Test-System Development Guide

Choosing Your Test System Software Architecture

Application Note 1465-4

This application note is part of the Test-System Development Guide series, which is designed to help you quickly design a test system that produces reliable results, meets your throughput requirements, and does so within your budget.

The information presented here will help you choose the direction for your software based on the application you have in mind and the amount of experience you have. We will explore the entire software development process, from gathering and documenting software requirements through design reuse considerations.

The complete list of application notes for this series is available on page 19.

Table of contents

Agilent Technologies

Introduction

This white paper will help you understand the tools required to design, develop and deploy the software component of your test system (see Figure 1).

Gathering and documenting software requirements—Before gathering and documenting your software requirements, finalize your test system hardware design. Once finalized, start working with your R&D and manufacturing teams to collect the information you need to create software requirements specifications (SRS).

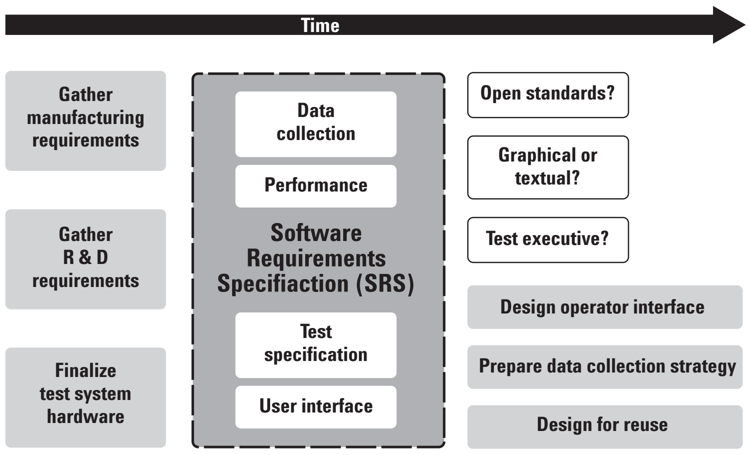
Programming and controlling your instruments—The control of instruments is rapidly evolving from proprietary test and measurement standards to open, computer-based industry standards. This trend affects the hardware that connects the PC to the instrument as well as the software and drivers that control the instrument.

Collecting and storing test data—Data collection is the science of obtaining, moving and formatting data. The integrity of your test system depends on obtaining the right data at the right time.

Designing the user interface—One of the most important (and easily overlooked) aspects of test systems is the Graphical User Interface (GUI). This is what the test engineers, operators and technicians see when they interact with your software.

Choosing the development environment—The software environment and tools you choose will have a significant impact on the overall cost of your test system. When choosing your software environment, consider more than just the purchase price of the software. Also, consider how easy it is to learn and use the software, how hard it is to connect to other languages, devices or enterprise applications, as well as support and maintenance costs. Over the life of a test system, software support and maintenance costs alone can exceed hardware costs.

Figure 1. Test-system software development process overview


Time
Gather manufacturing requirements
Gather R & D requirements
Finalize test system hardware
Data collection
Performance
Software Requirements Specifiaction (SRS)
Test specification
User interface
Open standards?
Graphical or textual?
Test executive?
Design operator interface
Prepare data collection strategy
Design for reuse

Working with open standards—Today, the industry trend is to move away from closed, proprietary development environments. More and more people are embracing open, industry-standard development environments as their platform of choice for test-system development projects. Making the right choice now will give you the flexibility and capabilities you need in the future.

Developing a test sequence—Test executives are applications designed to run a series of tests quickly and consistently in a pre-defined order. Of the 93% of test-system developers who use test equipment, approximately 37% use a commercial test executive for test sequencing, while the remaining 56% use a "home-grown" test executive.

Planning for software reuse—Designing for code reuse means you and your co-workers won't have to re-create your software components every time you start a new project. Instead, you can build up a company knowledge base of best ideas, best practices, and software components. This knowledge base will bring uniformity and consistency to your company's product testing functions.

This application note will provide you with a solid overview of the test-system software architecture as outlined above. For more in-depth information, refer to the sources listed throughout this document. Now, let's get started with the first phase of choosing your test-system software architecture—gathering and documenting your software requirements.

Gathering and documenting software requirements

The Software Requirements Specifications (SRS)[1] is a prioritized list of required test-system software capabilities and information on the software's external interfaces, performance requirements, system attributes and design constraints. Typically, some requirements "musts" are essential and others "wants" can be traded for time (e.g., to meet a project deadline).

The IEEE Society identifies the following areas you should address in your SRS:[2]

- Functionality–What is the software supposed to do?
- External interfaces–How does the software interact with people, the system's hardware, other hardware and other software?
- Performance–What is the speed, availability, response time and recovery time of various software functions?
- Attributes–What are the portability, correctness, maintainability and security considerations?
- Design constraints–What industry standards do I need to follow? Do I need to use a specific language? What about internal policies for database integrity, resource limits and operating environments?

Ideally, the SRS will describe WHAT you need the software to do, not HOW the software will do it. In other words, you can look at the software as a "black box" that controls a set of external resources such as instruments, a computer monitor and other components (see Figure 2).

The SRS will include implementation details only if those requirements are imposed externally. For example, your company may require that a portion of the system be implemented in a specific programming language.

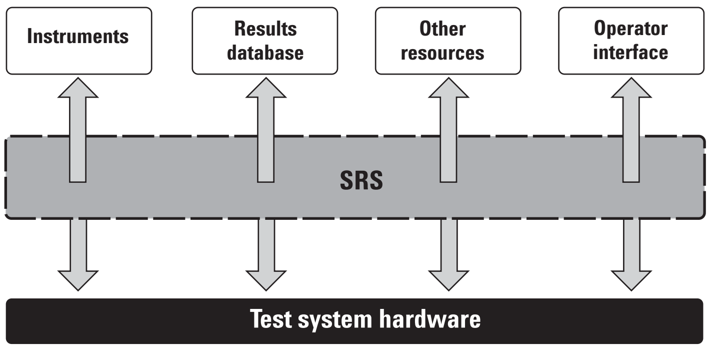
A good SRS should answer the following questions:

1. What measurements and tests are required to exercise the device under test (DUT)?
2. How will the measurements and tests be performed given the available instruments and devices?
3. What types of data need to be collected?
4. Where will the data be stored?
5. What are the external constraints (i.e., performance and time specifications)?
6. How will the operators, test engineers and technicians interact with the software?

Within the product development lifecycle, the R&D department should provide a formal list of testing requirements to the test-development department. The System Requirements Specifications, also referred to as a Project Requirements Specification, refers to the system as a whole and therefore is different from the Software Requirements Specifications. Furthermore, the manufacturing department will have their own requirements, such as safety standards. It is the combination of R&D and manufacturing specifications that determine the hardware requirements of a test system and provide the basis for the Software Requirements Specifications.

It's important to note that trying to build or design software while the test system hardware is still in a state of flux typically results in additional software re-work and re-design. This is one of the challenges you will face in the real world of test-system development!

Figure 2. Scope of the SRS



1 May be referred to as an ERS or simply as "the requirements."

2 For more information, refer to the IEEE Standard 830-1998 "Recommended Practice for Software Requirements Specifications" and the IEEE Standard 1233-1998 "Guide for Developing of System Requirements Specifications" located on the IEEE web site (http://standards.ieee.org).

Figure 3 provides an SRS template and a requirements example. As shown in the template, SRS is more than requirements. Document within the SRS what the software is meant to do and provide definitions for the terms you are using. Document the external constraints imposed upon you and the external resources you have available. Describe your users in detail and the modes of operation for each user class. Finally, include appendices and an index. Once you've completed these tasks, you're ready to describe the specific requirements.

The requirements example (user interface of a test sequencer) is a snippet from a larger set of requirements divided by function. The words "MUST" and "HIGH WANT" are a way of ranking the relative importance of the requirements. You can break up requirements into more manageable hierarchies based on function, program mode, or some other classification system that will make the requirements section easier to navigate.

IEEE says that requirements must be correct, unambiguous, complete, consistent, ranked for importance, verifiable, modifiable and traceable. You can see that the above format meets a number of those goals, but some additional practices are necessary to meet them all. If you refer to requirements in more than one place, you will need to cross-reference them using a unique number (3.4.3, for example) so that if a requirement changes, you will know where to fix it elsewhere in the document.

Each written requirement needs to be verifiable and unambiguous to ensure the test program behaves as expected. As you write the SRS, refer to the System Requirements Specifications whenever possible. This is called backward-traceability, helping to explain why certain requirements are included and not just an arbitrary restriction.

The SRS must describe what testing resources (instruments) are required (e.g., the type of voltmeter, switches, computer monitor, etc.) and whether any factory resources are needed (e.g., a results database). In addition, you need to define within the SRS the data collection method, user interface requirements, performance constraints and, most importantly, the specific DUT test requirements. For example, if you need to perform a specific resistance measurement and you know you have an Agilent 34401A multimeter, the SRS would specify a single-sample 4-wire measurement including a description of the proper switching path, thus ensuring access to the pins on the DUT.

In order to accurately describe the test-system software user interface requirements, you should develop specific use cases for the different users of the test system (e.g., operators, test engineers, managers, etc.). Use cases are scenarios describing the users' interactions with the software.

Taking the time to develop well-written requirements specifications up front will save you time later in the development process. The SRS process forces you to think about the scope of your project and helps to identify poorly understood areas of your software. This means you will spend less time re-writing and re-testing software due to confusion over what was truly required in the first place. A well-written SRS will help ensure that the project portion you want to contract out or redistribute will not require re-work on your part.

Figure 3. SRS template and requirements

Example SRS template

Table of contents
1 Introduction
- 1.1 Purpose
- 1.2 Scope
- 1.3 Definitions, acronyms and abbreviations
- 1.4 References
- 1.5 Overview

2 Overall description
- 2.1 Product perspective
- 2.2 Product functions
- 2.3 User characteristics
- 2.4 Constraints
- 2.5 Assumptions and dependencies

3 Specific requirements
Appendices
Index

Example requirements

3.4 User interface functionality:
3.4.1 (MUST) The UI allows the user to create, modify, run and debug sequences.
3.4.2 (MUST) The UI allows users to view and export, load and store sequence run result data.
3.4.3 (MUST) The UI represents sequences in a hierarchical manner, which may be expanded or collapsed to view or hide internal details of the sequence.
3.4.4 (HIGH WANT) The UI can represent shared (used several places) sequences separate from the main sequence hierarchy.
3.4.5 (HIGH WANT) The UI will use graphical icons to denote variations in state of sequence items.

Programming and controlling your instruments

When designing your test-system architecture, you need to think about how your PC will communicate with different instruments. The two most important factors are 1) how to physically connect the PC to other instruments, and 2) what software will you use to control and communicate with other instruments.

Physically connecting the computer to other instruments

For decades, the IEEE-488 bus, commonly known as the general-purpose instrumentation bus (GPIB), set the standard for connecting test instruments to computers and for providing programmable instrument control. While GPIB is still a common and effective instrument interface technology, PC-based standards such as USB and LAN tend to be more cost effective solutions (see Table 1).

USB is the best choice for R&D applications where the number of instruments in a system is usually small and a quick and easy interface set-up is desired.

USB 2.0 and Ethernet-based LAN are good choices for design verification and manufacturing applications where data-throughput performance, cost, remote access, and ease of system assembly are top priorities.

Given the choice between USB 2.0 and Ethernet-based LAN, most people choose LAN because of its inherent flexibility and remote system access and control capabilities. In addition, LAN performance is on par with USB—it has captive cable connectors (which aren't found on USB), and LAN has the capability for wireless operation.

The I/O software layer

Once you've decided how your computer and instruments will be physically connected, you need to decide what I/O software you will use to control and communicate with the instruments (see Figure 4).

The I/O software is the layer of software that sits between the software application and the instruments' physical interfaces. Once again, you have two choices. You can write directly to the instrument (Direct I/O) or you can use an instrument driver. Even though standard instrument drivers are popular because they are easier to use, they only express a subset of the instrument's functionality.

Figure 4. I/O software layer

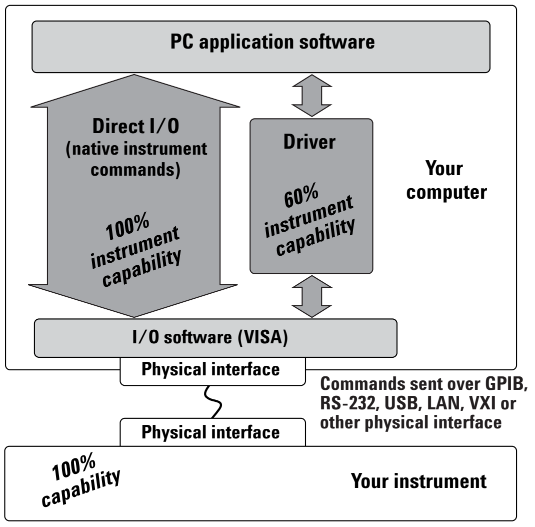

PC application software
Direct I/O
(native instrument commands)
100% instrument capability
Driver
60% instrument capability
Your computer
I/O software (VISA)
Physical interface
Commands sent over GPIB, RS-232, USB, LAN, VXI or other physical interface
Physical interface
100% capability
Your instrument

Table 1. GPIB, USB and LAN advantages and disadvantages

Interface	Theoretical Interface Speed	Advantages	Disadvantages
GPIB	• 8 Mb/s transfer rates	• Ubiquitous interface on test instruments • Maximize throughput for all block sizes • Low cost	• Expansion slot required • Must open PC housing to install card • Relatively expensive • Limited cable lengths permitted between computer and instruments
USB	• USB 1.1 12 Mb/s • USB 2.0 480 Mb/s	• Quick, easy setup • Low cost • Good data-throughput performance	• Does not work with Windows® NT, • Not available on most deployed instruments
LAN	• 10/100/1000 Mb/s transfer rates	• Good data-throughput performance • Low cost • Remote access makes it easy to control system from remote location	• Requires LAN knowledge to set up • Not available on most deployed instruments

So, how do you decide? Here are a few factors to consider.

You may want to use Direct I/O if:

- You need to use instrument features not supported by the available drivers (the other 40% to approximately 80% of the instrument's capabilities). You can often use a combination of direct I/O and instrument drivers in this case. Some drivers make it even easier by providing a direct I/O connection for such scenarios.
- You have instrument programming experience or access to programming experts.
- You need the absolute maximum in system throughput speed.
- You need to control the exact configuration of the instruments in your system.
- You have a large volume of legacy SCPI-based code.

You may want to use an instrument driver if:

- A driver is available that works with your development environment and I/O software, and supports the majority of instrument features you want to use.
- You want the ease of use gained by an easy-to-understand hierarchical organization of instrument functionality provided by drivers.
- You want to simplify the process of developing and maintaining your code over time so there is a single point of interface to update or change.
- You need to simplify maintaining the system when instruments need to be exchanged.
- Development time is a paramount concern.

If you choose an instrument driver, consider using an industry standard IVI-COM (Component Object Model) driver[3] together with a Visual Studio® .NET-compliant development environment (such as the Agilent T&M Programmers Toolkit). IVI-COM drivers have the following advantages.

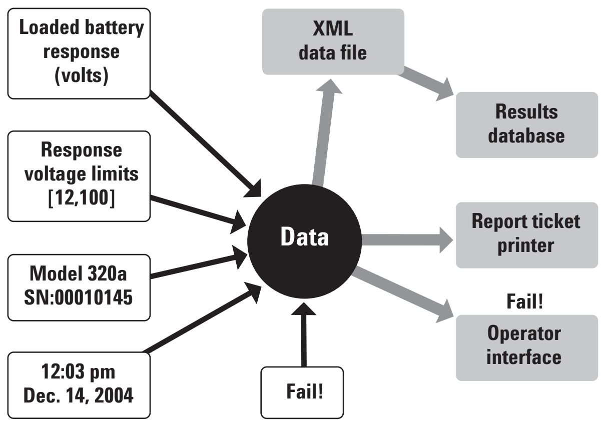
1. It works with all popular PC languages and most T&M languages.
2. It uses the most popular types of I/O.
3. It can be used in the latest .NET technologies.

By using IVI-COM drivers in the development of your test-system architecture, you'll save time[4] and have a higher degree of hardware and software interchangeability. You also will find that your software is easier to maintain and is more extensible in the future.

Collecting and storing the test data

Data collection is the science of identifying, collecting, formatting and distributing important information about the behavior of your test system and the devices it tests (see Figure 5). Quality data collection is the foundation for controlling your manufacturing and test processes—the ultimate goal of a manufacturing test engineer. Quality data also can be used to support many functions throughout your organization and support products throughout their development lifecycle.

Figure 5. Overview of data collection process



3 IVI-Component drivers are based on Microsoft's® Component Object Model (COM). IVI-C (NI) drivers are based on C dll's.

4 A recent survey conducted by Agilent Technologies found that test programmers experience a 20%-30% reduction in test program development time when using Visual Studio .NET.

Communicating results of a test sequence is one use of test data. Test data also may be used to ensure regulatory standards are met, document performance standards, or provide traceability for the DUT. Given these applications and others, you may want to collect more data than your R&D or manufacturing colleagues request.

In addition to external data requirements, recorded data can be used to debug a test sequence in ways debugging runs cannot. Debugging means slowing down and subtly changing the behavior of your test sequence. This means a defect you see in a normal run may not show up in a debugging run (and vice-versa). One way to reduce the burden of diagnosing test software, and its associated DUT, is to always collect the data you need to debug a problem. You will need to balance the benefits of collecting extra data with the costs in performance and time for your test software.

Just as important as the standard types of data (e.g., test limits, measured values and pass-fail judgments) are the contextual data. Contextual data are used to communicate everything relevant to the DUT's testing environment. This includes the test-system configuration, software version, driver versions and other factors.

The more variables you record, the more correlation points you and your colleagues can analyze during debug. For example, in one particular manufacturing test situation, a DUT would fail in the afternoon. The test engineer was able to correlate the time of day to the time of the failure and use that information to look more closely at a photoelectric component of the DUT. It turned out that sunlight would strike that component directly at certain times of the day, causing the component to charge a capacitor and cause the test to fail. A DUT may fail due to the temperature variations or relative humidity. Capturing contextual information and measurement conditions can save days of effort.

You want to ensure the writing or formatting of your data does not affect the behavior of your test system. Today's PCs use a variety of caching techniques that can dramatically affect how long it takes for a given file or network I/O command. If the time it takes to cache your data varies between each test run, you will get inconsistent test results. For that reason, it's a good idea to keep your data in RAM until the end of your DUT testing and then do your formatting and data transmission.

Data is useless unless it can be understood. Features of good data include:

- **Identifiable**—information to identify the circumstances surrounding the data and the condition in which it was collected.
- **Searchable**—regular structure or fields that are uniquely identifiable, making it easy for a script or software tool to identify and compare across multiple records or datasets.
- **Transformable**—raw data must be interpreted and displayed (insight is the goal). This means that software algorithms can perform operations on some or all of the fields of your data and create a new data format or data visualization based on your original data.
- **Permanent**—data must remain available and comprehensible. Relational databases tend to be the best choice for long-term storage of data as these databases are highly searchable. If your company does not already have a database for manufacturing information, you may want to consider a database solution. This decision depends on your company's data storage policies, practices and budget.[5]

Table 2 lists some common data file formats and relevant characteristics.

Table 2. File data format comparisons

	Binary	Unformatted text	Comma-separated variables (.csv)	XML (Extensible Markup Language)
Identifiable	Only with special tools	Only for small data sets	Needs good column format design	No major issues
Searchable	Only with special tools	Difficult and error-prone	No major issues	Excellent, but requires XML expertise
Transformable	Only with special tools	Difficult and error-prone	No major issues	Excellent, but requires XML expertise
Permanent	Only with special tools	Only for small data sets	No major issues	No major issues
Example: spreadsheet analysis	Only with special tools	Not importable	Supported by Excel, others	Excel 2003 format available

5 *Tufte, Edward R. "The Visual Display of Quantitative Information." Graphics Press, 2001.*

Binary formats have the fundamental issue of not being self-describing. In addition, you need to acquire a separate software application to interpret the data. Depending on the software application you use for interpreting the data, you also may be limited in the number of transformation functions.

Text files are hard to search and transform, and are not very identifiable. Since plain text files do not have regular fields, a text search for the number 12, for example, could return the hour twelve, the limit value 12, or the DMM measurement 12.

Comma-separated value (dot-csv) text formats are a good choice since they are easy to import into Microsoft Excel. With Microsoft Excel, it's easy to make a table of results with the rows containing the results and each column containing a unique description. Another advantage is most data analysis software can easily read this format. The downside of this format is that it cannot store hierarchical data or easily parse data sets. You must decide up front as to the number and types of columns, with each column containing one unique data field.

XML[6] is self-describing, very transformable, and has excellent search characteristics. There is an XML language called Extensible Stylesheet Transforms (XSLT) that can apply arbitrary algorithms to convert your XML data into new XML formats, HTML, or simple text formats.[7] A number of data analysis programs, including Microsoft Excel 2003, can import XML data.[8] If you fail to output your data in the right XML format for an analysis tool, you can write a relatively small XSLT that will convert all your XML data into the desired format. XSLT also provides a powerful search feature, making it much easier to identify data values or data structures.

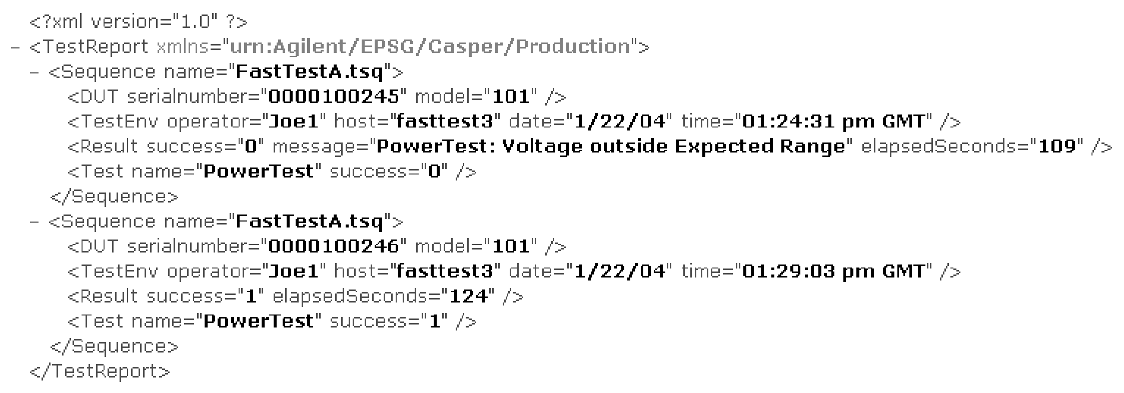
The manufacturing test industry has already begun adopting XML. Some test executive applications support XML data logging. There is a standard called IPC 2547 [9] that defines an XML format for communication of manufacturing test data.

Figure 6 is an example of a standard test run in XML format. You will still want to know the test sequence ID, the variant of the test, if the test limits are modifiable on the "PowerTest" and the hardware configuration of the test system.

If this were a .csv file, we would have to create a field for every record to answer those questions. Using XML, we can insert a record type called <TestSequence ID="32"> and fully describe the current test sequence in that record. We can then add an XML attribute called "IDREF" to refer to that test sequence record in our <TestRun> records.

In summary, the data format you choose will have a large impact on its value over time. You need to consider how easy or difficult it will be for someone else to read and interpret the data once you are no longer involved in the project.

Figure 6. XML report file



```
  <?xml version="1.0" ?>
- <TestReport xmlns="urn:Agilent/EPSG/Casper/Production">
  - <Sequence name="FastTestA.tsq">
      <DUT serialnumber="0000100245" model="101" />
      <TestEnv operator="Joe1" host="fasttest3" date="1/22/04" time="01:24:31 pm GMT" />
      <Result success="0" message="PowerTest: Voltage outside Expected Range" elapsedSeconds="109" />
      <Test name="PowerTest" success="0" />
    </Sequence>
  - <Sequence name="FastTestA.tsq">
      <DUT serialnumber="0000100246" model="101" />
      <TestEnv operator="Joe1" host="fasttest3" date="1/22/04" time="01:29:03 pm GMT" />
      <Result success="1" elapsedSeconds="124" />
      <Test name="PowerTest" success="1" />
    </Sequence>
  </TestReport>
```

6 *Extensible Markup Language: http://w3.org/xml.*

7 *Holzner, Steve. "Inside XML." New Riders, 2000.*

8 *XML in Microsoft Office: http://www.microsoft.com/presspass/press/2002/Oct02/10-25XMLArchitectMA.asp.*

9 *IPC 2547: http://webstds.ipc.org/2547/2547.htm*

Designing the user interface

When a user logs into a test system, what they see should depend upon their user class. The user class could be an operator, test engineer, technician, or service and calibration engineer. A well-written SRS will define the commands and/or menu selections available to each user class. You will want to provide each user class with only the capabilities and information they need to do their job. The more choices you provide, the greater the possibility for confusion and mistakes.

To ensure security, you can create a unique login for each of the users. Each user login should be linked to the appropriate class.

You can verify that your GUI meets the users' needs with a methodology called "User-Centered Design", or UCD, which consists of prototyping and storyboarding.[9,10] In general, a test system's GUI should be able to:

1. Customize its behavior based on the user class.
2. Provide or allow input of detailed information about the DUT.
3. Provide information about the state of the system.
4. Provide operations for controlling the system's state and potentially its configuration.
5. Display the DUT testing results.

Double footnote 9 on pg 8.

No text reference to footnotes 9 and 10 on this page.

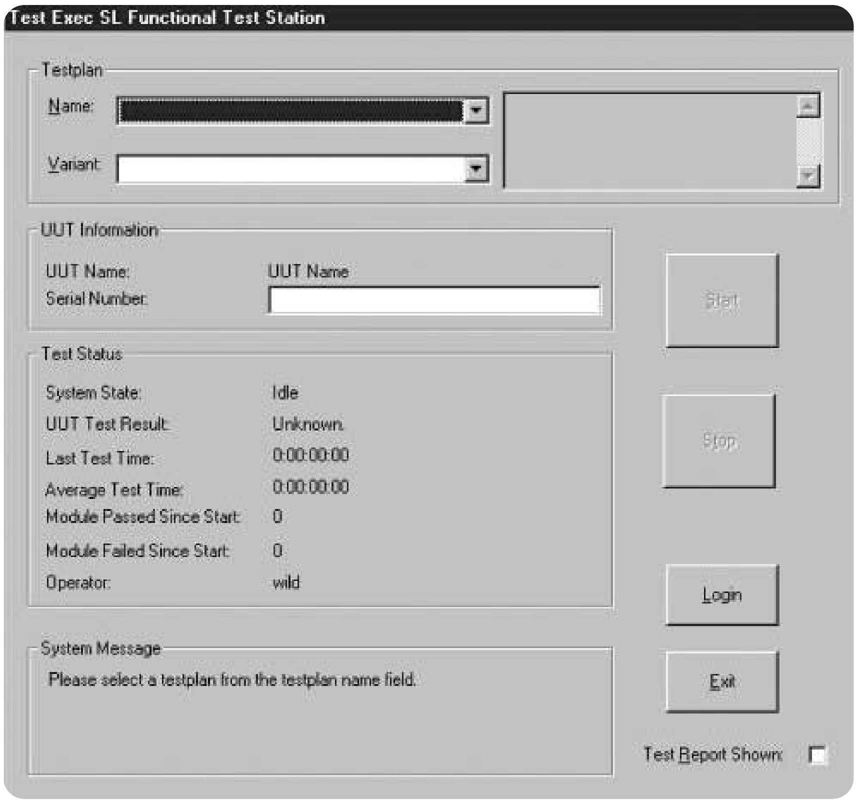
For an operator, the interface you design should always show the state of the test system (e.g., running a test, paused or stopped). For example, you could use a large color-coded graphic on the PC monitor in conjunction with lights mounted on the test system. The operator also will need a way to control the state of the test system as well as a way to input DUT information (unless this is done automatically via a bar code scanner).

As a general rule, the test program you design will require the following.

1. Commands for starting and stopping the test sequence.
2. Commands for sending test results to various kinds of printers (defect report ticket, etc.).
3. Control of the behavior of the test sequence (i.e., picking a DUT variant from a drop-down list).
4. A way to display a more detailed description of test results. The quality of a test results message can help in providing a quick diagnosis of a user error or a recurring hardware problem and may ultimately eliminate the need for a test engineer to visit the factory floor.

The user interface shown in Figure 7 was designed for an operator in a low-to-mid-mix/high-volume test application. The operator starts by logging into the test system, selecting the name and version of the testplan and entering the DUT information. The test status portion of the display is a little less prominent and visible than recommended for a manufacturing test environment, which may necessitate the addition of test status lights to the test system.

Figure 7. Low-mix, high-volume user interface



Test Exec SL Functional Test Station

Testplan
Name:
Variant:

UUT Information
UUT Name: UUT Name
Serial Number:

Test Status
System State: Idle
UUT Test Result: Unknown
Last Test Time: 0:00:00:00
Average Test Time: 0:00:00:00
Module Passed Since Start: 0
Module Failed Since Start: 0
Operator: wild

System Message
Please select a testplan from the testplan name field.

9 Vredenburg, Karel, et al, "User-Centered Design, an Integrated Approach." Prentice Hall PTR, 2002.

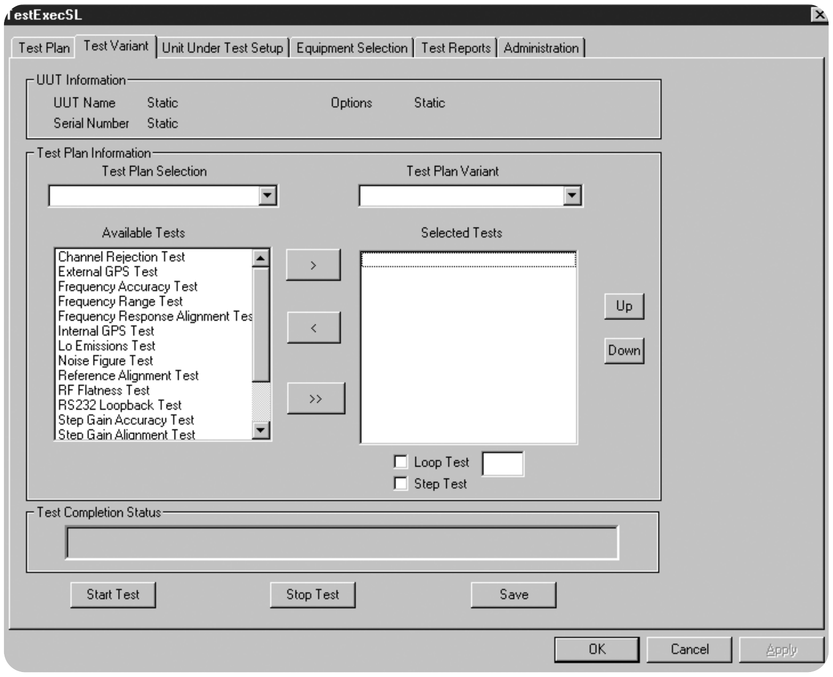
10 Norman, Donald A., "The Design of Everyday Things." Basic Books, 2002.

The system message field displays the test result information as well as instructs the user on what to do next. To help the test engineer during the debugging process, the system message field also can display error messages.

The user interface shown in Figure 8 was designed for a high-mix, very low-volume testing situation (e.g., cell phone base stations). It also can be used for test sequence development or debugging. The class of user for this interface is highly skilled and possesses detailed knowledge of the purpose and function of the available tests, the DUT, and the test system configuration. An unskilled test operator would not be able to use this interface effectively.

The two GUIs were created with the same test software, though they vary considerably in complexity. The operator GUI in Figure 7 hides unnecessary choices and information critical to the software developer.

Figure 8. Software developer's interface



Choosing the development environment

The next step in choosing your test-system software architecture is to select a software development environment. The software environment and tools you choose will have a significant impact on the overall cost of your test system. When choosing your software environment, consider more than just the purchase price of the software. You need to consider how easy it is to learn and use the software, how hard it is to connect to other languages, devices or enterprise applications, as well as support and maintenance costs. Over the life of a test system, just software support and maintenance costs can exceed hardware costs.

You have a number of options when it comes to software development environments, from writing everything yourself in a language such as C, C++, C#, VB, VB .NET, VEE or LabVIEW, to using an off-the-shelf test executive with pre-written third party tests. The software environment you choose needs to accomplish two goals: 1) meeting your time-to-first-test requirements and 2) meeting your test-throughput requirements. How fast can you get your test system up and running, and how can you get the greatest throughput?

Software development environments can be grouped into two categories: graphical or textual. Graphical environments, such as Agilent's VEE Pro 7.0 (see Figure 9) or LabVIEW, are considered easy for engineers to learn and use, largely because of engineers comfort with the schematic environment. In addition, it is easier to modify small to medium size graphical programs versus textual programming languages. Historically, textual programming languages ran faster in the manufacturing environment and yielded higher throughput. Today, there is less difference between the runtime speeds of a graphical environment and a textual environment.

Even though graphical environments are easier to use than textual environments, textual environments are used more commonly in manufacturing test systems. Only about 22% of the half-million-plus users who write code for test and measurement equipment use a graphical programming language.[11]

Graphical or textual programming?

Before you can decide on which development environment is best for your application, it's important to understand the use model of each in greater detail.

11 For more information on development environments, refer to www.agilent.com/find/vee; www.softwire.com, www.ni.com/labview, and Richter, Jeffrey, Applied Microsoft .NET Framework Programming, Microsoft Press, 1 edition, January 23, 2002.

Graphical programming is accomplished by manipulating images, called icons or objects, and the lines that connect these images. The images represent pre-made commands while the lines represent the program flow, control points, and /or how data are generated and consumed. The icons and interconnecting lines are contained within the integrated development environment (e.g., the software program).

Many graphical programming environments provide the ability to create compiled or packaged programs that do not need the programming environment to run. There are several graphical programming environments targeted at test and measurement engineers. These programs tend to have extensive I/O and instrument drivers, and T&M-specific math and graphing operations.

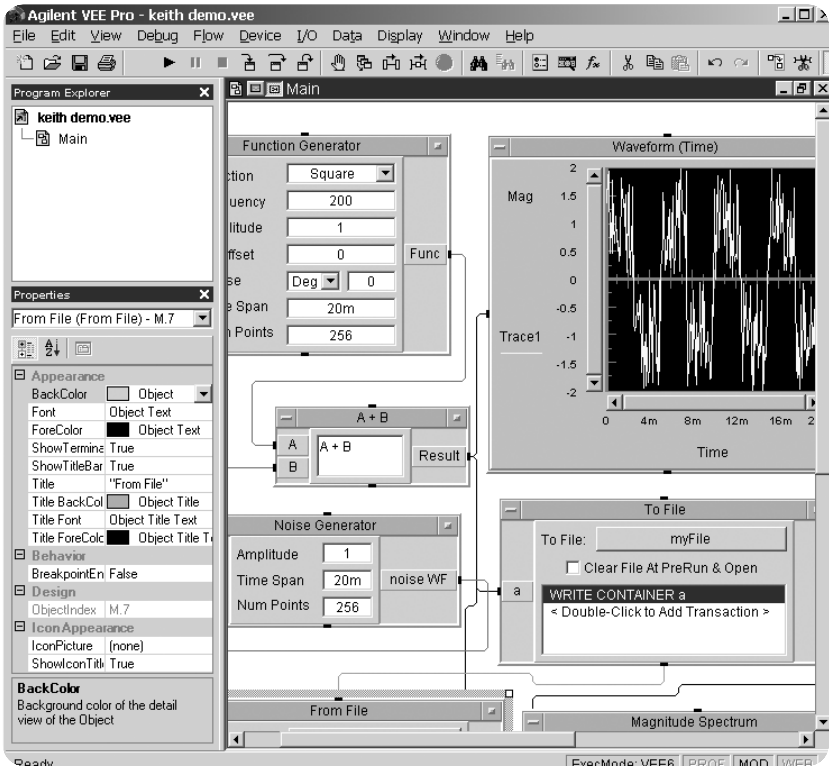
Some of the advantages of graphical programming languages over textual languages are as follows:

1. **No complex syntax**–The program instructions, typically presented as a group of icons connected by lines, are more immediately understandable.

2. **Easier to visualize the paths of execution and interaction**–Multiple concurrent activities rely on what is called a data-flow model, where a command needs to have all its data available before it will execute. This is easier than using multi-threaded programming techniques in textual programming languages such as C++ or Java.

3. **Can use real life metaphors**–The icons representing the commands can use metaphors (images) that represent real-world equivalents of the actions carried out by the icon. Most test engineers find graphical programming to be more intuitive and user-friendly than textual programming.

Figure 9. Agilent VEE Pro 7.0 graphical programming environment



Properties

From File (From File) - M.7

Appearance	
BackColor	Object
Font	Object Text
ForeColor	Object Text
ShowTermina	True
ShowTitleBar	True
Title	"From File"
Title BackCol	Object Title
Title Font	Object Title Text
Title ForeColo	Object Title T
Behavior	
BreakpointEn	False
Design	
ObjectIndex	M.7
Icon Appearance	
IconPicture	(none)
ShowIconTitl	True

BackColor
Background color of the detail view of the Object

Function Generator: ction Square; uency 200; litude 1; ffset 0; se Deg 0; e Span 20m; n Points 256

A + B: A + B

Noise Generator: Amplitude 1; Time Span 20m; Num Points 256

To File: To File: myFile; ☐ Clear File At PreRun & Open; WRITE CONTAINER a; < Double-Click to Add Transaction >

Agilent VEE Pro 7.0 and T&M Programmers Toolkit

Agilent VEE Pro 7.0

- Description—easy to use, powerful graphical instrument programming environment
- Applications—data acquisition, design, low volume manufacturing test
- Purpose—graphical program creation to acquire and analyze instrument data
- Features—easy test-system control, sequencing, support of Microsoft .NET framework, MATLAB® analysis and visualization, full support of ActiveX

Agilent T&M Programmers Toolkit

- Description—test code development (in VB .NET, C++ or C#) integrated into Visual Studio .NET
- Applications—design characterization, design validation, manufacturing
- Purpose—writing complex programs with a variety of drivers in a PC standard environment
- Features—instrument I/O and communication, test code debug, data collection, display and analysis, support for IVI-C, IVI-COM, VXI*plug&play* drivers

4. **Rapid prototyping**—With the intuitive nature of a graphical programming language, it can be easier to quickly build a prototype of your system. The prototyping capability is less useful when dealing with a large test system, but prototyping can aid development of systems of any size.[12]

5. **Easier to share and learn existing programs**—Using real-life metaphors as visual cues can make it easier to share and learn existing programs and increase productivity.[13]

Textual programming languages use special words and syntax to represent the program's operations and flow. Most, but not all textual programming languages are based on open standards. This means you will have a choice of vendors when it comes to your programming environment and software tools. Textual programming languages have a much larger set of third-party drivers, tools, and add-ins because they are based on open standards and are more widely used than graphical languages. This benefits the test engineer.

Some of the advantages of textual programming languages over graphical languages are as follows:

1. Textual programming languages are better suited for creating larger, more comprehensive programs.

2. For larger programs, textual programming languages are easier to navigate and comprehend. A person can observe only about 50 graphical objects at a time before the information becomes too complex or too small to see.[14] If a user is forced to move around in a program to see all its objects, he or she can lose track of the control and data lines and find it difficult to understand the overall flow of the program. With that said, you can improve the understandability of large graphical programs by breaking up the program's large operations into smaller sub-operations. This is called functional decomposition and is achieved by putting a series of commands into a "black box". You then send commands to the functional block and receive its output as appropriate. Make sure the graphical program you use supports this functional decomposition[15] if you plan on working with larger programs in a graphical environment.

3. While the use of a textual programming language can improve overall system throughput, it's the time spent during instrument operations that will have a greater impact. For example, you'll see a negative impact on performance in a test system where the DUT to instrument switching is inefficient, independent of which programming language is used (graphical or textual).

4. You also have a greater choice of development environments with textual programming languages. For example, there are few graphical programming languages that have development environments provided by multiple vendors. This means that today's graphical languages are less likely to have the advantages created by competition between vendors.

Graphical programming tends to be easier to learn and comprehend while textual programming is more pervasive and open. Table 3 summarizes the differences between the two programming environments.

Table 3. Graphical versus textual programming

	Graphical	Textual
Free and open	Few open standards, less extensible	Dominated by open standards, very extensible
Rapid prototyping	Excellent T&M prototyping features	Some code wizards, (T&M Programmers Toolkit, for example) but slower
T&M support	Several graphical environments targeted at T&M, many drivers	Several T&M-specific 3rd-party tools available, many drivers
3rd-party tools	Hundreds	Tens of thousands
Learnable and shareable	Easy to pick up and use programs programs are easy to share	Only small or very-well-designed

Working with open standards

In addition to choosing between graphical and textual programming, you need to consider whether the environment you choose will be based on industry standards or propriety, vendor specific technology. C++, Visual Basic, and C# are all examples of industry standard programming environments. Agilent VEE Pro and NI LabVIEW are examples of proprietary development environments although Agilent VEE Pro 7.0 does allow for access into industry standard technologies such as .NET.

Several factors to consider when deciding between an industry standard and a proprietary development environment are 1) cost, 2) industry support, 3) upgradeability, and 4) extensibility.

12 *Rahman, Jamal and Lothar, Wenzel, "The Applicability of Visual Programming to Large Real-World Applications," 1995, http://www.computer.org/conferences/vl95/html-papers/wenzel/paper.html.*

13 *Blackwell, Alan F. and Green, T.R.G., "Does Metaphor Increase Visual Language Usability?," IEEE Symposium on Visual Languages VL'99, 1999, pp. 246-253.*

14 *Begel, Andrew, "LogoBlocks: A Graphical Programming Language for Interacting with the World," 1996, http://www.cs.berkeley.edu/~abegel/mit/begel-aup.pdf.*

15 *Glinert, E. P., "Visual Programming Environments," IEEE Computer Society Press, 1990.*

Development environments for open-standard programming languages have a greater feature set and are less expensive than their proprietary counterparts. Simply stated, an open-standard environment tends to create greater competition, which in turn tends to drive down prices and create innovation.

Open-standard languages generate a lot of interest from both software tool vendors and open-source developers. Both of these groups spend considerable time understanding the needs of the test-system programmer and, as a result, develop both free and for-pay tools and applications to meet those needs. A good example is the tremendous number of C and C++ libraries available on the market, both from vendors and from end-users. These libraries save development time and money given that it is faster and less expensive for a developer to buy the domain-specific software (such as mathematical analysis libraries) than create it from scratch.

Open standard environments also have a time-to-market advantage, as most proprietary environments cannot quickly take advantage of emerging technologies. Emerging programming technologies are developed with the most common open standard programming languages in mind. It takes longer for a vendor to release a new version of proprietary software that takes advantage of new technology.

The .NET framework—The .NET Framework is an open, multi-platform, multi-vendor set of software technologies for programming computers. The C# language has been submitted to a standards body as an open language. The underlying .NET "Common Language Infrastructure" technology, also an open standard, is available in multiple operating systems, including Microsoft's Windows and Linux.

The .NET technology has excellent support and applicability to both web development and PC software development environments. The .NET technology has many of the advantages of Java language without many of Java's drawbacks. For example, the .NET technology eliminates programmer memory leaks, makes software deployment easier, and provides a rich Application Programming Interface (API) for system and GUI development. The .NET technology is fully compiled via a Just-In-Time (JIT) compiler. The JIT compiler takes the operating system (OS) and platform-independent code and creates optimized, machine-level code for the target platform.

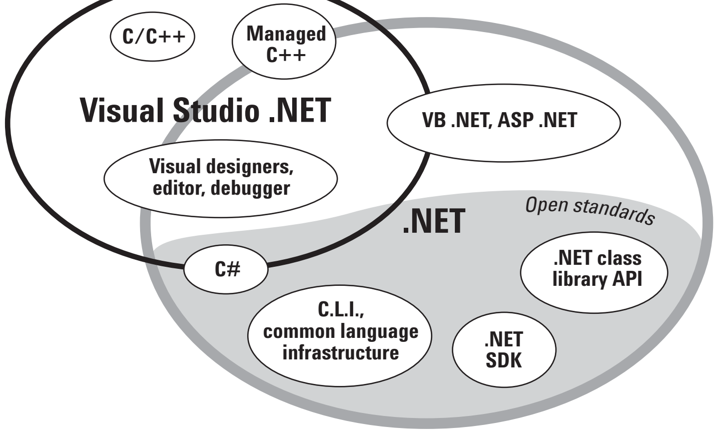
While there is some additional overhead required to load the .NET framework runtime, programs written with .NET are comparable, or run faster, than their C/C++ counterpart.[16] The reason programs can run faster in the .NET environment is due to the inefficiencies inherent in the linker operation of older languages.[17]

A survey of programmers and a number of case studies have shown significant improvements in productivity via the .NET environment over the programmers' old environment.[18]

The .NET Framework (the collection of API services and helper code used by the .NET languages) is not the same thing as Visual Studio .NET. Visual Studio .NET is Microsoft's development programming environment with support for the .NET technologies. As shown in Figure 10, there are multiple .NET development environments and programming languages available from a number of different vendors and supported on multiple platforms.

The best-known .NET languages are C# and Visual Basic (VB) .NET. C# is a lot like Java in structure and features, but its syntax is meant to be an evolution of C++. A C++ programmer familiar with object orientation and exception handling could easily move to the C# programming environment.

Figure 10. Programming languages within the .NET framework



16 *Wilson, Matthew, "Does C# Measure Up?," Windows Developer, Volume 2, Issue 13, Fall 2003, http://www.wd-mag.com/wdn/webextra/2003/0313*

17 *Johnson, Mark S. and Miller Terrence C., "Effectiveness of a machine-level, global optimizer," 1986, http://portal.acm.org/citation.cfm?id=13321&dl=ACM&coll=portal*

18 *http://www.microsoft.com/net/casestudies*

VB .NET is an upgrade to Visual Basic 6. Engineers with existing VB 6 applications must use an upgrade wizard to migrate to VB .NET. Once the upgrade process is complete, access to .NET applications and the additional power and flexibility provided by .NET can be achieved.

Microsoft's C++ language also has been enhanced to include a new version called Managed C++. Managed C++ makes it easier to execute calls within the .NET software. Microsoft provides the original unmanaged C++ in Visual Studio .NET as well.

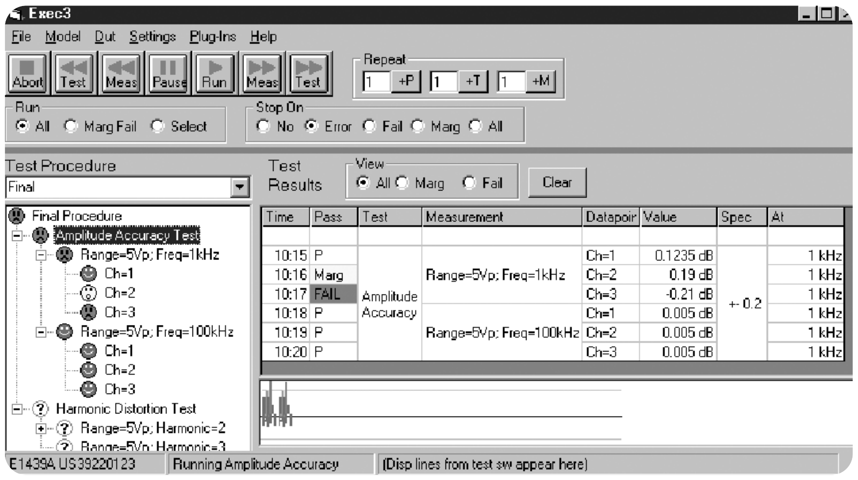
One significant advantage of .NET over older programming technologies is its extensibility. Microsoft engineered .NET so that it avoids a lot of the DLL installation frustrations Windows programmers experienced in the past. There are already a large number of third-party tools for .NET. Many of these third-party controls (i.e., advanced graphing visual controls) are useful to test-system programmers. Additionally, several test and measurement vendors, including Agilent Technologies, National Instruments, and Measurement Computing, have released .NET-compatible tools. For a complete list of released .NET-compatible tools, refer to Microsoft's .NET partner web site at **www.vsippartners.com.**

Agilent Technologies' first add-in for Visual Studio .NET is called the Test and Measurement Programmers Toolkit (see the sidebar on page 11 of this application note). The T&M Programmers Toolkit provides I/O tools, graphing and mathematical libraries, T&M specific help and example generators, and .NET wrappers for instrument drivers and other software. The T&M Programmers Toolkit is fully integrated into the Visual Studio environment. For more information on Agilent's solutions, go to **http://www.agilent.com/find/toolkit** or **http://www.agilent.com/find/iolib.** To download .NET-related I/O source files, which also work with the Agilent I/O Libraries, go to Agilent Developer Network (ADN) Web site at **http://www.agilent.com/find/adn.**

Developing a test sequence

In a survey of more than 2,500 test and measurement equipment users, 93% of the respondents said they use multiple test instruments and /or are connecting their test instruments to a PC. Of that, 37% said they use a commercial test executive for test sequencing. The remaining 56% of these respondents use internal or "home grown" software for test sequencing.

Figure 11. The test executive test sequencer



Repeat: 1 +P 1 +T 1 +M

Run: [✓ All] Marg Fail Select

Stop On: No [✓ Error] Fail Marg All

Test Procedure: Final

```
Final Procedure
  Amplitude Accuracy Test
    Range=5Vp; Freq=1kHz
      Ch=1
      Ch=2
      Ch=3
    Range=5Vp; Freq=100kHz
      Ch=1
      Ch=2
      Ch=3
  Harmonic Distortion Test
    Range=5Vp; Harmonic=2
    Range=5Vp; Harmonic=3
```

Test Results — View: [✓ All] Marg Fail

Time	Pass	Test	Measurement	Datapoir	Value	Spec	At
10:15	P	Amplitude Accuracy	Range=5Vp; Freq=1kHz	Ch=1	0.1235 dB	+- 0.2	1 kHz
10:16	Marg			Ch=2	0.19 dB		1 kHz
10:17	FAIL			Ch=3	-0.21 dB		1 kHz
10:18	P		Range=5Vp; Freq=100kHz	Ch=1	0.005 dB		1 kHz
10:19	P			Ch=2	0.005 dB		1 kHz
10:20	P			Ch=3	0.005 dB		1 kHz

E1439A US39220123 | Running Amplitude Accuracy | (Disp lines from test sw appear here)

A test executive is a software application designed to run a series of tests quickly and consistently in a predefined sequence. If any of the tests within the test sequence fail, then the DUT fails. Over the years, test executives have improved considerably both in terms of flexibility and capabilities. First-generation test executives were language-specific and not powerful enough for a mission critical manufacturing environment. Second-generation test executives, such as Agilent's TxSL and NI's TestStand are more powerful but more expensive. They also lack the flexibility required for a low-volume, high-mix manufacturing environment.

Each of the tests within the test sequence is a separate module. Commercial test executives come with a standard set of test modules and allow the user to create additional test modules from scratch (as well as customize existing test modules). Test executives control the data to and from the test module and, after collecting and analyzing all of the data, determine if the DUT passed or failed.

One reason for using a test executive is it provides a structured framework for manufacturing test systems. Test executives work best in medium- to low-mix, and medium- to high-volume manufacturing test environments.

Test executives are written so that sequence design, individual test design, and test limits and configuration management are treated as separate tasks. Keeping the three tasks separate results in greater flexibility, higher quality, and an increased opportunity for code reuse. It is the test executive that provides the infrastructure and helper services required to connect each of the separate tasks into a complete program.

One of the most important features of a test executive is its test sequencer. As shown on the left side of Figure 11, the test sequencer is a sequence of tests that can be manipulated in design mode. Various test executives provide different levels of flexibility in this sequence, such as “test looping.”

At a minimum, test executives should perform the following tasks.

1. Capture the results (and any extra data) using their own data collection model.
2. Keep track of the test limits and test setup data, passing the setups to the tests at execution time.
3. Provide limit checkers.
4. Provide run-time analysis of the test results (pass or fail reporting).

Additionally, test executives may include a software repository for maintaining the test modules (and for encouraging the reuse of tests). With a software repository, the test engineer can look for a specific test by doing a search within the test module repository. If all the engineers in a company settle on one test executive, it then becomes possible to share test modules between different product and manufacturing groups.

Test executives may use a switching model that makes it possible to map the physical layout of the test system’s control and data lines (and any switch boxes) to the DUT and instrument’s I/O pins. This allows the test engineer to think in terms of logical connections between instruments and the DUT, rather than worry about how the system is wired.

Finally, some test executives include tools for building the operator interface. While this feature tends to be less flexible than using one of the development environments discussed earlier, it does provide a fast and simple alternative.

Planning for software reuse

Aside from the use of standard libraries and operating system API’s, most software reuse tends to be opportunistic. A typical reuse scenario is when a programmer encounters a problem and remembers a similar problem handled by a co-worker. The programmer searches through the old source code of previous programs to find the desired code. If the code is found, the programmer decides how and if the software can be adapted to the current test situation. After modifications are made, the software must then to be re-verified. Most of the time, adapting software in these situations is faster than creating software from scratch.

The scenario above could have been improved with a systematic software reuse approach. The advantages of a systematic approach is in the reduced time it takes to search, find, verify, and adapt test code for new test situations. A systematic reuse approach requires following specific coding and architectural styles, as well as adherence to standardized company policies and practices.

Discussing all of the considerations for implementing a complete company-wide systematic reuse program is outside the scope of this paper, but there are decisions you can make to help achieve a more systematic approach for yourself, your team, and even your company. Reuse considerations should begin after you’ve gathered system requirements and before you begin the software development.

Professional test executive or custom software?

How do you decide if you should create your own test executive or buy an off-the-shelf version? Here are a few factors you will need to consider.

1. The first thing to look at is whether you need a test executive at all. If you don’t have a relatively fixed sequence of tests, test executives are probably not right for you.
2. If your company has an internal test executive, or more likely, several internal test executives, you’ll need to investigate their quality, features, availability of support, and the collection of tests or other auxiliary software available for them.
3. If you find a reasonable choice, it doesn’t hurt to look at the cost of porting existing code over to use a professional test executive.
4. You may decide to use a professional test executive because of its support, quality or features.
5. A professional test executive most likely will have better outsourcing characteristics. Third-party software contractors and consultants may already have experience with such a test executive, and third-party libraries may be available.
6. A professional test executive should include a complete set of documentation.

If you choose to go with a professional test executive, make sure it’s from a company that provides high-quality service and support.

The design reuse process

The first step in the design reuse process is to complete a domain analysis. This is accomplished by 1) systematically analyzing the functions and parts of your software domain, and 2) using this information to develop a software architecture with well-defined component types and algorithms.

Next, you will want to look for natural boundaries in your software. One software design practice of finding and documenting the natural boundaries is known as Design Patterns.[19] To find the natural boundaries, look to those areas where one type of activity or data set links with another type of activity or data set. These areas can then be grouped into separate modules and documented accordingly. Once documented, the same type modules can then be swapped for one another.

Once you have identified, collected and documented your modules, components and /or individual parts, you will need to thoroughly test them before they go into the repository (or are passed on to your co-workers). This will save you and your co-workers from problems later in the process.

Finally, reusable components are reusable only if your co-workers know they exist. You need a repository (such as a relational database) for your modules where anyone in your team, division, or company (if appropriate) can browse and search for them based on what the components are and what they do.

19 *Shalloway, Alan and Trott, James R., "Design Patterns Explained: A New Perspective on Object-Oriented Design," Addison-Wesley Pub Co, 2001.*

20 *This is a good example of a design pattern specific to the test and measurement domain.*

A design reuse example

A good model for design reuse of individual test modules is the test executive—here's why.

1. Some test executives break test software up into swappable tests, sequencers, limit checkers, test sequence and test limit data.[20]

2. Test executives rely on the concept of modules. For example, you can have a module that provides the ability to perform a single pass or fail judgment, including the sequencer data type, the sequence execution operation, and the test types.

3. Test executives allow reuse of tests in different test sequences with no change to the test code. The sequencer provides the necessary data to the tests to customize their operation for the current test sequence.

4. Test executives keep the tests in separate modules or files from the test sequencer or test executive application. This allows you to easily swap tests in and out without recompilation.

5. Some test executives allow you to write your own custom limits checkers or sequencers.

All of these modules are able to interoperate because test executives use well-defined application programming interfaces (APIs) for each module. The modules are placed on natural boundaries between different types of data and functions within the test executives.

You can achieve similar reuse success in your own code with good architecture influenced by the natural boundaries of your software's functions, types and data. To accomplish this, put information that changes frequently, such as the limits for a test, into a Data File. Put less flexible elements, such as a test class, into Types or "classes." Functions, or "procedures," should be reserved for the least flexible elements.

Design reuse and .NET

While the definitions of the boundaries of your software domain are not specifically influenced by the programming language or software environment, some environments are better than others in helping to keep your software modular and swappable.

.NET provides software tools that make it easier to develop a formal software reuse program within your department or company. Since .NET is object-oriented, it's good at representing boundaries between different types of objects, such as tests or sequencers. Nonobject-based languages, such as C, require you to keep track of which functions apply to which objects, without much context-sensitive help or compile-time error checking.

.NET also includes improved versioning and deployment features. In addition, .NET has the ability to tell Windows that you will only accept a certain version of an external library. This eliminates one of the common frustrations with earlier versions of Windows where you rely on an external library (DLL), but then the DLL changes and your software no longer functions correctly.

Design reuse benefits

In summary, the reasons for implementing a design reuse program include improved software quality, increased software development efficiency, and better use of expert knowledge.

Design reuse improves quality in a couple of different ways. First, software errors are reduced as a result of the extra architectural analysis, improved system design, and flexibility and transparency. With good reuse policies implemented throughout the organization, you have access to thoroughly tested and verified components, reducing the opportunities for creating new defects.

Design reuse increases software development efficiency by reducing duplication of effort. Components need to be designed, implemented and tested only once. Good reuse practices make it easier to reuse an existing component as opposed to re-writing or even re-creating a new component.

Design reuse takes advantage of an organization's expert knowledge. For example, most software developers spend time specializing on a particular set of skills and will write components based on those skills. With time, the set of available components for reuse becomes the set of the best knowledge of your organization. The company's expert skills and deep knowledge will be evident in a rich set of reusable software components.

These benefits are not theoretical. The Software Engineering Laboratory at the National Aeronautics and Space Administration's (NASA) Goddard Flight Center achieved significant benefits by implementing software reuse in the development of software products in its Flight Dynamics Division. According to the software engineering lab, NASA realized a 35% reduction in the effort needed to deliver a line of code, a 53% increase in daily productivity, and an 87% increase in code quality.[21]

Design reuse summary

Systematic design reuse across your company requires that your management value the extra efforts required by designing for reuse. Failure to invest and do the job right the first time will lead to frustration and wasted time down the road. One or more repositories of software components must be made available to all the engineers who will need them. You also need to be aware of any copyright or patent limitations of the code you plan to reuse. For example, if your software is written under contract with another company, they may have exclusive rights to that code.[22]

Summary

Before you begin writing code for your test system, you need to make a number of important decisions about the system's software architecture. You will want to start by creating a detailed software requirements specification that defines what you want the system to do and how it should operate. The SRS should include an outline of how you will gather, store, analyze and present your data as well as how end users will interact with your system.

Another important decision you need to make upfront is which programming environment and language you will use for writing your code. Using a standards-based environment such as Visual Studio .NET maximizes your flexibility and helps you prolong the useful life of your software. By combining Microsoft's Visual Studio .NET with Agilent's T&M Programmers Toolkit, you can wrap objects written in a variety of languages such as Agilent VEE Pro 7.0. This allows you to pull them forward into your new programming environment, making the most of your legacy code investment.

Whether you choose a graphical or textual environment will depend on the size and complexity of your system, your skill set, your company standards, and the size of your programming team. The decision usually comes down to which environment–graphical or textual–will make you more productive. Textual environments are almost always the best choice for creating code for large, high-throughput manufacturing test systems because they offer the most power and flexibility, and they allow faster throughput.

Finally, you need to decide whether to use an off-the-shelf test executive or write your own test routines. Test executives can speed up your test-system development and lower your costs but will require an up-front training investment. If you are only performing a few tests, you may want to consider writing your own code.

21 *Proceedings of the Sixteenth Annual NASA/Goddard Software Engineering Workshop: Experiments in Software Engineering Technology, Software Engineering Laboratory, December 1991.*

22 *Defter, Frank W, et al, "Software Reuse: Major Issues Need to Be Resolved Before Benefits Can Be Achieved," United States General Accounting Office, 1993, http://www.defenselink.mil/nii/bpr/bprcd/vol2/272c.pdf.*

Appendices

Glossary

ActiveX—A standard method for encapsulating COM-compliant software modules so they can be used in standard PC applications. ActiveX controls can be used in any ActiveX-compliant application, regardless of where they were created.

ADE (Application Development Environment)—An integrated suite of software development programs. ADEs may include a text editor, compiler, and debugger, as well as other tools used in creating, maintaining, and debugging application programs. Example: Microsoft Visual Studio

API (Application Programming Interface)—An API is a well-defined set of software routines through which an application program can access the functions and services provided by an underlying operating system or library. Example: IVI Drivers

C# (pronounced "C sharp")—New C-like, component-oriented language that eliminates much of the difficulty associated with C/C++.

COM — See Microsoft COM.

Direct I/O—Commands sent directly to an instrument, without the benefit of or interference from a driver. SCPI Example: SENSe:VOLTage:RANGe:AUTO

Driver (or device driver)—A collection of functions resident on a computer and used to control a peripheral device.

DLL (Dynamic Link Library)—An executable program or data file bound to an application program and loaded only when needed, thereby reducing memory requirements. The functions or data in a DLL can be simultaneously shared by several applications.

IDE (Integrated Development Environment)—See ADE.

Input/Output (I/O) layer—The software that collects data from and issues commands to peripheral devices. The VISA function library is an example of an I/O layer that allows application programs and drivers to access peripheral instrumentation.

IVI (Interchangeable Virtual Instruments)—A standard instrument driver model defined by the IVI Foundation (**http://www.ivifoundation.org**) that enables engineers to exchange instruments made by different manufacturers without rewriting their code.

IVI COM drivers (also known as IVI Component Drivers)—IVI COM presents the IVI driver as a COM object. You get all the intelligence and all the benefits of the development environment because IVI COM does things in a smart way and presents an easier, more consistent way to send commands to an instrument. It is similar across multiple instruments.

Libraries—Files containing reusable software operations or functions meant to be used by other programs. They can be C based libraries, Visual Basic libraries, .NET libraries, COM libraries, or based on other software technologies.

Microsoft COM (Component Object Model) — The concept of software components is analogous to that of hardware components: as long as components present the same interface and perform the same functions, they are interchangeable. Software components are the natural extension of DLLs.

Microsoft developed the COM standard to allow software manufacturers to create new software components that can be used with an existing application program, without requiring that the application be rebuilt. This capability allows T&M instruments and their COM-based IVI-Component drivers to be interchanged.

.NET Framework—The .NET Framework is an object-oriented API that simplifies application development in a Windows environment. The .NET Framework has two main components: the common language runtime and the .NET Framework class libraries. New frameworks can be added by anyone.

Plug and Play drivers (also known as universal instrument drivers)—An important category of proprietary drivers. Plug and Play driver standards were originally developed for VXI instruments and were known as VXI*plug&play* standards. When these standards were adapted for non-VXI instruments they became known simply as "Plug and Play" drivers. Library functions are in accessible C-language source and you can call them from programs written in C, Basic, VEE, LabVIEW, or LabWindows/CVI.

SCPI (Standard Commands for Programmable Instrumentation)—SCPI defines a standard set of commands to control programmable test and measurement devices in instrumentation systems. Learn more at **http://www.scpiconsortium.org.** See "Direct I/O" for example.

Universal drivers—Another name for Plug and Play drivers

VISA (Virtual Instrument Software Architecture)—The VISA standard was created by the VXIplug&play Foundation. Drivers that conform to the VXI*plug&play* standards always perform I/O through the VISA library. If you are using Plug and Play drivers, you will need the VISA I/O library. The VISA standard was intended to provide a common set of function calls that are similar across physical interfaces. In practice, VISA libraries tend to be specific to the vendor's interface.

VISA-COM—The VISA-COM library is a COM interface for I/O developed as a companion to the VISA specification. VISA-COM I/O provides the services of VISA in a COM-based API. VISA-COM includes some higher-level services not available in VISA, but in terms of low-level I/O communication capabilities, VISA-COM is a subset of VISA. Agilent VISA-COM is used by its IVI Component drivers and requires that Agilent VISA also be installed.

VXI*plug&play*—A hardware and software standard that allows interoperability between VXI instruments made by different manufacturers. Learn more at **http://www.vxipnp.org**

XML (eXtensible Markup Language)—A subset of SGML constituting a particular text markup language for interchange of structured data. The Unicode Standard is the reference character set for XML content.

Related literature

Data sheets

Agilent VEE Pro, 7.0
pub. no. 5988-6302EN

Agilent Toolkit,
www.agilent.com/find/toolkit

Application notes

Test-System Development Guide:

- *Introduction to Test-System Design*
(AN 1465-1) pub. no. 5988-9747EN
http://cp.literature.agilent.com/litweb/pdf/5988-9747EN.pdf
- *Computer I/O Considerations*
(AN 1465-2) pub. no. 5988-9818EN,
http://cp.literature.agilent.com/litweb/pdf/5988-9818EN.pdf
- *Understanding Drivers and Direct I/O*
(AN 1465-3) pub. no. 5989-0110EN
http://cp.literature.agilent.com/litweb/pdf/5989-0110EN.pdf
- *Choosing Your Test-System Software Architecture (AN 1465-4)*
pub. no. 5988-9819EN
http://cp.literature.agilent.com/litweb/pdf/5988-9819EN.pdf
- *Choosing Your Test-System Hardware Architecture and Instrumentation*
(AN 1465-5) pub. no. 5988-9820EN
http://cp.literature.agilent.com/litweb/pdf/5988-9820EN.pdf
- *Understanding the Effects of Racking and System Interconnections*
(AN 1465-6) pub. no. 5988-9821EN
http://cp.literature.agilent.com/litweb/pdf/5988-9821EN.pdf
- *Maximizing System Throughput and Optimizing Deployment*
(AN 1465-7) pub. no. 5988-9822EN
http://cp.literature.agilent.com/litweb/pdf/5988-9822EN.pdf
- *Operational Maintenance*
(AN 1465-8) pub. no. 5988-9823EN
http://cp.literature.agilent.com/litweb/pdf/5988-9823EN.pdf
- *Using LAN in Test Systems: The Basics*
(AN 1465-9) pub no. 5989-1412EN
http://cp.literature.agilent.com/litweb/pdf/5989-1412EN.pdf
- *Using LAN in Test Systems: Network Configuration*
(AN 1465-10) pub no. 5989-1413EN
http://cp.literature.agilent.com/litweb/pdf/5989-1413EN.pdf
- *Using LAN in Test Systems: PC Configuration*
(AN 1465-11) pub no. 5989-1415EN
http://cp.literature.agilent.com/litweb/pdf/5989-1415EN.pdf
- *Using USB in the Test and Measurement Environment*
(AN 1465-12) pub no. 5989-1417EN
http://cp.literature.agilent.com/litweb/pdf/5989-1417EN.pdf
- *Using LAN in Test Systems: Applications,*
(AN 1465-14) (available in February 2005)

Simplified Instrument Communication and Programming Using Textual Programming Languages (AN 1409-2),
pub. no. 5988-6617EN

Other resources

Agilent Developer Network (ADN)
http://agilent.com/find/adn

To discover more ways to simplify system integration, accelerate system development and apply the advantages of open connectivity, please visit the Web site at **www.agilent.com/find/systemcomponents.** Once you're there, you can also connect with our online community of system developers and sign up for early delivery of future application notes in this series. Just look for the link "Join your peers in simplifying test-system integration."

www.agilent.com

Agilent Email Updates

www.agilent.com/find/emailupdates
Get the latest information on the products and applications you select.

Agilent Open Connectivity
Agilent simplifies the process of connecting and programming test systems to help engineers design, validate and manufacture electronic products. Agilent's broad range of system-ready instruments, open industry software, PC-standard I/O and global support combine to accelerate test system development. More information is available at **www.agilent.com/find/openconnect.**

By internet, phone, or fax, get assistance with all your test & measurement needs

Online assistance: www.agilent.com/find/assist
Phone or Fax

United States:
(tel) 800 829 4444
(fax) 800 829 4433

Canada:
(tel) 877 894 4414
(fax) 800 746 4866

China:
(tel) 800 810 0189
(fax) 800 820 2816

Europe:
(tel) (31 20) 547 2111
(fax) (31 20) 547 2390

Japan:
(tel) (81) 426 56 7832
(fax) (81) 426 56 7840

Korea:
(tel) (82 2) 2004 5004
(fax) (82 2) 2004 5115

Latin America:
(tel) (650) 752 5000

Taiwan:
(tel) 0800 047 866
(fax) 0800 286 331

Other Asia Pacific Countries:
(tel) (65) 6375 8100
(fax) (65) 6836 0252
(e-mail) tm_asia@agilent.com

Agilent Technologies