Testing Times in Computer Validation

Ian Lucas

SeerPharma Pty Ltd.

With so much written and talked about the different phases of testing, it is easy to become confused about what constitutes the 'correct' amount of testing for any specific application or implementation. This article attempts to simplify some of the many testing terms, and provide some practical guidance in performing effective testing. The advice offered is focused on process control/automated manufacturing systems, but is also applicable to other computerized systems.

As with all aspects of computer validation, a basic knowledge of the theory must be understood before testing can be effective. Although this article is a personal viewpoint, I have included several document references for further guidance on the subject.

> **"The fundamental objective of testing is to prove some assertion. This sounds so obvious as to be silly , but the biggest cause of ineffective testing I have seen is the loss of clarity of this objective."**

The fundamental objective of testing is to prove some assertion. This sounds so obvious as to be silly, but the biggest cause of ineffective testing I have seen is the loss of clarity of this objective. People lose focus about what they are trying to prove, or only superficially test assertions. It is always easier to test what is expected than what is not. The reason why users can find problems in systems that are not found during defined testing times is that they are not restricted to a defined set of test data to use.

We need to ensure that we are clear on what we are trying to prove, and that our test data truly covers all aspects of that assertion.

The next section describes how paragraphs on testing are generally written and expanded upon in project documentation.

At the highest level 'Overall Test Objective' stage, the purpose is almost always correct (e.g., show that system X works as described in the requirements specification). At this stage, objectives are very generic, so anything that conveys the general idea of implementing a quality system can be construed as correct.

At the next level down, the 'test cases' stage, test objectives are again fairly generic (e.g., show that the user security system works as expected). At this stage, errors are usually not found in what is written, but rather what is not. A common cause for test omissions is to write the cases based on the programs to test, rather than the functions to test. This then leads to an almost one-to-one mapping between programs and tests, and makes the testing easier to perform from a testing point of view. Unfortunately, from the project point of view, the idea is not to make the testing easier, but to ensure as much effective testing of the critical areas of the system is tested in the time provided.

From the test cases, the test scripts are written. These describe the actual actions to perform, and the expected results that should be observed to prove the intent of the case. There is a real skill required to be

able to write these, so that the correct emphasis is placed on the intent of the case, rather than the peripheral actions required to set-up the case.

There is also a skill required in asking the tester to produce, annotate, and verify hard copy attachments that form 'defendable proof' for that case (without producing attachments simply for the sake of documentation). There will be more information on this topic in the *Journal of Validation Technology* at a later date.

Even with a clarity of focus on the test objectives for a single test exercise, it is easy to lose focus as to where that exercise sits in relation to the total project. This can be due to the high number of testing terms, and where they all fit in relation to each other. It is also attributable to the fact that many people's definitions of these terms are different than other individuals.

Included are my definitions and general testing philosophies below to assist with the understanding of the examples supplied.

As a general rule, I'm an advocate of the 'X' model (GAMP 3)[1] of system development and testing (see *Figure 1*). This takes into account that systems are developed and tested (to varying degrees) in conditions away from their final environment.

The overall testing exercise therefore covers both environments, with common sense and experience being used to determine the amount of testing being performed in each sub phase for each individual project.

The aim is to minimize the duplication of testing throughout the phases, and to use each sub phase to build upon the previous phases to provide comprehensive testing coverage.

(Note that in the interest of not over complicating this paper, I have not added the documents created on the left side of this diagram. Varying requirement specifications and designs can be on both the USER and SUPPLIER sides, depending upon whether a product or custom solution is being developed.)

The Supplier Testing

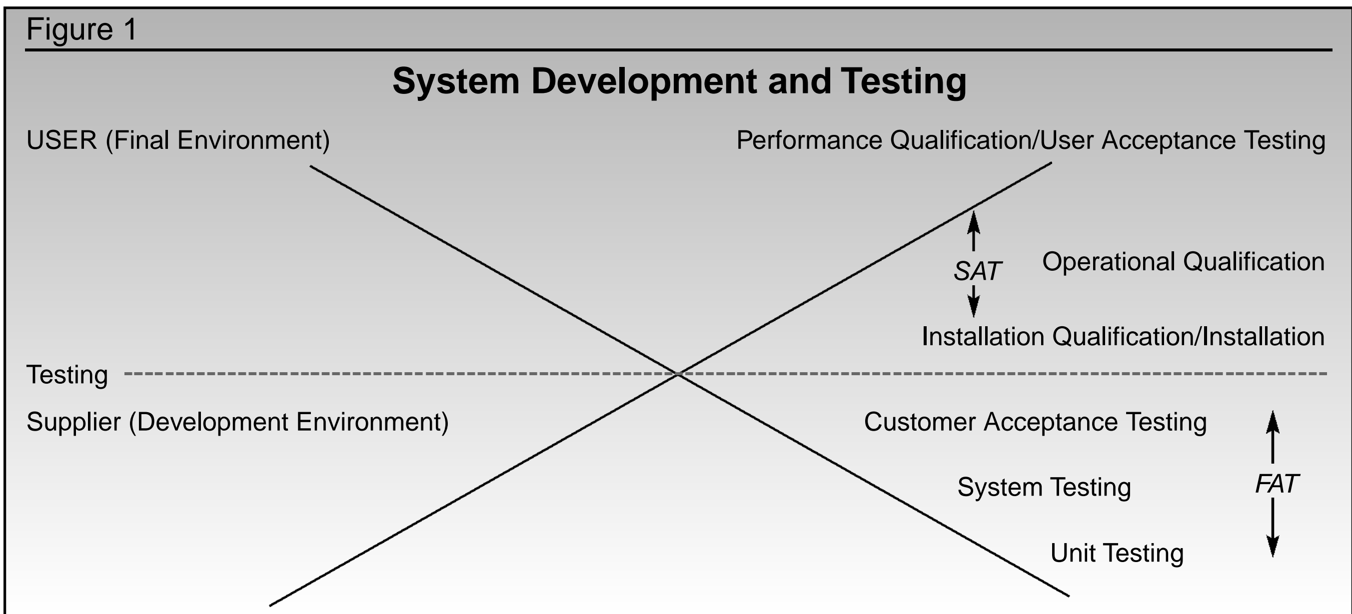
Unit Testing

For systems where new code is developed, 'Unit Testing' is used to provide structural (also referred to as 'white box') testing of this code. This involves checking that the code has been written to defined coding standards, that the code reflects the design (usually the pseudo code process specifications), and varying degrees of stand-alone module performance checks. These need to fit the criticality of the module, and involve input/output parameter checking and branch/code coverage. Note that for this testing there is an assumption that the design is correct as approved for development (a costly assumption in some cases).

As this testing is usually performed by another developer, emphasis is placed on the technical aspects, rather than the functional aspects. However, it must be noted that additional code cannot magically appear in the executable program after the source code is written. It is therefore, imperative, that the coder add a detailed level of implicit 'safety-net' (my definition) code, as well as the explicit 'requirements' code.

Figure 1

System Development and Testing



Without going into an overly technical explanation, please refer to *Figure 2*, which is the following two pseudo-code listings. Both describe the process for summing a set of numbers, working out their average, and then saving the information in a data table.

Figure 2

Pseudo-Code Listings Describing the Sum Process and Averages

Listing One	Listing Two
Set Sum = 0, Count = 0	Set Sum = 0, Count = 0
While not exit selected Read Number	While not exit selected Read Number
	If input not numeric
	Report error, and do not add/increment
Add Number to sum	Add Number to sum
Increment count	Increment count
End while	End while
	If count = 0
	Report message, and do not calculate total or store
Set average = Sum/count	Set average = Sum/count
Store average in data table	Store average in data table
	If store fails, Report error

The design pseudo-code may not explicitly have the bold lines in the Listing Two described, but it is important that the code review pick up that a level of this 'safety-net' code has been written.

For the above example, a typical functional test may be to run two or three sets of standard data through the algorithm, and check the expected result against the one in the database. Note that both sets of codes would pass this test. It would only be when non-numeric data, no data, or when a problem with the database occurred, that the code generated from Listing Two would show its robustness over Listing One.

The above example is a little contrived, but the point is that in a large system many of these 'safety-net' checks will not be tested (as time is given to testing the expected paths). The robustness of the system only comes out over time, as inadvertently, unexpected data or situations are encountered.

This highlights the importance of unit testing, and auditing the developers of critical software products to ensure that they have performed these code reviews on their products. As mentioned before, functional testing cannot put this safety net back in – only show that it is not there.

Unit Integration Testing – A Link Between Unit and System Testing

For complex systems, a level of testing of some of the integrated software modules may be warranted. This may involve writing software shell 'driver' programs, and calling up several modules to test flow and control between these modules.

Note that this control will be tested again during system testing, and should only be performed if the jump between a single unit test and full functional testing is considered a substantial risk.

System Testing

The supplier 'system testing'is where the system is functionally put through its paces. As well as the standard expected operational tests, other stress tests should be devised. These include:

- Power Tests – think about the most inopportune times to lose power on the system and try these
- Time – if applicable, wind back and forward the time on the server and client Personal Computers (PCs) to simulate daylight savings time
- Status Transitions – look for processes that use status'to determine operation, and check that these cannot deadlock the process. I have seen several processes where a product/batch had to be in status Ato initiate the process, moved to status B during the process, then finished with status C. The problem was that if the process failed in process B, there was no way to restart or finish that process on that product/batch.

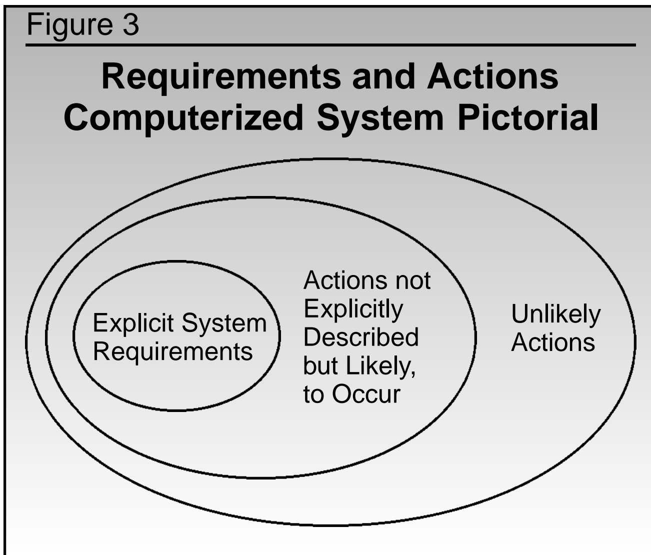
Amajor difference between system testing (also read OQ) and customer/user acceptance testing (also read PQ) is that system testing is not just trying to show the system does what it is designed to do from the specifications – it is also trying to find errors.

There is a fundamental mindset change between these two objectives that has not been captured in most guideline reference material. Take most V-model diagrams (see *Figure 4*) that associate requirements and design documents to the different testing phases. The

common thread is that tests can ONLY be written against requirements documented in those specifications. We are also constantly told that all testing must be planned before being performed.

Both of these philosophies are contributing to testing not being as effective as it should be.

There needs to be a realization that some tests need to 'think outside the specification.' In *Figure 3*, a computerized system is pictorially represented as sets of requirements and actions.



Almost all testing is quite naturally focused on the inner ellipse. If users always acted exactly as expected, this would be fine. However, over a system's life, 'actions not explicitly described but likely to occur,' most probably will. Actions including:

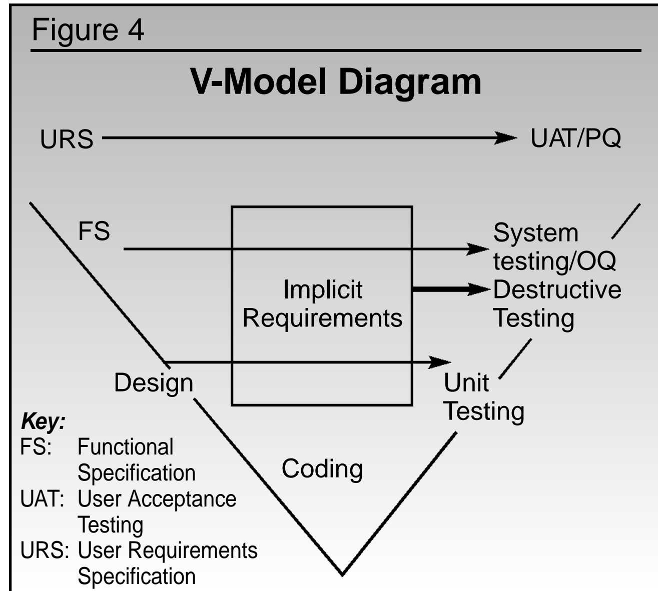
- Non-numeric values where numbers are requested
- Escape, control, function, and ALT characters pressed when not expected
- Opening and using windows out of sequence
- Running multiple unexpected processes at the same time
- Not entering mandatory data
- Data files/tables growing larger than expected

These actions cannot be stopped – only mitigated by performing expected reactions when they occur.

Note that the perfect system is not one where the tester has thought of all these unspecified actions, but where the developer has.

Given that almost all computerized systems have some degree of actions not specified in the requirements that will cause unexpected results, these are best picked up (in the development environment) in the unit testing (by another developer) or in supplier system testing.

A more relevant V-model diagram (although less aesthetic) would resemble *Figure 4*.



In this destructive testing phase (which is really a sub phase of system testing), tests would mainly be developed from common sense and experience. Guidelines should be developed to aid inexperienced personnel in areas to consider. Other tests also need to be developed by looking at common problems associated with the following areas, including:

- The software development tool (if known)
- The product itself
- The customer site
- Interfaces to the system

The idea of this testing is not to follow a structured plan, but to purposely try to find errors. All tests and results must be documented, and it will be up to the discretion of the project/validation team afterwards to review each issue to determine the likelihood of the problem occurring, the risk if it does, and the cost of the rectification.

Customer Acceptance Testing

Prior to a system being installed in the live environment, the actual users should check that they are indeed receiving what they expect. This level of testing (and to varying degrees, the system testing) is often called the

Factory Acceptance Test (FAT) for process control systems.

A set of tests should be developed from the requirements documents that test the system running through a typical process flow in chronological order. To obtain the maximum benefit out of this testing, both the writing of the protocols and the performing of the testing should be conducted by different personnel than those who performed the system testing. However, the system testing results and report should be reviewed prior to customer acceptance testing to ensure that testing time is not being wasted in simply reperforming previous tests.

Keep in mind that each testing exercise needs to be as comprehensive or narrow as deemed appropriate for the risk of the system.

For a purchased product where extensive system testing cannot be guaranteed, a full set of customer acceptance tests may be warranted; whereas for a custom developed solution where detailed unit, system and destructive testing have occurred, a reduced customer acceptance testing may be acceptable.

The Testing Exercise

In all stages of testing, the exercise can only be as effective as the protocols developed, and the skill and attentiveness of the tester. The ability to hypothesize and observe the entire system environment, whilst testing specific assertions is a valuable asset in any testing team. The idea is to not only look for the obvious expected result, but to be ready for the obscure discrepancies.

In developing protocols, a common question is "how detailed does the test step description text (also referred to as 'actions' text) need to be?"

A couple of serious problems arise when the actions are made too detailed (e.g., place the cursor over batch 12345 and press F5 (Select). Enter '1.564' in the Weight field and 'Acceptable' in the Pass/Fail field. Press F5 (Continue), then F7 (Update), etc):

❶ To write the actions, the protocol developer usually has to actually run the product with the test data to see what happens. If the tester then repeats these actions during testing, this is not independent testing, but simply confirming what the test developer did. In this way, the test developer can sway the tester in the performance of the testing. It also totally removes any testing in the ease of use of the system as the tester is simply entering data as instructed.

❷ Even with an experienced tester, the emphasis is placed on reading the many instructions and performing each step, with many more expected results than is really necessary. It is not always clear which action is significant, and which are only there to prepare for a significant step.

Another problem that can make the testing less effective is to ask for too many hard copy print outs. These form important test result evidence when requested (and annotated) correctly, but can add unnecessary time and effort when requested at each test step. Each print out takes time to physically produce, collect, collate, annotate, and review.

Also, from an auditor's point of view, producing unnecessary output shows that the testing team is not certain what the critical control points in the process are.

Generally, it is better to give the clear intent of the test with the expected result, and only aim to give as many explicit actions as is truly necessary. Of course, there will always be situations where more detail is necessary (e.g., regression test suites where the aim is to run a known set of data through a previously tested process to ensure that it still operates as it formerly did).

Annotating Printouts

Many testers fall at the last hurdle when it comes to tying up the test results. There are many formats for the test scripts, but they should minimally contain:

- Intent – what the test is trying to demonstrate
- Actions – what actions need to be performed to show the intent
- Expected Result – what constitutes acceptance of the intent
- Actual Result – what actually happened. There should be room provided to note an incident or anomaly number, and to reference hard copy printouts (if requested)
- Pass or Fail – does the actual result either match the expected result, or can it be explained as to why it doesn't match, but can still be deemed as a pass
- Signature, date, and time

An example is shown in *Figure 5*. Note that log files can sometimes be hundreds of pages long. To

simply print this, and include it as an attachment to the completed test scripts does not automatically show obvious success. As well as annotating a two-way reference between the test and the output, all relevant statements on the print out should be highlighted, initialed, and dated, and reasons written why the statement should be accepted as proof or rejected. For an example, see *Figure 6*.

Note that if follow-up actions are required as part of output justification, additional attachments (correctly annotated back to the original document) will be required for completeness. When reviewed (by an independent reviewer), all outputs should also be signed and dated to complete the 'defendable proof' package.

Another method of recording test results is to use a tester and a witness whilst testing, and not produce hard copy output. Although this may reduce the overall testing and review time, it does prevent further independent review, and is less defendable, should issues arise later.

As mentioned before, a combination of both methods (with hard copy outputs limited to critical control/decision points) is best.

Installing the Software in the Live Environment

Once the FAT has been approved, the system is ready to be commissioned and qualified in the live environment. For most implementations, this is more than just physically moving the test environment. It may involve:

- Running two systems in parallel
- Decommissioning the current system to implement the new one
- Moving current live data to the new system
- A defined time period in which to work
- Coordinating multiple activities and disciplines
- Downtime of critical business processes

From this, it is clear that an installation plan (and subsequent Installation Qualification [IQ]) is vital to ensure that the system configuration tested in the development environment is the same system to be commissioned in the live environment. Without this assurance, the case for using assertions proved in the development environment to minimize testing in the live environment declines.

Remember, effective testing relies on building the overall validation package throughout the layers of testing.

Qualifying the System

There has always been much discussion about what activities should be in the qualifying phase of a system. Statements such as 'all IQ activities should be performed with the power off' may be true for many cases, but there would be few software installation checks using that philosophy.

I believe it doesn't matter whether activities occur in your IQ, Operational Qualification (OQ), or Performance Qualification (PQ), as long as they are done.

For some systems (especially control systems), qualifying the total solution will involve many individual qualifications. For example, the physical hardware, operating system, database, and application may all be installed separately. It is important to get the order and timing of these correct to again build on previous assertions.

Installation Qualification

If the actual installation plan has been written and performed correctly, the actual IQ should be a fairly simple exercise. The intent of the IQ is to prove that the system has been installed as per the developer's instructions. This includes:

Figure 5

Annotating Printout

Intent*: To prove that the standard import feature can import an export file successfully.*

Test Number	Actions	Expected	Actual	Pass Fail	Signature	Date Time
4.5	Use the import feature to load the export file into the database. Note the log file name, print it out, and review.	Data imports without errors. All duplicate records can be explained.	Log file name:_________ Attachment no.:_______ Incident no.:__________			

Figure 6

Log File Example

Attachment To: Import File Testing
Exhibit Number: Test 45 Att: 1
Page 1 of 1 (of this attachment)
Initials: I.L. Date: 29/05/2002

test_output.txt

```
SQL> set echo on
SQL>
SQL> connect dbase/name@password
Connected.

SQL> -- ************ Alter and update timers ******************
SQL>
SQL> update current_value
  2  set description = 'Bulk SIP to Filling Timer'
  3  where point_name = 'TIMERPTM'
  4  /

0 rows updated

SQL>
SQL> update cip_sip_table
  2  set initiate = 'START SEQUENCE 225'
  3  where sequence_number = 223
  4  and step_number = 1
  5  /

1 row updated

SQL>
SQL> insert into cip_sip_version
  2  select 225, version, version_comment, user1, user2, ddate
  3  from cip_sip_version
  4  where ss_number = 215
  5  /

0 rows created

SQL>
SQL> update cip_sip_table
  2  set batch_number = 'XXXXXXXXXXXXXXXX'
  3  where sequence_number >= 410 and sequence_number <= 480
  4  /
update cip_sip_table
Error at line 1:
ORA-01401: inserted value too large for column
```

Expected. Point deleted as part of CR2002-562 Ian Lucas 29/5/2002

Expected. Version for 215 will not exist until approved. Statement will then be reperformed. See Test 45 Att: 2 Ian Lucas 29/5/2002

Expected. A valid batch number will be obtained from the live system and the update reperformed. See Test 45 Att: 3 Ian Lucas 29/5/2002

- Proving the right product has been installed (serial numbers, software versions, etc.)
- Proving the product has been installed correctly. Although it may be a fairly subjective decision, the idea is to perform the minimum of checks to give enough confidence to continue to the more thorough OQ.
- Proving that support manuals and Standard Operating Procedures (SOPs) exist (without actually executing them)

From the above criteria, you can see how many IQs are checklists to indicate that the environment has been set-up before performing application dependent checks.

Operational Qualification

The intent of the OQ is to push the system processes and parameters through as many possible operating ranges in the time available. Note that 'possible' is a wider range than 'normal.'

As the finish line (i.e., live production) is usually very close by the time OQ is being performed, it is a fact

of life that the system will be much more visible to the greater user community, (especially senior management), and the commercial pressure will be on to minimize this OQ time.

This emphasizes why as much stress testing that can be performed during system testing (even if realistic simulations are used) is so valuable later in the project.

For process control systems, some or all of the IQ and OQ can be combined and called the Site Acceptance Test (SAT). As these systems involve physical hardware, cabling, Programmable Logic Controllers (PLCs), Supervisory Control and Data Acquisition (SCADA) or Distributed Control System (DCS), and potentially Manufacturing Execution System (MES) software, there may be a combination of several OQs or SATs to perform. For such complex systems, it is important to be able to clearly show how all these tests and qualifications mesh together to allow a qualified person to declare the system fit for delivery to the user department.

Performance Qualification

The PQ (sometimes referred to as User Acceptance Testing) is normally the qualification where both the process and the automation are tested as one. I like to refer to this stage as 'a typical day in the life of the system.' Although many systems do not have 'typical days,' the intent is to show the system will work as the user specified.

For batch-related systems, a qualification approach would be to run several batches through the full operation of the system to prove a correct and consistent result. Although three is a number bandied around, the aim should be three consecutive batches that give correct and consistent results. If this is not the case, more batches should be performed under formal qualification conditions until the problems are corrected and this is the case.

Some of the less obvious tests that should be performed at this time include:

- SOPs and manuals are correctly written for their purpose
- Spares exist where applicable
- Correct support procedures and Service Level Agreements (SLAs) exist to allow the system to go live

If confidence is high after OQ, PQ is sometimes performed on 'live' batches. These batches are obviously more keenly scrutinized, and their output/results kept to form the PQ results.

Conclusion

In conclusion, the testing exercise on any system is no different to any other area of computer validation. That is, there is no simple 'cook book' on how much to do, or what approach to take. The points made in this article are intended to show that there is no substitute for well constructed code that simply and intuitively performs the requirements of the user.

Testing does nothing to change this. It only shows or disproves this aim.

For all future systems, more effort should be made at this debugging and unit testing level, that will in turn, simplify testing by providing a more reliable base to work from. Disciplines that may assist in this area include:

- Questioning at all phases of the project to force clarification of requirements as early as possible
- Hypothesizing adverse actions and events and discussing solutions
- Managing tighter control over developers by making them more aware of the implications of their work

However, as systems will always be individually configured for specific applications, the testing phase will always remain a vital component in the total validation package. A logical approach needs to be taken in developing the overall testing strategy for each system that best utilizes the time, environment, and available resources. The key points to remember are:

- Construct the overall testing case by building on each testing phase in both the development and live environments
- Wherever possible, avoid duplicating tests on assertions already proved
- Test for both explicit and implicit requirements
- Keep realigning focus to remain clear on the objectives
- Ensure results and attachments are clearly anno-

tated, and form appropriate evidence for documented assertions

Only personnel who intimately know the requirements, and the workings of the computer system, can truly say how thoroughly a system was tested.

Involving these individuals early, together with an overall test strategy that uses common sense and creativity, will maximize the effectiveness of testing, by locating problems earlier in the lifecycle.

Good luck in these testing times! ❑

About the Author

Ian Lucas is the Director of SeerPharma Pty Ltd. in Melbourne, Australia. He has over 20 years experience as a software developer/manager, and over 13 years experience implementing computerized manufacturing solutions for the pharmaceutical industry. Lucas assists pharmaceutical companies in Australia and Asia in all aspects of computer system implementation and computer validation requirements. He can be reached by phone at 61-3-98971990, by fax at 61-3-98971984, or by e-mail at ian.lucas@seerpharma.com.au.

Reference

1. ISPE. GAMP. 3.0. "Appendix 10." (March) 1998.

Additional Guidance

- Many general software testing references can be found at www.testingstuff.com
- ISPE. GAMP 4.0. December 2001.
- ANSI/IEEE 829. "Software Testing Documentation." 1998.
- ANSI/IEEE 1008. "Standard for Software Unit Testing." 1987.
- ISO/IEC 12119. "Information Technology – Software Packages – Quality Requirements and Testing. 1994.
- BSI. BS-7925-1. "Software Testing – Vocabulary." 1998.
- BSI. BS-7925-2. "Standard for Software Component Testing." 1998.

Article Acronym Listing

DCS:	Distributed Control System
FAT:	Factory Acceptance Test
FS:	Functional Specification
IQ:	Installation Qualification
MES:	Manufacturing Execution System
OQ:	Operational Qualification
PC:	Personal Computer
PLC:	Programmable Logic Controller
PQ:	Performance Qualification
SAT:	Site Acceptance Test
SCADA:	Supervisory Control and Data Acquisition
SLA:	Service Level Agreement
SOP:	Standard Operating Procedure
UAT:	User Acceptance Testing
URS:	User Requirement Specification