

---

# FTI STUDIO™ PROGRAMMER'S MANUAL

Copyright © 2007 Focused Test Inc

## INTRODUCTION

The FTI Studio™ Programmer's Manual has two main sections. The first section is tutorial in nature, and teaches you how to write test programs and tests for FTI Studio™. The second section is reference information, which assumes reasonable familiarity with the tutorial parts. The reader is assumed to have some test programming experience, and a little experience using an Integrated Development Environment (IDE) such as Visual Studio.

Within the tutorial section, topics are shown by simple examples. These examples normally use random number generators to pretend to be measurements. This allows any user of FTI Studio™ to run these examples and experiment with them, independently of what instruments, devices and load boards are available. It is much easier to learn and debug problems when there is no DUT, load board and new instrumentation to confuse you.

All of the tutorial examples used are available on the FTI Studio™ release media.

## FTI STUDIO™ TEST PROGRAM CONCEPTS

### STRUCTURE OF AN FTI STUDIO™ TEST PROGRAM

FTI Studio™ test programs consist of one or more Flows, each Flow having one or more Test Steps. Each Test Step is implemented by a Test Method, and can have step-specific configuration data. FTI Studio™ also has Variables for the Program, the Site or the Test Step.

#### FLOWS

An FTI Studio™ test program can contain any number of Flows, but most programs will have only a few. Each flow represents a significantly different way of testing or developing the test program.

For example, a test program could contain the following flows:

- The primary flow used to test devices
- A fixture test flow used to test and calibrate any circuitry on the load board
- A development flow used to work on single test steps in a small environment before putting them back into the primary flow.

A flow defines the sequence of test steps run when that flow is used for testing. Testing starts with some flow start-up, runs each of the steps in the flow, and finishes with some flow tidy-up parts.

The normal process of running all of the test steps can be stopped for several different reasons:

- The flow specifies "Stop on Fail" and a test step fails the device.
- An alarm occurs, or an exception is caught and handled by FTI Studio™, which suggests that the program has gone badly wrong.
- During development, a code breakpoint may be reached. After debugging, the normal flow will be resumed.

Future versions of FTI Studio™ will include additional flow structure.

## TEST STEPS AND TEST METHODS

The Test Step is the basic unit of testing in FTI Studio™. Each Test Step uses a Test Method to define the actual programming operations needed for that step. The Test Methods can either be local to the test program, or they can be in a shared library, where many test programs can use the same Test Method.

When a Test Method is used several times in one test program, or when it is used in several different test programs, it needs information that tells it the specific details about each step. This information is the Configuration of the Test Method. Configuration Data will usually specify the test conditions, the test limits, and the signals and instruments used.

To say that again: a Test Method is the definition of the general way of performing some test; a Test Step uses the Test Method, plus the Configuration Data specific to that Test Step, to create the actual test that is run. The Flow is a sequence of such Test Steps.

## VARIABLES

Each FTI Studio™ program has a collection of named Variables available. These can be used to specify such things as limits in one place, rather than having them distributed (and copied with mistakes) in the configuration data for every Test Step. The Variables can have different values for Wafer, FT (Final Test), QA (Quality Assurance) and a Custom setting.

In addition to the Program Variables, FTI Studio™ makes a Variables collection available for each Site and for each Test Step, with the one for the test step referred to as MethodVariables.

## MULTISITE MECHANISM

Multisite is the ability of a test program to test several devices "at the same time". For example, a wafer probe test program may connect to four devices at a time. Multisite mechanisms need to ensure that the test results are the same as you would get if you tested the devices individually.

FTI Studio™'s architecture provides support for several different mechanisms that you may want to use for multisite.

- Each site can run in a separate thread. This is the only mechanism available in the first release of FTI Studio™.
- A group of sites can run a test step together
- Parts of the test program can be run on individual sites one at a time.

If the test system has enough instrumentation that each site can be tested with instruments that are not used by any other sites, you will want to run separate threads.

If there is instrumentation that does identical actions on many channels at one (digital instruments often behave like that) then the steps will want to run several sites together.

When there is some instrument that is expensive, but only used for a short time in testing a device, then running a set of steps that need that instrument on one site at a time gives most of the performance benefit of multisite, while keeping the system cost low.

## TEST METHODS

There are several base classes that you can use for defining Test Methods with FTI Studio™. There are some differences between these base methods.

### TESTMETHOD

TestMethod is the base TestMethod class, as provided by the Kernel of FTI Studio™. All test method classes must be derived from TestMethod.

You'll probably never derive directly from TestMethod!

### CONFIGURABLETESTMETHOD

ConfigurableTestMethod is one of the methods you're likely to use as the base for your own test methods. ConfigurableTestMethod adds several important capabilities to the basic TestMethod.

- Configurations, that are saved with the test program, are supported.
- A simple graphical editor for the configuration data.
- Easy access to the Program, Site and Method variables.
- Methods for testing values against single-sided or double-sided limits.
- Data Logging facilities.

We'll see a lot more about these facilities later on.

### DESIGNABLETESTMETHOD

DesignableTestMethod is an extension of ConfigurableTestMethod that also supports writing site-independent test method code for testing a single site at a time through the Instrument Resource map.

The normal set of callbacks and their arguments that DesignableTestMethod uses are slightly different from the plain ConfigurableTestMethod events and arguments. Extra callbacks are used to Run for a single site: when the base Run is called for several sites, it translates to single site calls.

## C# BACKGROUND

FTI Studio™ tests are written in the C# language. There are many excellent textbooks about C# available, and one of these will enable you to learn the C# that you need. Experienced C++ and Java programmers will probably find that “C# in a Nutshell” meets their needs. Those who have not used either C++ or Java extensively may prefer “Learning C#” or “Programming in the Key of C#”. Whichever text or course you choose, you do not need to be either a C# expert or a .Net expert to use FTI Studio™ effectively.

## C# IS MANAGED CODE

The .Net environment that FTI Studio™ builds on is termed “managed code”. The “runtime” takes on responsibility for managing many things that programmers had to worry about in older programming systems. Managed code environments are not new (there were some in the mid-1950s for the Lisp language). Managed code uses a portable form of compiled code that is checked and converted to actual machine instructions before the code is run. The managed environment can check that array subscripts are within the space that the array actually occupies, and can check that pointers actually point to something they are allowed to (to be precise, pointers are not used in managed environments, instead a reference is used).

Managed environments take over many aspects of memory management on behalf of the programmers, using techniques that reclaim memory once it can no longer be accessed by the program, without the programmer having to remember to release it correctly. This removes a whole class of common, subtle and hard to find bugs in programs.

Many people are apprehensive that managed code must be slow because of the things the environment handles. Certainly, an expert can hand-code things to out-perform a managed environment. But they usually don't! And the steady increase in processor and memory speeds means that a managed environment will be fast enough in a few months time if such hand-coding is needed today.

## CLASSES AND OBJECTS

The C# language, the .Net environment and FTI Studio™ itself are object-oriented. This means that have strong organization in terms of classes and objects. Everything that you develop for FTI Studio™ is a class, and while your program is running there may be many objects for each of your classes. For instance, each test step has an object that is an instance of Test Method class. If the Test Method is used five times in the program, for different steps that each has their own configuration data, then there would be five of these objects.

Each object has several things within it:

- Data that makes one object different from others. Some of this data is available only within the class itself, other parts are exposed as Properties.
- Code routines, often called “methods”. Some of the methods are available only within the methods of the same class, others are available outside the class.

## PROPERTIES

Properties are pieces of data in an object, where the designer has control over how (and whether) the data can be accessed. Instead of the data being just a field (as it is in a C struct), a Property specifies the code to be used to get the value and to set the value. To the client code using the object, it looks just like accessing a field in the object. Property notation provides protection against changes in the way that the data is implemented, without the client code being aware of the change.

Many properties do provide direct access to a field in the underlying object. Because of that it is very common to see fragments that look like this:

```
private double bilbo;

public double Bilbo
{
    get {return bilbo;}
    set {bilbo = value;}
}
```

That fragment shows a private field “bilbo” being exposed through a public property called Bilbo, for both reading (get) and writing (set). It is a common convention for the private version to have a lower-case first character, but otherwise be the same as the public name..

## METHOD OVERLOADING

Method overloading is the facility provided by C# and other modern languages that several methods can have the same name, but use different types of parameter. The compiler selects the appropriate one based on the number and types of the parameters in the call.

Method overloading avoids having to have separate names for each of these variations, when the purpose is the same, or for having parameters that say which variation is being requested, and so which other parameters are meaningful.

The utility pieces provided in ConfigurableTestMethod for limits checking and data logging are good examples of using method overloading.

## EVENTS AND DELEGATES

Events and Delegates can be confusing. A delegate is similar to callbacks in traditional programming languages: a piece of code that will be called when something happens. The “something” is an event. Delegates and the event methods in C# that FTI Studio™ uses are more powerful yet simpler than using conventional callbacks.

Part of the confusion is that there are several different levels of abstraction used in the C# delegate mechanism:

- A delegate statement defines a new “delegate type” that has a name, and specifies the parameters and return type of the “callback function”.

- The delegate type is then used with a “new” operator to create a “delegate object” that can be passed around, stored and eventually called.
- The delegate object is usually “added” to an “event variable” to register the callback. Registering the callback specifies that the delegate will be called when the corresponding event happens. Several different delegates can be added to an event, and if so all of them will be called when the event happens.

For the details on delegate and event syntax, consult your favorite C# textbook. FTI Studio™ wizards or Visual Studio generate most of the delegate and event code that you will need, so don't worry if you find it confusing.

#### EVENT CALLBACK PARAMETERS: SENDER AND EVENTARGUMENTS CLASSES

There is a very strong convention used throughout .Net about the parameters for an event callback. FTI Studio™ follows this convention.

Every event callback procedure gets two parameters, conventionally called “sender” and “args”.

Sender is the object that is sending the event to the callback. This can be any type of object, but particular events may always have specific types of object as the Sender.

Args is an object that holds any information about the event that may be needed. By convention the class for these objects is always a subclass of EventArgs, and always has a name then ends in EventArgs.

The callback procedure looks at properties in the args parameter to get at the information.

You will discover that sometimes there is no useful information in the Sender, nor in the EventArgs parameter. You may wonder why the convention is being followed when there is nothing there! The reason is that this convention makes your event code robust against future changes when we find useful things to put in the EventArgs parameter. Packaging all of the information in a single object also makes the procedure's interface robust when extra information needs to be provided by a future release of FTI Studio.

#### EXCEPTIONS

Many things can go wrong that a test program needs to handle. The C# language and the managed runtime provide an Exception mechanism that simplifies handling most of the problems.

With Exceptions, when something goes wrong, the place that notices the problem “throws an exception” as a forcible way of making the program know that there's something unexpected to handle. Throwing an exception is sometimes also called “raising an exception”.

Throwing an exception makes a sudden stop in the procedure that throws the exception, and program execution continues from the nearest place that is prepared to handle that type of exception. The `try` statement is used to say that this is a block that is prepared to handle exceptions. It should be unusual to have your own try statements in test code, but there will be some behind the scenes.

Let us consider some of the things that can “go wrong” in a test program and how exceptions can help simplify the test code.

- A device measurement produces a failing value. This is somewhere exceptions should **not** be used! The test executive parts of FTI Studio™ will see that a fail has occurred and run further steps as specified in the flow.
- You call an instrument routine with parameters that are not allowed. If these problems are not caught at compilation, the instrument routine should throw an exception.
- You make a mistake and use an array index that is too large for the array. The managed execution environment will spot this and throw an exception.
- The instruments detect a very abnormal situation such as the instrument's temperature sensor indicating overheating. An interrupt can be turned into an exception that stops the test program.
- The operator presses a Stop button. This can again be turned into an exception that stops the test program.

As you can see, the issues that exceptions should handle are a mixture of your programming mistakes and conditions that are hard to anticipate. And how often will you make mistakes in API call parameters, yet correctly check the status code returned?

Unfortunately, many API libraries for instruments have conventions from the days before exceptions were available in programming languages. These libraries may indicate problems by returning a status code on each call. Some libraries even use the very dangerous mechanism of storing such a status code in a global variable the caller is expected to examine after each call. (In case you're wondering why this is dangerous, multithreaded execution of the test program cannot use global variables in that way).

If you have to work with libraries that return status codes, we recommend that you follow each call that returns a status code with a call to `Debug.Assert` that specifies the expected value for that status code. This throws an exception if the status code is incorrect.

You can also throw a `com.FTI.Kernel.Alarm` when an unexpected status code is returned.

By using exceptions, the main body of the test code is close to straight line code that makes the instruments apply the intended stimulus, measures the response, and then makes pass/fail decisions. All of the complicated conditions to handle bad status returns or programming mistakes are eliminated.

## GARBAGE COLLECTION AND POINTERS

If you're from a C or C++ background, you'll be surprised by the fact that C# does not have any pointers. Soon after that you'll convince yourself that its "reference types" are just a way of having pointers but not saying so. Later on, you'll realize that references are not the same as pointers (and certainly not the same as C++ references).

A reference to an object is very similar to a pointer to that object, but with some important differences. The object can move, and the reference stays valid. You cannot do any of the forms of pointer arithmetic that C and C++ use extensively. The managed environment is in control of memory use. And needs to know which values are "pointer-like".

In return, the managed environment knows when there is nothing else referring to a piece of memory, and it can then reclaim the memory for reuse. There is no more need to “free” a piece of memory when you believe it will no longer be used. The environment does that for you with its garbage collector.

If you've never used a language with garbage collection, it sounds scary. Once you have used such a language, you never want to see again the burden of getting memory ownership and release right.

## WRITING TEST METHODS IN THE TEST PROGRAM

### VISUAL STUDIO SOLUTIONS AND PROJECTS

We recommend that each Test Method should be its own file, don't put several methods in one C# source file. We also recommend using the Test Method name as the filename. Nothing will go wrong if you have several Test Methods in a single source file, but this convention makes finding things in Visual Studio much simpler. (If the `DesignableTestMethod` is used, only one test per file is allowed because the Visual Studio designer requires it.)

A Visual Studio Project will produce an Assembly. You may well want a single project for all of the tests specific to a particular program. During development you'll probably have some times when almost all of the changes are in one or two Test Methods, and the rest of the Test Methods are staying unchanged. When that happens, you may want to create a Project for just the Test Methods that are changing.

### MAKING THE PROJECT RUN FTI STUDIO™ TO DEBUG

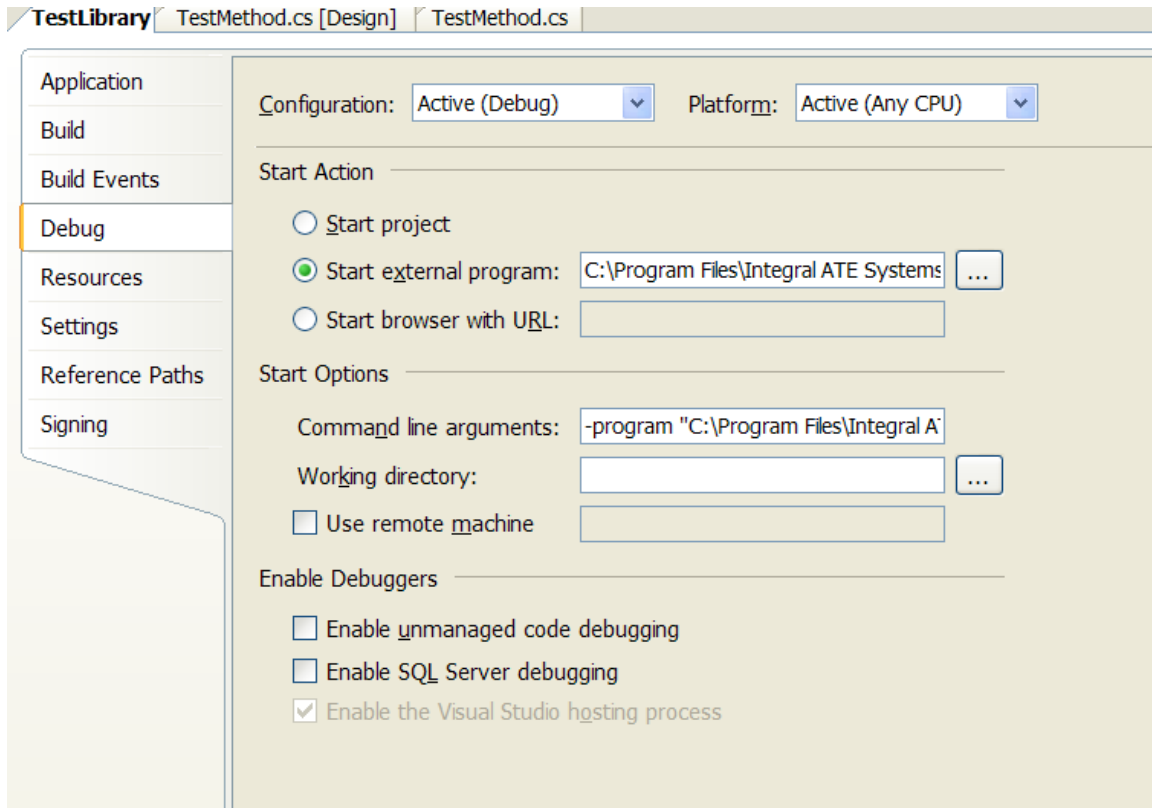
There are several ways of running and debugging a new test method with the combination of FTI Studio™ and Visual Studio.

The simplest is to set the project in Visual Studio so that FTI Studio™ will be run whenever Visual Studio knows that you want to run your new method.

The following steps do this.

1. With the project opened in Visual Studio, select the *Project/Name* Properties menu item (usually the last menu item under the Properties menu, *Name* is the project name).
2. Select the Debug tab.
3. Under Start Action, select Start external program
4. Set Start Application to the FTI Studio™ Application
5. Add `-program program_path` to the arguments substituting the `program_path` with the test program file path

The next screen shot shows these settings.



Now, when you are in Visual Studio and debug this project, FTI Studio™ will be started by Visual Studio.

## TEST METHOD INTRODUCTION

### FIRST EXAMPLE: RANDOM NUMBER

To start learning about FTI Studio™ test methods, we'll use some simple examples, and examine them in detail.

Our first example just calculates a simple random number and checks that it is between high and low limits. The random number will have a Gaussian (also known as Normal) distribution. Each step that uses the test method can specify the mean, standard deviation, low and high limits.

An example this simple allows you to explore using and programming FTI Studio™ without the need to have specific instruments and DUTs available.

### COMPLETE TEST METHOD EXAMPLE

To start, here is the complete test method that we'll look at. Following the listing of the complete method, we'll split it up into short pieces and look at each of them.

GaussianTestV1.cs [Design]		GaussianTestV1.cs					
Data		Resources					
	VariableName	Type	Mode	ProgrammingName	DefaultValue	Category	
	Mean	Double	Simple		0.0	Configuration	
	Deviation	Double	Simple		0.0	Configuration	
▶	LowLimit	Double	Variable	LowLimit	0.0	Limits	
	HighLimit	Double	Variable	HighLimit	0.0	Limits	
*							

```

using System;
using System.ComponentModel;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
using System.Drawing.Design;
using FTI.Exceptions;
using FTI.MethodDesignerKit;
using FTI.Standards;
using FTI.Formatters;
using FTI.Utility;

namespace TestLibrary
{
    [Method(@"Gaussian Random V1")]
    public partial class GaussianTestV1 : DesignableTestMethod
    {
        // We have a single linear random number generator
        // shared between all of the steps that use this method

        private static Random randomGenerator;
        private Gaussian gaussianGenerator;

        private void PreRun(GaussianTestV1PreRunArgs e)
        {
            if (randomGenerator == null)
            {
                randomGenerator = new Random();
                gaussianGenerator = new Gaussian(randomGenerator);
            }
            gaussianGenerator.Mean = e.Config.Mean;
            gaussianGenerator.Sigma = e.Config.Deviation;
        }

        private void Run(GaussianTestV1RunArgs e)
        {
            double value = gaussianGenerator.NextDouble();
            e.TestAndLogData(0, e.Config.LowLimit, value,
                e.Config.HighLimit, "Gaussian V1", "");
        }

        private void PostRun(GaussianTestV1PostRunArgs e)
        {
        }
    }
}

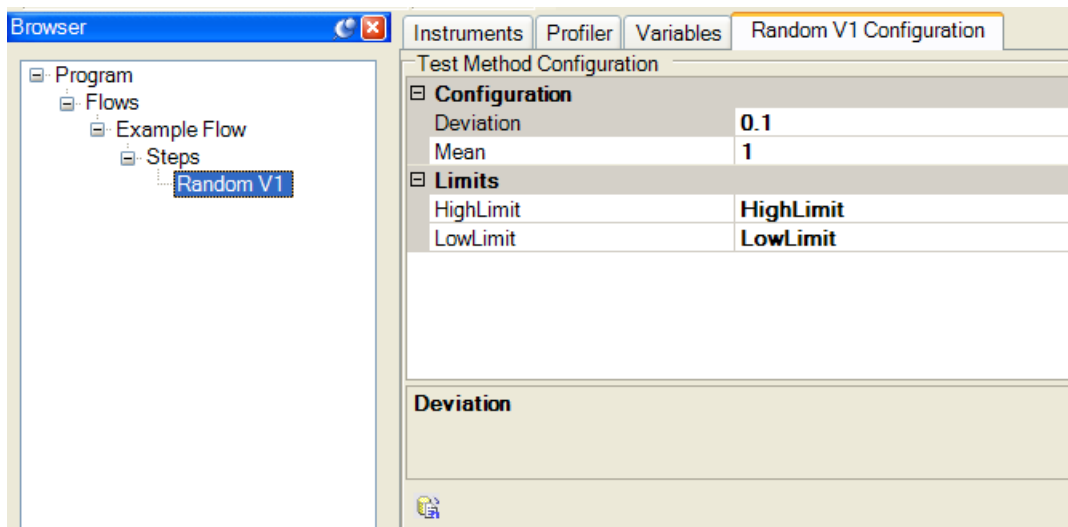
```

## BREAKDOWN OF THE TEST METHOD

The pieces of the test method are each fairly simple. We'll go through looking at them in detail. The designer view of the method displays the configuration values:

GaussianTestV1.cs [Design]		GaussianTestV1.cs					
Data		Resources					
	VariableName	Type	Mode	ProgrammingName	DefaultValue	Category	
	Mean	Double	Simple		0.0	Configuration	
	Deviation	Double	Simple		0.0	Configuration	
▶	LowLimit	Double	Variable	LowLimit	0.0	Limits	
	HighLimit	Double	Variable	HighLimit	0.0	Limits	
*							

Mean and Deviation control the Gaussian distribution. The low and high limits determine the pass and fail values. Because mean and standard deviation use the simple mode, they are directly set in the configuration panel. The limits are variable type, therefore they are found in the limits table and are linked to the test on the configuration panel. The category column separates values in the GUI. When the configuration is displayed, it looks like this:



The test method code starts with several “using” directives:

```
using System;
using System.ComponentModel;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
using System.Drawing.Design;
using FTI.Exceptions;
using FTI.MethodDesignerKit;
using FTI.Standards;
using FTI.Formatter;
using FTI.Utility;
```

Many of these directives are written for you by the FTI Studio™ wizard when you create a new test method in Visual Studio.

These directives allow the use of the straightforward names for many classes rather than its full name. For example the class Gaussian that we'll see shortly could have been FTI.Utility.Gaussian. Most of the time we'll prefer the shorter name. The full names are sometimes found in examples when they have been generated on your behalf. We'll see this shortly.

---

```
namespace TestLibrary
{
```

The rest of our method is all inside a namespace. You'll normally want to use an explicit namespace for all of your methods. If you do not name the namespace, your methods will go into the default namespace, which is technically called the "global namespace". Your company or department may well have conventions for what you should use as the namespace. Most of FTI Studio™ uses a convention that its own namespaces start with "FTI".

Be careful if you need to change the namespace after you re using the Test Method in your programs. The Test Program "program.xml" file contains the full name of Test Methods, including namespaces, and if you change the namespace, you will need to alter program.xml so that it has the new name. Avoid the issue by choosing good long-term names straight away.

---

```
[Method(@"Gaussian Random V1")]
public partial class GaussianTestV1 : DesignableTestMethod
{
```

The `[Method("Gaussian Test V1: Category Added")]` line is our first example of an attribute. This attribute is telling any interested parts of FTI Studio™ that the class following is a test method. The string is the "name" that will be shown in user interfaces, and in this case I've used the name with spaces where appropriate.

The class is called GaussianTestV1. It is specified as being `public` so that it can be accessed from outside the assembly. GaussianTestV1 is a subclass of `DesignableTestMethod`, which is one of the base test method classes that FTI Studio™ provides.

---

```
// We have a single linear random number generator
// shared between all of the steps that use this method

private static Random randomGenerator;
private Gaussian gaussianGenerator;
```

Now we declare the engines that will actually make the random numbers we will be using as our test values. The first declaration is a `Random` (which is in the `.Net System` namespace). A `Random` generates a uniformly distributed value between 0.0 and 1.0. We specify the `Random` as static, which means that there is only one copy of this in the test program. All of the steps and all of the sites that use this method will be sharing the same base generator.

`Gaussian` is an FTI Studio™ class (in `FTI.Utility`). It provides a Gaussian random number with a specified mean and standard distribution. `Gaussian` needs a `Random` as its source of uniformly generated random numbers (if you're interested, the algorithms are in standard textbooks such as the "Numerical Recipes" series, or "The Art of Computer Programming").

We declare the `Gaussian` `gaussianGenerator` as private. That means that nothing outside our test method class can see this variable and use it.

`gaussianGenerator` is definitely not static. We use a different `Gaussian` object for each test step that uses this method, because we want it to hold the mean and standard deviation specifically for the step. All sites share the same `gaussianGenerator` for the step.

---

```
private void PreRun(GaussianTestV1PreRunArgs e)
{
    if (randomGenerator == null)
    {
        randomGenerator = new Random();
        gaussianGenerator = new Gaussian(randomGenerator);
    }
    gaussianGenerator.Mean = e.Config.Mean;
    gaussianGenerator.Sigma = e.Config.Deviation;
}
```

The first callback we see is `PreRun`, which is called when the test program is run, and so is a useful place to do things that need to be done at the beginning of a test. `PreRun` is called once each time the Test Step is run.

This particular `PreRun` does two things. First, it creates a `Random` for the static `randomGenerator` if that is needed. Second, it creates a `Gaussian` (using the `Random`) just for this step, and sets the `Mean` and `Sigma` (Standard Deviation) properties from the properties we have already seen.

`PreRun` also sets the mean and sigma of the generator to the mean and deviation defined in the designer and set in the configuration panel in the GUI.

---

```
private void Run(GaussianTestV1RunArgs e)
{
    double value = gaussianGenerator.NextDouble();
    e.TestAndLogData(0, e.Config.LowLimit, value,
        e.Config.HighLimit, "Gaussian V1", "");
}
```

The second callback is Run. This runs the test step.

The "GaussianTestV1RunArgs" parameter gives the call access to configuration parameters and log methods.

A value is generated with the generator by calling NextDouble. Then, the value is logged as subtest 0 and the limits from the limit table are passed in along with a description of the value. The TestAndLogData method will test the value against the limits and take care of pass/fail decisions.

The complete test method ends with a couple of right braces "}" to close the class and the namespace.

## TEST METHOD CALLBACKS AND EVENTS

FTI Studio™ uses several callbacks and events to organize the work that test methods need to do. FTI Studio™ calls the callbacks and event handlers as the user takes actions. In this section we'll describe all of these callbacks and events and when they are called. To help you to see what these callbacks and events are, and when they are called, we've provided a simple Test Method called TraceEvents, which writes everything that happens to the Commander window.

When the test program is being loaded, the Load event handler is called for each step that has one.

There is an additional event called even earlier during the load sequence, called Setup, but Setup is called very early, and is used by FTI Studio™ for internal operations.

When a device is tested, each step sees a sequence of three callbacks, PreRun, Run and PostRun. If any exceptions occur during PreRun, then Run is not called. If there is an exception in Run, PostRun is called so that any cleanup can be done.

When the test program is closed, the Unload event handler is called.

All of these events follow the .Net conventions of having two arguments, the first is the "sender" of the event, the second has type which is based on EventArgs, and has additional information for some of the events. For these events, sender is always the test step object itself, and so is the same as "this". However at the moment sender should never be used, as it may change to something more helpful that is different from "this".

### SETUP

There is no specific information in the SetupEventArgs argument.

### LOAD

There is no specific information in the LoadEventArgs argument.

### UNLOAD

There is no specific information in the UnloadEventArgs argument.

### PRERUN

Contains the device number and gives access to the configuration data.

## RUN

Contains the PreRun information plus access to logging and limit testing and advanced access to variables. Resource information for multisite independent programming is also available.

## POSTRUN

See Run for the description of the TestNamePostRunArgs contents.

## WRITING TEST METHODS FOR USE IN MANY TEST PROGRAMS

When you want your test methods to be used in several different test programs, you need to make a few small changes.

The first decision you need to make is whether they will be a new Assembly, or be added to an existing Assembly of your Test Methods. Remember that all of the methods with the TestMethod attribute in an Assembly are available to a program using that Assembly.

Your company's policies probably state how new methods should be released, and whether they should be released in existing Assemblies or as new Assemblies.

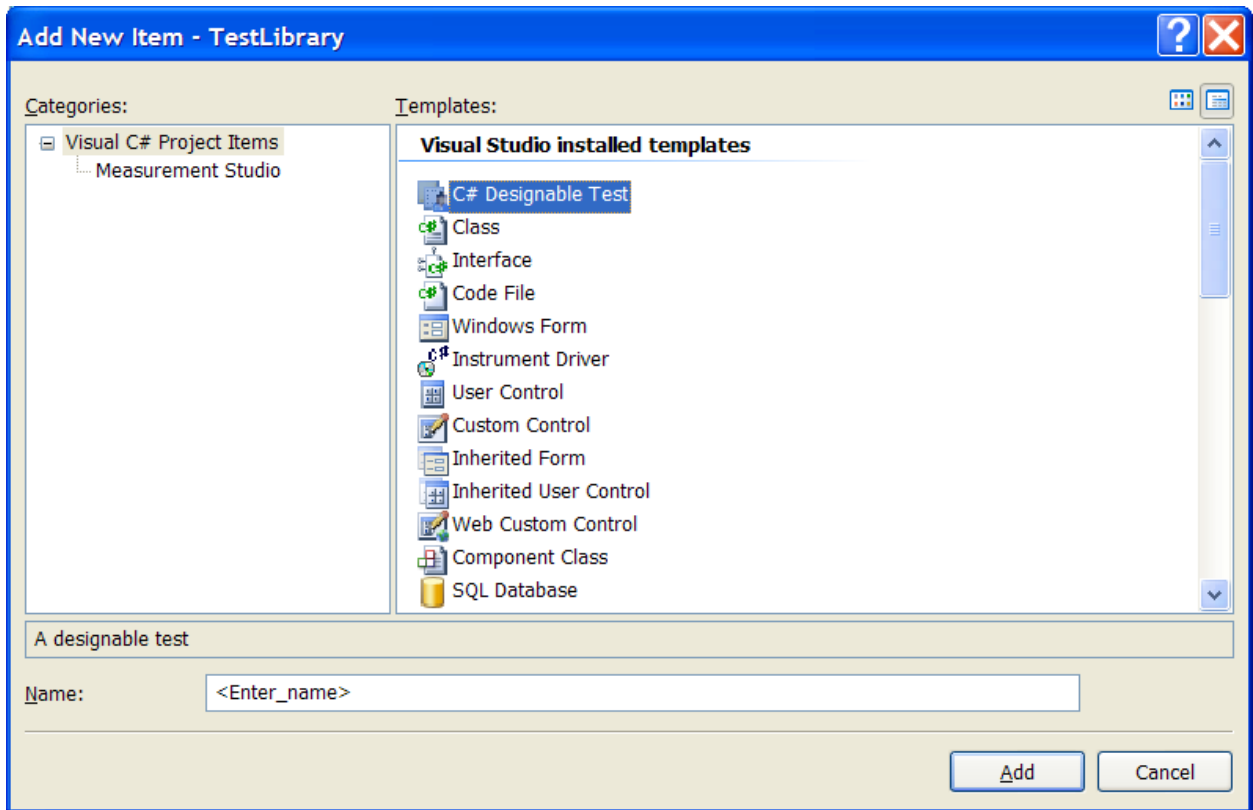
Test Methods behave differently for "New step" depending on whether they are local to the program or shared.

If local to the program, they are presented by class name. If shared, they are presented by the contents of the "TestMethod" attribute.

## ADDING A METHOD TO AN EXISTING ASSEMBLY

The simple way to create a new Test Method class is to use the wizards that are installed with FTI Studio™.

By right-clicking on the Project for the Assembly, selecting Add, then Add Class, you will get the dialog shown below, where you can choose the Test option with FTI Studio™ to create a new Test Method class.



## CREATING A NEW ASSEMBLY FOR THE METHODS

The simplest way to get an assembly for the methods is to start with one that is in a test program.

The file AssemblyInfo.cs for a test method assembly that is to be shared has to have a few lines that are different from those created in a program, although they do no harm if they are in a program's assembly.

Here's an example of the start of AssemblyInfo.cs for a shared test library.

```
using System.Reflection;
using System.Runtime.CompilerServices;
using FTI.Kernel;

//
// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
//
[assembly: MethodAssembly("ProgrammingExamples", "Focused Test Inc", "Small
Example Test Methods")]
```

The using statement that references FTI Studio™ Kernel must be added, and the MethodAssembly attribute must be added.

Note: If you forget the MethodAssembly attribute, you may get a misleading error message when adding the assembly with the MethodAssemblyManager tool. The message will say that AssemblyTitle, AssemblyCompany and AssemblyDescription are missing, and you will think that it is referring to the assembly attributes with those names, when it seems to be referring to the parameters on the MethodAssembly attribute.

Also don't forget to have meaningful names on all of the [TestMethod("Meaningful Name")] attributes. Otherwise you will see some strange blank entries in the menu of choices!

## DESIGNABLETESTMETHOD UTILITIES

DesignableTestMethod provides several useful pieces that can simplify development of test methods.

### TESTING VALUES AGAINST LIMITS AND LOGGING

The general concept is that there are overloads for when testing against actual values, and for when testing against values held in Variables. Each of these returns a bool that is true to indicate that the test has failed. So the return value is usable as a "fail" Boolean on other FTI Studio™ routines.

Here is the collection of limit test routines currently available.

```
public bool TestRange (
    double minLimit,
    double val,
    double maxLimit)

public bool TestRange (
    string minLimit,
    double val,
    string maxLimit)

public bool TestMin (double minLimit, double val)
public bool TestMin (string minLimit, double val)
public bool TestMax (double val, double maxLimit)
public bool TestMax (string val, double maxLimit)
```

Here is the collection of logging routines currently available:

```
public bool LogData (
    int subtest,
    double val,
    bool fail,
    string description,
    string message)

public bool LogData (
```

```
        int subtest,  
        object val,  
        bool fail,  
        string description,  
        string message)  
  
public bool LogData (  
    int subtest,  
    double val,  
    bool fail,  
    string description,  
    string message,  
    IDataFormatter formatter)  
  
public bool LogData (  
    int subtest,  
    object val,  
    bool fail,  
    string description,  
    string message,  
    IDataFormatter formatter)
```

When using these test and log instructions you pass the result of the test routines into the log routines. However, most people use the combined routine:

```
public bool TestAndLogData (...)
```

## WRITING TO THE CONSOLE

The routine `WriteConsoleMessage` takes a string parameter and writes it to the FTI Studio™ Console, which is visible as the Commander tab in the data logging area.

For more complicated formatting and variable data you can always use `Console.WriteLine` directly to get messages to that same Console. Note that the Operator interface does not have a Console, so any data sent to the Console with the production operator interface will not be seen.

## DATA LOGGING

Normal values are logged using `LogData`, which has eight overloaded versions. One set of the overloads formats the data in a default manner, the other allows you to provide a formatting object that gives you full control of the appearance of text datalog entries.

Alarms are logged using `LogAlarm`, or they will be logged by the system for you if you do not catch them yourself.

## CUSTOM FORMATTERS FOR DATALOGGING

The datalog format is controlled by any object that implements the IDataFormatter interface. The definition of the interface is:

```
public interface IDataFormatter
{
    string Format (
        uint subtest, object val,
        bool fail, bool postProcessFail,
        bool alarm, string alarmMessage,
        string description, string message);
}
```

This means that any object you want to use for formatting must have a method called Format that returns a string, and takes the following parameters.

subtest	An (unsigned) integer that gives the subtest number for this particular log entry.
val	This is defined as "object" so that the interface supports logging of anything at all. Most actual formatters will insist that their value be of a specific type. A really simple default for values that aren't understood specifically is to use val.ToString() to get a string representation of the value.
fail	A bool to indicate whether this is regarded as a failure or not.
postProcFail	A bool to indicate whether this is regarded as a post process failure or not.
alarm	A bool to indicate whether this is regarded as an alarm.
almMess	A string to describe the alarm
description	A string usually used for a description of the purpose of the subtest
message	A string that gives any further explanation you want of the result.

To demonstrate this capability, let us modify the GaussianTestV1 example to use a custom formatter, using one that's provided with FTI Studio™ called DoubleFormatter.

We'll create a new test method and copy the existing GaussianTestV1 into it, renaming it to be GaussianTestV2. But in order to use the provided DoubleFormatter we need to make a couple of extra changes.

The first is to add a line saying that we'll be using the namespace that DoubleFormatter is in.

```
using FTI.Formatters;
```

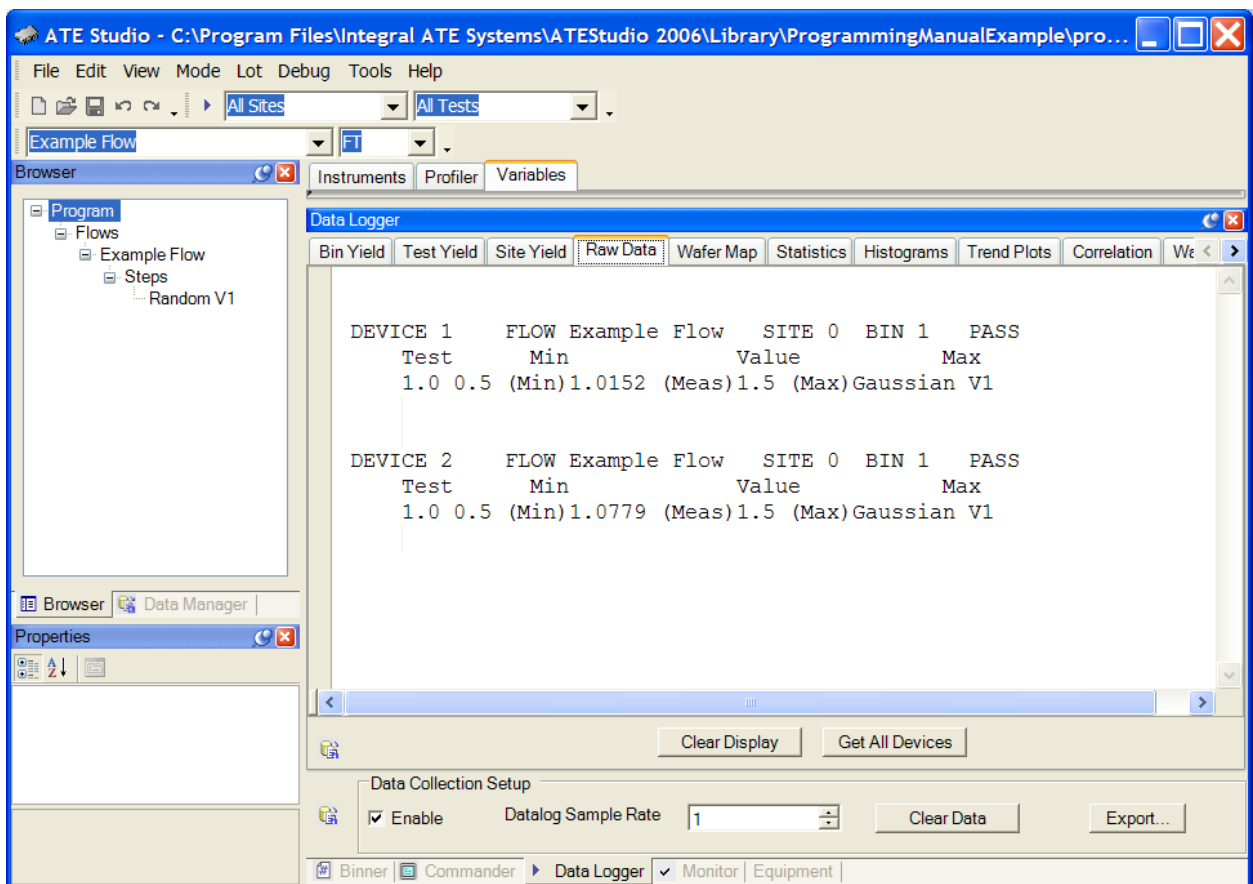
To use DoubleLogger, we need to make some very small changes to the test method code. Here's the new version of Run, with the changes highlighted.

```
private void Run(GaussianTestV1RunArgs e)
{
    double value = gaussianGenerator.NextDouble();

    DoubleFormatter formatter =
        new DoubleFormatter(e.Config.LowLimit, e.Config.HighLimit, 6, 4);

    e.TestAndLogData(0, e.Config.LowLimit, value, e.Config.HighLimit,
        "Gaussian V1", "", formatter);
}
```

Here's a screen shot of a datalog using the provided DoubleFormatter.



Your company and department may have rules on datalog appearance and a Data formatter that implements those rules. Simply implement your own formatter.