spaghetti code that would have been far more puzzling than it turned out to be, except for the helpful comments built into it, thanks to considerate programmers. Any other documentation that it might have had had been lost in the mists of time.) If your project *doesn't* go well, then

quite possibly a few well-written comments can help avoid many frazzled nerves. Time spent on helpful comments is an investment.

5.2.9.6.3.6 EDIT THE CONSTRUCTOR CALL

By now, **`testValues[0]`** contains a constructor call that can look like this if we specify all the parameters:

```
        new FibTestCase (Id : "01 Low, valid input",
          N0Low : 0,
          N1High : FibTestLimit,
          ExceptionExp : DefaultExceptionMessage,
          NumCallsExp : -1
        ),
```

Though the in-line comments are now gone from this source code, they are still visible in the IntelliSense pop-up as we edit the parameter list, or when the mouse I-beam cursor is moved to a parameter name. The nearby screen shot shows that placing the cursor onto **`N0Low`** displays this text:

```
(parameter) int N0Low = 0
Lower value of argument used in this test
```

This pretty well covers the contents of the original line in the anonymous-type **`testValues[0]`**, before we began using FibTestCase:

```
N0Low = 0, // Lower value of argument used in this test
```

Looking at our default values, it occurs to us that the value we chose as the default value for **`N1High`** is not one that we would use "frequently". More often, we might want its value to match that of **`N0Low`** so that we could use the test case to deliver only one value to the working code instead of the usual set of several values. We could then run only that one test case, to assist us with tracing into that working code. We can already do that, but this would let us do it more concisely by not having to specify the same value for both **`N0Low`** and **`N1High`**.

So… we set a new default value (-1) for **N1High** and edit the constructor to treat it as meaning that we shall run the test case using only the **N0Low** value:

```
        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="Id">
        /// Test case identifier
        /// </param>
        /// <param name="N0Low">
        /// Lower value of argument used in this test
        /// </param>
        /// <param name="N1High">
        /// Upper value of argument used in this test,
        /// but the default value is <see cref="N0Low"/>
        /// </param>
        /// <param name="ExceptionExp">
        /// Expected exception
        /// </param>
        /// <param name="NumCallsExp">
        /// Expected # of invocations
        /// of the recursive function,
        /// or -1 if we are not checking it
        /// </param>
        public FibTestCase(string Id
          , int N0Low = 0
          , int N1High = -1
          , string ExceptionExp = DefaultExceptionMessage
          , int NumCallsExp = -1
          )
        {
          this.Id = Id;
          this.N0Low = N0Low;
          this.N1High = N1High == -1
                ? N0Low  //The real default value of N1High
                : N1High;
          this.ExceptionExp = ExceptionExp;
          this.NumCallsExp = NumCallsExp;
        }  // end: FibTestCase()
```

The only changes we have made here are in the treatment of the parameter **N1High** and in its XML comments.

Now the XML comments for the properties and the constructor parameters no longer fully match. The original comment for the property **N1High** is still accurate, but the comment for the parameter **N1High** accounts for the special case of having it match **N0Low**. This comment is necessary because the IntelliSense (and Object Browser's) summary of the constructor's signature shows that the default value of **N1High** is now -1, but that is misleading, since the property's value will default to whatever value is specified for **N0Low**.

If we take advantage of the default values, we can abbreviate this first **testValues[0]** constructor to look more like this, and giving the **Id** parameter a more appropriate description:

```
    new FibTestCase ("01 Using default values",
      N1High: FibTestLimit),
```

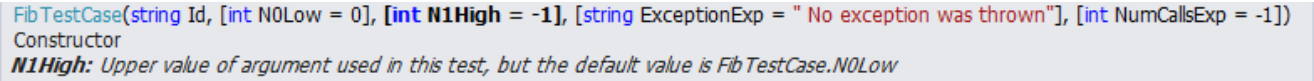
Well, OK, now this is *very* concise, probably more so than necessary, but with this test case definition the test still passes. We could have made it even more concise by omitting the **N1High:** line, and the test would still

have passed, but only the **`NewCode.Fib(0)`** call would have been run, with no indication in the test report that any values were missing.

When you're using the VS source-code editor, even this stripped-down constructor call contains some hidden documentation.

- Replace the contents of **`testValues[0]`** with this abbreviated version.
- Click on the parameter value (in this case, click inside the right parenthes

is), then press <shift><control><space> in VS (or type and erase a comma), to display the following IntelliSense pop-up or something similar:



All of the parameter names, their types, and their default values are displayed here, along with comments about the constructor and its **`N1High`** parameter, and this information doesn't clutter the code when you don't need to see it. (However, you do need to be using VS's code editor or the Object Browser to be able to view it.)

5.2.9.6.3.7 REMOVE REDUNDANT CONTENTS FROM `TESTVALUES[]`

The following steps are optional; they remove material that is now redundant, making the code more concise and, one hopes, easier to read and maintain, but the code will still compile and run correctly without making these changes. In the stripped-down version of **`testValues[0]`** that you just now used to replace the previous one, these steps have already been done, but the other test cases in **`testValues[]`** will need them. We shall also remove some other comments left over from the original TdsTest snippet.

In each of the remaining **`testValues[]`** elements,

- (optionally) Delete unneeded parameter names and references to default values.
- (optionally) Delete any comments that are reflected in the IntelliSense pop-ups because they now exist as XML comments in the named class, **`FibTestCase{}`** and are therefore redundant.

Comments on special values of the properties, however, still apply to those special values and don't exist anywhere else, so they should remain in their **`testValues[]`** elements.

5.2.9.6.3.8 DE-COMMENT AND CONVERT OTHER TEST CASES

We commented out the remaining elements of **`testValues[]`** to reduce the work involved in converting them to constructor calls while we defined the constructor. We could de-comment and edit these initializers now to convert them to constructors, or it might be easier to copy the constructor from **`testValues[0]`** and edit the copies using values copied from the commented-out test cases.

As described in section 5.2.9.6.3.3, having de-commented an anonymous-type initializer, we can

- insert the type name, **`FibTestCase`**,
- replace the outer braces, "{" and "}", with parentheses, "(" and ")",
- replace the first equal sign in each parameter, "=", with a colon, ":",
- erase the comma following the last parameter, and
- (optionally) delete unneeded parameter names and references to default values
- (optionally) delete any comments that are now reflected in the IntelliSense pop-ups because they have been copied to XML comments in the named class.

After we de-comment some earlier anonymous-type objects that we earlier (in section 5.2.8.1.1) converted to comments, with Id tags "02", "03", and "10", edit them to recast them into **FibTestCase** object constructors, and order them by tag number, the contents of **testValues[]** consist of these:

```
        new FibTestCase ("01 Using default values",
              N1High : FibTestLimit),

        new FibTestCase("02 High number (18) sent to slow version",
              N0Low : 18,
              NumCallsExp : 8361),

        new FibTestCase("03 High number (20) sent to slow version",
              N0Low : 20,
              NumCallsExp : 21891),

        new FibTestCase("10 This should raise an exception",
              FibTestLimit + 1,  //This should cause overflow
              ExceptionExp : "n must be"),
```

Parameter names that are omitted in these constructor calls are still visible via IntelliSense. Any optional parameters following the last unnamed parameter may be rearranged. Parameters that are to be given their default values may be omitted.

To improve the documentation, I also replaced "48" in test case "10" with "**FibTestLimit + 1**". Even though I don't expect the value to need to be changed, the name does carry with it a comment explaining why it was chosen, and "48" conveys no such information. On the other hand, clicking on the name "**FibTestLimit**" does show that its value is 47, so no information is lost in that way. Probably there is no longer any need for the in-line comment on that line, but since it was a comment addressing a specific value in the original code, I left it there.

Seeing that the original test case "11" (for "**N0Low : FibTestLimit**") is redundant, since its tests are already handled by test case "01", we drop that one, but we can easily add some other test cases:

```
        new FibTestCase("20 Out of bounds negative",
          -3, -2,
          ExceptionExp : "n must be"),

        new FibTestCase("21 Out of bounds negative",
          -1,  //Special case because of default value
          ExceptionExp : "n must be"),

        new FibTestCase("22 Wrong order of limits",
          8, 6, //Ending value must not be lower than starting
          ExceptionExp : "N1High must"),
```

▶ Run the tests after adding these elements to **testValues[]**.

We should see that all tests pass.

The reason for test case "21 Out of bounds negative" is that our choice of -1 for the default **N1High** value got into the way of using that value for its usual purpose of identifying the highest value of a set of two or more. Was there an easy way to send the value -1 to the method? We used the extra test case as a workaround (with **N0Low** and **N1High** both equal to -1). To verify that the argument -1 is being sent to the working code, we

could temporarily edit test case "21" to change its value of **ExceptionExp** and run the test; the failure message (due to an unexpected exception) would show the value of the parameter sent to **Fib()**.

Expecting new test case "22" to fail, since we haven't provided code for any exception beginning "N1High", when we run the tests including these new test cases, case "22" accidentally passes. This was one of those cases where we specified conditions that we plan to address soon in the tested code, but where we wanted the test to fail until we've done that. Oops! It didn't fail at all. (We might notice that it didn't do anything else, either, such as calling the working code.) We would like to disallow such events, since specifying nonsensical inputs (in this case, it was listing the limits in the wrong order, but there are lots of other ways to make silly mistakes) may not raise an exception or otherwise attract attention, but they are still misleading. For example, this mistake might lead me to think that I am running tests that are actually not being run at all.

We already have safeguards in the code relating to negative arguments, as tested by test case "20", in which the working code, **Fib()**, raises an exception. Since the current problem appears to lie entirely within the TDS code, we need to address it within **FibTest()**.

For this problem, we could check for inconsistent limits in the test cases by adding code to **FibTest()**, at the beginning of the "**foreach (var tCase in** ..." loop. Even though we're not examining working-code output here, we can still use an **Assert** statement to report our problem.

► In the **FibTestCase()** constructor, add the following code immediately before the end of the constructor's body:

```
        //Check tCase properties for consistency
        Assert.IsTrue(this.N1High >= this.N0Low,
              string.Format(
@"N1High must be no lower than N0Low.
    FibTest() internal error in test case ""{0}"" :
      N0Low = {1}; N1High = {2}"
                , this.Id  //{0}
                , this.N0Low  //{1}
                , this.N1High  //{2}
              ));
```

If this assertion fails, which will likely be right away, we'll know it's an internal mistake and can take care of it promptly by editing the identified test case.

```
  - FibTest()
      Exception message:
Assert.IsTrue failed. N1High must be no lower than N0Low.
    FibTest() internal error in test case "22 Wrong order of limits" :
      N0Low = 8; N1High = 6
```

Even though the expected exception for this test case matches the message generated by the exception, that is unimportant, as the test case itself is erroneous and needs to be corrected.

► Delete or comment out test case "22".

5.2.9.6.3.9 IDENTIFYING THE LOOP INDEX (IF ANY)

This section applies only if you are placing some of your **Assert** statements inside a repeating structure such as a **for()** or **foreach()** loop. If so, it will likely be helpful to include in the **Assert** messages some information about which iteration of the loop was active when the **Assert** exception occurred. Similarly to taggaing the message to identify which of several **Assert** statements issued it, as we did in section 5.2.8.5.2.3,

since this example does involve loops, let's also include information that can help identify the conditions (such as which iteration through the loop it is) that gave rise to the error. In this example, the index to the **`for()`** loop, **`nValue`**, can be included in the **`Assert`** statement's message (and already is included in those that we have added since we created the loop). For example, we might modify the **`Assert`** statement that tests for an expected but missing exception to make it look like this:

```
            //Test that if no exception occurred, none was expected.
            Assert.IsTrue(
                  exceptionMsgExp == DefaultExceptionMessage,
                  MsgForMissingException(
                    "Fib", tCase.Id,
                    exceptionMsgExp
                  ) + "  Found in call to Fib("
                  + nValue + ")."
                );
```

The reference to **`nValue`** that we added near the end of the statement makes it easy to determine which iteration of the loop encountered the error.

- To demonstrate that this is working, add the following (failing) item to **`testValues[]`** and run TDS:

```
        new FibTestCase(
          "30 Expected event didn't materialize",
          -3, +3, "n must be"),
```

This test item is expecting all the specified argument values (-3, -2, -1, 0. 1. 2. 3) to raise an invlaid-parameter exception, and they all did so until we reached **`Fib(0)`**, as revealed by the error message:

```
FibTest(), test case 30 Expected event didn't materialize:
  No Exception was raised in this test case,
  but Exception "n must be" was expected.  Found in call to Fib(0).
```

5.2.9.6.3.10 CHECK COVERAGE OF INPUT ARGUMENT VALUES

In case we want to monitor which arguments we are actually sending to **`Fib()`** (but there's no need to actually do it in this exercise), we could place into **`FibTest()`** a statement such as

```
              Console.WriteLine(
                @"FibTest(): Fib({0}) is called by test case ""{1}""."
                      , nValue  //{0}
                      , tCase.Id  //{1]
                    );
```

after the "**`TODO: FibTest() -- Provide a suitable calling expression`**" Task comment, and directly before the

```
    actual = NewCode.Fib(nValue);
```


statement, and run the tests. The test report would now contain several lines of output identifying the actual data sent to the working code. In the following example, I have omitted some lines from test case "01" for brevity.

```
***** TDS.Test.FibTest()
***** InitializeTestMethod() was called at 2016-12-27T09:10:18.1184979-06:00 .
FibTest(): Fib(0) is called by test case "01 Low, valid input".
FibTest(): Fib(1) is called by test case "01 Low, valid input".
FibTest(): Fib(2) is called by test case "01 Low, valid input".
…
FibTest(): Fib(45) is called by test case "01 Low, valid input".
FibTest(): Fib(46) is called by test case "01 Low, valid input".
FibTest(): Fib(47) is called by test case "01 Low, valid input".
FibTest(): Fib(18) is called by test case "02 High number (18) sent to slow
version".
FibTest(): Fib(20) is called by test case "03 High number (20) sent to slow
version".
FibTest(): Fib(48) is called by test case "10  This should raise an exception".
FibTest(): Fib(-3) is called by test case "20 Out of bounds negative".
FibTest(): Fib(-2) is called by test case "20 Out of bounds negative".
FibTest(): Fib(-1) is called by test case "21 Out of bounds negative".
***** CleanupTestMethod() is complete.
***** (End of test)
```

A visual check in this case could confirm that we had included all the values we wished to test, though we might notice that we used 18 and 20 as arguments twice (to check **`NumCallsExp`**) and 47 twice (more than necessary). The test report also shows us that all of the tests passed.

Doing the checking this way would be practical only with a small number of such values, as we have here; we would use a different technique if we needed to verify coverage of a large set of inputs. Here, having checked this output, we could then erase the **`Console.WriteLine()`** statement that we added.

5.2.9.6.3.11 CONSTRUCTOR OVERLOADS

You don't need to limit yourself to using only one constructor in **`testValues[]`**. You may have reason to define constructor overloads that accept parameters that may not even directly correspond to the object's properties, but rather are used to calculate them. (We sort of did that with the **`N1High`** parameter in the **`FibTestCase()`** constructor; that parameter isn't exactly the same as the **`FibTestCase.N1High`** property.)

For example, suppose that we want to define several test cases using **`FibTestCase{}`** in which we want to specify only a single value for the argument, which we have been doing by giving the **`N1High`** property a special default value, but suppose this trick can't be used because no suitable value is available, or we think doing this makes the code difficult to read. Notice that with our current default value of -1 for **`N1High`**, we cannot specify a set of test arguments in which -1 is the highest value. (For the sake of argument, assume that we might actually want to do something like that.) We could instead define an alternate constructor having an

integer parameter called **SingleValue**, whose value would be given to both properties, and omitting the **N0Low** and **N1High** properties from that overload's parameter list. It's easily defined and might look like this:

```
        /// <summary>
        /// Constructor specifying a single argument
        /// and a corresponding exception
        /// </summary>
        /// <param name="Id">Test case identifier</param>
        /// <param name="SingleValue">Argument used in this test</param>
        /// <param name="ExceptionExp">Exception to be raised
        /// by this value, "" if none is expected</param>
        public FibTestCase(string Id
          , int SingleValue
          , string ExceptionExp
        ) : this(Id, SingleValue, SingleValue
            , ExceptionExp : ExceptionExp )
        { }  // end: FibTestCase(string, int, string)
```

In this constructor overload, I made **ExceptionExp** be a required parameter to keep this constructor from being ambiguous with the first one. I do include a more detailed version of the comment[92] on its closing brace here.

This constructor might be called using the following **testValues[]** element:

```
            new FibTestCase("30 Single argument with exception",
                  4, ""),
```

Similarly, you might have a set of test cases in which several fields or properties keep the same values throughout that set of cases. Instead of specifying those values repeatedly, you might use a parameter in the constructor (maybe an **int** or **enum**) that identifies the set, and use the value of that parameter to determine what set of values those fields or properties should be given[93].

You might be able to use the tag on the **Id** property for this purpose; this could save time and make the code slightly more concise. However, if you do this, I suggest being a bit cautious — that tag would then have multiple purposes. It would still identify a failing test case (for use in failure messages) and it could still be used to group related test cases (as we shall do later, in sections 5.3.17.4.3 and 5.3.17.6.1), but it would now also be put into use to specify sets of values of multiple fields or properties.

Why might this be a problem? Consider that, in a relational database, it's often a good idea to make every table's primary-key field be completely meaningless beyond its use as the primary key. Otherwise, the database designers or administrators might get the idea that that key value can also be used for some

---

[92] Usually, for brevity, on the "**// end:** …" comment on the closing brace of a method with no overloads, I omit the parameters that would normally appear within the parentheses. On this overload, I included the types of its parameters, to help match with the opening brace. Perhaps, for consistency I should update the comment on the closing brace of the original version as well… but I usually do so only if it's needed to avoid confusion, and with short definitions like these, confusion seems unlikely. In this example, the comment isn't needed for matching the braces, since they share the same line, but I also use it for matching with the first occurrence of the name. You are, of course, welcome to totally omit any such comments.

[93] See the discussion in section 4.4.4 of the **OverloadSig** property used in the **testValues[]** array of **TimeRoundedTest()**. We do something similar in section 5.3.17.4.1.2, where the **DocNum** property of the **testValues[]** array is used as an index to an array of documents to determine which one is to be sent to the working code.

additional purpose, thereby making it a permanent part of the table. This might cause future versions of the database to be required to maintain a field that was once used as a primary key but is now needed only because it acquired that secondary purpose, and except for that historical reason its function could have been accomplished in a more efficient manner. **`testValues[]`** may be thought of as a database table, with the **`Id`** property's tag as its primary key. These multi-purpose **`Id`** tags might thus become difficult to change later if you wished to reorganize the test cases.

5.2.9.6.4 WHY NOT ALWAYS START **`TESTVALUES[]`** USING A NAMED TYPE?

You might choose, as a matter of course, to set up a named type for the elements in the **`testValues[]`** array of each of your TDS methods. If I were following a TDD protocol (see section 1.8.1), this is probably what I would do. The reason I did not do that in this example, and that I usually use anonymous types for **`testValues[]`** elements, is that, early in the TDS method's life, less work is needed to define or redesign anonymous objects, when only a few of them exist. Most changes tend to be needed soon after a new TDS method is defined, while the function member being tested is still being developed and its interface with its environment is somewhat malleable. During that time, **`testValues[0]`** may be the only test case needed.

Many TDS methods (as I usually use them) will employ only a small number of test cases or will otherwise never gain much benefit from having a named type. Much of the development of the called function member may be complete by the time I need to define a second **`testValues[]`** element, and two or three such elements may be all that I shall ever need for the TDS method. In my projects, only if I expect ahead of time that I will need to use a named type do I use one from the beginning. However, you may find that it saves time to set up your TDS methods the same way each time — with named types for the test cases in all of your TDS methods — so that no conversion of initializers to constructors will ever be needed for them.

5.3 Example: Modifying an XElement via a new method

5.3.1 Overview of this example

The previous examples have demonstrated using TDS to help develop some simple-minded methods that you might have been able to develop in your sleep. (Well, at least once you have the desired algorithm for **`Fib()`**, expressing it in C# code is fairly trivial.) Now we want to give TDS a function member that it can sink its teeth into, but without creating much additional complexity in the TDS method itself.

This example will illustrate using TDS to help build a new function member, which will again be a method. Even though this method will write to the Console, allowing us to observe some of its inner workings, this time we do not intend to automatically test the values of those messages. (You've already seen that done with **`TestableConsoleMethodTest()`** in section 4.4.3.3 and elsewhere, so there's no need to repeat it here.)

The method to be created in this example will modify some XML documents. A moderately large XML document is trickier to trace than simple types such as Boolean or integer values, and the benefit of using TDS to help with development will perhaps be more apparent in this example than it was with the simpler types used earlier. Even if you have no special interest in XML-valued objects, you can think of them as surrogates for other types of complex objects (word-processor documents, **`Regex`** transformations, or databases, for example) that a function member might need to access, construct, or modify.

Incidentally, the parser from which the XML Schema Definition language (XSD) schema used in this example is derived employs some of the powerful but sometimes-puzzling .NET **`System.Text.RegularExpressions`** members (including the **`Regex{}`** class), and I used TDS in developing and debugging the parser, but this example works well enough without using any tricky **`Regex{}`** objects. So… the only complex objects used in this example are XML-valued objects.

The new method, to be called **`InsertSymbol()`**, might look a bit hairy, but you will be able to simply copy code from this *TDS User's Guide* into it as a simulation of developing the method. I'll describe the purpose of the change, and the copyable code will reflect the results of implementing the change. Following the details of developing **`InsertSymbol()`** is not an important part of this discussion — the main objective is to set up the TDS method that will run and test it, in an environment in which the function member under development is undergoing major changes and needs heavy support from the TDS method during those changes.

The process shown here of writing the code for **`InsertSymbol()`** will be a much-abbreviated version of reality. Depending on your coding style, the actual writing of a function member like this might involve more refactorings[94] than are shown here, and those details are immaterial to the use of TDS. So, we shall hint at how to refactor the new function member (and show the results), but will focus more on how its TDS method, **`InsertSymbolTest()`**, might need to be modified to continue to match its shifting requirements and, of course, to verify that the refactorings had no material effect on its behavior. (If you have never needed to deal with shifting requirements, be thankful — many people have.)

[94] Refactorings = changes of expression in the code that do not alter any behavior that we care about.

5.3.2 Learning objectives

When you complete this example, you will have…

- used TDS to assist in debugging a method that uses XML-valued parameters and reads an attached data file (an XSD file)
- constructed XML-valued objects and edited them for use in test cases
- incrementally modified the new method's interface with its caller, as the method is developed
- used TDS to monitor refactorings, such as naming common subexpressions to facilitate software maintenance

5.3.3 Statement of purpose of the code in this example

The end product of this exercise is to be a method that, given an XML document that represents an English-language sentence,

- validates the document according to the pre-specified XSD schema (to be given in a file that we shall attach to the project),
- inserts into the document some additional XML elements (terminal **`<Symbol>`**s) representing words in the sentence,
- validates the resulting modified document, and
- returns a string summarizing the words in the changed document.

Let's assume that, at this stage, the XSD schema has been agreed upon and we are not allowed to change it, but the exact format of the inputs and outputs is, for now, more fluid than that, allowing us considerable latitude in specifying them. We may find, as we develop and exercise our code, that some work is being duplicated or that some variables are being specified in an inefficient or hard-to-read manner. The act of altering the method's design will be easy at first, trickier later, a bit like firing a clay pot or physically printing a book — after publishing the method, improving on the design will no longer be a low-cost option, since any subsequent changes will involve other people. For this example, we will assume that we haven't yet reached that point, and that we would like to do now whatever we can do to reduce the need for later changes.

5.3.4 Analysis

This example pretty much omits any analysis of the problem. Yes, I remember what I said before (in section 5.2.7) about the importance of analysis, but here the XSD file keeps track of any grammatical problems in the XML code, so any improper (syntactic) changes to the sentence are usually caught promptly. (We assume that we don't care here about semantic mistakes, as they are immaterial to anything **`InsertSymbol()`** does; it merely copies elements without changing them.)

I'll try to show in the testing phase that, as we think of new ways to cause undesired results, we can easily add test cases to identify them before they can cause trouble. Adding test cases in this way is similar to the analysis that we might do in a mathematical proof, taking care of special cases after we've addressed the majority of the domain we're dealing with. In a proof, we might say something like "the divisor must not equal zero"; similarly, in a method we might raise an exception if a needed input has a value of zero — and test that by feeding the method a zero and verifying that it raises the proper type of exception.

Incidentally, concerning that XSD schema, I originally tried to have Visual Studio construct it automatically, inferring it from an example XML document (using VS menu "XML, Create Schema"). That wasn't very satisfactory, as I wanted the **`<Symbol>`** tags to be defined recursively, and to exist as two species, terminal and non-terminal. Of course, Visual Studio also couldn't offer any comments to describe what I had in mind for

each type of tag, so, without much automated assistance, I defined my own version of the XSD schema, including the comments. Having done so, I was delighted to notice that, when I was editing XML documents with the help of this XSD schema, I enjoyed IntelliSense assistance similar to what I am accustomed to seeing when editing C# code, such as the "Complete Word" feature and pop-ups presenting lists of expected values. It is even context sensitive, offering differing, and correct, comments for each type of **<Symbol>**. Nice work, Microsoft!

5.3.5 Requirements for the `InsertSymbol()` method

For this example, we'll even go a bit further than assuming a slight fluidity in the method's interface — we'll assume that it's not even obvious at the outset exactly how we want the method to proceed, only that we want to develop some convenient way to add desired elements to an existing XML **<sentence>** document and to summarize the results. We'll let experience from running prototypes, and maybe consultation with potential users (ideally, with written records of such meetings available to us), guide us.

I understand that this approach — starting development without fully detailed specifications – may be far afield from the way you normally develop code, but in my experience the final requirements have not always been in place at the time I have begun coding a function member. (On the opposite side, trying to define a complete set of requirements with no opportunity to play with the results is no picnic, either — I imagine that the most successful specification writers draw heavily on their experience implementing previous systems and recovering from mistakes.) Starting to code without fully defined requirements, while keeping in mind the likelihood of having to modify the code later as the specifics of the requirements become evident, can help to clarify the problem by providing feedback to the requirements-definition process.

Since the purpose of the present discussion is illustrating how to use TDS to support development of a function member, I shall not go into much detail concerning the writing of the code, so if some of this procedure seems unpleasantly sloppy or out-of-sequence to you, please feel free to skim over it to the description of the results. (In this example you'll mostly copy code fragments and observe what they do.) Regardless of how the code in the new method was designed, I expect that the final version, as it will be presented in this example, will work dependably, and that you will have an easy-to-use mechanism (the TDS test method) to detect any remaining defects. (Incidentally, even despite this expectation, I offer no guarantees here, but TDS has worked well for me in similar circumstances.)

To begin with, we know that the **`InsertSymbol()`** method must be able to access an existing **<Sentence>** document and perform on it the operations specified in section 5.3.3 above.

We shall allow this method to display output onto the Console, but we do not plan for it to accept any input from the keyboard, and we do not intend to do any automatic analysis of the Console output, so we shall use the **`TdsTest`** snippet as a basis for our TDS method. (This is our usual TDS template, one that does not specifically address input or output involving the Console.)

For the moment, we shall also assume that we will have considerable latitude in specifying the interface, since nothing has been published yet and therefore nobody else depends on that interface. If that assumption later proves wrong, we may have to redo some of the early work, but if what we develop early is modular, testable, and sufficiently documented, then we ought to be able to re-use much of it to satisfy the revised requirements.

5.3.6 Set up a new function-member stub and its TDS method

(We developed the previous example, in section 5.2.6, similarly to this one.)

5.3.6.1 Create a TDS method to exercise `InsertSymbol()`

5.3.6.1.1 SET UP A PROJECT WITH TDS CODE

If you already have a VS Solution with a TDS Project to which you want to add this example code, open that Solution and skip to section 5.3.6.1.2.

► Follow the steps in section 4.14.7, "Setting up a stand-alone TDS Project", to construct a new VS Solution.

The VS Solution that you have just now constructed should contain only a (mostly empty) "ConsoleApp1" Project and the "TDS" Project that we shall use to exercise the **`InsertSymbol()`** method, which will be our working code in this example.

5.3.6.1.2 ADD A TDS METHOD

► Choose a location for your new test method.

It needs to be somewhere in the TDS Project, so either file TDS_Ex01.cs or TDS.cs would be suitable, or we could create a new file for it, as we did in section 4.10. For this exercise, we'll use TDS.cs.

The location I would use is somewhere between the "**`TODO: New TDS methods may be placed here:`**" Task List comment and the line

```
}  // end: Test{}
```

near the end of the file, but it must be located at the top level within the **`TDS.Test{}`** class. If you have other TDS methods defined here, their relative order (alphabetical, for example, which is what I usually use) is unimportant.

► At your desired location, use the **`TdsTest`** code snippet to generate a TDS method for the to-be-defined method **`InsertSymbol()`**, as we did in sections 4.8.2.1, 5.1.5.1.1.2, and 5.2.6.1.2. Type the name "InsertSymbol" into its "TestableFunctionMember" field and press <enter>.

This new TDS method will be called **`InsertSymbolTest()`**.

5.3.6.1.3 CREATE AN EXAMPLE WORKING-CODE NAMESPACE

You may place the new working-code method **`InsertSymbol()`** into an existing working-code namespace or create a new one for this example. Let's use the same class, "**`Working_Code.NewCode{}`**", that we used in the previous example.

► If namespace **`Working_Code`** and class **`Working_Code.NewCode{}`** already exist (from the previous example), then you may skip to section 5.3.6.1.4 below.

► Insert the following code at the end of file Program.cs in the ConsoleApp1 Project, following all of the existing code in that file (that is, below the closing brace, "**`}`**", of namespace **`ConsoleApp1`**):

```
/// <summary>
/// Simulated working code to be exercised by TDS methods
/// </summary>
namespace Working_Code
{
  /// <summary>
  /// Class containing methods to be developed
  /// with the help of TDS
  /// </summary>
  public class NewCode
  {
  }  // end: NewCode{}
}  // end: Working_Code namespace
```

This is the same code shown in section 5.2.6.1.3. (The XML comments on the **Working_Code** namespace are ignored by the Object Browser, but they are harmless.)

► Set a Reference in the TDS Project to the working-code Project, **ConsoleApp1**, if necessary. (See section 4.4.1.2.)

► At the "**TODO: Usings -- Include "using" statements for the namespaces of the code**" Task in TDS.cs, add this **using** statement:

```
using Working_Code;
```

At first, this statement will be grayed out in the VS editor, since the namespace is not being used yet.

Next we'll add some working code to the **Working_Code** namespace.

5.3.6.1.4 CUSTOMIZE THE TDS METHOD

Now we update the code in the new TDS method **InsertSymbolTest()** to invoke the to-be-defined method **InsertSymbol()** and create a stub for the new method.

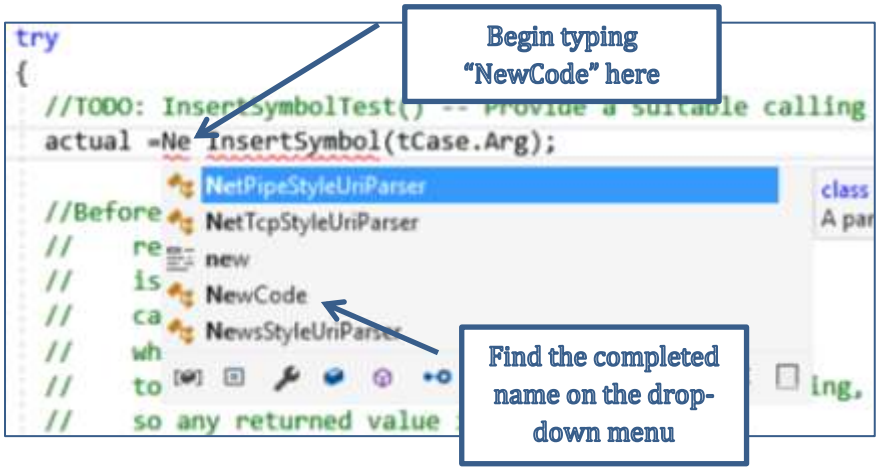
► To link this TDS method to the new method that it is to call, go to Task "**TODO: InsertSymbolTest() -- Provide a suitable calling expression**" and change the

```
    actual = InsertSymbol(tCase.Arg);
```

statement to invoke the new method, now to read

```
    actual = NewCode.InsertSymbol(tCase.Arg);
```


As before (in section 4.10.3, for example), **`InsertSymbol`** is appearing here for the first time (so VS has no clue as to what it might mean), so you'll have to type (or paste) the namespace name, "**`NewCode.`**", into this statement.



Even though the class name, "NewCode", is properly recognized now, the "InsertSymbol" name appearing here is not yet defined (we'll do that in section 5.3.7), so for now it will still have a wiggly red underline indicating that it's undefined.

► In the Task labeled "**`TODO: InsertSymbolTest() -- Use a suitable default value.`**", change the

```
var actual = 0;
```

statement to be

```
var actual = false;
```

We want the method to return a **`(bool)`** value indicating whether the requested changes were successful.

5.3.6.1.5 ADD THE TDS METHOD'S NAME TO TESTMETHODSTOBERUN

► In TDS.cs, into the literal string following the Task comment "**`TODO: TestMethodsToBeRun -- List all TDS test methods to be run.`**", enter the name of the TDS method that we have just now defined, "**`InsertSymbolTest()`**".

This name is case sensitive, but the parentheses are optional.

► Since we want to focus on this new TDS method, temporarily comment out or erase any other tests listed in **`TestMethodsToBeRun`**, as we did in section 4.8.2.5.

We are right now interested in exercising only **`InsertSymbol()`** and its new TDS method. Messages in the TDS test report when we do a TDS run will remind us that the other TDS methods are being skipped.

5.3.6.2 Generate a value for the input parameter

We need to be able to hand to the new method a simple version of the **`<Sentence>`** document, giving it enough detail to allow us to trace what the method does to it.

► Use Task "**`TODO: InsertSymbolTest() -- Define inputs and expected outputs.`**" to locate **`testValues[0]`**.

► In **testValues[0]**, select the existing **Arg** property and rename it to **Doc**,

In case VS does not support menu "Edit, Refactor, Rename" for this property, select all the code in **InsertSymbolTest()** and use menu "Edit, Find and Replace" to change the three occurrences of "**Arg**" in the "Selection" (not the "Current Document", and with the "Match whole word" option set) to "**Doc**".

The name "Arg" suggested an argument for a function, but "Doc" is much more suggestive of an XML document, which is the type of value we plan to pass to the new method.

► Give **testValues[0].Doc** the following value (instead of "**Doc = 3,**")

```
            Doc =  // Original value of <Sentence> document,
                   //   to which elements are to be added
                new XDocument(
                  new XElement("Sentence",
                    new XElement("Symbol",
                      new XAttribute("sentence", "true"),
                      new XElement("Sense", "W.Sentence"),
                      new XElement("Description", "Full Sentence"),
                      new XElement("Symbol",
                        new XAttribute("sentence", "false"),
                        new XElement("String", "BASKET"),
                        new XElement("Sense", "W.Noun"),
                        new XElement("Description", "Container")
                      )
                    )
                  )
                ),
```

► VS warns you that **XDocument** is undefined; right-click on it, choose the "Quick Actions" pop-up menu item (or hover over it and open the drop-down list), and choose the "**using System.Xml.Linq;**" item to add the needed statement to the **using** list.

5.3.7 Begin coding the method

► Generate the method stub based on the call in the "**actual =**" statement.

In Task "**TODO: InsertSymbolTest() -- Provide a suitable calling expression**", in the "**actual =**" statement, move the mouse cursor to the (currently underlined) **InsertSymbol** identifier, click on the light-bulb flag ("Quick Actions") appearing nearby, and click on "**Generate method 'NewCode.InsertSymbol'** ". (The wiggly underline should disappear.)

VS generates a basic version of the new method. We will need to add parameters later, to account for the inserted **<Symbol>**s and the returned summary, but we can begin with passing our example **<Sentence>** document to the method and having it validated. **tCase.Doc** contains the **<Sentence>** document, and the returned value is to be **true** iff no errors are encountered.

The generated stub should appear at the end of **NewCode{}**; move it if you wish, or leave it where it is. (I usually try to maintain alphabetical order.) To navigate to its definition, click on its name, then press <F12>.

The new method should look like this:

```
    public static bool InsertSymbol(XDocument doc)
    {
      throw new NotImplementedException();
```



```
    }
```

► To keep track of braces, I would add a comment on the method definition's closing brace to make it look like this:

```
    }  // end: InsertSymbol()
```

► (optional) Add an XML comment just before the definition of **`InsertSymbol()`** with something like

```
    /// <summary>
    /// Modify a &lt;Sentence&gt;
    /// </summary>
    /// <param name="doc">Document to be modified</param>
    /// <returns>True iff result is valid</returns>
```

We'll replace this later, in section 5.3.11, so for now there's no need to change the code to add it. The reason I suggest it here is that I think it's a good habit to always add XML comments describing new code, and it's easiest to remember at the same time that the code is created.

► To allow the name **`XDocument`** to be used without qualification in file Class1.cs, since we expect to use it several times, add the following statement near the beginning of Class1.cs if it's not already present:

```
using System.Xml.Linq;
```

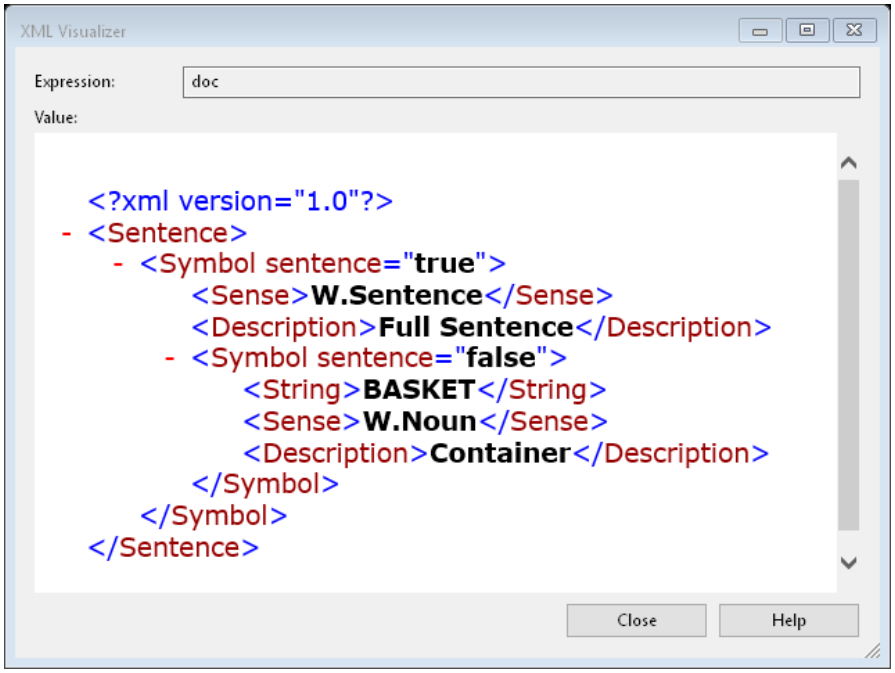
5.3.8 Do a smoke test on the new TDS method

► In file Program.cs, in **`Working_Code.NewCode.InsertSymbol()`**,place a breakpoint on its **`throw`** statement.

► In Solution Explorer, set TDS as the Startup Project (as in section 4.4.3.1), if necessary.

► Use VS menu "Debug, Start Debugging" (or <F5>) to run to the breakpoint.

Observe the value of parameter **`doc`**. It should look something like this, if we use VS's XML visualizer:



► Use VS menu "Debug, Continue" (or <F5>) to resume running.

If an "Exception User-Unhandled" pop-up message appears for a **`TDS.AssertFailedException`** exception, uncheck the "Break when this exception type is user-unhandled" box (in the "Exception Settings" menu) and press <F5> again to resume processing.

We should see the following among the lines of output in the Command Prompt window:

```
  The following test method returned a status of Failed:

  - InsertSymbolTest()
      Exception message:
Assert.IsTrue failed.
InsertSymbolTest(), test case 01 Sample test:
  The expected exception should start with " No exception was thrown".
  This unexpected exception was thrown:
    "The method or operation is not implemented."
```

► After viewing this output, close the window and remove the breakpoint.

5.3.9 Set up for validation

5.3.9.1 Copy the XSD file

We'll need an XSD schema that can specify correct syntax for our XML documents. This will let us do for an XML file what the C# compiler does for C# source code with XML comments — display IntelliSense information and AutoComplete text to make typing easier, and to generate error messages if any of the XML code is malformed. As with C# code, of course, syntax isn't the same as semantics, and it's quite possible for the XML code to be well formed but meaningless. Syntax checking can give us some protection from silly typing errors, though, and that's why we're using it here.

► In VS's Solution Explorer window, click on the startup project, "ConsoleApp1" (not the Solution with that name).

► Use VS menu "Project, New Folder" to create a file folder and name it "Data Files".

► Click on the "Data Files" folder. Use VS menu "Project, Add Existing Item", navigate to your Demo\TdsSource\ folder, and add a copy of file Sentence.xsd to the Solution.

This file extension is "xsd" instead of "xml". It uses XML syntax, so we can use VS's XML editor to edit it, but it's a specialized version of XML, with its own file extension.

The reason for placing this file in the "ConsoleApp1" Project is that it's a part of the working code, Any files used by the working code need to be available while it is running, and the TDS files are not intended to be used then, only during debugging and testing.

The new XSD file may be opened for editing, though there's no need to do that now unless you're just curious about what's in it.

5.3.9.2 Configure the XSD file

Unlike TDS.cs, this XSD file is copyrighted, but that should not be a major inconvenience for you, because it is provided only as part of this documentation. You need this XSD file to be able to build this example, but you won't need it when you use the TDS system on your own projects. (This copyright allows you to use it elsewhere if you wish, just like the rest of this *TDS User's Guide*, but with some restrictions, such as not being able to claim that you own the copyright on it.)

► In VS's Solution Explorer window, in the "ConsoleApp1" project, in its "Data Files" folder, right-click on the Sentence.xsd file, and choose "Properties".

► In VS's Properties window, which now shows the Data Files\Sentence.xsd file properties, change Build Action from "None" to "Content".

► Set Copy to Output Directory to "Copy always".

We'll need this XSD for use in validating some XML documents.

► In the Solution Explorer window, right-click on the ConsoleApp1 project and choose "Properties".

► In the ConsoleApp1 window, the Publish tab, in the "Install Mode and Settings" pane, click on "Application Files…" (button on the right side of the pane).

► In the "Application Files…" window, set the Publish Status of Data Files\Sentence.xsd to "Include (Auto)".

If it is not visible in the list, you might re-check its properties in Solution Explorer.

► Click "OK" to close the Application Files window.

► Close the Properties tab for the ConsoleApp1 Project.

5.3.9.3 Add the new schema to the active schema set

5.3.9.3.1 CREATE A FIELD TO CONTAIN THE SCHEMA

► In file Program.cs, in the **`Working_Code.NewCode{}`** class, perhaps immediately after the opening brace, add the following lines:

```
/// <summary>
/// Relative pathname to the file containing the
/// validation schema for &lt;Sentence&gt; documents.
/// <para>Current value is
/// "Data Files\Sentence.xsd".</para>
/// </summary>
/// <remarks>This path, instead of simply "Sentence.xsd",
/// allows NUnit, as well as Main(), to find the file.</remarks>
const string SchemaFile = @"Data Files\Sentence.xsd";

/// <summary>
/// Schemata for validating XML test documents.
/// </summary>
/// <remarks>These are set up in
/// the static <see cref="NewCode()"/> constructor,
/// based on the contents of
/// the file named by <see cref="SchemaFile"/>.
/// </remarks>
static XmlSchemaSet Schemata = new XmlSchemaSet();
```

5.3.9.3.2 ADD **`USING`**

Hmm... the type **`XmlSchemaSet`** seems to be undefined.

► Click on (or hover over) its name, click on the light-bulb flag that appears, and choose "**`using System.Xml.Schema;`**"

A suitable **`using`** statement magically appears at the beginning of the file, and the wiggly underline under **`XmlSchemaSet`** disappears.

5.3.9.3.3 POPULATE THE NEW FIELD

► If class **NewCode{}** does not yet have a static constructor, add one using the following code:

```
/// <summary>
/// Static constructor
/// </summary>
static NewCode()
{
  #region open file dialog
  var openFileDialog1 = new System.Windows.Forms.OpenFileDialog();
  openFileDialog1.FileName = SchemaFile;
  try
  {
    var myStream = openFileDialog1.OpenFile();
    Console.WriteLine("***** Schema file {0} is opened."
            , SchemaFile  // {0}
          );
    Schemata.Add("", System.Xml.XmlReader.Create(myStream));
    Console.WriteLine("***** New schema is available.");
  }
  catch (Exception e)
  {
    Console.WriteLine("***** " + e.Message);
    Console.ReadKey(true);  //Allow user to read the message
  }
  #endregion open file dialog
}  // end: static NewCode()
```

If this class already has a **static** constructor, then add the code in this **#region open file dialog** region to that constructor.

► In Solution Explorer, in the ConsoleApp1 project, right-click on References, click on "Add Reference…". In the "Assemblies, Framework" tab, add "System.Windows.Forms".

By the time the new **InsertSymbol()** method is called, this code will already have been run, and the schema for **<Sentence>** documents will be available for use.

5.3.9.4 Validate the document

► Add some code at the beginning of **`InsertSymbol()`**'s method body (before the **`throw`** statement) to validate the document:

```
      //This becomes False on validation errors.
      var isNoValidationError = true;

      #region Validate and compile doc
      doc.Validate(Schemata,
            (sender, e) =>
            {
              Console.WriteLine(
@"InsertSymbol @{0} at {1}:
  The following validation error occurred:
    ==> ""{2}"". "
                    , "doc1"  //{0}
                    , "initial validation"  //{1}
                    , e.Message  //{2}
                  );
              isNoValidationError = false;
            },
            true);
      if (!isNoValidationError) return false;
      #endregion Validate and compile doc
```

5.3.10 Delete the `throw`

► Replace the **`throw`** statement at the end with

```
      return true;
```

We expect to return values now and therefore no longer need the **`throw`**. We can always return **`true`** at the end because we plan, if we detect any error (in an earlier statement), to return **`false`** immediately and never reach that last statement.

5.3.11 Add XML comments

► (optional) Add some XML comments before **`InsertSymbol()`**, replacing the present ones, to reflect what we've done so far:

```
    /// <summary>
    /// Insert XElements (to be specified)
    /// into the &lt;Sentence&gt; in <paramref name="doc"/>,
    /// validating after each one,
    /// stopping on failure.
    /// Return True if all changes are valid.
    /// Write intermediate results to the Console.
    /// </summary>
    /// <param name="doc">Original XML document,
    /// into which the specified &lt;Symbol&gt;s
    /// will be inserted.</param>
    /// <returns>True iff no validation errors were detected
    /// after any of the insertions.</returns>
```

These comments are accurate for this stage of development, but we shall continue to update them; a fuller version, to replace these, will appear in section 5.3.15.3.1 below.

- Set a breakpoint on the

```
        return true;
```

statement and run (via "Debug, Start Debugging" or <F5>).

Reaching that statement demonstrates that the **`doc1.Validate()`** call was successful. (In case it wasn't obvious that anything interesting happened here, we'll demonstrate some failures soon.)

- Stop debugging (use <shift><F5>).

Leave active the breakpoint on the **`return`** statement, while we make some major changes to this method.

5.3.12 Calculate the summary string

5.3.12.1 Return a summary result

We intended to summarize some of the contents of the transformed sentence. We can do the calculation and return the result in a new parameter; let's call the parameter **`wordlist`**. We don't have to be very formal in specifying what the result should look like; we'll just give an example in the comments. For now, we'll assume that the calculation will not change and that we can specify the processing directly in code. With sufficiently clear code and comments, it should be easy for a future developer to make the needed editing changes, should the requirements change.

5.3.12.2 Add a parameter

- Change the signature of the **`InsertSymbol()`** method to include parameter **`out string wordList`** following **`doc.`**

The method header might now look like this:

```
    public static bool InsertSymbol(XDocument doc
          , out string wordList
      )
```

5.3.12.3 Edit XML comments

- To the XML comments, add, following the **`<param name="doc">...</param>`** element, the following element:

```
    /// <param name="wordList">
    /// <para>Comma-separated list
    /// of nouns and verbs, nouns first.
    /// </para><para>Example: "water (Noun),
    /// plant (Noun), carry (Verb)"</para>
    /// </param>
```

This will keep our documentation current with the new method signature.

5.3.12.4 Specify default

- Give the parameter a default value. Following

```
    var isNoValidationError = true;
```

insert the following lines:

```
      //Return this value if the unmodified document is invalid
      wordList = "(No list -- invalid <Sentence>)";
```


5.3.12.5 Calculate summary value

- Before the

```
    return true;
```

statement at the end of **`InsertSymbol()`**, insert the following lines to calculate the value of the summarizing string:

```
      #region Calculate wordList
      //Return a comma-separated list of selected words,
      //  sorted by part of speech.
      //The first 2 characters of each <Sense> value
      //  (the "W." part) are omitted.
      //Example: "water (Noun), plant (Noun), carry (Verb)"
      wordList = String.Concat(
            from node
            in doc.XPathSelectElements(
              "//Symbol[Sense='W.Noun' or Sense='W.Verb']")
            let partOfSpeech =
              node.Element("Sense").Value.Substring(2)
            orderby partOfSpeech
            select String.Format(", {0} ({1})"
              , node.Element("String").Value.ToLower() //{0}
              , partOfSpeech //{1}
            )
          );

      if (wordList.Length < 2)
        wordList = "(No nouns or verbs were found.)";
      else
        wordList = wordList.Substring(2);

      Console.WriteLine("{0}: {1}"
              , "Nouns & verbs in this sentence" //{0}
              , wordList //{1}
            );
      #endregion Calculate wordList
```

5.3.12.6 Add references

We notice that **`XPathSelectElements()`** is undefined.

- Hover the mouse pointer over its name and in the drop-down list choose "using System.Xml.XPath;" .

5.3.13 Update the call to `InsertSymbol()`.

- In Task "**`TODO: InsertSymbolTest() -- Use a suitable default value.`**", after

```
          var actual = false;
```

add local variable **`wordlist`**, as in

```
    //Summary of contents of updated <Sentence>
    var wordList = "";
```

- Change the "**`actual =`**" statement to add the **`wordList`** parameter:

```
    actual = NewCode.InsertSymbol(tCase.Doc, out wordList);
```

5.3.14 Check the revised code

- Run the program in Debug configuration. At the breakpoint on the

```
    return true;
```

statement, look in the Locals window at the about-to-be-returned value of **`wordlist`**.

Its value should look like this:

```
basket (Noun)
```

- Stop debugging (<shift><F5>).

5.3.15 Add a parameter specifying editing

5.3.15.1 Determine how to specify changes

At this point, our method demonstrably does most of what we asked it to do in section 5.3.2 above, "Statement of purpose", except for adding words and validating the results.

Since, for now, we have no constraints in our specification on how to do this, we can exercise our own judgment in what to do, bearing in mind that we may be overruled later and would have to redo some of this. (A conversation with the customer might be in order, but often the customer won't care about technical details, being more interested in how soon a working version will be ready to use.)

In this grammar, the terminal **`<Symbol>`**s that we want to allow the method to add contain only three components that we care about:

- a terminal string, the **`<String>`** value
- a part of speech, the **`<Sense>`** value
- comments, in the **`<Description>`**

We could pass the values of these three components to **`InsertSymbol()`** in the form of three **`String`**–valued parameters, or as three properties of a single object, but after a bit of pondering we decide on using one parameter containing a delimited **`string`**. (Such pondering might involve writing some code to compare the choices for legibility and ease of maintenance, and maybe checking with the customer.)

In addition to the value of the **`<Symbol>`** to be inserted, we need to be able to specify where to put it, and we decide to specify as well a subset of the **`<Sentence>`** to be validated, so that we don't spend time re-validating the unchanged parts of the XML document. We also specify a comment describing the change, to be displayed on the Console as part of a progress report.

All of these values can be specified as **`String`** values, so we can use a single **`String[]`** array to handle all of the remaining inputs. If we needed to include some non-**`String`** values, we would probably instead define a type (maybe a **`struct`** or a **`System.Tuple`**) with suitable properties.

5.3.15.2 Add another parameter

Since we actually want to specify multiple changes on each call to the method, what we will pass will be an array of these string arrays, one element specifying each of the XElements (representing words in a sentence) that we want to insert. This will allow us to do multiple insertions, but to keep the testing simple at first, we'll start by passing an array that contains only one element.

Note that our choice of an array of arrays of strings makes it a bit difficult to give suggestive names to the array elements (something we could do if we used an object with named properties), and we can't apply XML

comments to these, so we also won't get any IntelliSense help with the element values. However, in favor of our decision to use a string array, it is easy to define, so for now we'll go with this quick and easy definition.

► Add to the parameter list in the **`InsertSymbol()`** definition, between **`doc`** and **`wordList`** as the new second parameter,

```
            , string[][] editingParams
```

5.3.15.3 Update XML comments

Having changed the design (again), we need to update the XML comments to match.

► Update the XML comments on **`InsertSymbol()`** to give more details about what the method is doing now, and to account for the new parameter, specifying what is expected to be in each array element.

In real life, I would just make the needed editing changes to the XML comments. For this exercise, it's simpler to replace all the existing XML comments for this method with the following:

5.3.15.3.1 XML COMMENT CODE

```
/// <summary>
/// Insert the XElements specified by <paramref name="editingParams"/>
/// into <paramref name="doc1"/>, validating after each one,
/// stopping on failure.
/// Return True if all changes are valid,
/// and return a summary of some of the elements.
/// Write intermediate results to the Console.
/// </summary>
/// <remarks>XML exceptions due, for example, to malformed parameters
/// are passed on to the caller.</remarks>
/// <param name="doc1">Original XML document,
/// into which the &lt;Symbol&gt;s specified in
/// <paramref name="editingParams"/> will be inserted.</param>
/// <param name="editingParams"><para>
/// Specification of XElements
/// to be inserted and validated in sequence.
/// In each element of this array,
/// </para><para>[][0] = context, a description of
/// the type of change to be made.
/// Example: "before editing"
/// </para><para>[][1] = label, XPath specification
/// of area to be validated.  Example: "Sentence"
/// </para><para>[][2] = insertionPoint, Xpath identifying the XElement
/// following which the new XElement is to be inserted.
/// Example: "Sentence/Symbol"
/// </para><para>[][3] = insertedElement, components of the
/// &lt;Symbol&gt; to be inserted, separated by '|' characters.
/// </para><para>Example: "GOLD|Noun|Collectible treasure"
/// </para><para>[0] = &lt;String&gt; value, e.g. "GOLD"
/// </para><para>[1] = &lt;Sense&gt; value, e.g. "Noun"
/// </para><para>[2] = &lt;Description&gt; value,
/// e.g. "Collectible treasure"
/// </para></param>
/// <param name="wordList"><para>Comma-separated list of
/// nouns and verbs in the &lt;Sentence&gt;, nouns first.
/// </para><para>Example: "water (Noun),
/// plant (Noun), carry (Verb)"</para>
/// </param>
/// <returns>True iff no validation errors were detected
/// after any of the insertions.</returns>
/// <exception cref="ArgumentException">The values of tags for the
/// &lt;Symbol&gt; must be specified in the format
/// <para>"string_value|sense_value|description_value".</para></exception>
```

With these added/updated comments, we have specified more precisely what we expect the parameters to contain, and what the method will do (such as raising an exception) if it encounters malformed values.

5.3.15.3.2 VIEWING RESULTS IN VS'S OBJECT BROWSER

The Object Browser (reachable via VS menu "View, Object Browser") may be used for documentation and navigation, and for proofreading XML comments, as illustrated in the following sections.

5.3.15.3.2.1 DOCUMENTATION

Having placed these revised XML comments (from section 5.3.15.3.1 above) into the code, selecting **`InsertSymbol()`** in the Object Browser displays the following (and more legibly formatted than in the C# code) version of the comments:

public static **bool** **InsertSymbol**(**System.Xml.Linq.XDocument** *doc*, **string[][]** *editingParams*, out **string** *wordList*)
Member of **NewCodeNamespace.NewCode**

Summary:
Insert the XElements specified by editingParams into doc1, validating after each one, stopping on failure. Return True if all changes are valid, and return a summary of some of the elements. Write intermediate results to the Console.

Parameters:
doc1: Original XML document, into which the <Symbol>s specified in editingParams will be inserted.
editingParams: Specification of XElements to be inserted and validated in sequence. In each element of this array,

[][0] = context, a description of the type of change to be made. Example: "before editing"

[][1] = label, XPath specification of area to be validated. Example: "Sentence"

[][2] = insertionPoint, Xpath identifying the XElement following which the new XElement is to be inserted. Example: "Sentence/Symbol"

[][3] = insertedElement, components of the <Symbol> to be inserted, separated by '|' characters.

Example: "GOLD|Noun|Collectible treasure"

[0] = <String> value, e.g. "GOLD"

[1] = <Sense> value, e.g. "Noun"
s
[2] = <Description> value, e.g. "Collectible treasure"
wordList: Comma-separated list of nouns and verbs in the <Sentence>, nouns first.

Example: "water (Noun), plant (Noun), carry (Verb)"

Returns:
True iff no validation errors were detected after any of the insertions.

Remarks:
XML exceptions due, for example, to malformed parameters are passed on to the caller.

Exceptions:
System.ArgumentException: The values of tags for the <Symbol> must be specified in the format

"string_value|sense_value|description_value".

If the XML comments are malformed in some way (such as containing an unmatched "<" character), this entire displayed text is abbreviated to

public static **bool** **InsertSymbol**(**System.Xml.Linq.XDocument** *doc*, **string[][]** *editingParams*, out **string** *wordList*)
Member of **NewCodeNamespace.NewCode**

Sorry, this is not very informative as to what might be wrong with the XML comments. Please see section 4.14.9.2 for suggestions on debugging malformed ones.

5.3.15.3.2.2 NAVIGATING USING VS'S OBJECT BROWSER

The names of types (left panel of Object Browser) and type members (upper right panel of Object Browser) are hyperlinks; double-click on one to be transported to its definition in the code. (The Class View window supports this, too, but without displaying the XML comments.)

The underlined names in the lower-right panel, where the XML comment contents are displayed, are links, too, but not to the code — they link to types in the left panel.

5.3.15.3.2.3 PROOFREADING THE XML COMMENTS

If I suspect that some of my XML comments are unusable, for example due to a stray "**<**" character in some copied text, I can select the top member listed in the upper-right panel and use arrow keys to run through the list to see if any XML comment text is missing.

5.3.15.4 Update the call in `InsertSymbolTest()`

To keep things simple, we can begin by assuming that the value of the new parameter **`editingParams`** specifies only one change, so at first we will pass an array that contains only one element.

▶ In Task "**`TODO: InsertSymbolTest() -- Define inputs and expected outputs.`**", in **`testValues[0]`**, immediately before the line containing

```
    ExceptionExp = DefaultExceptionMessage,  // Expected  exception
```

, insert the following expression:

```
    EditingParams = new[]{
        new [] {"after first insertion", "Sentence/Symbol",
          "Sentence/Symbol/Symbol", "DIAMOND|Noun|Treasure"},
      }, // Elements to be inserted
```

Since we are keeping this and the values of the other inputs close to each other in the code, we can (somewhat) easily keep them consistent with each other as we exercise the new method.

What this requests is for **`InsertSymbol()`** to insert a new **`<Symbol>`**, "DIAMOND", following the first 2nd-level **`<Symbol>`**, "BASKET", that appears in the **`<Sentence>`**, and validate the result starting at the first top-level **`<Symbol>`**. (See section 5.3.15.8 below for an XML Viewer view of the same expected result.)

For example, even though we provide for the method's raising an exception if it receives unusable inputs, right now we don't want those — we'd prefer to have at least one set that the method can digest and use to return the desired result. We can handle the weirder cases soon, using later **`testValues[]`** elements.

▶ In Task "**`TODO: InsertSymbolTest() -- Provide a suitable calling expression`**", update the call to **`InsertSymbol()`** to add a value for the new parameter:

```
            actual = NewCode.InsertSymbol(tCase.Doc
                  , tCase.EditingParams
                  , out wordList);
```

As I've done in some other places in the code, I've put these commas at the beginnings of lines of code to make insertions, reorderings, or deletions slightly easier to do.

5.3.15.5 Calculate a returnable value

▶ In file Program.cs, in method **`Working_Code.NewCode.InsertSymbol()`**, before the line containing

```
      #region Calculate wordList
```


, insert the following lines to perform the editing specified by the new parameter, **editingParams** (but don't try running the program using them until you have done the step following, that adds the closing brace).

```
      foreach (var paramSet in editingParams)
      {

        #region Perform specified editing and check validity
        //paramSet[0] = context
        //paramSet[1] = area to be validated
        //paramSet[2] = XPath to insertion point
        //paramSet[3] = components of the inserted <Symbol>
        var symbolSpecs = paramSet[3].Split(
              new[] { '|' }, StringSplitOptions.RemoveEmptyEntries);
        if (symbolSpecs.Count() != 3)
        {
          var message1 = string.Format(
  @"Values of tags for the <Symbol> must be specified in the format
  ""string_value{0}sense_value{0}description_value""."
                  , '|'  //{{0}
                );
          throw new ArgumentException(message1);
        }
        var insertedXelement = XElement.Parse(
              String.Format(@"
<Symbol sentence=""false"">
  <String>{0}</String>
  <Sense>W.{1}</Sense>
  <Description>{2}</Description>
</Symbol>
"
                  , symbolSpecs[0]  //{0}
                  , symbolSpecs[1]  //{1}
                  , symbolSpecs[2]  //{2}
                ));

        if (paramSet[2].Substring(0, 1) != "(")
          try
          {
            doc.XPathSelectElement(paramSet[2])
                  .AddAfterSelf(insertedXelement);
          }
          catch (Exception e)
          {
/*
```

```
*/
            Console.WriteLine(
@"InsertSymbol @{0} at {1}:
  The following validation error occurred:
    ==> ""{2}"". "
                , paramSet[1]  //{0}
                , paramSet[2]  //{1}
                , e.Message  //{2}
              );
            wordList = "(none)";
            return false;
          }

        #region XformValidate
        //This becomes True on validation errors.
        var hasValidationError = false;

        #region valHandler(sender, e)
        ValidationEventHandler valHandler = (sender, e) =>
        {
          Console.WriteLine(
@"InsertSymbol @{0} at {1}:
  The following validation error occurred:
    ==> ""{2}"". "
              , paramSet[1]  //{0}
              , paramSet[2]  //{1}
              , e.Message  //{2}
            );
          hasValidationError = true;
        };
        #endregion valHandler(sender, e)

        Console.WriteLine("Validating {0} {1}..."
                , paramSet[1]  //{0}
                , paramSet[0]  //{1}
              );

        var element = doc.XPathSelectElement(paramSet[1]);
        element.Validate(element.GetSchemaInfo().SchemaElement,
              Schemata, valHandler, true);

        Console.WriteLine(@"Area ""{0}"" {1}."
                , paramSet[1]  //{0}
                , hasValidationError
                  ? "is not valid" : "is valid"  //{1}
              );

        #endregion XformValidate

        if (hasValidationError)
        {
          wordList = "(none)";
          return false;
        }
        #endregion Perform specified editing and check validity
```

5.3.15.6 Close the loop

- Before the line at the end of **`InsertSymbol()`** containing

```
    return true;
```

, insert the following line to close the **`foreach()`** loop:

```
    }  // end:foreach (var paramSet ...
```

Only the brace, "**`}`**", is significant here, but I included the comment to help match this brace with the one opening the loop. Collapsing the outermost "**`#region`**" code may help to make the structure of the code easier to view.

5.3.15.7 Comments on the process

What this code does is to split out the values to be given to the three tags in the new **`<Symbol>`**, throw an exception if there are not exactly three of them, construct an **`XElement`** containing them, insert the new **`XElement`** if a place is specified for it, and validate the part of the document containing the insertion.

As you might guess, I didn't write all this in one breath, and what you see here is (close to) the final version. During the process, with the help of **`InsertSymbolTest()`**, and being able to see, step by step, that the intermediate values were being calculated properly based on the original input, helped assure me that the later steps had the inputs they needed to be able to proceed. I also made minor changes to the inputs (located in **`testValues[0]`**), such as changing the separator in

```
testValues[0].EditingParams[0][3]
```

from '/', which originally[95] seemed suitable, to '|' to make the values easier to read. At that stage, the change was easy to make, as only three locations were involved: **`testValues[0]`**, the expression in **`InsertSymbol()`** that referred to it, and the XML comment in **`InsertSymbol()`** that described its **`editingParams`** parameter. Of course, if any other persons had known of my earlier convention, I would have needed to notify them of the proposed change and gotten their agreement to it, so it saved time to notice the need for this change early.

5.3.15.8 Check results

- Run to the breakpoint on the method's last statement,

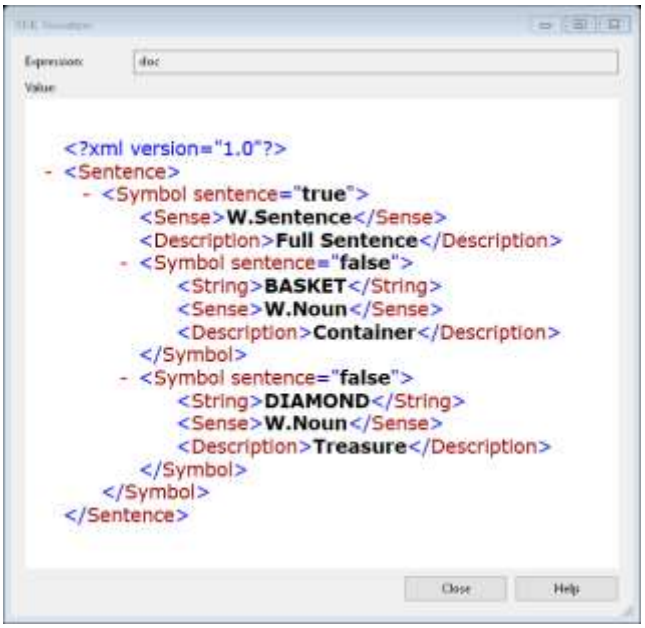
```
        return true;
```

At the breakpoint, using VS's Locals window, we can see that the value of **`wordlist`** that is about to be returned is

```
basket (Noun), diamond (Noun)
```

and the value of **`doc`** has now, as we expected, been changed to include a new **`<Symbol>`**. In the XML Visualizer it now looks something like this:

[95] The earlier version was "**`DIAMOND/Noun/Treasure`**".



This looks good — the "DIAMOND" **<Symbol>** has been inserted following the first 2nd-level **<Symbol>** in the <Sentence>, as we requested.

► Stop debugging (use <shift><F5>).

5.3.16 Add specifications for insertions of additional <Symbol>s

5.3.16.1 Specify multiple <Symbol>s

Adding one **<Symbol>** to the **<Sentence>** seems to have been successful; we are apparently ready to handle adding multiple **<Symbol>**s. We can do this by specifying a more complex value for **EditingParams**.

► In Task "**TODO: InsertSymbolTest() -- Define inputs and expected outputs.**", to **testValues[0].EditingParams**, add some new elements, making its value become

```
            EditingParams = new[]{
                new [] {"before adding anything", "Sentence/Symbol",
                  "(beginning)", "a|b|c"},
                new [] {"after first insertion", "Sentence/Symbol",
                  "Sentence/Symbol/Symbol", "DIAMOND|Noun|Treasure"},
                new [] {"after second insertion", "Sentence",
                  "Sentence/Symbol",
                  "Carry|Verb|Take the named object with you"},
              }, // Elements to be inserted
```

We are entering an element before the original one and one following it. The first of these three doesn't actually do any editing, but it does prompt **InsertSymbol()** to validate the original document, before changing it, as a safeguard against mistakenly specifying a malformed original.

5.3.16.2 Run the program using the new value

► In Task "**TODO: InsertSymbolTest() -- Provide a suitable calling expression**", set a breakpoint on the "**actual =** ..." statement calling **NewCode.InsertSymbol().**

► Run to this breakpoint (the one we just now set in **InsertSymbolTest()**).

We can examine the inputs here before continuing, if we wish.

- Step over the **`actual =`** ... statement (via <F10>).

- At the breakpoint on the **`return true;`** statement in **`InsertSymbol()`**, clear the breakpoint, then (perhaps using VS menu "Debug, Step Out") return to the calling statement.

- Step over the **`actual =`** ... statement. (Yes, again.)

Having called and returned from **`InsertSymbol()`**, we can examine the results. Looking at the messages in the Console window shows us, for example, that the edited document apparently remained valid, according to our XSD file, after each change.

5.3.16.3 Be aware of permanent changes

Since the **`<Sentence>`** that we are modifying is passed in the call-by-reference parameter **`doc`**, changes that we make to it in **`InsertSymbol()`** become permanent changes in the document passed in **`doc`**, even if errors occur that cause this method to return control before completing its instructions.

You may not always want this to happen, and to assure atomicity (= being sure that an operation is never only partly complete) in your transactions, you may need to make sure that your new function member can either correctly complete all of the requested operations before it commits any of them, or else that it performs a rollback to the original state before returning an error message.

We could have done that here by copying the original document from **`testValues[0].Doc`** and operating on the copy, for example changing the lines containing

```
try
{
  //TODO: InsertSymbolTest() -- Provide a suitable calling expression
  actual = NewCode.InsertSymbol(tCase.Doc,
        tCase.EditingParams, out wordList);
```

to code that looks like this:

```
//Document copy that may be modified
var docCopy = new XDocument(tCase.Doc);

try
{
  //TODO: InsertSymbolTest() -- Provide a suitable calling expression
  actual = NewCode.InsertSymbol(docCopy,
        tCase.EditingParams, out wordList);
```

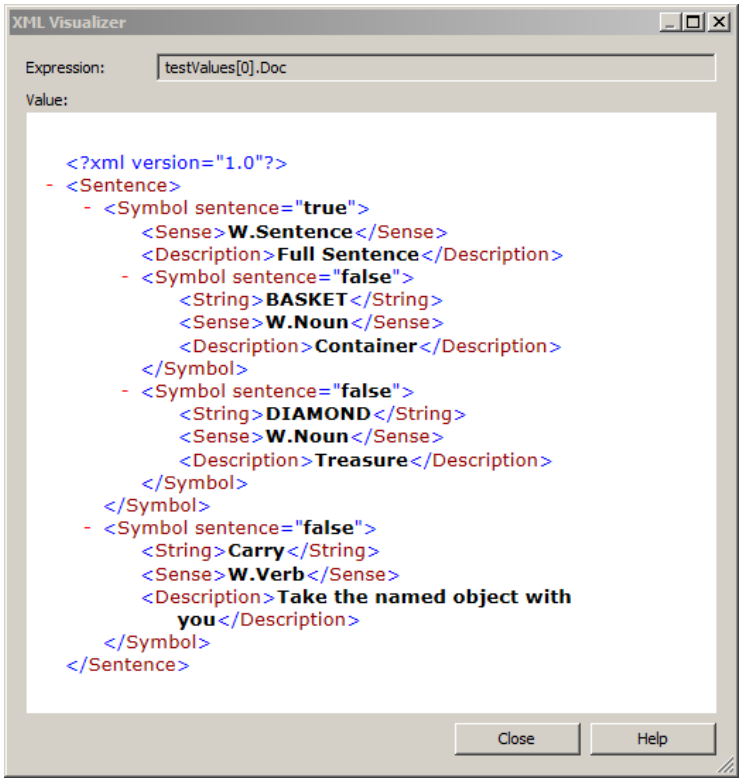
Altering the copy, as we intend to do, would leave the XDocument referred to in **`testValues[0].Doc`** untouched, and we shall do this before long, in section 5.3.17.4.1.3 below.

In case of error, the caller is notified, so that it can discard or undo the results. Each call from the TDS method, **`InsertSymbolTest()`**, would include a fresh copy of the document, so there would be no permanent damage if the document is partially changed — we could simply discard the changed copy. If copying is impractical (for example, if the object is a relational database), we would need to ascertain whether the transaction could be completed without error before deciding to commit it or roll it back. Specifically how to do that is a topic for another day.

In the present example, we assume that there is no need to take such care, since we shall use each **`testValues[]`** element only once in this TDS method. If there were a possible need to reuse any of the objects referred to by the **`testValues[]`** properties, a comment warning of that possibility would definitely be appropriate.

5.3.16.4 Compare values

► While paused, after stepping over the "actual =" statement, observe the values, in VS's Locals window, of the returned variables. **`actual`** should be **`true`**, **`wordList`** should contain "**`basket (Noun), diamond (Noun), carry (Verb)`**", and **`testValues[0].Doc`** should now contain added **`<Symbol> XElement`**s



containing "DIAMOND" and "Carry" (viewed using the XML Visualizer):

If these variables do not contain the expected values, check the Console output — there may be messages there identifying what might have gone wrong.

If you did not want your **`testValues[0]`** to have its contents changed by a test, then your code should reflect your intentions.

► Stop debugging (use <shift><F5>).

5.3.17 Begin automatic testing

5.3.17.1 Specify expected values

By now, we know that **`InsertSymbol()`** is working well enough to return usable values at least sometimes, and we are ready to have the TDS method compare these returned values with what we expect them to be. Like the inputs that we have been using, those expected outputs are (or will be) in **`testValues[]`**. You may

navigate to **`testValues[0]`** using Task "**`TODO: InsertSymbolTest() -- Define inputs and expected outputs.`**".

- In **`InsertSymbolTest()`**, in **`testValues[0]`**, delete the line containing

```
    ValueExp = 4,  // Expected returned value
```

and insert the following lines in its place:

```
    ValidXmlExp = true,  //True iff we expect the result to parse
    WordListExp = "basket (Noun), diamond (Noun), carry (Verb)",
        // Expected returned value
```

The included comments in these lines are intended to describe the properties, not these specific values.

5.3.17.2 Check returned values

- In the Task "**`TODO: InsertSymbolTest() -- Provide suitable non-exception tests here:`**", replace the **`Assert.AreEqual()`** statement with the following two statements, to compare the returned values with the expected values that we specified:

```
            Assert.AreEqual(
                  tCase.ValidXmlExp,
                  actual,
                  string.Format(@"
InsertSymbolTest(), ValidXmlExp test case {0}."
                    , tCase.Id  //{0}
                    )
                  );

            Assert.AreEqual(
                  tCase.WordListExp,
                  wordList,
                  string.Format(@"
InsertSymbolTest(), WordListExp test case {0}."
                    , tCase.Id  //{0}
                    )
                  );
```

Similarly to what we did in section 5.2.8.5.2, the message in each of these statements includes the name of the TDS test method, the test-case identifier, and a name identifying which **`Assert`** statement produced the message, to make it easy to track down the conditions producing the failure message. The format is slightly different, but the same information is present.

- In Task "**`TODO: InsertSymbolTest() -- Remove the Assert.Inconclusive()`**", delete the **`Assert.Inconclusive()`** statement at the end of the **`InsertSymbolTest()`** method body, and the preceding "TODO:" comment.

- Run the program.

- When you reach the breakpoint, remove the breakpoint and continue running (use <F5>).

At the

```
Press the <enter> key to finish . . .
```

message in the Console window, the TDS method should return a status of Passed:


```
Passed: 2  Failed: 0  Inconclusive: 0
```

(If not, check the earlier messages for evidence of why not. For example, one of my test runs Failed because of an extra space in the value of **WordListExp**.)

You may also see a message

```
The TestMethodsToBeRun list does not match the [TestMethod] methods.
```

reminding you that some other TDS methods are being skipped.

► Close the Console window.

5.3.17.3 Define another starting document

5.3.17.3.1 CREATE A LOCAL ARRAY TO CONTAIN THE TEST DOCUMENTS.

We are ready to try testing using a variety of inputs. Besides inserting various **<Symbol>**s into the given **<Sentence>**, we would like to experiment with different **<Sentence>**s, but we would rather not have to specify them multiple times in **testValues[]**. To avoid that, we shall create a local variable, **docs[]**, to contain some sample **<Sentence>** documents, and use a new property in **testValues[]** to select one of these.

The first one, **docs[0]**, will simply be a copy of **testValues[0].Doc** , which we specified in section 5.3.6.2. The value sent to **InsertSymbol()** using **testValues[0]** will not change.

5.3.17.3.1.1 COPY CODE FOR DOCS[0]

► In Task "**TODO: InsertSymbolTest() -- Define inputs and expected outputs.**", immediately following the

```
    #endregion testValues[]
```

line, insert the following code, mostly copied from the value of **testValues[0].Doc** but also defining the new **docs[]** array. The comment at the beginning refers to the not-yet-existing property **DocNum**, which we

shall define soon (in section 5.3.17.4.1) to select one of the array elements. The comment "**`docs[0] – short, valid document`**" at the end will help to distinguish this document from others in the list.

```
#region docs[], containing <Sentence> documents
// testValues[].DocNum specifies which is to be used.
var docs = new[] {

  new XDocument(
    new XElement("Sentence",
      new XElement("Symbol",
        new XAttribute("sentence", "true"),
        new XElement("Sense", "W.Sentence"),
        new XElement("Description", "Full Sentence"),
        new XElement("Symbol",
          new XAttribute("sentence", "false"),
          new XElement("String", "BASKET"),
          new XElement("Sense", "W.Noun"),
          new XElement("Description", "Container")
        )
      )
    )
  ),  // docs[0] – short, valid document

//TODO: InsertSymbolTest -- Add other test documents above here
};  // end:docs[]
#endregion docs[], containing <Sentence> documents
```

Expecting to add more test documents, we use a "TODO:" Task to mark the place. We could, of course, search for the "#endregion docs" string, which is unique in this file, but I think using a Task is more convenient.

5.3.17.3.2 USE VS'S XML EDITOR TO CONSTRUCT A NEW XML DOCUMENT

The second element of the new array, **`docs[1]`**, we shall develop with the help of the XML editor in VS, and it will be expressed in (what I consider to be) a more legible, compact format than that of the value of **`docs[0]`**, but this is a matter of style. Choose what works for you.

We intend to specify the second **`<Sentence>`** more concisely than we did the first one, omitting the explicit **`XElement()`**, etc., references. By calling **`XDocument.Parse()`**, using an XML literal, our C# code can look more similar to the values we will see as we examine the results.

OK, I used some non-standard terminology here. Unlike Visual Basic®, C# doesn't really have any special syntax for "XML literals" — what we'll use here will be an ordinary string literal value — but this string will be easy to format to look like XML code. Looking similar to the source document(s) that we might copy into our source code makes these copies easy to inspect for inconsistencies and thus can help us notice and correct misspellings and similar mistakes. The intent is to make the job of maintaining the TDS test method, and keeping it consistent with its corresponding function member, as easy as possible.

Incidentally, in case you're disappointed that C# doesn't syntax-check the XML code here, consider that its treating the value as a string allows us to perform all the normal string operations on it (including **`Regex{}`** operations or **`String.Format()`** constructions) that we wish to use, before treating it as XML. Also, even if we had compile-time syntax checking, the results would be incomplete because the compiler would have difficulty applying the validation schemata that we might intend to use. Anyway, it's immaterial in this program because we're syntax-checking all of our XML at run time, using the accompanying XSD file.

5.3.17.3.2.1 OPEN AN XML WORK FILE

The following steps illustrate a convenient way to populate an XML document, such as the expression to be used in **`docs[1]`**, using VS's XML editor. If you wish to skip the steps involved in building this example document, you may copy the completed code for **`docs[1]`** from section 5.3.17.3.3.2 below and continue from there.

► In VS's Solution Explorer window, right-click on the ConsoleApp1 project and select "Add, New Item".

► Click on "Data", then "XML File".

► Accept the default file name of "XMLFile1.xml" and click on Add.

The name is unimportant because we'll use it only for editing, then erase it.

An editing window for this new file will open, containing only the line

```
<?xml version="1.0" encoding="utf-8" ?>
```

► Use VS menu "XML, Schemas…" to bring up a dialog box listing current schemata. File "Sentence.xsd" should be among them (maybe near the top), since we are already using it.

If not, click on "Add…", navigate to the "ConsoleApp1\Data Files" folder and select Sentence.xsd .

► On the line for sentence.xsd, in the "Use" column, select "Use this schema" (check mark appears), then click "OK".

In the XMLFile1.xml editing window, IntelliSense auto-completion pop-ups should appear, to assist you in generating valid XML according to this schema. (If not, use VS menu "Tools, Options", select the "Text Editor, XML" tab, and choose the desired properties.) Also, hovering the mouse pointer over the tags that you have entered will display their comments in IntelliSense pop-ups.

5.3.17.3.2.2 BEGIN ADDING ELEMENTS

► Enter suitable tags. On line 2 enter "<" and press control-space (if necessary) to see a menu of choices.

► We'll begin with a comment; double-click on "**`!--`**", or press <enter> when it's highlighted, and type "**`Parsing tree for a Sentence`**"

► Press the <end> key, then the <return> key.

This should complete the comment and move the cursor to the beginning of the next line.

► Enter "<" and (if necessary) press <control><space>, to see a menu of choices.

► Click on "Sentence", or type "s", or use the arrow keys, to highlight the "Sentence" menu choice.

In a moment, an IntelliSense pop-up appears with a description of the **`<Sentence>`** element (copied from comments in the XSD), as seen here:

► Double-click on "Sentence", or press <enter> or <tab> when it's highlighted, and type "**>**", then <enter>, to close the tag.

The closing tag **</Sentence>** appears, but with an error flag (wiggly blue underline). The document should now look like this:

```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <!--Parsing tree for a Sentence-->
3  <Sentence>
4
5  </Sentence>
```

► Hover the cursor over **</Sentence>** to see what is wrong.

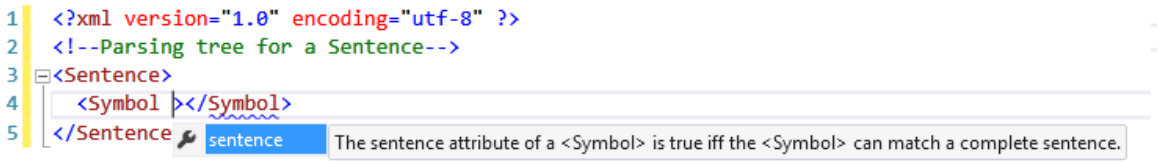
Apparently a **<Symbol>** element is needed; we see the message

```
The element 'Sentence' has incomplete content.  List of possible elements
expected: 'Symbol'.
```

► Enter a **<Symbol>**, as you did the **<Sentence>**.

► Inside the opening **<Symbol> tag,** immediately before the "**>**", type a space; a pop-up menu offering a "**sentence**" attribute appears.

An IntelliSense pop-up with a description of the "**sentence**" attribute also appears:



► Select "sentence"; that attribute is inserted, offering a list of choices. For this instance, give it a value of **"true"**.

5.3.17.3.2.3 ADD ELEMENTS VIA ANOTHER METHOD

A possibly easier way to enter XML elements is the following:

► Immediately after the **</Symbol>** tag, type "<S" to begin another **<Symbol>** element.

► Press <tab> to enter the name "Symbol".

We should now have this, with the cursor at the end of the word "Symbol":

```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <!--Parsing tree for a sentence-->
3  <Sentence>
4    <Symbol sentence="true"></Symbol><Symbol
5  </Sentence>
```

► Press <tab> again to generate a generic **<Symbol>** element.

Now we get a complete **<Symbol>** element, with default values assigned to the fields. Some fields will have to be filled in, but now the document looks like this:

```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <!--Parsing tree for a sentence-->
3  <Sentence>
4    <Symbol sentence="true"></Symbol>
5    <Symbol sentence="boolean">
6      <String>string</String>
7      <Sense>string</Sense>
8    </Symbol>
9  </Sentence>
```

5.3.17.3.2.4 ADD REMAINING ELEMENTS

Continue adding elements, attributes, and contents as desired to construct a syntacticaly correct example that will let you exercise the features of your function member.

For this example, we shall add the elements shown in section 5.3.17.3.2.5 below. You may copy the XML code from there into this XML document and examine it using the VS editor.

However, in your own XML projects, you will likely enter elements like these (short ones, anyway) from the keyboard. If you do that with this **<Sentence>** document, you may notice that the IntelliSense comments for the **<Symbol>** elements helpfully differ depending on their context — only the IntelliSense comments for the correct type of **<Symbol>** (terminal vs. non-terminal) are displayed.

5.3.17.3.2.5 ADD THE REMAINING ELEMENTS FOR THIS TEST CASE

- Erase the second **<Symbol>** (the one we just now added).

We should now have a comment and a **<Sentence>** element, and the **<Sentence>** should contain only an unfinished **<Symbol>** element.

► Within the unfinished **<Symbol>** element, insert the following six (counting the comment) XML elements:

```
    <Sense>W.VtWithObj</Sense>
    <Description>Transitive verb and object, with adverb.</Description>
    <Symbol sentence="true">
      <String>POUR</String>
      <Sense>W.Verb</Sense>
      <Description>Action with object</Description>
    </Symbol>
    <Symbol sentence="false">
      <String>WATER</String>
      <Sense>W.Noun</Sense>
      <Description>Pourable liquid</Description>
    </Symbol>
    <!-- The following <Symbol> could modify the noun,
          but not in this case -->
    <Symbol sentence="false">
      <Sense>W.Adverb</Sense>
      <Description>Modifier of a verb</Description>
      <Symbol sentence="false">
        <String>ONTO</String>
        <Sense>W.Prep</Sense>
        <Description>Preposition, introducing a modifier</Description>
      </Symbol>
      <Symbol sentence="false">
        <String>PLANT</String>
        <Sense>W.Noun</Sense>
        <Description>Movable potted plant</Description>
      </Symbol>
    </Symbol>
```

► Reformat the XML document (use VS menu "Edit, Advanced, Format Document"), to make it easier to read.

► Immediately before the closing **</Sentence>** tag (last line of the file), insert a second **<Symbol>** element, to look like this:

```
  <Symbol sentence="false">
    <Sense/>
    <Description/>
    <Symbol sentence="false">
      <String> .</String>
      <Sense>W.End</Sense>
    </Symbol>
  </Symbol>
```

5.3.17.3.3 MOVE THE NEW XML DOCUMENT INTO `Docs[]`.

Our test document is now complete and is valid according to our XSD (as indicated by the lack of wiggly underlines, or at least I hope you have none of those now), at least for the moment. Mistakes could yet creep into the XML text, perhaps the result of our later making what we think are minor changes, for example to improve the appearance or legibility of the XML. That is why I chose to start the first call to **`InsertSymbol()`** for a given document with an editing specification that does no editing but does validate the unchanged document; we could omit that do-nothing call on later references to the same document.

You could do further editing, creating a document that you could use to thoroughly exercise the **`InsertSymbol()`** method, but for now, we'll assume that this test document is in its final form. We'll next

reformat it for use in C# source code, then copy that C# code into **`InsertSymbolTest()`** as the new **`docs[1]`** test document.

5.3.17.3.3.1 REFORMAT IT FOR USE AS C# CODE

Before copying the XML code, we'll need to escape the quotation marks by duplicating all of them.

► In the XMLFile1.xml editing window in VS, replace each double quotation mark (`"`) with a pair of double quotation marks (**`""`**) (20 occurrences).

The intent is to create escaped quotation marks, which the @"…" string in C# will translate back to their original form. The text will, of course, no longer be a valid XML document, but we will have no further use for this XML file anyway, so no harm is done.

► In TDS.cs, immediately before the "**`TODO: InsertSymbolTest -- Add other test documents above here.`**" Task comment, insert the following expression as the new (and currently empty) **`docs[1]`**, including a descriptive comment:

```
            XDocument.Parse(
@"
"
            ),  // docs[1] -- long, valid document
```

► Copy the entire (modified) contents[96] of file XMLFile1.xml and paste them immediately following '`@"`', as the new **`docs[1]`** value.

Since the new code is between the unpaired double quotation marks, those identify the beginning and end of the inserted string, and the escaped quotation marks within the string are correctly interpreted.

[96] The first line, "`<?xml version="1.0" encoding="utf-8"?>`", may be omitted in these XDocuments.

5.3.17.3.3.2 FINISHED DOCS[1] CODE

The new **`docs[1]`** value should look like this:

```
        XDocument.Parse(
@"<!--Parsing tree for a sentence-->
<Sentence>
  <Symbol sentence=""true"">
    <Sense>W.VtWithObj</Sense>
    <Description>Transitive verb and object, with adverb.</Description>
    <Symbol sentence=""true"">
      <String>POUR</String>
      <Sense>W.Verb</Sense>
      <Description>Action with object</Description>
    </Symbol>
    <Symbol sentence=""false"">
      <String>WATER</String>
      <Sense>W.Noun</Sense>
      <Description>Pourable liquid</Description>
    </Symbol>
    <!-- The following <Symbol> could modify the noun,
          but not in this case -->
    <Symbol sentence=""false"">
      <Sense>W.Adverb</Sense>
      <Description>Modifier of a verb</Description>
      <Symbol sentence=""false"">
        <String>ONTO</String>
        <Sense>W.Prep</Sense>
        <Description>Preposition, introducing a modifier</Description>
      </Symbol>
      <Symbol sentence=""false"">
        <String>PLANT</String>
        <Sense>W.Noun</Sense>
        <Description>Movable potted plant</Description>
      </Symbol>
    </Symbol>
  </Symbol>
  <Symbol sentence=""false"">
    <Sense/>
    <Description/>
    <Symbol sentence=""false"">
      <String> .</String>
      <Sense>W.End</Sense>
    </Symbol>
  </Symbol>
</Sentence>"
        ),  // docs[1] -- long, valid document
```

5.3.17.3.3.3 NOTES ON THE DOCS[1] CODE

If you're looking at a color version of this document, you'll notice that all of the coloring identifying XML syntax is gone. The entire document, now in **`docs[1]`**, is correctly shown as a C# string literal value, and XML editing assistance (such as AutoComplete or IntelliSense) is no longer available for it. If we wish to do further extensive editing of the XML, we can copy the value back into a suitable editing window in VS and de-escape the quotation marks, replacing each '`""`' with '`"`'. We might also need to specify which XSD schema to use.

If we want to add other XML documents to this list for further testing, we can copy them from XML code appearing elsewhere into an empty XML doucment in VS or create and edit them using the XML editor, then escape the quotation marks and paste the results into C# source code, as we did here for **`Docs[1]`** . This usage is concise and legible, but the C# compiler can't help detect syntax errors in the XML code; those will show up only at run time. It was to help mistakes in this code show up as soon as possible that we had **`testValues[0].EditingParams[0]`** request no editing changes at all, calling **`InsertSymbol()`** only to validate the document in its original form, and we plan to do that with this new document as well.

5.3.17.3.3.4 ADD DOCS[2] AND DOCS[3] CODE

- Add the following code as additional test documents:

```
        XDocument.Parse(
@"<!-- Invalid &lt;Sentence&gt; -->
<Sentence>
  <Symbol sentence=""false"">
    <Sense/>
    <Description/>
  </Symbol>
</Sentence>"
      ),  // docs[2] -- invalid document

        XDocument.Parse(
@"<Sentence>
  <Symbol sentence=""true"">
    <Sense>W.Phrase</Sense>
    <Description>Disjointed word</Description>
    <Symbol sentence=""false"">
      <String>INTO</String>
      <Sense>W.Preposition</Sense>
      <Description>Use with ""PUT""</Description>
    </Symbol>
  </Symbol>
</Sentence>"
      ),  // docs[3] -- no noun nor verb
```

This will allow us to test that **`InsertSymbol()`** properly detects a faulty starting document, and that it correctly reports that a document does not contain any nouns or verbs (if that be true). These branches are rarely used, but the tests are needed to ensure that all of the code functions as intended, at least some of the time (less embarrassing than discovering later, for example, that some of the code is never reachable).

5.3.17.3.4 DELETE THE WORK FILE

Instead of deleting XMLFile1.xml now, you might choose to keep it active for a while so that you could do further editing, to create additional XML test cases. For this example we'll assume that we now have plenty of examples for our purposes.

- When you have no further use for the XML work file, XMLFile1.xml , delete it.

For example, in VS's Solution Explorer window, right-click on XMLFile1.xml, click on the "Delete" menu item, then click the "OK" button. The file's editing window should disappear as well.

We created XMLFile1.xml only to make editing the XML document easier, and its contents are now saved in our source code.

However, don't delete file Data Files\Sentence.xsd file in the same project. It will continue to be needed as long as we are using **`InsertSymbolTest()`**., even though it's part of the "ConsoleApp1" working-code Project. Depending on how development of the working code proceeds, the XSD and other auxiliary files may become a permanent part of the working code.

5.3.17.4 Update `testValues[]`

5.3.17.4.1 MODIFY THE `TESTVALUES[0]` PROPERTIES

5.3.17.4.1.1 ID PROPERTY VALUE

We can use the XML documents in **`docs[]`** from here on, instead of getting them from **`testValues[]`**. OK, actually you might have a need to do some of each, but for this example we'll just go with references to **`docs[]`**.

- In **`testValues[0]`** (at Task "**`TODO: InsertSymbolTest() -- Define inputs and expected outputs.`**"), change the value of the **`Id`** property to reflect what this test case is supposed to do:

```
    Id = "01 Short document, adding noun and verb",
```

5.3.17.4.1.2 REPLACE THE DOC PROPERTY

- Delete the **`Doc`** property and its comments from **`testValues[0]`**.

The first element of the new **`docs[]`** array was copied from **`testValues[0].Doc`**, so we won't need that property any longer.

With the XML document now removed from **`testValues[]`**, we need to identify the document to be used in each test case. We shall add a new property, **`DocNum`**, in the same location as the former "**`Doc =`** … " line, to allow us to specify which of the XML documents in **`docs[]`** is to be sent to **`InsertSymbol()`**.

For example, immediately before the

```
EditingParams = new[]{
```

line in **`testValues[0]`**, we could insert the following new lines (but don't do that yet; we're about to do something else):

```
            DocNum = 0,   //Index into docs[] member
                // containing a <Sentence> document,
                // to which XElements are to be added
                // Current members:
                //   0 = Short, valid document
                //   1 = Longer, valid document
                //   2 = Invalid document
                //   3 = Valid document with no noun nor verb
```

This may seem like a long list of comments for this simple property. However, since **`docs[]`** is a local variable, it is difficult to comment its elements in a useful way. The comments that we have put on the closing braces within **`docs[]`** do contain summaries, but they are buried among the XML code and are not easy to find. They are definitely not candidates for display via IntelliSense tags. Since we will normally access **`docs[]`** via the **`DocNum`** property in **`testValues[]`**, these comments seem like an easy-to-locate place to describe the documents. If we later add other elements to **`docs[]`**, we can list them as well in these comments, and thus have and maintain a short list here that identifies the test documents.

However, depending on how extensively you might use this type of reference, it might make sense to use an **enum** instead of an **int**; the **enum** definition would provide a way to add IntelliSense support to these references, and the **enum** members could have names that are more suggestive of their function than a "1" or "3" would be.

► Place the following definition immediately following the end of the definition of **InsertSymbolTest()**:

```
    /// <summary>
    /// Reference to element of docs[] collection
    /// of XML documents
    /// in <see cref="InsertSymbolTest"/>
    /// </summary>
    internal enum InsertSymbolTestDoc
    {
      /// <summary>
      /// Short, valid XML document
      /// </summary>
      ShortValid = 0,
      /// <summary>
      /// Long, valid XML document
      /// </summary>
      LongValid,
      /// <summary>
      /// Invalid document
      /// </summary>
      Invalid,
      /// <summary>
      /// Valid document, but missing noun and verb
      /// </summary>
      NoNounNorVerb
    }  // end: enum InsertSymbolTestDoc
```

Making it **internal** instead of **private** allows its XML comments to be visible in the Object Browser.

With this definition, we can use the **enum**'s XML comments to replace the in-line comments that we might have used with the **int**-valued references. Yes, it takes a few minutes to define **enum InsertSymbolTestDoc**, and (as with the comments we included on the "**DocNum = 0,**" line) its list of members will need to be updated whenever we add new XML documents to the **docs[]** collection, but doing so will let the references be more self-explanatory. (Also, chasing down even one bug due to using the wrong index value might easily take longer than defining this **enum** would.)

To illustrate the use of this **enum** in place of the **int** that we first suggested, we shall use the **enum** in this example.

► In **testValues[0]**, insert the definition of new property **DocNum** immediately before that of property **EditingParams** (where property **Doc** was previously defined), to look like this:

```
    DocNum = InsertSymbolTestDoc.ShortValid,   //Index to a docs[] member
```

(The comments that I previously suggested, such as

```
                //   0 = Short, valid document
```

, are no longer needed; the **enum** does a better job.)

Hovering the mouse pointer over "`InsertSymbolTestDoc`" in this line of code produces an IntelliSense message describing the general purpose of including the **`DocNum`** property in the test cases; it's a property-specific comment. Similarly, hovering the mouse pointer over "**`ShortValid`**" produces an IntelliSense message offering a brief description of the specific **`docs[]`** member identified by this reference; it's a value-specific reference. Please see section 5.2.9.6.3.2 for a description of how comments about a property might be treated differently from comments about a specific value appearing in a **`testValues[]`** element.

These comments include reminders of how the documents differ, since the value of the index (such as "0") is not as informative as a name like "ShortValid" (the **`enum`** member's name) or "Short, valid XML document" (on the IntellSense pop-up).

5.3.17.4.1.3 SEND A COPY, NOT THE ORIGINAL

For the reasons mentioned earlier, in section 5.3.16.3, we shall send only a copy of the original XML document to **`InsertSymbol()`** to be modified for our tests, rather than sending the original.

- In Task "**`TODO: InsertSymbolTest() -- Use a suitable default value.`**", following the

```
    var wordList = "";
```

statement, delete these lines (if you added them in section 5.3.16.3)

```
    //Document copy that may be modified
    var docCopy = new XDocument(tCase.Doc);
```

- Insert the following lines (even if you didn't include a **`var docCopy`** statement earlier):

```
            //Copy of the selected source <Sentence> document
            //   to which <Symbol> elements are to be added
            var docCopy = new XDocument(docs[(int)tCase.DocNum]);
```

If we were to define **`DocNum`** as an **`int`** instead of as an **`enum`**, then the statement would be similar, except that in this statement, the "**`(int)`**" cast would be unnecessary and therefore omitted.

- In Task "**`TODO: InsertSymbolTest() -- Provide a suitable calling expression`**", change the first parameter in the "**`actual =`**" method call from **`tCase.Doc`** to **`docCopy`**, if you did not do so in section 5.3.16.3.

Also, since **`InsertSymbol()`** generates some Console output that can look somewhat cluttered, we can insert a **`Console.WriteLine()`** statement to identify the test case that generates that output.

The statements should now look like this:

```
            Console.WriteLine(@"
Test case {0}"
                  , tCase.Id  //{0}
                );

            actual = NewCode.InsertSymbol(docCopy
                  , tCase.EditingParams
                  , out wordList);
```

Now that we intend to reuse the original documents, we shall pass only copies of them, not the originals, to **`InsertSymbol()`**. We shall continue to discard results when we have finished examining them, instead of attempting to roll back any changes made by **`InsertSymbol()`**.

5.3.17.4.1.4 CHECK RESULTS

- Run (using <F5>).

In the test report, **`InsertSymbolTest()`** should have a status of Passed. This test passed to **`InsertSymbol()`** the same inputs that we used in the previous test, including the short **`<Sentence>`** document.

- Close the Command Prompt window.

5.3.17.4.2 ADD A SECOND TEST CASE

This time, we shall use the second, longer starting **`<Sentence>`** document, that we are calling "**`LongValid`**". As before, the first edit will make no change, but instead will merely validate the unchanged document.

- Add a second test case, element **`testValues[1]`**, containing the following code:

```
        new {
          Id = "02 Long document, adding noun & verb",
          DocNum = InsertSymbolTestDoc.LongValid,
          EditingParams = new[]{
            new [] {"before adding anything", "Sentence/Symbol",
              "(beginning)", "a|b|c"},
            new[]{"after first insertion", "Sentence/Symbol",
              "Sentence/Symbol/Symbol", "DIAMOND|Noun|Treasure"},
            new [] {"after second insertion", "Sentence",
              "Sentence/Symbol[2]",
              "Carry|Verb|Take the named object with you"},
          },
          ExceptionExp = DefaultExceptionMessage,
          ValidXmlExp = true,
          WordListExp = "diamond (Noun), water (Noun),"
            + " plant (Noun), pour (Verb), carry (Verb)",
          },
```

If, having added this code, you notice that everything in your **`testValues[]`** array is suddenly full of wiggly red underlines, the problem may merely be that some property in **`testValues[0]`** is out of order, possibly **`DocNum`**. Moving it so that all the properties match should correct the problem.

5.3.17.4.3 EDIT TAGS IN TEST-CASE `ID` PROPERTIES

Since we may now use either of two starting **`<Sentence>`** documents, we decide to change the tags in the **`Id`** properties, prefixing each with "**`L`**" (long) or "**`S`**" (short) to identify which document is specified by the **`DocNum`** property. This will allow us, if we wish, to use a filter in **`testSelectionList`** that runs only the test cases that employ the selected one of the two documents.

Perhaps you think it's silly that we now have redundant specifications (both in the tag in **`Id`** and the value of **`DocNum`**) of which source document to use. If you consider the **`Id`** tags to be stable and dependable, you can put code into the **`InsertSymbolTestCase()`** constructor[97] to compute the value of the **`DocNum`** property based on the value of the tag, and remove **`DocNum`** from the parameter list. (Oh, yes, and also update the XML comments to match that change.) We won't overload them that way in this example, however, to avoid

[97] This constructor will be available after we convert the anonymous-type **`testValues[]`** elements to be of **`InsertSymbolTestCase`** type, as we shall do in section 5.3.17.5.

encumbering the **`Id`** tags in ways that could make them difficult to change later, for reasons mentioned in section 5.2.9.6.3.11 above.

How you identify test cases is your choice, but I suggest that if you come up with a really fancy design, it might be polite to include a comment describing how it works. Whoever follows you may appreciate it.

► Change the value of **`testSelectionList`**, near the beginning of the "**`#region testValues`**" region (just before Task "**`TODO: InsertSymbolTest() -- Define inputs and expected outputs.`**"), from "**`01`**" to "**`S01`**".

► Change the tag at the beginning of the first **`Id`** value from "**`01`**" to "**`S01`**".

► Add a comment to **`testValues[0]`** describing this new convention.

This could be something like 'Prefix "S" refers to the short XML document, prefix "L" refers to the longer one.'.

► Change the second tag, in **`testValues[1].Id`**, from "**`02`**" to "**`L01`**".

► To verify that we have made no major mistakes, run a test (using "Start Debugging" or <F5>).

The test should pass, but with the usual note that only **`InsertSymbolTest()`** and **`AllTestsAreToBeRunTest()`** were run.

► Close the command prompt window.

5.3.17.5 Change testValues[] to use a named type

5.3.17.5.1 WHAT NEEDS TO BE DONE

We want to add some other test sets, but, as we did above in section 5.2.9.6, if we wish to give them a named type we should do so now, so that we won't have to do extra work to revise more of them later.

5.3.17.5.2 REFORMAT **`TESTVALUES[]`** MEMBERS AS CONSTRUCTORS

We could copy and edit the example class definition from the TDS.cs template, but, as described in section 5.2.9.6, it is probably easier to generate a new one, as we shall do now. (You may instead skip ahead to section 5.3.17.5.3, which contains a copyable version with these steps completed.)

► Comment out the definition of **`testValues[1].`**

► Following the instructions in section 5.2.9.6.3.1, change the first line of **`testValues[0]`** to include the class name "**`InsertSymbolTestCase`**".

Thaat line should now look like this:

```
    new InsertSymbolTestCase {
```

► Generate a nested class from this initializer; change its accessibility from "**`private`**" to "**`internal`**".

► Add XML comments to the class, for example 'Specifications for a test case in <see cref="InsertSymbolTest"/>'.

In this example, the "<" and ">" are legitimate parts of the XML code and should not be escaped.

► Add XML comments to the properties in **`InsertSymbolTestCase{}`**.

Much or all of this material can be copied from **`testValues[0]`**. I would split the editing window to help with this. See section 5.2.9.6.3.2.

► Reformat **`TDS.Test.InsertSymbolTest.testValues[0]`** from an initializer into an instance-constructor call. See section 5.2.9.6.3.3.

In this example, there are no stray "=" signs, so they may be replaced using a "replace all within selection" operation. (There should be six occurrences.)

► Generate a corresponding instance constructor, as shown in section 5.2.9.6.3.4.

Although I usually give parameters names that begin with lower-case letters, in this case I left them in the upper-case form generated by VS, feeling that the effort needed to change them was not warranted. These are used in defining **`testValues[]`** elements and nowhere else, and there seems to be little chance of confusing them with the same-named properties, which are used in other contexts, such as in calculations.

Nevertheless, if you do wish to rename them, I suggest doing that, for each parameter, by selecting its name, using VS menu "Edit, Refactor, Rename" (or by pressing <F2>), changing the name, leaving "Include comments" and "Include strings" unchecked, and clicking on "Apply". Its occurrence in the **`<param>`** element of the XML comments will be updated as well, even though "Include comments" is not selected.

► Change the "**`set;`**" accessors on the properties to be "**`private set;`**".

► Add XML comments to the constructor.

Since all of the parameters directly reflect their same-named properties, just copy the properties' XML comments to these **`<param>`** elements.

► Add default values to the constructor's parameters.

Parameters **`Id`**, **`EditingParams`**, and **`WordListExp`** do not have values that are likely to be used often enough to be suitable as defaults, so they will be required. Since I think the parameter list is easier to read when each parameter is on a separate line, I shall display them this way in the example code below, but since doing this makes the code less concise, you may prefer a more compact format. Putting the comma at the beginning of each line makes it easy to add or remove default values.

For the other three parameters, I choose the following as their default values:

```
DocNum = InsertSymbolTestDoc.ShortValid
ValidXmlExp = true
ExceptionExp = DefaultExceptionMessage
```

These should be easy to remember and are likely to be used frequently. Move them to the end of the parameter list; they must (C# rule) follow all those without default values.

5.3.17.5.3 EXPECTED RESULTS

When we have completed these steps, the new class's definition should look something like this:

```
    /// <summary>
    /// Specifications for a test case
    /// in <see cref="InsertSymbolTest"/>
    /// </summary>
    internal class InsertSymbolTestCase
    {
      /// <summary>
      /// Test-case constructor
      /// </summary>
      /// <param name="Id">
      /// Test case identifier (required),
      ///  consisting of a unique 2- or 3-character tag, a space,
      ///  and a short description of the test case.
      ///  <para>Prefix "S" refers to the short XML document,
      ///  prefix "L" refers to the longer one.</para>
      /// </param>
      /// <param name="DocNum">Index to a docs[] member</param>
      /// <param name="EditingParams">Elements to be inserted</param>
      /// <param name="ValidXmlExp">True iff we expect the result to parse
      /// </param>
      /// <param name="WordListExp">Expected returned value</param>
      /// <param name="ExceptionExp">
      /// Expected exception
      ///  This specifies a string that the beginning
      ///  of the exception message, if any, is expected to match.
      ///  "" is treated as "No exception is expected".
      /// </param>
      public InsertSymbolTestCase(string Id
            , string[][] EditingParams
            , string WordListExp
            , InsertSymbolTestDoc DocNum =
                InsertSymbolTestDoc.ShortValid
            , bool ValidXmlExp = true
            , string ExceptionExp = DefaultExceptionMessage
            )
      {
        this.Id = Id;
        this.DocNum = DocNum;
        this.EditingParams = EditingParams;
        this.ValidXmlExp = ValidXmlExp;
        this.WordListExp = WordListExp;
        this.ExceptionExp = ExceptionExp;
      }  // end: InsertSymbolTestCase()

      /// <summary>
      /// Index to a docs[] member
      /// </summary>
      public InsertSymbolTestDoc DocNum { get; private set; }

/*
```

```
*/
        /// <summary>
        /// Elements to be inserted
        /// </summary>
        public string[][] EditingParams { get; private set; }

        /// <summary>
        /// Expected exception
        ///  This specifies a string that the beginning
        ///  of the exception message, if any, is expected to match.
        ///  "" is treated as "No exception is expected".
        /// </summary>
        public string ExceptionExp { get; private set; }

        /// <summary>
        /// Test case identifier (required),
        ///  consisting of a unique 2- or 3-character tag, a space,
        ///  and a short description of the test case.
        ///  <para>Prefix "S" refers to the short XML document,
        ///  prefix "L" refers to the longer one.</para>
        /// </summary>
        public string Id { get; private set; }

        /// <summary>
        /// True iff we expect the result to parse
        /// </summary>
        public bool ValidXmlExp { get; private set; }

        /// <summary>
        /// Expected returned value
        /// </summary>
        public string WordListExp { get; private set; }
    }  // end: InsertSymbolTestCase{}
```

Given this definition, we can use the following constructor in **`testValues[0]`**:

```
        new InsertSymbolTestCase (
          Id : "S01 Short document, adding noun and verb",
          EditingParams : new[]{
              new [] {"before adding anything", "Sentence/Symbol",
                "(beginning)", "a|b|c"},
              new [] {"after first insertion", "Sentence/Symbol",
                "Sentence/Symbol/Symbol", "DIAMOND|Noun|Treasure"},
              new [] {"after second insertion", "Sentence",
                "Sentence/Symbol",
                "Carry|Verb|Take the named object with you"},
            },
          WordListExp : "basket (Noun), diamond (Noun), carry (Verb)"
        ),
```

The main work to be done here was to delete the three optional parameters, which are thus given their default values in this call. We have also deleted all of the comments describing the properties (having moved them into the new type definition), so we have made this constructor call more compact and easier to read, with no loss of information.

▶ In Task "**TODO: InsertSymbolTest() -- Define inputs and expected outputs.**", format **testValues[1]** as a constructor call.

It has been sleeping in **testValues[]** as a collection of in-line comments, but we are ready to express it in its new format.

It should look something like this when finished:

```
        new InsertSymbolTestCase (
          "L01 Long document, adding noun & verb",
            new[]{
              new [] {"before adding anything", "Sentence/Symbol",
                "(beginning)", "a|b|c"},
              new[]{"after first insertion", "Sentence/Symbol",
                "Sentence/Symbol/Symbol", "DIAMOND|Noun|Treasure"},
              new [] {"after second insertion", "Sentence",
                "Sentence/Symbol[2]",
                "Carry|Verb|Take the named object with you"},
            },
            "diamond (Noun), water (Noun), plant (Noun),"
              + " pour (Verb), carry (Verb)",
            InsertSymbolTestDoc.LongValid
            ),
```

For brevity, though possibly abbreviating the code a bit more than necessary, I have omitted the parameter names in this constructor call. In doing so, I also needed to arrange the required parameters in the order specified by the constructor.

The result of building and using the new **InsertSymbolTestCase{}** type, though it involved considerable work, is that we now have more flexibility in specifying **testValues[]** elements, where much of the work in specifying unit-test cases will take place. We can now avoid explicitly including what is now redundant information, if we wish, via omitting optional parameters or using alternate constructors[98]. We may make the comments invisible except when we wish to see them (such as while we are specifying parameter values). Unless only a few elements of **testValues[]** are specified, keeping these specifications short and as simple as possible can make the code easy to read and understand.

5.3.17.5.4 NOTE ON XML COMMENTS

The XML comments on some of the **InsertSymbolTestCase{}** properties, especially on **EditingParams**, are kind of skimpy, containing more of a reminder of their contents than a full specification. That is not entirely due to haste or laziness — I wanted to list these details only once, for ease of maintenance, and some of the properties duplicate, or closely resemble, the XML comments on parameters to **InsertSymbol**(). Instead of maintaining nearly duplicate XML comments, I felt that the XML comments in working code, such as in **InsertSymbol()**, should be the essential ones, those that a user would need in order to use the method. They should be the more authoritative specifications, the governing ones in case of differences.

In contrast, comments in the TDS methods, as well as in their nested types, such as **InsertSymbolTestCase{}**, are intended for use only by developers, not also by users of the working code,

[98] We used only one constructor in this example, but see section 4.8.4.2 for an example of using multiple constructors in **testValues[]**, used in **TimeRoundedTest()**.

and the developers have easy access to the **`InsertSymbol()`** comments, for example via IntelliSense, so there's less need for duplicate comments in the TDS code.

For example, in the statement

```
actual = NewCode.InsertSymbol(docCopy,
      tCase.EditingParams, out wordList);
```

, hovering the mouse cursor over "**`EditingParams`**" displays some brief information ("Elements to be inserted") about the **`EditingParams`** property of an **`InsertSymbolTestCase{}`** instance, but this comment is more of a reminder than an authoritative description. A detailed description of this parameter may be seen via IntelliSense while editing the value of an **`editingParams`** parameter to **`InsertSymbol()`**, or in the Object Browser window while examining the description of the **`NewCodeNamespace.NewCode.InsertSymbol()`** constructor.

In contrast, clicking at the end of "**`EditingParams`**" in this statement (but before the comma) and pressing <shift><control><space> (or typing and erasing another comma after "**`EditingParams`**") will pop up an IntelliSense box containing the XML comments from the code in **`NewCode.InsertSymbol()`** for the **`editingParams`** parameter of **`InsertSymbol()`**, displaying parameter information on the **`InsertSymbol()`** call, rather than a description of the **`EditingParams`** property of **`tCase`**. In the following screen shot, both the parameter comments and the property comments are displayed, and they should look like these:

Two sets of IntelliSense comments are visible here; the more detailed, upper one comes from the XML comments on the **`NewCode.InsertSymbol()`**'s parameter **`editingParams`**, and the shorter, lower one comes from the XML comments on the **`InsertSymbolTestCase.EditingParams`** property.

5.3.17.5.5 REMOVE EXPLICITLY STATED DEFAULT VALUES FROM `TESTVALUES[]` ELEMENTS

Because this constructor gives default values to some parameters, such as **`DefaultExceptionMessage`** for the **`ExceptionExp`** property, we can omit some commonly used values from these expressions.

- Remove the properties in **`testValues[]`** that are explicitly given their default values.

You can display the default values using IntelliSense by putting the cursor on a parameter name and pressing <shift><control><space> (or by typing and erasing a nearby comma). Removing the no-longer-necessary lines can reduce clutter. What you are probably most interested in seeing is whatever distinguishes each test case from the others.

- (optional) Remove the "**`Id :`**" name from the constructor calls.

I suggest this to keep the test cases short, as we may eventually include many test cases in **`testValues[]`** . This is merely a matter of style, and I mention it only to explain why the example code in section 5.3.17.5.3 might not match your code, though both might be correct.

5.3.17.5.6 TEST IT

- Having refactored the contents of **`testValues[]`**, test (using Start Debugging or <F5>).

The result, as before, should be

```
Passed: 2  Failed: 0  Inconclusive: 0
```

5.3.17.6 Add new test cases

Now we are ready to add a few more test cases to the two that we have already defined; most of the heavy lifting has been done by now. As in our previous exercise, we will do things like trying to annoy **`InsertSymbol()`** so that it will raise exceptions, or like placing unusual XML elements into the document to verify that they are placed correctly.

Since each **`testValues[]`** element occupies several lines of code, I have surrounded each one with **`#region`** ... **`#endregion`** directives, to allow them to be temporarily hidden. (I did this manually; TDS does not provide any automatic means of adding **`#region`**s to the source code.)

To illustrate the use or omission of parameter names, I have omitted them in some of the test cases. You may judge for yourself if the names help identify the parameters as you read the code, or if they simply occupy space needlessly.

5.3.17.6.1 DEFINE NEW **`TESTVALUES[]`** ELEMENTS

In toto, the **`Id`** tags are "**`S01`**", "L**`01`**", "**`S02`**", "**`S03`**", "**`S04`**", "**`L05`**", "**`S06`**", "**`S07`**", "**`X01`**" , "**`V01`**", and "**`V02`**".

You may wish to compare your current versions of the first two test cases shown below; there should not be major differences.

► Copy the following test cases and paste them into **`testValues[]`** in your own code, replacing the existing two elements. (I converted the page breaks to comments.)

```
      var testValues = new[] {

        //TODO: InsertSymbolTest() -- Define inputs and expected outputs.
        #region Test case S01
        new InsertSymbolTestCase (
          "S01 Short document, adding noun and verb",
          EditingParams : new[]{
              new [] {"before adding anything", "Sentence/Symbol",
                "(beginning)", "a|b|c"},
              new [] {"after first insertion", "Sentence/Symbol",
                "Sentence/Symbol/Symbol", "DIAMOND|Noun|Treasure"},
              new [] {"after second insertion", "Sentence",
                "Sentence/Symbol",
                "Carry|Verb|Take the named object with you"},
            },
          WordListExp : "basket (Noun), diamond (Noun), carry (Verb)"
        ),
        #endregion Test case S01
        #region Test case L01
        new InsertSymbolTestCase (
          "L01 Long document, adding noun & verb",
            new[]{
              new [] {"before adding anything", "Sentence/Symbol",
                "(beginning)", "a|b|c"},
              new[]{"after first insertion", "Sentence/Symbol",
                "Sentence/Symbol/Symbol", "DIAMOND|Noun|Treasure"},
              new [] {"after second insertion", "Sentence",
                "Sentence/Symbol[2]",
                "Carry|Verb|Take the named object with you"},
            },
            "diamond (Noun), water (Noun), plant (Noun),"
              + " pour (Verb), carry (Verb)",
            InsertSymbolTestDoc.LongValid
            ),
        #endregion Test case L01
        #region Test case S02
        new InsertSymbolTestCase(
          "S02 Misplaced <Symbol>",
          DocNum : 0,
          EditingParams : new[]{
            new [] {"after misplaced insertion",
              "Sentence/Symbol",
              "Sentence/Symbol/Symbol/Description",
              "Arsenic|Noun|Poison"},
          },
          ValidXmlExp : false,
          WordListExp : "(none)"
        ),
        #endregion Test case S02
/*
```



```
*/
        #region Test case S03
        new InsertSymbolTestCase(
          "S03 Misplaced <Symbol>, short document",
          DocNum : 0,
          EditingParams : new[]{
            new[] {"after adding a verb", "Sentence",
              "Sentence/Symbol/Description",
              "CARRY|Verb|Take an object with you"},
            new [] {"after misplaced insertion",
              "Sentence/Symbol",
              "Sentence/Symbol/Symbol/Description",
              "Arsenic|Noun|Poison"},
          },
          ValidXmlExp : false,
          WordListExp : "(none)"
        ),
        #endregion Test case S03
        #region Test case S04
        new InsertSymbolTestCase(
          "S04 Valid <Symbol>, short document",
          DocNum : 0,
          EditingParams : new[]{
            new[] {"after adding a verb", "Sentence",
              "Sentence/Symbol/Description",
              "CARRY|Verb|Take an object with you"},
            new [] {"after misplaced insertion",
              "Sentence/Symbol",
              "Sentence/Symbol/Description",
              "Arsenic|Noun|Poison"},
          },
          WordListExp : "arsenic (Noun),"
          + " basket (Noun), carry (Verb)"
        ),
        #endregion Test case S04
        #region Test case L05
        new InsertSymbolTestCase(
          "L05 Malformed EditingParams, long document",
          EditingParams : new[]{
            new[] {"after adding a verb", "Sentence",
              "Sentence/Symbol/Description",
              "CARRY/Verb/Take an object with you"},
              //Wrong punctuation
          },
          ExceptionExp : "Values of tags for",
          ValidXmlExp : false,
          WordListExp : "(No list -- invalid <Sentence>)"
        ),
        #endregion Test case L05
/*
```

```
*/
        #region Test case S06
        new InsertSymbolTestCase(
          "S06 Malformed XML, short document",
          DocNum : 0,
          EditingParams : new[]{
            new[] {"after adding a verb", "Sentence",
              "Sentence/Symbol/Description",
              "CARRY|Verb|Take <Symbol>an object</Symbol> with you"},
          },
          ValidXmlExp : false,
          WordListExp : "(none)"
        ),
        #endregion Test case S06
        #region Test case S07
        new InsertSymbolTestCase(
          "S07 Malformed XML, unbalanced '<', short document",
          DocNum : 0,
          EditingParams : new[]{
            new[] {"after adding a verb", "Sentence",
              "Sentence/Symbol/Description",
              "CARRY|Verb|Take <Symbol an object</Symbol> with you"},
          },
          ExceptionExp : "'object' is an unexpected token.",
          ValidXmlExp : false,
          WordListExp : "(No list -- invalid <Sentence>)"
        ),
        #endregion Test case S07
        #region Test case X01
        new InsertSymbolTestCase(
          "X01 Faulty document that should be rejected",
          DocNum :InsertSymbolTestDoc.Invalid,
          EditingParams : new[]{
            new [] {"before adding anything", "Sentence/Symbol",
              "(beginning)", "a|b|c"},
          },
          ValidXmlExp:false,
          WordListExp:"(No list -- invalid <Sentence>)"
        ),
        #endregion Test case X01
        #region Test case V01
        new InsertSymbolTestCase(
          "V01 Short document with no noun nor verb",
          DocNum : InsertSymbolTestDoc.NoNounNorVerb,
          EditingParams : new[]{
            new [] {"before adding anything", "Sentence/Symbol",
              "(beginning)", "a|b|c"       },
            new [] {"after first insertion", "Sentence/Symbol",
              "Sentence/Symbol/Symbol", "QUICKLY|Adverb|Speed=2"},
          },
          WordListExp : "(No nouns or verbs were found.)"
        ),
        #endregion Test case V01
/*
```

```
*/
        #region Test case V02
        new InsertSymbolTestCase(
          "V02 Short document -- invalid insertion",
          new[]{
            new [] {"after first insertion", "Sentence/Symbol",
              "Sentence", "QUICKLY|Adverb|Speed=2"},
          },
          "(none)",
          InsertSymbolTestDoc.NoNounNorVerb, "",false
        ),
        #endregion Test case V02
};
```

5.3.17.6.2 TEST USING THE NEW TEST CASES

- Having added these, test again (using Start Debugging).

The result, as before, should be

```
Passed: 2  Failed: 0  Inconclusive: 0
```

You might notice that the test report contains plenty of detail regarding the progress of the various tests. If these are not of importance to you, you may want to suppress them — none of them indicate any error conditions (since all of the current test cases Passed). Reducing or eliminating unneeded, non-error messages from the TDS method (or, if practical, the working code as well) will be especially helpful after the working code is largely debugged, when other tests are also generating output to the test report. Of course, whenever an error is detected during a TDS test run, messages describing it should be written to the test report in enough detail to allow quick resolution of the problem.

5.3.18 Do some housekeeping (refactor subexpressions)

In real life, if you were developing code similar to that shown in this example, you might already have done some of the refactorings that I shall suggest here, but I have delayed them until now to simplify the presentation. The code works without them, and this section may be skipped, continuing with section 4.7, "Run the working code without TDS", without affecting the subsequent discussion. The purpose here is to illustrate that, as you refactor the code, wherever it's important that all of the behavior that you care about should remain unchanged, running the TDS tests frequently can give you some assurance that none of your refactorings have changed any of the outputs that you are testing. If a test fails, you (we hope) immediately know where to look for trouble so that you can correct it.

It is probably a good idea to have **`Assert`** statements in your TDS test method that cover all the outputs that interest you or your customer. Notice that in this example, instead of comparing the entire returned **`<Sentence>`** to its expected value, which would be easy for the computer to do but would involve code that would be tedious for us to maintain, we summarize those contents in the short string **`wordList`** returned by **`NewCode.InsertSymbol()`**. Otherwise, we might have needed to maintain a complete copy of the expected XML tree, so we could compare the one returned by the method with it. Here we have made a conscious decision that, if the returned value is grammatically correct (verified using the XSD) and the summarizing string, returned in **`wordList`**, matches what we expect it to contain, then that's good enough evidence for our purposes that the method is working correctly. If we learn later that this was a poor assumption, we can revise our tests accordingly at that time.

5.3.18.1 Give names to literal values

Something that we may want to do is to give symbolic names to literal values in the code In this example, we have numerous instances of literal values scattered throughout the code in **`InsertSymbol()`**, and giving them names would allow us, for example, to put all of their definitions at the beginning of the method, where we can more easily examine them, change them if desired, and keep them consistent.

For example, the indices to some of the arrays used in **`InsertSymbol()`** are not as suggestive as names would be. (Do you remember which string is passed as **`editingParams[0][2]`**? Neither do I.) Using a name like **`editingParams[0][ixPs2InsertionPoint]`**, if the name is chosen well, can help keep track of what the value means.

In addition to using assignment statements such as

```
internal const string DefaultExceptionMessage = " No exception was thrown";
```

to give names to literal values, we might also define and use locally defined types with named and commented properties.

In this example, we defined **`enum InsertSymbolTestDoc`** (in section 5.3.17.4.1.2) to attach names (and XML comments describing them) to some of our test data documents, and we defined the nested **`class InsertSymbolTestCase`** (in section 5.3.17.5) to help make the test case definitions in **`testValues[]`** easier to read and write (with explanations embedded in its XML comments). Both involved some extra work at the time they were created but promised to make the code where the objects defined with their help are used easier to understand as one reads the code. I think it can also make the code easier to modify; for example, if I add another document to the **`docs[]`** array and its correspoinding name and description to **`InsertSymbolTestDoc`**, then that name and description automatically become available every place that the new document is used thereafter, with no further effort required. Removing it later is easier, too, as we can search for its name (or let the compiler complain that the name's definition is missing).

In the example shown in section 5.3.18.4 below, "Refactored code example", most of the literal strings have been replaced by names, but in some cases the code might have been easier to read (though possibly trickier to maintain) if the literal value is left alone. (I sometimes do both, then keep only the one that looks ecbetter.)

Giving names to the literal values also allows these names to be searched by a "Find All References" operation, which produces a more focused list than a general "Find" operation. Possibly your customer could require all literals to be given names, as a matter of style, in which case your decision becomes easy. (I once had a boss who required even literal values like 0 to be given names. That wasn't difficult to do, though it seemed kind of pointless — but the customer is always right!)

5.3.18.2 Give names to common subexpressions

Inspecting the code in **`InsertSymbol()`** may reveal the existence of common subexpressions that could be factored out into named variables with values that could be changed once and applied in multiple places. We did something similar in section 4.8.3.5, replacing two occurrences of similar code with calls to new method **`CheckForIndexException()`**, and enabling us to update both of the original occurrences in one place as needed.

Naming common code (such as by extracting a method from it, and replacing that code with calls to the method) can improve consistency; for example, if there is no reason for two messages to have different formats, we could have them share a single, more maintainable expression that specifies the one format to be used by both. The compiler probably already does this, invisibly, but if you name the expression yourself, you

get the power to determine where it's used (and maybe those places are common subexpressions themselves). If you need to make an improvement to the message, you need to update it only in one place, and the change is applied automatically everywhere that it is used.

Naming an expression can also help during debugging, as the name and value can appear in VS's "Locals" window while stopped at a breakpoint. (We did something like this earlier, in section 5.1.5.2.2, intentionally giving the name **`result`** to an expression that we would use only once.)

It is possible that an expression that appears only once might need to be changed later, so giving it a name could make it easy to find and update. For example, in **`InsertSymbol()`** we use a test for determining if we want to apply the **`AddAfterSelf()`** method; this involves determining if the path to the insertion point begins with "(". We may want to be able to change what starting character we look for in this case, or we might want to use some other test entirely. So, instead of just making the "**`(`**" be a named string that we could change if we wish, in the refactored code we pull this entire test out and give it the name **`IsNotDummyXpath()`**, defining it to look like this:

```
Func<string, bool> IsNotDummyXpath =
      xpath => xpath.Substring(0, 1) != "(";
```

With this kind of definition, we could easily change the criterion to be a test that depends on the final character of the string instead of the first one, or the length of the string, or something else entirely, by changing just this one line of code. Of course, we could even go a bit further and refactor this test into an entire separate method if we thought it might also be useful elsewhere.

Another example of common code given a name is in **`ReportException()`**, an **`Action`** that displays exception messages in a standard format. If the message needs to be changed, for example, one change will take care of all uses of it. (If our main purpose in this case had been to keep the code as short as we could, we should probably have undone this refactoring after seeing the results. The result of the change proposed here probably makes the overall code slightly longer than the original version, due to the length of the definition of the common code.)

5.3.18.3 Refactor large blocks of code into methods

We won't break out any code into separate methods here (it's an exercise for the reader ☺), but sometimes doing that can make the containing function code easier to read and test. Any such new method will need its own XML comments and should probably be given a corresponding TDS method, so doing this involves some extra work, but the result is often code that is easier to maintain than a single, long method would be.

In an effort to improve legibility, I have used **`#region`**…**`#endregion`** directives, as you have probably noticed, to allow some of the code to be collapsed out of sight (VS menu "Edit, Outlining, Toggle Outlining Expansion") whenever it is of no current interest. The content of such a **`#region`**, if it is lengthy enough, is a prime candidate for extraction into a method, even if that method is only ever called from that one place.

In my own code, about the time a block of code occupies more space than I can see at one time, about 30 lines, I start looking for convenient ways to extract a method from it, even if I expect that the method will never be called from anywhere else.

I hesitate to mention performance penalties, as they are probably insignificant, but overdoing the extraction of very short code blocks into methods could unnecessarily consume run-time resources. (Or maybe not, if the optimizing C# compiler notices that the methods can be in-lined instead of being called.) Also, using too many

short methods could clutter the namespace, making code hard to find, if you don't have an easy-to-use naming convention for the method names.

An alternative to extracting short blocks into methods might be extracting them into **`Func()`**s or **`Action()`**s, as we did in section 5.2.8.5.2.1. I think of **`Func()`** and **`Action()`** as C#'s strongly typed version of those marvelously versatile pre-processor "**`#define`**" directives in C++ that can make a program a dream to write but possibly a nightmare to read and understand later, if those directives are used clumsily. A **`Func()`** or **`Action()`** is more limited than a C++ macro (and it applies at run time, not at compilation time), but it can act as a mini-method definition, and if you decide later that you'd like to extract it into a real method, that would not be difficult to do[99].

Along with saving your work frequently as you refactor your code, I suggest running the current TDS method frequently, to help you notice any easy-to-correct mistakes.

5.3.18.4 Refactored code example

The code in **`InsertSymbol()`** works correctly, as you have no doubt noticed, without any of the refactorings that I am suggesting, but one possible result of applying these refactorings is shown below, and this is the version to be used in the subsequent tests in this example. (The XML comments are unchanged, since the behavior is expected to be unchanged.) If you saved the version of your new function member that you had before beginning to refactor it, you could visually compare that saved version with the refactored version, but that could be kind of tedious with a long method like **`InsertSymbol()`**. See section 5.3.18.5 below for a low-cost, low-hassle alternative.

The refactored code, using the similar name **`InsertSymbol2()`**, might look like what follows.

[99] One change that might be needed would be adding parameters to the method call for variables that are visible to the **`Func()`** or **`Action()`** that would otherwise be hidden from the method.

► (Optional) Copy and paste the following code immediately below the end of the definition of **`InsertSymbol()`**:

```
    /// <summary>
    /// Insert the XElements specified by <paramref name="editingParams"/>
    /// into <paramref name="doc"/>, validating after each one,
    /// stopping on failure.
    /// Return True if all changes are valid,
    /// and return a summary of some of the elements.
    /// Write intermediate results to the Console.
    /// </summary>
    /// <remarks>XML exceptions due, for example, to malformed parameters
    /// are passed on to the caller.</remarks>
    /// <param name="doc">Original XML document,
    /// into which the &lt;Symbol&gt;s specified in
    /// <paramref name="editingParams"/> will be inserted.</param>
    /// <param name="editingParams"><para>
    /// Specification of XElements
    /// to be inserted and validated in sequence.
    /// In each element of this array,
    /// </para><para>[][0] = context, a description of
    /// the type of change to be made.
    /// Example: "before editing"
    /// </para><para>[][1] = label, XPath specification
    /// of area to be validated.  Example: "Sentence"
    /// </para><para>[][2] = insertionPoint, Xpath identifying the XElement
    /// following which the new XElement is to be inserted.
    /// Example: "Sentence/Symbol"
    /// </para><para>[][3] = insertedElement, components of the
    /// &lt;Symbol&gt; to be inserted, separated by '|' characters.
    /// </para><para>Example: "GOLD|Noun|Collectible treasure"
    /// </para><para>[0] = &lt;String&gt; value, e.g. "GOLD"
    /// </para><para>[1] = &lt;Sense&gt; value, e.g. "Noun"
    /// </para><para>[2] = &lt;Description&gt; value,
    /// e.g. "Collectible treasure"
    /// </para></param>
    /// <param name="wordList"><para>Comma-separated list of
    /// nouns and verbs in the &lt;Sentence&gt;, nouns first.
    /// </para><para>Example: "water (Noun),
    /// plant (Noun), carry (Verb)"</para>
    /// </param>
    /// <returns>True iff no validation errors were detected
    /// after any of the insertions.</returns>
    /// <exception cref="ArgumentException">The values of tags for the
    /// &lt;Symbol&gt; must be specified in the format
    /// <para>"string_value|sense_value|description_value".</para></exception>
    public static bool InsertSymbol2(
        XDocument doc,
        string[][] editingParams,
        out string wordList)
    {
/*
```

```
*/
      #region Constants
      //TODO: InsertSymbol2 -- Some of these strings could be parameterized,
      //  to allow callers to insert other types of nodes
      //  or to extract different types of elements.

      //Separator for editingParams[][3] values
      const char elementSeparator = '|';

      //Constants identifying array indices to paramSet[]
      const int ixPs0Context = 0;  //Description of change
      const int ixPs1ValArea = 1;  //Area to be validated
      const int ixPs2InsertionPoint = 2;  //XPath to insertion point
      const int ixPs3SymbolSpecs = 3;   //SymbolSpecs string,
                                        // components of the inserted <Symbol>

      //Constants identifying array indices to symbolSpecs[]
      const int ixSsString = 0;  //<String> value
      const int ixSsSense = 1;  //<Sense> value
      const int ixSsDescription = 2;  //<Description> value

      //Constants related to validation messages
      const string validatingNodeMsgFmt = "Validating {0} {1}...";
      const string validationStatusMsgFmt = @"Area ""{0}"" {1}.";
      const string validationStatusMsgBad = "is not valid";
      const string validationStatusMsgValid = "is valid";

      //Error message format for validation errors
      const string validationErrMsgFmt =
@"InsertSymbol @{0} at {1}:
  The following validation error occurred:
    ==> ""{2}"". ";

      //Constants related to the word list
      const string wordListDefault = "(No list -- invalid <Sentence>)";
      const string wordListDescription = "Nouns & verbs in this sentence";
      const string wordListEmptyMsg = "(No nouns or verbs were found.)";
      const string wordListEmptyPlaceholder = "(none)";
      const string wordListItemFmt = ", {0} ({1})";
      const string wordListListFmt = "{0}: {1}";
      const string wordListSpec =
            "//Symbol[Sense='W.Noun' or Sense='W.Verb']";
      //Constants related to XML in the document
      const string xmlDocString = "String";
      const string xmlDocSense = "Sense";
      const string xmlInsertedXelementFmt = @"
<Symbol sentence=""false"">
  <String>{0}</String>
  <Sense>W.{1}</Sense>
  <Description>{2}</Description>
</Symbol>
";
/*
```

```
*/
      const string xmlInsertedXelementFmtErr1 =
@"Values of tags for the <Symbol> must be specified in the format
  ""string_value{0}sense_value{0}description_value"".";

      #endregion Constants

      #region bool IsNotDummyXpath(string xpath)
      //Test that a name is intended to specify
      //    an XPath location for inserting an XElement,
      //    instead of being a placeholder.
      //  For example, IsNotDummyXpath("(beginning)") returns False.
      Func<string, bool> IsNotDummyXpath =
            xpath => xpath.Substring(0, 1) != "(";
      #endregion bool IsNotDummyXpath(string xpath)

      //Return this value if the unmodified document is invalid
      wordList = wordListDefault;

      #region Validate and compile doc1
      //This becomes True on validation errors.
      var isValidationError = false;

      doc.Validate(Schemata,
            (sender, e) =>
            {
              Console.WriteLine(
                    validationErrMsgFmt
                    , "doc1"  //{0}
                    , "initial validation"  //{1}
                    , e.Message  //{2}
                );
              isValidationError = true;
            },
            true);
      if (isValidationError) return false;
      #endregion Validate and compile doc1

      foreach (var paramSet in editingParams)
      {
        #region Perform specified editing and check validity
        #region ReportException(string msg)
        //Display an exception message in a standard format,
        //  using validationErrMsgFmt
        Action<string>
              ReportException = (
              msg  //Exception message
              ) =>
                Console.WriteLine(validationErrMsgFmt
                      , paramSet[ixPs1ValArea]  //{0}
                      , paramSet[ixPs2InsertionPoint]  //{1}
                      , msg  //{2}
                    );
        #endregion ReportException(string msg)

/*
```

```
*/
        #region Insert the element
        var symbolSpecs = paramSet[ixPs3SymbolSpecs].Split(
              new[] { elementSeparator },
              StringSplitOptions.RemoveEmptyEntries);
        if (symbolSpecs.Count() != 3)
        {
          var message1 = string.Format(
                  xmlInsertedXelementFmtErr1
                  , elementSeparator  //{0}
                );
          throw new ArgumentException(message1);
        }
        var insertedXelement = XElement.Parse(
              String.Format(xmlInsertedXelementFmt
                , symbolSpecs[ixSsString]  //{0}
                , symbolSpecs[ixSsSense]  //{1}
                , symbolSpecs[ixSsDescription]  //{2}
              ));

        if (IsNotDummyXpath(paramSet[ixPs2InsertionPoint]))
          try
          {
            doc.XPathSelectElement(paramSet[ixPs2InsertionPoint])
                  .AddAfterSelf(insertedXelement);
          }
          catch (Exception e)
          {
            ReportException(e.Message);
            wordList = wordListEmptyPlaceholder;
            return false;
          }
        #endregion Insert the element

        #region Validate the changed part of the document
        //This becomes True on validation errors.
        var hasValidationError = false;
        #region valHandler(sender, e)
        ValidationEventHandler valHandler = (sender, e) =>
        {
          ReportException(e.Message);
          hasValidationError = true;
        };
        #endregion valHandler(sender, e)

        Console.WriteLine(validatingNodeMsgFmt
                , paramSet[ixPs1ValArea]  //{0}
                , paramSet[ixPs0Context]  //{1}
              );

        var element = doc.XPathSelectElement(paramSet[ixPs1ValArea]);
        element.Validate(element.GetSchemaInfo().SchemaElement,
              Schemata, valHandler, true);

/*
```

```
*/
        Console.WriteLine(validationStatusMsgFmt
                , paramSet[ixPs1ValArea]  //{0}
                , hasValidationError
                  ? validationStatusMsgBad
                  : validationStatusMsgValid  //{1}
              );

        #endregion Validate the changed part of the document

        if (hasValidationError)
        {
          wordList = wordListEmptyPlaceholder;
          return false;
        }
        #endregion Perform specified editing and check validity

        #region Calculate wordList
        //Return a comma-separated list of selected words,
        //  sorted by part of speech.
        //  The first 2 characters of each <Sense> value
        //  (the "W." part) are omitted.
        //  Example: "water (Noun), plant (Noun), carry (Verb)"
        wordList = String.Concat(
                from node in
                  doc.XPathSelectElements(wordListSpec)
                let partOfSpeech = node.Element(xmlDocSense)
                  .Value.Substring(2)
                orderby partOfSpeech
                select String.Format(wordListItemFmt
                  , node.Element(xmlDocString).Value.ToLower()  //{0}
                  , partOfSpeech  //{1}
                )
              );
        if (wordList.Length < 2)
          wordList = wordListEmptyMsg;
        else
          wordList = wordList.Substring(2);
        Console.WriteLine(wordListListFmt
                , wordListDescription  //{0}
                , wordList  //{1}
              );
        #endregion Calculate wordList

      }  // end:foreach (var paramSet...

      return true;
    }  // end:InsertSymbol2()
```

There is no need to examine this code in great detail. Its purpose is to illustrate an implementation of a method (in this case, **`InsertSymbol2()`**) that performs the same work as the code it replaces, but is expressed in a way that is intended to be easier to read, understand, and update than the original, even though it may look quite different. In section 5.3.18.5 we address a way to try to determine if it actually does perform in the same way.

If you are reading this document using Adobe Acrobat Reader, be aware that the process of copying the source code for **`InsertSymbol2()`** from the above listing will likely have introduced some errors into the source code — the leading spaces on some lines of output may have become lost. Much of the indentation in the source code may be lost as well, but that affects only the legibility of the source code. However, output is affected in a couple of lines, and these need to be corrected before running the comparison that we shall perform in section 5.3.18.5.3. One type of affected statements is in the

```
        #region Perform specified editing and check validity
```

`region` of **`InsertSymbol()`**, in the string literal

```
@"InsertSymbol @{0} at {1}:
  The following validation error occurred:
    ==> ""{2}"". "
```

, in which the second line is indented two spaces and the third line is indented four spaces. The version copied from the PDF version of the *TDS User's Guide* and now located in the

```
      #region Constants
```

at the beginning of the definition of **`InsertSymbol2()`**, may look like this after being pasted into the code in file Program.cs:

```
      //Error message format for validation errors
      const string validationErrMsgFmt =
      @"InsertSymbol @{0} at {1}:
The following validation error occurred:
==> ""{2}"". ";
```

► Reformat this expression to make it look like the original version, two spaces in front of "The" and four spaces in front of "==>".

This formatting would be unimportant, except that we want to compare the two versions of output to look for significant differences, and the lines differing only in spaces would be distracting.

5.3.18.5 Testing refactored code

5.3.18.5.1 PURPOSE OF THIS QUICK TEST

In real life, we would probably have performed the refactorings mentioned above as we developed the code, and would not have available to us two widely separated versions that we could compare (as we shall do here). Also, the comparison to be described here assumes that the function member's code generates text using **`Console.WriteLine()`** (or similar) statements, and the main purpose of this comparison is to verify that this text, which is not monitored by the TDS method, is not changed as a result of the refactorings.

A more thorough test might involve automatic comparison of the text output under program control, as we did in **`TestableConsoleMethodTest()`** .

Assuming that we do have on hand versions of the code both before and after it was refactored, that it does produce some console text that we want to verify was not changed by the refactoring, and that we do not want to take the effort to test that text in a TDS method, it is easy to apply the technique to be described here.

We want to gain some confidence that the refactored version of the **`InsertSymbol()`** code behaves the same as the previous one, but without our having to do much additional work to verify that. Before erasing the old version, we shall run the tests we currently run, once with each version. In addition to the TDS summary

output, we shall collect all the text output from each version and compare both sets, looking for minor errors like miscounted spaces or inconsistently capitalized names.

5.3.18.5.2 ALTER THE OLD VERSION, IF NECESSARY

Depending on the nature of your code, you might decide to intentionally alter some of the text output by one of the versions to make it obvious that you can detect differences. (We did something similar in an earlier example, in the discussion of buggifying tested code in section 5.2.9.3, to show that we could correctly display unexpected results and notice the difference from the expected output.)

In this case, since the output already contains time stamps that are never the same twice, we shall simply compare them and check that the time-stamped lines are the only ones that differ. Except for the time stamps, we should see that the refactored version produces the same output that the previous version did.

5.3.18.5.3 RUN BOTH VERSIONS

► In VS's Solution Explorer, right-click on the TDS Project, choose "Add, New Item, Visual C# Items, General, Text File"; name it "xxOld.txt".

If file xxOld.txt already exists in the TDS Project, open it as an existing item and erase its current contents.

We shall use this file to collect the Console output from the TDS test run using the older definition of **`InsertSymbol()`**.

► Do this again and name the file "xxNew.txt".

We shall use this file after getting output from **`InsertSymbol2()`**.

► Check that TDS is still the Startup Project.

► Run the TDS program using "Debug, Start Debugging" (or <F5>).

This runs a test of **`InsertSymbol()`**, the old version, and displays its output, including messages about the results of each test in **`testValues[]`** of **`InsertSymbolTest()`**.

Don't close the Console window when TDS pauses after displaying the summary at the end of the test.

► Copy the output from the Console window, as we did in section 4.8.3.2.

However, instead of using "<alt-space>EK" to select some of the text, use "<alt-space>ES<enter><enter>", which will copy all of the text in the Console window to the Clipboard and will close the Console window.

► Paste the copied text into the xxOld.txt editing window in VS, replacing any existing contents of xxOld.txt..

This may be accomplished via <control>A<control>V.

► Save the file (for example via "<control>S").

► In the body of **`InsertSymbolTest()`**, in the "**`TODO: InsertSymbolTest() -- Provide a suitable calling expression`**" Task, change the line

```
actual = NewCode.InsertSymbol(docCopy
```

to be

```
actual = NewCode.InsertSymbol2(docCopy
```

► Run the program again (using "Debug, Start Debugging" or <F5>), but this time save the output to file xxNew.txt.

We now have collected the console text output from each version, including its TDS test report.

5.3.18.5.4 COMPARE THE RESULTS

You may compare the results visually or, as we shall do here, by using a comparison app.

- To compare these results, open a Windows Command-Prompt window or a Windows PowerShell window.
- Navigate to your Demo\TDS\ folder (via "CD" command in Command Prompt or PowerShell).

If you're using a Windows Command Prompt window, run this command:

```
FC /N xxOld.txt xxNew.txt
```

, or in a Windows PowerShell window, run this command (all .on one line):

```
Compare-Object (Get-content .\xxOld.txt) (Get-content .\xxNew.txt) –SyncWindow
1 | Format-List
```

The results generated by this comparison should reveal that nothing in the output (except for the three lines containing the non-matching time stamps) is affected by the changed line in the code. When we examine the output in Windows Command Prompt, it begins with one of the time-stamp messages, somehing like this:

```
Comparing files xxOld.txt and XXNEW.TXT
***** xxOld.txt
    8:  ***** TDS.Test.InsertSymbolTest()
    9:  ***** InitializeTestMethod() was called at 2017-07-30T14:49:37.9940746-
05:00 .
   10:
***** XXNEW.TXT
    8:  ***** TDS.Test.InsertSymbolTest()
    9:  ***** InitializeTestMethod() was called at 2017-07-30T14:57:36.6329049-
05:00 .
   10:
*****
```

In Windows PowerShell, the output begins with lines similar to these, where the "SideIndicator" value "=>" refers to xxOld.txt and "<=" refers to file xxNew.txt:

```
InputObject   : ***** InitializeTestMethod() was called at 2017-07-
30T14:57:36.6329049-05:00 .
SideIndicator : =>

InputObject   : ***** InitializeTestMethod() was called at 2017-07-
30T14:49:37.9940746-05:00 .
SideIndicator : <=
```

5.3.18.5.5 REMOVE THE OLD CODE

- (If you copied the code for InsertSymbol2() into file Program.cs) In file Class1.cs, in **`NewCode{}`**, erase the old **`InsertSymbol()`** definition, including its XML comments.

We used the same XML comments in both versions, since we expected the behavior of the method not to change.

- Rename "**`InsertSymbol2`**" to "**`InsertSymbol`**".

The name **`InsertSymbol()`** appears at the beginning of the method definition as well as in some comments.

You may use VS menu "Edit, Refactor, Rename" (or <F2>), including updating the comments, to rename it. You could instead just edit the code using menu "Edit, Find and Replace". Also, in the "**`TODO: InsertSymbolTest() -- Provide a suitable calling expression`**" Task, change its name in the calling expression if necessary.

Having satisfied ourselves that the two versions produce equivalent results, we have removed the older definition of **`InsertSymbol()`** and renamed the newly refactored version to replace the older one.

5.3.18.6 Comments on this example

It is possible that we could discover additional ways to test **`InsertSymbol()`** that would call for new **`testValues[]`** test cases, **`Assert`** statements, or both. Although refactorings should not call for any changes to the TDS method, newly discovered bugs would. Any new error/bug discovered after the TDS method is (thought to be) complete was apparently not properly detected by it, so even before we examine the code to determine the cause of the bug, it would be useful to add a test case to the TDS method to reproduce the conditions that caused the bug to appear.

We might discover that a function member being tested contains a major flaw that cannot be corrected without a change to its interface with the rest of the system (for example, by changing a parameter or adding a reference to a global property). If such a flaw appears, we may be able to avoid starting over with developing a replacement function member and its corresponding TDS method, by adding test cases and **`Assert`** statements to address the changes, while continuing to use the existing test cases to help ensure that any existing behavior that we wanted to keep is unchanged.

For now, it's apparently reasonable to assume that **`InsertSymbol()`** (newly refactored version) is being tested adequately and that we no longer need to concentrate exclusively on its TDS method, **`InsertSymbolTest()`**. So, let's return to running all of our other tests as well.

5.3.19 Re-enable all TDS tests

- De-comment all of the TDS tests in **`TestMethodsToBeRun`** and/or include the names of any TDS methods listed in the TDS report following this heading, if it be present:

```
The following TDS methods have [TestMethod] attributes
    but are not in the TestMethodsToBeRun list:
```

- Run TDS to get a complete test report, then close the Console window.

5.3.20 Summary

As you might guess, the presentation here has been compressed somewhat in an attempt to avoid useless detail. The actual development of the **`InsertSymbol()`** method involved some code that I later discarded because it was no longer needed. Similarly, the test cases did not all magically appear after the method was written, even though it might appear that way from the order of these paragraphs. As I wrote or rearranged code, I used existing test cases, in **`testValues[]`**, to help check for obvious run-time errors. While concentrating on this method, I commented out all of the TDS tests in **`TestMethodsToBeRun[]`** except for **`InsertSymbolTest()`**. After adding the second test case, I de-commented **`//#define RunOnlySelectedTestData`** at times, to allow me to look bonly at the results of specific branches taken with certain types of inputs. As I added code that might raise exceptions, I also added corresponding test cases to generate those exceptions, while the surrounding conditions were still fresh in my mind. The **`Assert`** statements were slightly less interesting, as they had been pretty much determined by the time the inputs and outputs had settled down, but

they, too, were developed over a period of time, not all at once. After the initial definition of the TDS method, both it and the function member to be tested were developed together — most changes to **`InsertSymbol()`** were paired with corresponding changes to **`InsertSymbolTest()`**. The refactorings described in section 5.3.18 above are an exception, as they were intended to leave the method's behavior unchanged. By that time, the TDS method was serving mostly as a watchdog to detect mistakes, rather than to verify that the function member was properly exhibiting new or changed behavior.

Of course, your preference might differ. You might choose to specify all the test cases first, TDD style, then write the code to try to pass them. Or you may prefer to run a full set of tests every time you run any test, instead of skipping some of them. The purpose of the TDS framework is to give you some choice in how to proceed, without having to redo the infrastructure (= stuff like support for **`RunOnlySelectedTestData`**) every time you construct a new function member that you might eventually want to unit test.

5.3.21 Test using NUnit

As in earlier examples, you can run NUnit to unit-test your TDS test methods(see section 4.5.1) by double-clicking on Demo\Project1.nunit in Windows® Explorer.

Even though NUnit can give you a summary of the results of large numbers of tests, telling you which of the tests did not pass, it does not provide the same support for debugging the tested function members that TDS does. You may want to retain your TDS code to allow you to debug your function members, for example with "**`#define RunOnlySelectedTestData`**" enabled at times in one or more of the TDS source-code files, or with some members of **`TDS.Test.TestMethodsToBeRun[]`** commented out, to exercise some part of your code that is of particular interest to you in investigating a bug.

5.4 Example: Testing a Visual Basic Project

Although most of the examples of "working code" presented in this *TDS User's Guide* are written in C#, it seems appropriate to include an example of working code written in some other language; in this example, it is Visual Basic ("VB"). Although the TDS methods are expected to be written in C#, the working code in this example is written in VB.

5.4.1 Create & run an example Project3

I suppose it might be instructive to include an example that not only uses VB code, but also features a connection to a Web page or a database, or a Web page that connects to a database, but for now those examples are out of scope for this *TDS User's Guide*. If you use such objects, I assume you already know how to connect to them. For debugging and testing, I would choose to put the needed connection code into either

- **`InitializeTestMethod()`** (navigate there via the Task "**`TODO: InitializeTestMethod() -- Add other test-setup code here`**"), to be run once at the beginning of each TDS method, or
- **`InitializeClasses()`** (navigate there via the Task "**`TODO: InitializeClasses() -- Add other class-setup code here`**"), to be done once at the beginning of the entire test session.

Code to close these connections may be added at Tasks "**`TODO: CleanupTestMethod() -- Add other test-cleanup code here`**" and/or "**`TODO: CleanupTestSession() -- Add other end-of-session code here`**".

For this example, besides using a different language for the working code, the instructions are shortened a bit from what you saw in the Tutorial. Whereas we played around with several (simulated working code) function members in the Tutorial, to explore the features of TDS, here we'll instead create a simple method that we will not edit at all (except for temporarily buggifying it, to show that the test is working). We'll just run it, once by

itself to show that it works (or at least doesn't crash), and once using TDS to generate a test report. The idea is that you may use the steps in this example as a guide for using your own code instead of the "Module1.vb" example code used here, to allow TDS to exercise your code. To use this example as intended for adding TDS to your own Solution, substitute a function member of your choosing for the code shown in the box below, or make a copy[100] of some existing VS Solution, and follow the steps of this example beginning in section 5.4.2, suitably modified to invoke your working code.

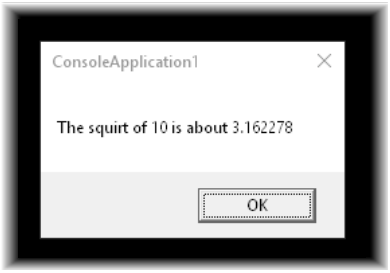
► In VS, close the existing Solution (if one is open), create a new Project, an "Other Languages, Visual Basic, Windows, Classic Desktop" Project, choose "Console App" (or "Console Application"), and set the new Project's name to "ConsoleApp1" or "ConsoleApp2" (the default in VS 2017)[101]. As we did in section 4.3.6.1, save the Solution in an empty folder called "Demo".

► Edit its "Module1.vb" contents to contain the following code:

```
Public Module Module1

    ''' <summary>
    ''' Display the square root of 10.
    ''' </summary>
    Sub Main()
        MsgBox("The squirt of 10 is about " & Squirt(10.0))
    End Sub

    ''' <summary>
    ''' Return the square root of the argument
    ''' </summary>
    ''' <param name="val">Number whose square root we want</param>
    ''' <returns>The square root</returns>
    Function Squirt(val As Single) As Single
        Return Math.Sqrt(val)
    End Function  ' end: Squirt()

End Module  ' end: Module1
```

Hey, I didn't claim that it would do anything useful! As usual, I include some XML comments; even though this code is trivial, I'd rather write the comments when the code is fresh in my mind than to have to examine the code later to determine what comments to add.

We gave Module1 a "**Public**" modifier (default is "**Friend**") to make it accessible outside its Project. (See section 4.8.8 for suggestions on testing inaccessible working code.)



► Run it (for example, via <F5>).

It should pop up in a message box the output that we expect, an approximate value of $\sqrt{10}$.

---

[100] I suggest playing with a copy, instead of the original code, to avoid the possibility of damaging code that you depend on. You may need to edit the copy to, for example, make tested function members visible (as we did in section 4.8.8.3).

[101] Older versions of VS use a default value of "ConsoleApplication1". To be consistent with the instructions for this example, I suggest using the name "ConsoleApp1".

► Click on "OK" to end the run.

5.4.2 Add the TDS Project to the Solution

Follow the steps in section 4.14.7.1 through section 4.14.7.5 to add a TDS Project to this new Solution.

5.4.3 Construct a TDS method

► As in section 4.8.2, we create a TDS method following the "**`TODO: New TDS methods may be placed here:`**" Task in file TDS.cs and enter the name "Squirt" into the **`TdsTest`** snippet[102], giving us the TDS method "**`SquirtTest()`**".

► In Solution Explorer, set Project TDS as the Startup Project (as in section 4.4.3.1).

As this is the first TDS method we'll add to this Solution, a bit of additional housekeeping is needed; most of these changes are located at Task comments.

► In the TDS Project, using Solution Explorer, add a Reference to ConsoleApp1 (in the "Projects, Solution" tab).

See section 4.4.1.2; be sure that the Project's name appears in the list of References for the TDS Project.

► At the "**`TODO: Usings`**" Task, add "**`using ConsoleApp1;`**" (instead of the previous "**`using NewCodeNamespace;`**").

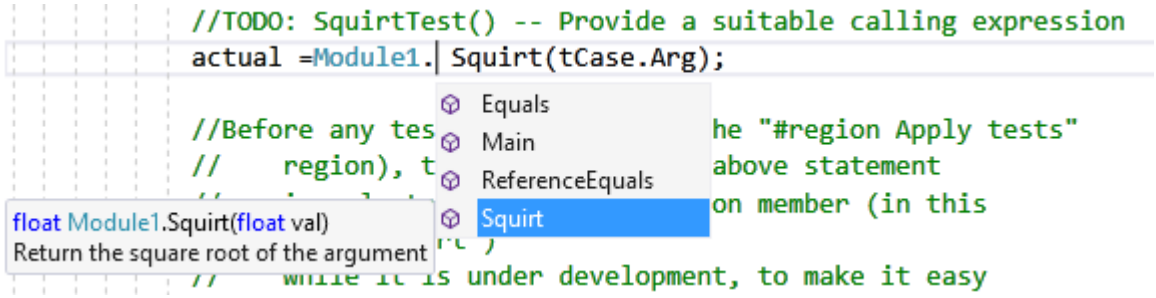
► At the "**`TODO: SquirtTest() -- Define inputs and expected outputs.`**" Task, change the "**`Arg =`**" value from "**`3`**" to "**`2F`**", and change the "**`ValueExp =`**" value from "**`4`**" to "**`1.414214F`**".

► At the "**`TODO: SquirtTest() -- Use a suitable default value.`**" Task, change "**`var actual = 0;`**" to "**`var actual = 0f;`**", declaring **`actual`** to be a **`float`**.

► At the "**`TODO: SquirtTest() -- Provide a suitable calling expression`**" Task, insert "**`Module1.`**" before "**`Squirt`**", so that it will read thus:

```
            actual = Module1.Squirt(tCase.Arg);
```

The VS editor should offer help with the names after you begin typing:



► At the "**`TODO: TestMethodsToBeRun`**" Task, in the list, replace the names of any listed TDS methods with

```
SquirtTest()
```

[102] If this snippet is undefined, please see section 4.4.4.

This name is case sensitive, but the parentheses are optional. As ususal, the names of TDS methods that you are repacing may be commented out instead of being erased.

Since we erased or commented out the names of the example TDS methods in section 4.14.7.3, "**`SquirtTest()`**" should be the only one active.

5.4.4 Run the TDS tests

► Optionally, hide (as shown in section 4.4.2) the "TDS.AssertInconclusiveException" pop-up that we expect will appear if we don't hide it.

► Run the Project (use <F5>).

The TDS method **`SquirtTest()`** should return "**`Inconclusive`**".

► If a "TDS.AssertFailedException" or a "TDS.AssertInconclusiveException" dialog box pops up, hide that exception type as we did in section 4.4.2.1: in "Exception Settings", uncheck "Break when this exception type is thrown", close the dialog box, and resume running TDS (via <F5>).

Some of the above housekeeping chores will not need to be repeated when you add other TDS methods at the end of TDS.cs, to test other function members in the working code. For example, it is likely that the TDS Project's list of References (including the working code's namespace) will still be valid, the new form of the "**`using`**" statement will still be correct, the **`Assert`** exceptions will already have been hidden, and the revised list of TDS files will still be correct.

5.4.5 Clean up the code

► When you have finished examining the results and have no further use for the new VS Solution that you cconstructed for this example, you may delete it.

5.4.6 Re-enable all TDS tests

► De-comment all of the TDS test names in **`TestMethodsToBeRun[]`**, and include the names of any TDS methods listed near the end of the TDS test report following this heading, if it be present:

```
The following TDS methods have [TestMethod] attributes
    but are not in the TestMethodsToBeRun list:
```

► Run TDS to get a complete TDS test report.

You may navigate there via the "TODO: TestMethodsToBeRun -- List all TDS test methods to be run." Task.

If, after running the tests, you do not see the message "**`All listed TDS test methods passed.`**" near the end of the Console output, check the rest of the TDS test report to determine what did not "Pass", so you can correct it.

► Close the Console window.

6 That's all for now

I claim (even though I haven't actually demonstrated it in these examples — maybe in a later version) that TDS can be used to help develop not only methods but also various other kinds of function members, such as properties, indexers, events, operators, or constructors. Methods seemed easiest to use as illustrations in these examples, so those are most of what you've seen here. As you saw, each method being developed was invoked in a statement such as

```
actual = …
```

where the method call followed the "=" sign.

Adapting TDS for use with other categories of function members would involve suitable changes to this statement. Even if they don't explicitly return values, they could be made testable by, for example, changing the values of fields that are accessible to the TDS methods.

Using TDS, even absent any plans you might have for for unit testing, is intended to help you standardize the inputs to function members under development, by packing these values into the **`testValues[]`** arrays of TDS methods.

If you intend to unit test your function members, using TDS can also help you to standardize their expected output values, again using the **`testValues[]`** array, and to organize the **`Assert`** statements, packing those into the "**`#region Apply tests`**" region, where they can easily be managed.

If you *don't* intend to do any unit testing, please let me suggest one last time that unit testing means never having to say, "I'm sorry," or in some shops never having to buy a round of raspberry-filled jelly doughnuts for the rest of the shop because it was *your* code that broke the build! What, never? Well… hardly ever. I suggest that you save the doughnut buying for a happy occasion like someone's birthday or promotion.

OK, these are enough suggestions. I hope you find that TDS saves you time and effort. Good luck!

7 Glossary

The following terms are used, in some cases, with senses peculiar to this *TDS User's Guide*. More detailed descriptions of their uses may be found in the referenced sections of the document.

Term	Definition/description	References
Analysis	Mathematical or logical examination of a problem intended to assist in the design of a computer program. Combined with testing of prototype code, this helps to verify that the software model of a problem reflects the nature of the problem sufficiently well to provide a reliable solution.	Section 5.2.7
Buggifying	Intentionally altering working code to return false results, to verify that the code testing it properly detects and reports on faults. (Caution: This is not a standard term, and you might look silly if you use it too freely!)	Sections 4.6, 5.2.9.3
"Easter egg"	Undocumented feature of a computer program, intended to be amusing and to be discovered accidentally in the course of using the software. (The TDS software does not contain any known Easter eggs.)	Section 5
filtering TDS methods	Choosing a selected subset of TDS methods to be run, to temporarily display only selected test results in the report. This may be accomplished by editing the list of TDS method names in the TestMethodsToBeRun string.	Section 4.8.2.5
filtering test cases	Choosing only a selected subset of test cases (elements of the testValues[] array) to be run, to temporarily display only the results of running the working code using the selected values. This may be accomplished by editing the testSelectionList string in a TDS method and using a "#define RunOnlySelectedTestData" C# pre-processing directive in the C# file containing that TDS method.	Section 4.8.7
happy path	The default path through the working code. This path should work even if nothing else in the code does. I usually make this the subject of my first test case in a TDS method.	Section 1.10.5
Id tag	Label in a testValues[] element that is used to identify a test case; it may be listed in failure messages in test reports to assist in locating the source of a failure.	Sections 4.8.3.4, 4.8.7
Iff	Abbreviation for the phrase "if and only if".	Various places in code and in text
platform, unit-test	Automatic system supporting unit tests; in this document, besides TDS itself, only the Microsoft Unit Testing Framework and NUnit Framework are described. Also called "framework".	Section 4.5 and elsewhere

Term	Definition/description	References
Refactoring	Altering executable code in a way that does not cause any unwanted change in the code's behavior. Unit tests may help to expose unwanted side-effects of an attempted refactoring.	Section 5.3.1
schema (XSD) file	Specification of the structure of an XML file, used to confirm that the XML file is well formed.	Sections 1.7, 5.3.3
smoke test	In hardware development, a "smoke test" involves applying power to a prototype circuit laid out on a breadboard. If one can see or smell smoke from a too-hot component, this indicates a serious problem that needs to be corrected before continuing.	Section 4.3.6.3
Stub	Stubby subprogram, a placeholder in code for a function member to be developed, consisting of comments that resemble program statements but do nothing useful. This serves as a reminder of work to be done.	Sections 1.8.1, 4.10.3
TDD (Test-Driven Development)	A traditional software-development methodology that involves specifying tests of functionality of to-be-developed function members. These tests are expected to be operational before those function members are written. (In contrast, TDS also supports concurrent development of working code and tests of that code.)	Section 1.8.1
TDS	"Test-driven scaffolding", program code usable as templates for test methods for function members of C# types, plus code supporting basic tests of those function members.	Section 1.4
testing	Running executable code in a (one hopes) realistic environment and comparing the expected results of running it with the actual results. Combined with analysis of the problem, this is intended to help detect flaws in the implementation of a problem solution.	Section 5.2.8.3.3
testValues[]	In a TDS method, the default name for the array of test-case objects whose properties or fields specify values of parameters for calling working code and values of expected results of executing those calls.	Sections 4.8.3.4, 4.8.3.1
TOC	Table of contents	Table of Contents, Section 2.3.4.1.1
TODO comments	Comments in C# code identifying tasks to be displayed in the Visual Studio "Task List" window. Most such comments in the examples begin with "//TODO:", but a few begin instead with "//HACK:" to indicate that they are present only as examples to be deleted after being observed.	Sections 3.3, 4.14.16

Term	Definition/description	References
working code	Term used in this *TDS User's Guide* for code in function members that is being developed with the help of TDS methods; this distinguishes it from the C# code in TDS methods that interacts with the function members by invoking them or reporting on the results of running them.	Sections **Error! Reference source not found.**, 1.8.1, 2.1.1
VS	Abbreviation, in this *TDS User's Guide*, for Microsoft® Visual Studio®.	Section 1.4

8 Subject Index

Selected topics are listed here if there is some discussion of them in the text. If what you seek is not here, you might check the Table of Contents or use a "Find" or "Search" operation on the document.

"► " flag, used for actions 25

#region, used to hide code 144

//// in comments 125

`//TODO:` ..... *See* `TODO` comments

`Action`, compared with method 182

Adobe Reader 25, 26, 27

Alarm, false 186

analysis compared with testing 173, 180

anonymous testValues[], converting to named .. 190, 248

atomicity of operations 232

braces, untangling mismatched ..... *See* comment, on closing brace

buggifying tested code 58, 268

Bunnies, Fibonacci 151

comment, on closing brace 137

comments on properties ..... *See* properties (`testValues[]`)

comments, documentation ..... *See* XML comments

comparing text, using FC

  white space 269

Console window, copying text from 67

Contents, table iii

copying code from this document 25

database keys, primary 207

default values

  in constructor definition 198

escaping special HTML characters ..... *See* HTML, escaping

FC.exe ..... *See* comparing text, using FC

Fibonacci sequence 151

filtering test cases 98, 169

fonts used in this document 25

framework, unit-test ..... *See* platform, unit-test

generating functions 152

Golden Ratio 157

happy path 22

housekeeping ..... *See* refactoring subexpressions

HTML, escaping, in XML comments 139

InsertSymbol() first example

  tree view 227

  XML Viewer view 230

Knuth, Donald 152

links ..... *See* navigating this document

maintenance, test-method 20

methods vs. `Actions`. *See* `Action`, compared with method

named `testValues[]` ..... *See* anonymous

navigating this document 26

Object Browser

  navigating 227

  using 225

  viewing XML comments in 111

overloading

  constructors 206

  `Id` tags 207, 247

phi or "φ" ..... *See* Golden Ratio

platform, unit-test 46
primary keys ..... *See* database keys, primary
properties (`testValues[]`)
  comments on 115, 166
  naming conventions 115
refactoring
  subexpressions 209, 258
  TDS methods into tests 46
requirements
  incomplete/fluid/malleable 211
  specification of 128, 158, 211
Schema (XSD) file ..... *See* XSD (Schema) file
smoke test 37, 216
Split the editing window 183, 192, 193
stub 15, 129
TDD (Test-Driven Development) 14, 22, 134
TDS method 11
testing vs. analysis ..... *See* analysis compared with testing
`testValues[]` properties ..... *See* properties
`throw` statement, replacing 143
TOC iii
`TODO` comments 122, 141
tracing ..... *See* variables, observing values
typography 25
Unicode, saving copied text as 26
unit tests, reasons for performing 145
variables, observing values while tracing 142
working code 11, 15, 18
wrapper method 81
XML comments
  IntelliSense display of 140
  viewing in Object Browser 111
XML editing
  literals in C# 236
  using XML editor in VS 237
XSD (schema) file
  description 217