FTI Studio™ Getting Started Manual

1. INTRODUCTION

Programmatic test executives have a long history. They have been built with various languages, tools, on various operating systems. Most of the current executives use a compiled language such as C or C++, and run on UNIX or Windows. These solutions suffer from several problems that are related to the technology they are based on.

The C based languages, including C++, have always be best suited for expert programmers who need total control over memory and performance. Given the limited performance and memory of past microprocessors, test engineers only had a low level of control over the machine's ability to achieve good production performance. However, several important trends have changed things.

The massive improvement in processor performance and the drastic cost reduction of memory has allowed a host of improved languages to emerge and become practical. These new languages manage memory for the programmer, are more dynamic, and protect the user from making mistakes. Thus we have languages like Java, J#, and C#.

These new languages relieve the test engineer from the burden of becoming a programming expert so they can focus on testing. In addition, the performance of the new languages is great enough that production throughput is not affected.

One might argue that the performance justifies the creation of a whole new programming paradigm, such as a graphical or an iconic programming scheme. However, this requires expensive training and a long learning period for those who grew up with procedural programming. Learning an intense iconic language can be a difficult as learning a new spoken language for some people.

FTI Studio™ accepts the current paradigm, but improves upon it by leveraging the newest languages and tools. Additionally, it leverages the current knowledge of those test engineers that already have procedural programming language skills. This allows them to gain from the new technology without the expense of training for a new paradigm.

INDUSTRY STANDARD TOOLS

The big choices in building a test executive are the core tradeoffs between enhancement of industry standard tools and the creation of a complete environment. In the latter case, the test executive creator has complete control of the user experience. This allows him/her to create a seamless code and debug environment. Thus, instrument panels may continue to function while at a break point in code.

Using off the shelf tools place some restrictions on what one can create. For example, when using Visual Studio .Net, break-pointing a thread stops all threads. This forces one to create instrument panels in a separate process. If one is using PXI instruments with National Instruments drivers, this causes many problems. NI drivers do not support calls from different processes. This forces one to create a debug process that both the test program and the instrument panels call. But this then makes the test program writer use custom API's, locking out the use of NI APIs.

If one uses an off the shelf iconic environment like NI Lab View, there are a different set of problems. One has to manage a single shot run of a NI Virtual Instrument (VI), and a continuous run that allows one to inspect the state of the instruments. Additionally, general purpose programming is difficult, thus making the numerous standard tools that test engineer's use difficult.

The big picture then is the tradeoff between the limitations of standard tools vs. custom tools, and between the leveraging of low cost robust tools vs. a more unreliable set of tools that potentially offer a better environment.

FTI Studio™ extends standard tools by adding to Visual Studio .Net and National Instruments Measurement Studio. Furthermore, it leverages industry standard tool kits like Infragistics NetAdvantage.

Open Systems

In any system design the designer must consider if and how it can be modified by others. To the degree that the system can be modified, it is more or less open. Open systems allow a multitude of people in different roles to contribute to a system, thus expanding its capability faster than a single company could. However, open systems may come with some collateral damage: lack of robustness and complexity.

Open systems create a scenario where contributions can destabilize the whole. This creates the need for rules and mechanisms to enforce them. Additions have to have stamps of approval.

Complexity lengthens development. The cost is thus passed on to the end user in the form of higher up-front cost and more costly maintenance.

Closed systems of course have much more control of their inner workings, and they can be very robust. But, they can also be totally useless, because a fast moving market can't wait for one vendor to add functions. It also means customers compete for the vendor's attention, which favors the bigger customers. This locks the product out of many smaller markets.

FTI Studio™ takes an open system approach. Tools and subsystems can be added simply by standard sub-classing FTI Studio™ kernel classes, and then dropping the resulting assembly into a well established directory. No messy registration or registry editing is needed. Tests are also coded in assemblies and added to the system in a similar manner. The GUI is built with a toolkit assembly that anyone can use to build their own custom user interface.

Production Systems

Laboratory environments and production environments have different purposes. In the lab, the primary goal is characterizing a small number of devices, and measurement time is not critical. On a production floor, millions of devices must be tested with maximum throughput.

While FTI Studio™ can be used in a lab, its primary purpose is production testing. It has been highly optimized to give maximum multi-site throughput at the expense of loosing some of the flexibility needed in a lab environment.

Workstations

A test platform can be designed to work on one computer system or designed as a client-server application. In this context, "client-server" does not mean sharing files with a file system mount or Windows Network Neighborhood; it means something more like a GUI communicating with a server process, akin to the Semiconductor Test Consortium's OpenStar platform.

Client server platforms require large and expensive teams to design, and can suffer from performance problems when used in unanticipated ways. Even so, they are appropriate for large companies implementing tests for logic or SOC designs, where test times are longer and client-server third party tool interaction is required.

In low-end cost sensitive environments, mainly analog and small mixed-signal, a client-server package just becomes baggage that gets in the way.

FTI Studio™ is intended for the ultra-low cost analog and mixed-signal market, and therefore it is an application that is designed primarily for single workstation use. However, because it is a .Net application, one could easily write a .Net WebService that could control a tester. It would be practical to do this for operator control, system monitoring, and other activities except for creation of test libraries and sub-systems. Creating these requires compilation, and that can not be made to operate the client-server.

Hardware Platform Independence

Most ATE software systems are hardware platform dependent. Many lab software systems are platform dependent, but support multiple platforms such as VXI, PXI, etc.

FTI Studio™ is designed to be hardware platform independent. As long as one can call traditional C/C++ or .Net API classes, on a Windows platform, one can use FTI Studio™.

To ease some of the programming burden, different hardware platforms can be integrated into FTI Studio™, presenting their own API. This adds functionality, and presents the test programmer with an API that is more like a traditional ATE environment. However, it is not necessary. It is perfectly reasonable to write tests using NI Measurement Studio classes, or CVI classes directly.

The Tradeoff Matrix

The tradeoffs are show below in bold type:

- **Procedural Language** vs. Iconic Language
- **Extended Standard** vs. Full Custom Tools
- **Current Paradigm** vs. New Paradigm
- **Open System** vs. Close System
- **Production Oriented** vs. Lab Oriented
- **Workstation** vs. Client-Server
- **Hardware Platform Independence** vs. Hardware Platform Dependence

Architecture

Test Studio is comprised of a set of .Net assemblies and executables. When the main engineering interface executable is run, the permanent assemblies are located and run by the .Net runtime. Then directories are searched for in assemblies that are dynamically loaded.

Dynamic assemblies are a mechanism for plug-and-play. The two main plug-and-play assembly types are subsystems and tools. These assemblies reside on the disk drive in the installation folder. FTI Studio™ searches these directories looking inside each assembly it finds, searching for an attribute that marks it as a subsystem or tool. If the attribute is present, the assembly is loaded, and the subsystem or tool is installed into FTI Studio.

Subsystems and tools have GUI controls built into them, and the FTI Studio™GUI automatically places the controls in the GUI at the proper location. The non-GUI portion hooks key events from objects in the permanent assemblies. For example, the datalogger subsystem receives start, stop, and test events. These events contain result data that the datalogger can display.

To add a subsystem to FTI Studio, a programmer creates an assembly with a class that implements a system or tool class. After the assembly is compiled, it is placed in the proper directory, and the next time FTI Studio™is run, the subsystem is loaded.

Test libraries are also dynamically loaded assemblies, but they are not discovered by searching directories. Test libraries are registered with the system with a tool. Libraries can be registered by reference or value. When registered by reference, the assembly location is where it was compiled. Any recompilation will immediately affect any test program that uses it. When registered by value, the assembly is copied to a well know directory and is then immune from changes.

INTERNAL STRUCTURE

The kernel assembly defines the internal structure of programs, flows, steps, subsystems, and tools. It is these classes that define where things are plugged in. Understanding this structure is important for subsystem and tool writers, as well as test writers. A rough outline of the structure will be presented here.

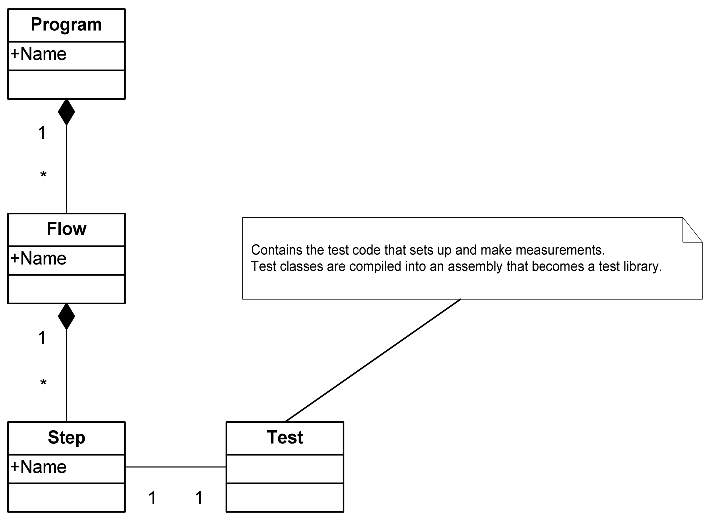
PROGRAM, FLOWS, AND STEPS

Program is the starting point for the object tree. Programs have flows, flows have steps, and steps are associated with tests. A program is a collection of flows, limits, variables, setup info, and all the things that are needed to test one device.

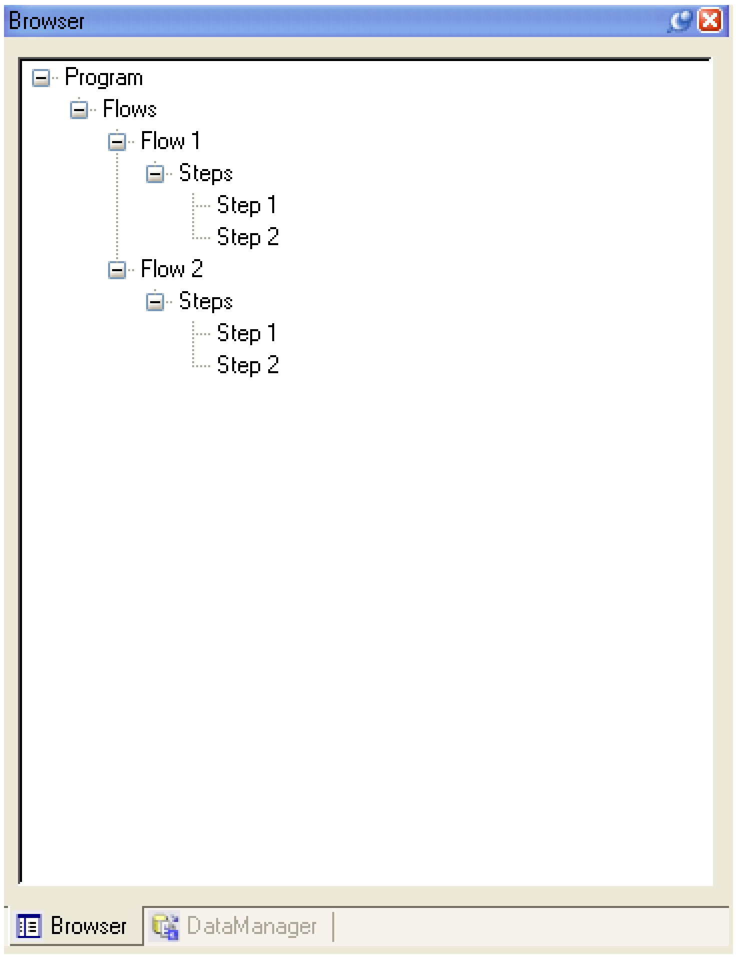
Flows define the order that things are tested in. One might create flows for characterization, final test, QA, etc. Flows are a key mechanism, because they allow one to share information, like data sheet values.

Steps are the unit of test. Each step measures one data sheet parameter, or a small collection of similar ones. A step performs the measurement with a test. A test can be used by more than one step, but a step can only be used once in a program. In a sense, a step has an identity in the test program which includes the flow that it is in, its position, test number, etc. A test defines the code that runs. This separation allows the system to use a test more than once, but maintain a unique identity for each location in the flow where it is used.

A UML diagram of the structure is show below:

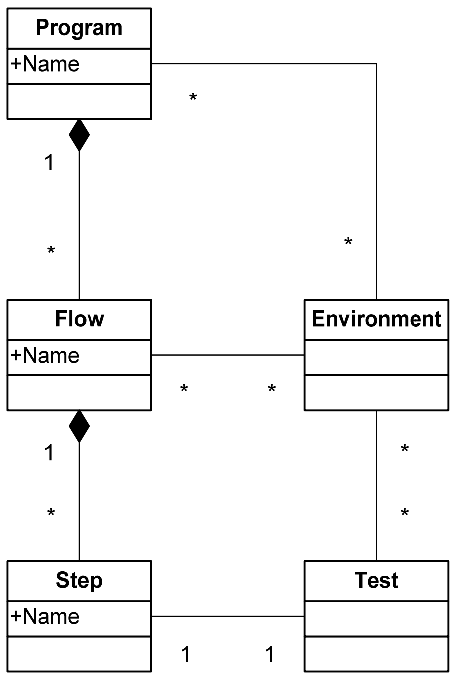

Program
+Name
1
*
Flow
+Name
1
*
Step
+Name
1
1
Test
Contains the test code that sets up and make measurements.
Test classes are compiled into an assembly that becomes a test library.

This UML diagram represents the internal structure. In the user interface, this structure is represented with a tree as shown below:

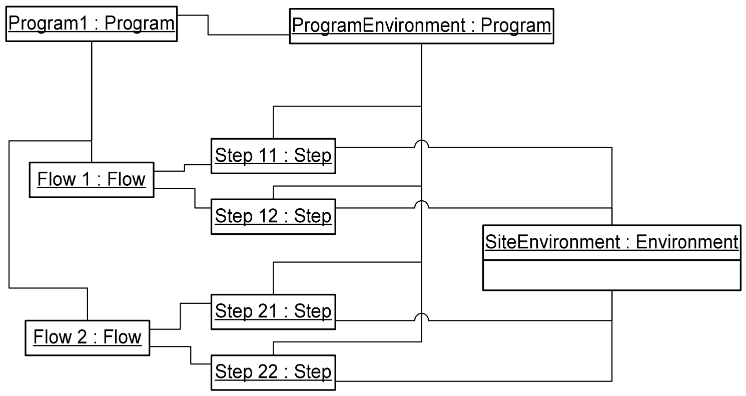


What are not shown in the tree are the tests that are associated with each step. Tests are hidden behind the steps because a step represents an instance of a test. Testing proceeds by executing steps in the order displayed in the GUI.

Because the internal structure is object oriented, there is no way for this simple structure to pass information around between steps and flows. Therefore, there are environment objects that act as places for steps and flows to share information. Rather than one large container, there are several containers are shared with different objects. The UML class diagram is shown below. Be patient, it will not be clear how this works until an object diagram is drawn so you can see the connections.



It should be obvious why the class diagram is useless. Everything is connected to everything. Let's look at a partial object diagram:



First notice the relationship between the program, flows, and steps. This matches the earlier diagram. Now, pay attention to the program environment. There are links to the program object, and all step objects. The program environment is like a global state. Anything stored in it can be accessed by any object. This is where global variables and limits are stored. The site environment is connected to every step. This allows steps to pass data between themselves. There is one site environment for each site; therefore each step has to use the proper environment depending on what site/s it is testing.

There are other environment relations that allow the storage of step configurations, tool data, etc. These will be covered in more detail in another document. What matters here is that an object oriented program structure requires an object oriented state storage system so that data can be passed around.

SUBSYSTEM

Subsystems are installed into environments. The mechanism for this is simple. A subsystem class is marked with an attribute that denotes what environments it should be installed into. When the subsystem loader dynamically loads them, it examines this attribute and installs an instance in every instance of these environments.

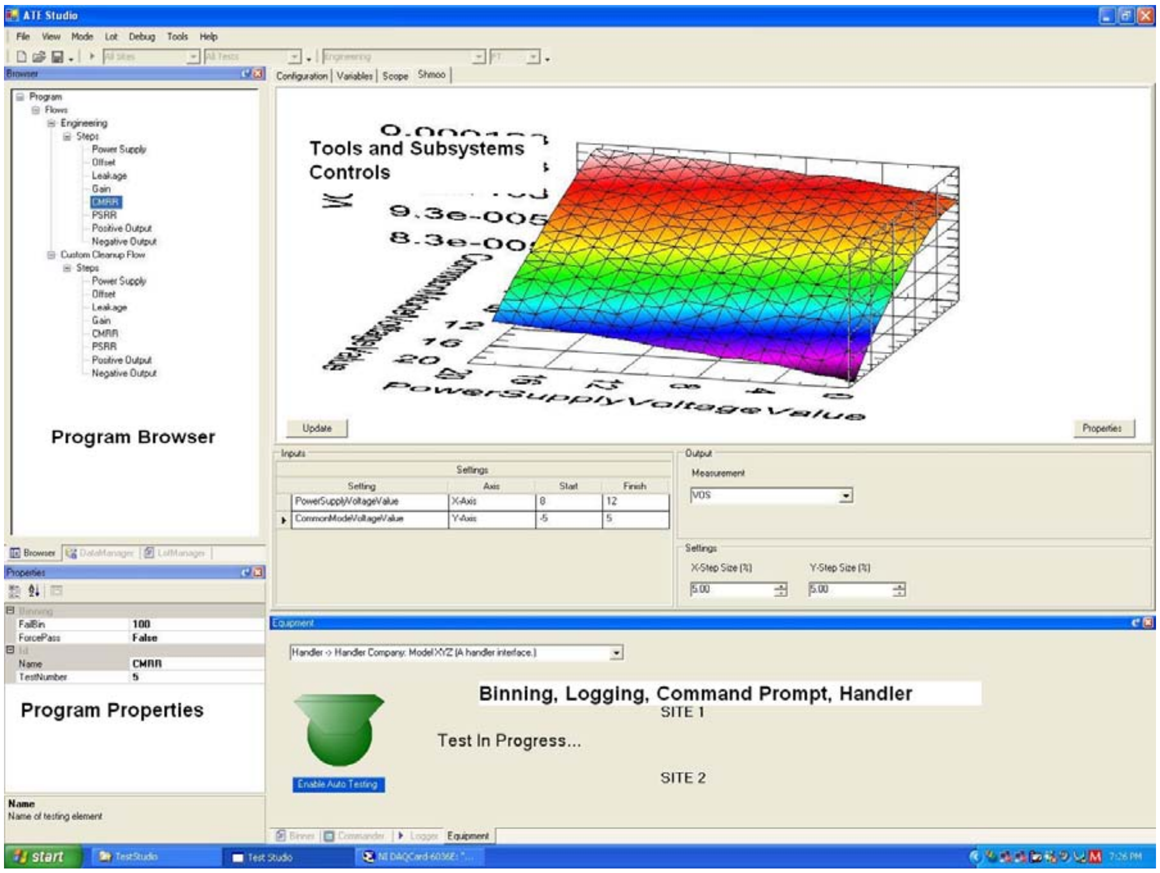
When the subsystem is installed, it is given a chance to register interest to events associated with each environment. Events such as test start, test done, program start, etc. These events are what drive the subsystems. Subsystems also can have GUI controls, so events can also come from the user.

Subsystems can also store state in their environment. So, a datalogger system could consist of two subsystems: one in the program environment, and one in each site environment. Test code would get the site environment subsystem and write data to it. The subsystem would store the data in the site environment. When a site is done testing, the site subsystem might get an event from the flow object that says testing is done, and then batch copy the data to the program environment. When the program object generates an event that all sites are done, it might process the data for all sites and update its GUI control which is the datalogger window they see.

It is by creative use of subsystems which plug into the environments that variables, datasheets, dataloggers, and other features are plugged into the system.

USER INTERFACE

The standard user interface is a Windows GUI. For those that must use a command line, there is a command line window in the Windows GUI. The user interface has 5 working areas.


ATE Studio
File View Mode Lot Debug Tools Help
Browser
Configuration Variables Scope Shmoo
Tools and Subsystems Controls
PowerSupplyVoltageValue
Program Browser
Program Properties
Update
Properties
Equipment
Enable Auto Testing
Binning, Logging, Command Prompt, Handler
Browser DataManager LotManager
Binner Commander Logger Equipment

Program
Flows
Engineering
Steps
Power Supply
Offset
Leakage
Gain
CMRR
PSRR
Positive Output
Negative Output
Custom Cleanup Flow
Steps
Power Supply
Offset
Leakage
Gain
CMRR
PSRR
Positive Output
Negative Output

Inputs

Setting	Axis	Start	Finish
PowerSupplyVoltageValue	X-Axis	8	12
CommonModeVoltageValue	Y-Axis	-5	5

Output
Measurement: VOS

Settings
X-Step Size (%): 5.00
Y-Step Size (%): 5.00

FailBin	100
ForcePass	False
Name	CMRR
TestNumber	5

Name
Name of testing element

Handler -> Handler Company: Model XYZ (A handler interface.)

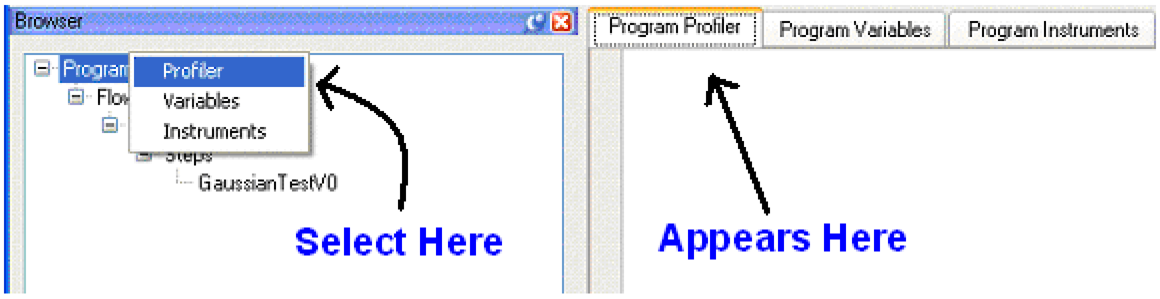
SITE 1
Test In Progress...
SITE 2

Along the top of the interface is the usual menu and tool bar.

The left top panel has a program browser and data manager. One can select which item is on top with the tabs at the bottom of the panels. The program browser can display one program at a time. By opening the nodes in the program tree, one can see the flows, steps, and other items.

Below the program browser is a properties panel. When you select a node in the program tree, its properties are shown in the properties window. This allows you to give flows and steps names, and set test numbers and bin numbers.

The top right has a tabbed panel that contains panels for the subsystems and tools. These tabs do not appear by default, and must be accessed by right clicking on a node in the Program Browser. After right clicking a node you may select the tab desired. (This feature allows the user to have multiple tabs open per node, and also to have more than one of the same tab open at once:)



The tool on top is a shmoo tool. There is also a scope tool, variables tool, and a configuration tool. The configuration tool allows you to set the parameters for a step. Each step has a set of parameters associated with its test that configure how the test will behave. This is how you create tests that are reusable. When you code the test, you code it with parameters so that you can customize its behavior when you write test programs.

The bottom right window has a tabbed panel for the yield display, datalog, and the command line window.

If you extend the system by adding subsystems, tools, and tests, their panels always end up in the top right tabbed panel. When you right click a node in the program tree and choose a tool it creates their tabs. To add panels to the lower right area, you have to emend the user interface toolkit. If you build your own GUI with the tool kit, which is possible with data binding, you can put windows wherever you want them.