1 Introduction to VSTEST and VSTESD

As part of introducing you to Visual Studio Team Edition for Software Testers (VSTEST) and Visual Studio Team Edition for Software Developers (VSTESD), we need to see where these tools fit within the life cycle of software development. To give you, the reader, a better understanding of this, we're going to discuss briefly the general software development process.

The Software Development Process and Software Development Life Cycle (SDLC)

The software development processes provide a general framework for developing software. The ''software development process'' in and of itself is not a set of guidelines by which to develop software. Rather, it's a set of terms in which you can describe and discuss many different development methodologies — Waterfall, CMMI, Scrum, and eXtreme are all different software development processes that vary greatly in their actual implementation but share common phases in one form or another. These are all under the larger umbrella of ''Software Development Life Cycle (SDLC).''

Often, these different processes are formalized by different organizations, and sometimes government organizations. CMMI, for example, is managed by the Software Engineering Institute, or Rational Unified Process by IBM.

Planning, Analysis, Design, and Development

Within any software design process, there are many common sections. Here, we'll discuss in brief the sections in which VSTEST and VSTESD do not play a huge role. These are catered for by many tools, and in a large part by the rest of the members of the Visual Studio Team System set of tools from Microsoft (Team Foundation Server, Team Architect, Database Professional) and other companies (Borland, IBM).

Planning

Planning is the phase in which you decide what needs to be built (e.g., the problem that the ultimate end product will solve) and who is to be involved in the project, set target dates, and make other decisions that need to be determined before you can start developing an application.

You must understand at a very high level the problem you are trying to solve. You will also begin to set some preliminary dates and time periods for the task at hand.

The planning phase is something that varies significantly from business to business, and project to project. With the ever increasing desire of teams to produce more and produce it faster, agile processes are changing how people think about planning. Instead of it being a lengthy process in which many details are set out (dates, what language to use, etc.), they are being moved out in the development process to be considered closer to the point at which they are needed, rather than in a disconnected process that may have little or no context related to the questions that need to be answered at the time.

Analysis

During the analysis phase, you gain knowledge of what the software needs to do to be considered successful. Through gathering of requirements and analysis of these requirements, you can scope the project into that which is truly needed to solve the problems that your customers (be they internal or external) need to be solved.

There are tools and processes in the marketplace that can help you manage your requirements and your requirements gathering. These provide a great way to track requirements against actual implementation — something that can be lost in the heat of software development. Some of these tools include CaliberRM from Borland and MindManager from Mindjet software. CaliberRM has integration into TFS to provide a rich linking interface with other work items such as test results, bugs, and any other work item type.

Design

After one has collected and digested the requirements, it is possible to start designing the application. This ranges from the high-level component architecture to the layout of the UI. This could easily stray into the development of the project, but ideally one usually avoids the nitty-gritty at this stage.

Development and Testing

These two phases are where, in case you hadn't deduced it, the Visual Studio toolset can really help. However, we'll save that discussion for later. Right now, it's important to note that both of these phases are catered to extensively by Visual Studio. Development is the process of taking your design and turning it into a real living and breathing piece of software. This is what we all know and love as ''programming'' — writing lines of code, creating UI, and building new functionality and experiences.

However, once you've built your application, you need to test it. How you do this varies widely from complex, forward-thinking processes to simple exploratory testing. Each different type of testing brings something unique to the table, along with many different advantages and disadvantages. Some processes don't have a formal testing phase; however, somewhere there is testing.

Implementation and Maintenance

Once you've built an actual system, your job is not done. You need to deploy the system to your customers so that they can reap the benefits of the wonderful application you have created. Often this is called deployment or implementation, in which you roll out the application in your customer's environment and your users use it every day, which will undoubtedly result in issues and defects being reported. This neatly brings us to maintenance.

Even after your application has shipped, its development is not complete. There will be issues with your application once it's been deployed to customers, and these often will need to be resolved. This is where you will need to maintain your application.

Existing Tools in the Marketplace

Testing has been around since the first computer programmers. This has often been a painful and laborious process: calling multiple functions with different inputs; moving through long, drawn out steps in an end-to-end scenario; making specific HTTP requests — the list goes on. Because of the need to perform these actions often and repeatedly, several different tools have been created over the years. We're going to take a look at some of the most common tools that match VSTEST for functionality, specifically unit testing and web/load testing. There are other features around testing and development, but for the purposes of this discussion, these are the ones that are standout features.

Unit Testing (NUnit, MbUnit)

Unit testing has been recently popularized by Kent Beck as part of eXtreme programming. By no means does this mean that the only use of unit testing is within eXtreme programming — it has many uses in many different development processes ranging from the standard waterfall model to the most agile processes today. It even has a place in the hobbyist developer's toolbox, providing a great way to validate the units of your code in an automated and repeatable fashion that, no matter how small or large your development project is, provides an enormous amount of value.

The name *unit testing* is the key to understanding what unit testing brings to the table: the ability to test individual units of code. A class, function, SQL stored procedure, web page, or even a specific part of a protocol (or maybe the whole protocol) can be considered a ''unit.'' Similar to the way the manufacturing world may have different units that combine to produce a complete product, units combine to form the complete application.

Using this approach, you can see that, as you build the units of your application, creating a set of validation tests to validate your components can give you significantly higher confidence when combining those units later on. By extension, this allows you to develop and fix bugs with greater confidence.

It is important to note that unit testing does not always have to be about testing specific units. It can also be used to perform any other type of programmatic testing. Examples include the way in which the VSTEST team made use of the unit testing features in VSTEST as the foundation for testing all the areas of the products, not only for the developer unit tests, but also for writing scenario tests, integration tests, UI tests, web tests, and even database tests, all using the VSTEST tools and technology.

NUnit

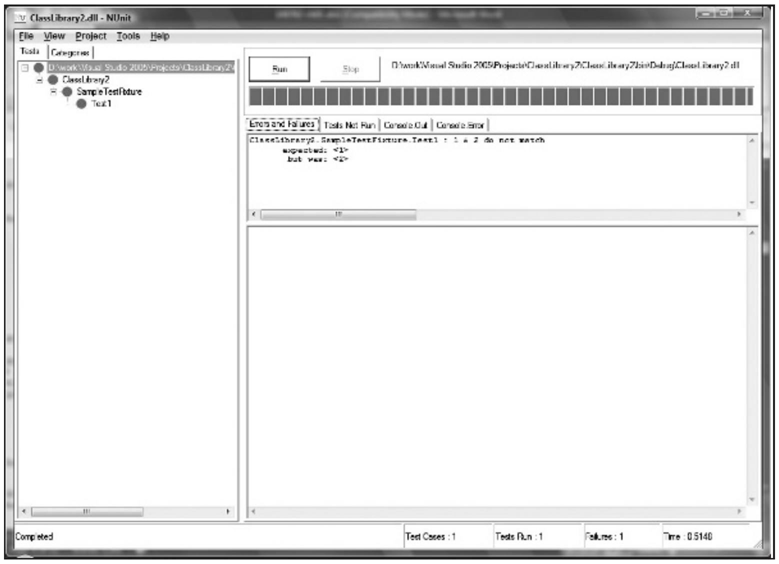
NUnit is the .NET port of the xUnit framework created by Kent Beck. However, it is more than a straight port — the NUnit team embraced some of the advantages provided by the .NET platform to create a much more holistic feel to the tests that are written. As a platform, NUnit provides a command line and GUI runner with different user experiences depending on the intended use. Additionally, it provides attributes, assertion classes (above the standard `Debug.Assert` classes), and configuration files for manipulating the environment for running the tests.

At the simplest level, a unit test within NUnit is simply a method appropriately attributed to indicate that they should be considered tests. However, those methods must reside within a class that has a different set of attributes on it. Here is a canonical example:

```
[TestFixture]
public class SampleTestFixture
{
    [Test]
    public void Test1()
    {
        Assert.Fail("This test does nothing meaningful");
    }
}
```

As one can see, the attributes make it easy to create and identify the test classes and test functions within a set of source code.

When you run this test in the GUI runner, you get an output like that in Figure 1-1.


ClassLibrary2.dll - NUnit
File View Project Tools Help
Tests Categories
D:\work\Visual Studio 2005\Projects\ClassLibrary2\
ClassLibrary2
SampleTestFixture
Test1
Run
Stop
D:\work\Visual Studio 2005\Projects\ClassLibrary2\ClassLibrary2\bin\Debug\ClassLibrary2.dll
Errors and Failures
Tests Not Run
Console Out
Console Error
Completed
Test Cases : 1
Tests Run : 1
Failures : 1
Time : 0.5140

```
ClassLibrary2.SampleTestFixture.Test1 : 1 & 2 do not match
        expected: <1>
         but was: <2>
```

Figure 1-1

As can be seen, you are provided with a list of test classes, and the tests contained within those test classes. You can run at the different levels (assembly, namespace, class) and see your results in the right-hand pane. As you can see from the sample test, our single test case failed — with the error message that was provided to the `Assert.Fail` call. This allows for easier diagnosis when your tests fail; having an error like

```
ClassLibrary2.SampleTestFixture.Test1 :
    expected: <1>
     but was: <2>
```

is unhelpful. However, an error like

```
ClassLibrary2.SampleTestFixture.Test1 : 1 & 2 do not match
    expected: <1>
     but was: <2>
```

is much more helpful, providing you with some context as to what the assertion was actually intended to validate, which can be invaluable when trying to diagnose test-case failures with code that you do not understand well. In the case of one specific test — or even tens of tests — this is not an issue, as the set of code that you have to understand is small. But when you have many hundreds of test cases, the code that you need to understand to diagnose and fix the failures may be prohibitively large.

In addition to the standard ''unit testing'' duties of providing an execution environment and assertion framework, NUnit actually goes to the next level by providing unit testing for both Windows forms and ASP.NET. These provide simple, but powerful, object model type unit testing functionally for the Windows forms and ASP.NET frameworks. It should be noted that these are not functional tests that simulate user clicks and mouse movements, but rather, they merely invoke and wrap the necessary methods that are required to simulate the events being fired on the controls. This is often just as powerful as functional testing, but it has its own set of caveats.

The last remaining piece of the NUnit puzzle is that it provides mock objects' functionality. Mock objects allow you to simulate the implementation of an interface or class without having to do the full implementation of that interface or class. This is performed by providing skeletal code that allows your real code to function as though it were using a real implementation. This is extremely powerful for validating when your code is consuming interfaces without having the potential headaches of a real implementation (e.g., IHeartMonitor is difficult to test without a real heart monitor).

MbUnit

On the surface, MbUnit looks like any other unit testing framework: It provides assertions, declarative markup, and execution tools. However, MbUnit is potentially very much more powerful than the other unit testing frameworks. This is because of the focus that MbUnit has on extensibility and allowing people easily and simply to extend and customize how tests are executed. The most obvious scenario is data-driven tests, which with MbUnit can be declared inline:

```
[TestFixture]
public class DivisionFixture
{
    [RowTest]
    [Row(1000,10,100.0000)]
    [Row(-1000,10,-100.0000)]
    public void DivTest(double num, double denom, double result)
```

```
        {
            Assert.AreEqual(result, num / denom, 0.00001 );
        }
    }
```

As can be seen, you can simply, easily, and clearly declare that a test is minimally data driven while providing a small set of values to be passed into the test. Additionally, because of the strong typing on the test methods' parameters, the actual test can be more clearly written. It's important to remember that these types of extensions can be written such that your custom attributes appear as first class citizens within your tests. This means that rather than having some alternative and unclear way of setting test behavior, it is right there with the test in a way that is integrated and clear. This gives a very different experience from creating your own solutions to the same problem, which results in you ending up with code that just isn't as clear.

It is interesting to note that it is the NUnit model that VSTEST is following rather than MbUnit. MbUnit is a recent entrant into the market and has certainly shaken it up somewhat, with the NUnit team looking closely to see what can be learned in the area of extensibility.

Web and Load Testing

With today's typical application being more than just a single form-based application written in Visual Basic sitting on a single user's computer, load testing has become a much more significant part of the software life cycle. This combined with the primary platform these days being a web-based one, web testing and load testing are the most common types of non-developer-orientated testing being done by organizations. Within this market there is only one clear leader today — Mercury Load Runner. This is a very expensive product, but it provides the whole gamut of load and web testing, from simple http testing through to SOAP testing, and profiling of those services.

Additionally, the Mercury toolset provides a tight integration across a wide range of testing tools, from test-case management to business process testing.

Where VSTEST and VSTESD Fit in the SDLC

The Visual Studio Team System attempts to provide tools and technology to cover the whole life cycle — bug/work item tracking, source code control, automated build system, testing, architecture, and so on. From the names *Visual Studio Team Edition for Software Developers* and *Visual Studio Team Edition for Software Testers*, it should be clear in which part of the SDLC these products fit — testing and developing.

VSTEST provides an ability to author tests, run tests, and, most importantly in the context of SDLC, publish and share the results with your team and the management of your team. This is thanks to the reporting functionality of the Team Foundation Server (TFS), which allows you to create reports covering all aspects of the data within TFS — Source Control checkins (number of files, number of changed lines, changes, code reviewer), work item changes, code coverage, and test results — over time within a SQL Reporting Services environment.

Where you see *testing* as part of your software life cycle (no matter what specific methodology it may be), VSTEST is a product that can help you fulfill those tasks. It's also important to note that SDLC is a huge topic that could be a separate book on its own. However, in the context of the Team System, we have covered the SDLC in more detail in Chapter 9.

What VSTEST and VSTESD Do

Visual Studio Team Edition for Software Testers and Visual Studio Team Edition for Software Developers are editions of Microsoft Visual Studio that go above and beyond the Visual Studio Professional Edition experience by providing role-specific extensions to the IDE experience. VSTEST is specifically targeted at the tester role by providing a set of features for testing applications and services. VSTESD provides tools enabling people in developer roles to have a greater understanding of their code — through performance, code analysis tools, and unit testing. This section only provides a very high-level overview of the features in VSTEST and VSTESD. For more information, see Chapter 2 (a general overview of VSTEST and VSTESD) and the specific chapters and appendixes for the specific areas of functionality.

VSTEST Features

With the focus in VSTEST being on testing, one can see that all of the extra functionality is testing based, focusing on several different test types that the user can take advantage of to test his or her application. Some of these are useful to the developer-focused tester (such as unit testing and web testing), while others are very much for the non-developer-focused tester (such as load testing and manual tests).

The experience for the user is intended to be the same across all test types — whether you are running a unit test or a load test, the basic experience is intended to be the same. This enables you to take advantage of a core set of experience when working with tests in VSTEST. Later in this book, all the test types and areas of the product will be covered in much more detail, where some of this shared user experience will become evident.

Unit Testing

The unit testing functionality provided by VSTEST is very similar to that of NUnit. You author the actual test code in the same fundamental way as NUnit, by adding appropriate attributes to the methods and classes that you wish to be considered tests. Additionally, unit testing in VSTEST supports being data driven, and also another type of test called ASP.NET tests. These are not to be confused with web tests. All of the main Microsoft programming languages are supported out of the box — Visual Basic, C#, and C++/CLI (formerly called Managed C++). Other languages are also supported; however, there is no integrated IDE support for running tests in those languages. An example here might be Visual J#: You can author tests in this language, but you will have to use the command line tools to execute these tests.

The data-driven support revolves around each test being executed *N* times (where *N* is the number of rows in a database table) and the test being provided with the data row for each invocation. Any ADO.NET consumable database may be used for data driving a unit test; however, XML (as an arbitrary markup) is not supported.

The ASP.NET support is twofold. Firstly, it allows your tests to be executed under the ASP.NET runtime environment, allowing access to all the standard System.Web namespace objects you have access to in a normal ASP.NET application. Secondly, your test has access to the System.Web.Forms.Page object that represents the page being requested for that unit test. This allows access to the controls, methods, and other items that are relevant to unit testing ASP.NET code.

More detailed information on unit testing can be found in Chapter 3, which covers all aspects of unit testing in VSTEST and VSTESD. Also, database unit testing is covered in detail in Chapter 4.

Web Testing

The underlying technology used to implement the web testing functionality in VSTEST is HTTP requests. The testing that VSTEST provides here is not a browser-based web testing solution — it won't drive your mouse clicks through your HTML and script. However, it is a very powerful HTTP-based web testing solution, providing everything from an easy-to-use browser-based recorder to a set of complex extensions for data validation of the returned data (extraction and validation rules), and being drivable from a database.

The Web Test Recorder allows you to simply record a web test by navigating between pages, filling any forms along the way. All the data are captured and turned into a full web test, which can be customized to your heart's content.

At a basic level, as long as your requests return ''OK'' responses as determined by the HTTP response code, then your tests will pass. But as is always the case, this is unlikely to meet your needs, which is where extraction and validation rules become useful. They allow you to extract certain pieces of information from the HTTP response and also to validate that response against a set of criteria. Additionally, it is possible to create your own rules — these require programming, but can be very powerful. All parts of the request (URL, form data, etc.) can be driven out of a database to ensure that not every test is identical. An example is if you have a URL that takes a query string parameter (such as `http://localhost/viewOrder.aspx?orderid = 1234`), you can use a database to fill in the ''1234'' part with different order ID, using the URL as a template.

Finally, any web test can be turned into a ''coded Web test,'' a code-based web test that will allow the authors of a test to have extremely fine-grained control over the test. This includes request parameters, choices/branches based on certain responses, and anything else one chooses to put in code.

Chapter 5 contains a deep overview of all the web testing functionality to be found in VSTEST. A walk-through of creating your first unit test can be found in Appendix C, which is a step-by-step guide showing you details of each step.

Load Testing

Within VSTEST, there are many test types, and those test types can be extended by third parties. It may be that many of these test types are exactly the kind of test that you would like to run against a server as a set of load tests. Both unit tests and web tests fit this criterion out of the box, and one can imagine many other types of tests that might fulfill the same role.

The load test functionality in VSTEST and VSTESD is a container test type. It contains many other tests (of any test type) and allows the user to set parameters for execution (number of users, variance over time). This powerful feature allows for great flexibility in how you author your tests. It allows you to author the test in the most appropriate way, rather than having to shoehorn one type of testing paradigm into another, that is, web tests as unit tests.

As a load test is executed, it will collect a large range of Windows performance counters from the local machine, any agents being used to distribute the test load, and any servers that the user specifies. All of these counters are stored for later review, allowing you to see bottlenecks and drill deeper into any that are found. The remote agent and controller functionality in VSTEST and VSTESD exists primarily to support running load tests under high load.

Chapter 7 contains a thorough discussion of the load testing functionality found in VSTEST, including discussion of analyzing the graphs of performance. Appendix D contains a walk-through for creating your first load test. Additionally, there is a discussion of performance analysis in Chapter 8 in the context of profiling your application.

Manual Testing

The manual testing functionality in VSTEST is somewhat limited, as it only provides a simple text document of steps that need to be performed along with pass/fail results and user comments. This is useful for small-scale projects but does not really scale out to managing thousands of test cases that need to be performed by many users who need to report bugs, screenshots, issues, and other pieces of information that may be relevant to the test.

It is important to note that there are two types of manual test in VSTEST, text-based and Microsoft Word-based. The text-based tests are just that, simple text documents. The Word-based tests allow you to author the test documents in Microsoft Word in any way that can be exported to HTML. Note that one area of the HTML that is restricted is the support for inline ActiveX objects — this is to ensure a secure environment when running manual tests.

Chapter 6 covers manual testing in more detail, with a walk-through for creating manual tests being provided in Appendix B.

Generic Testing

One problem that the Visual Studio Team Test development team encountered during the development was requests from users to be able to integrate their existing custom test harnesses into the Visual Studio Team System. People were looking for a low cost of entry, and a custom test type did not have this. Additionally, the existence of custom test types implies a desire to keep authoring tests in that manner. Many customers wanted to switch over to the Visual Studio Team System without having to lose many thousands of test cases. The solution was generic tests.

Generic tests are a test type that merely invokes a given executable with a specific environment and waits for the process to terminate, and depending on the return code determine a pass/fail of that test. This allows many automated tests to be integrated into VSTEST quickly and easily.

In the case of the pass/fail, or one high-level result not being fine-grained enough, the test harness can also output XML conforming to a specific schema. This output will be interpreted to provide detailed results and better integration of legacy test harnesses.

Chapter 6 provides detailed coverage of generic tests, and a walk-through is provided in Appendix E.

Ordered Testing

The general approach that VSTEST takes to the execution order of your tests is ''undefined'': Your tests can execute in a different order between runs, and VSTEST makes no guarantee of that order. However, there are times when you want your tests to execute in a very specific order. Take the example of unit testing databases, either stored procedures or a data access layer, for which you may have expensive test cases that make significant changes to the contents of the database. While you can back up/restore the state of the database, this could significantly increase the time to run your tests.

Ordered tests are the VSTEST solution to this problem. They allow you to combine any set of tests (including different types, such as unit with web tests) and specify an explicit order for them to run in. You can also choose to have the ordered test ‘‘fail’’ if any of its contained (or ‘‘inner’’) tests fail, or it can continue. When you run the tests, the order is as you explicitly specified it. When you are viewing results, you have to drill down into the ordered test results to see any specific results of the contained tests.

Chapter 6 covers ordered testing detail, and Appendix F provides a walk-through of creating ordered tests.

Code Coverage

The final high-level feature of VSTEST is code coverage. Code coverage allows you to see which parts of your application are being executed by your automated and manual test cases. It provides line-by-line information as to which have been executed and which have not. This allows for some great insights — both from the ‘‘we’re not testing enough’’ and the ‘‘this code is never executed, it is dead code’’ perspectives — providing valuable insight into your testing and your application.

Code coverage works through instrumenting the application binaries on disk. One of the things VSTEST can do is to instrument the binaries each time you run tests to ensure that your coverage is collected.

VSTESD Features

It’s important to note that the only shared feature area between VSTEST and VSTESD is unit testing (and ordered testing). This is the exact same feature that you see in VSTEST, but transplanted into VSTESD. It is the only part of the testing functionality that is available in VSTESD. The rest of the features are targeted toward developers.

Code Analysis

Static analysis or code analysis is a technique for inspecting the correctness of applications without actually executing the applications. This can include inspecting source code directly or using tools to examine the binary code directly. Within VSTESD, there are two sets of tools to analyze your code: PREfast (for C++) and FxCop, referred to as *Code Analysis* inside the Visual Studio IDE.

Both of these tools check for common programming errors and validate against a set of rules, which in the case of FxCop are extensible to enable an organization to enforce its own common programming rules. These may be as simple as spelling errors or naming conventions but also extend to ensuring that certain code-based security practices are taken into account.

Profiling

When your application is having a performance problem, it can often be very difficult to identify where the bottleneck is at the code level. Through the use of profiling, one can find these bottlenecks. This involves either running your application through the scenarios that show the performance bottleneck while sampling CPU-based ‘‘counters,’’ or using instrumentation. After the data have been collected, the Visual Studio toolset will allow you to see where that bottleneck is.

Main Differences Between VSTEST and VSTESD

It should now be clear that there is a large difference between the feature sets of VSTEST and VSTESD. When one steps back, the sharp demarcation between the intended roles for the two editions can be seen — testing versus development.

VSTEST provides a set of testing tools that encompass all aspects of testing — test types (from web, to unit, to manual testing), remote execution, and simple test-case management. On the other hand, VSTESD provides tools enabling you to write better applications as a developer — unit testing, profiling, and code analysis.

Feature	VSTEST	VSTESD
Unit testing	X	X
Web testing	X	
Load testing	X	
Generic testing	X	
Manual testing	X	
Ordered testing	X	X
Code coverage	X	X
Test manager	X	
Remote execution	X	
Code analysis		X
Profiling		X

Remote execution is a feature of VSTEST that enables you to run your tests on a remote set of machines — an *agent* in the terms of the product. However, all the components to run your tests remotely are not provided with VSTEST, and additional software is required. The Visual Studio 2005 Team Test Load Agent product provides both Controller and Agent components of remote execution that are licensed per CPU. Installing these two pieces of software is a simple process, and enabling remote execution from a VSTEST is as simple as editing the Test Run Configuration to set the remote machine to execute on. For more details about editing the test run configuration, see Chapter 2. It is important to note that the Load Test Agent is not a free product and must be purchased.

How Do They Help the SDLC?

One of the biggest challenges in any software team is being able to manage and track the work that is being undertaken. In the context of testing, this means being able to see your results over time as well as being able to actually perform the work that is required (i.e., test your product).

VSTEST helps you achieve this through its many testing features: unit testing, web testing, load testing, manual testing, generic testing, result publishing, and code coverage. Each of these individual components allows you to fulfill a different part of the SDLC, but when they are combined, they allow you to succeed with your testing across your entire product.

It’s clear that the ‘‘Testing’’ sections allow you to fulfill the need to test your product. However, the other parts are less clear. Results publishing is a good example of where it goes beyond simply getting the results into some other system to be reviewed by others. It allows, in combination with the Team Foundation Server, tracking and analysis of results over time. This should not be underestimated.

While seeing that you have 90 percent of your tests passing today is great, you need context to see what has happened over time. What were the results last week? What were the results a month ago? Tomorrow, when the results show only a 10 percent passing rate, you can see that this is a significant change from the previous day. This moves you from merely testing your product into being able to know about the state of your product *over time*. You can then plan throughout the product cycle, knowing your history and trends across both testing and development.

The key factor here is that there is an integrated experience across the whole of the Visual Studio Team System products that provides an integrated experience for the whole of the SDLC, and VSTEST dovetails into this by providing a great solution for testing.

Why Choose VSTEST Over Other Tool Sets?

The primary advantage VSTEST provides over other tool sets is the integration into Visual Studio. No other set of tools integrates into Visual Studio as VSTEST does. It is a first class citizen within the IDE, providing high-level windows for authoring, executing, and managing tests that have a consistent look and feel with the rest of the Visual Studio IDE. Although there might be programs in the industry that can compete in a limited capacity, when you look at VSTEST and VSTESD as part of an integrated solution, their power and reliability are much more compelling than a set of cobbled together tools from around the industry.

Integration into the IDE

Within the IDE, VSTEST provides several integration points: test projects and the Test View, Test Manager, and Test Results windows. These all serve a specific purpose. Note that Test Manager is not available in VSTESD — it is a VSTEST-only feature. These windows are what you need to know about to get around the product. For a deeper look at the individual parts of the IDE, take a look at Chapter 2, which will walk you through all the basic functionality of VSTEST and VSTESD.

Test Project

Test projects are the biggest unit of granularity when it comes to managing your tests. A test project is just like a class library project, except that it holds tests as well as source code. It provides all the normal project features (source control, folders, debugging, etc.) and comes in the three language flavors — Visual Basic, C#, and C++/CLI. You can find test projects under File ➪New ➪ Project ➪ <Language> ➪ Test in the Visual Studio IDE. When looking in the other test windows, it's important to note that only tests in test projects in the currently open solution will show up.

Test View

Test View is the primary window for viewing and running tests. It provides a linear, flat list of all the tests currently loaded. Double clicking a test in this window will open the test in its editor. You can use the Test View toolbar for running, debugging, grouping (one level tree like view), and filtering the list of tests. You can access this window from either the Test toolbar, or from the Test ➪ Windows menu item at the top of the IDE.

Test Manager

Categorization is an important factor when you have many hundreds of tests, and it is Test Manager that provides a simple way to categorize your tests. It does this through a concept of test lists, and the use of a ''metadata'' file that is shared across the solution. It provides a very similar interface to Test View, but with the added functionality of categorization, both for viewing and running the tests.

Test Results

This is the primary window where you will monitor and view your results. It provides both a flat list as well as a category view (a la Test Manager), with the same filtering/grouping functionality from Test View/Test Manager. You can re-run, pause, and stop a run that is currently in progress, as well as see results that have already completed during a run in progress. One important feature when you are using remote agents is the ability to disconnect from the run. This means you don't need to be connected for a 12-hour load test run. You can also save specific results and full runs from here for sharing with your team.

Integration with TFS

The final part of the VSTEST puzzle when being compared against other toolsets, is the integration with the Team Foundation Server. VSTEST provides several valuable integration points with TFS, allowing you to get test data into TFS easily. A step-by-step guide for installing and using the Team Explorer functionality of TFS can be found in Appendix A.

The simplest of these is being able to file a bug directly from a test result: When you see that failing test case, you can right-click on the test result and select ''Create Work Item.'' This will open the Create Bug form in TFS, pre-filling in the title and description with error details and attaching the results file to the bug. This means that when someone else comes to look at the bug, they have all the relevant information required to take a first pass at investigating the bug without having to try re-running the test.

Secondly, and arguably much more complicated, is the ability to publish your results into TFS. These results are associated with a build within TFS and are tracked as part of those builds over time. You can also associate failing test cases with bugs and see which bugs are tracking which failures at any given time. As you publish your results, you slowly build up a historical view over time, being able to see the total number of test cases, pass/fail rates, trending, and code coverage numbers for your test cases over time. While initially the data appear to be mostly academic in nature, they allow you to start building reports in which you can compare the actual performance of your team against the predicted/required performance to hit milestones.

Summary

The SDLC is a complex and huge area of discussion whose surface cannot even begin to be scratched with what we've covered here. However, we have shown that there are several areas where testing can help improve the quality of your applications by allowing those tools to fulfill parts of the SDLC.

We've also looked at the tools that are out there other than VSTEST and VSTESD, and what unique advantages those tools have. Also, we took a very brief overview of what features VSTEST and VSTESD have and what they bring specifically to the SDLC.

With this basic understanding of VSTEST and VSTESD, the context for discussion of the specific features and how they can be used and help you with developing and testing your application has been set.