

# **The 16 Commandments of Systems/Acceptance Power Testing: A First Hand Account**

BY WARREN S. REID COPYRIGHT 1998 ALL RIGHTS RESERVED.

"Quickly! Which is worse: Ignorance or indifference?"

Answer: "I don't know, and, I don't care!"

## **Introduction**

All too often, this is the answer expressed by the players (MIS experts, integrators, consultants, vendors, customers, attorneys, managers, purchasing agents, contract administrators, users, etc.) regarding why they didn't adequately prepare for, specify, and execute adequate systems and acceptance testing. Now, many of these individuals are answering the question in court or at the site of failed systems selections and installations.

I once asked a software engineer why the hardware engineers were able to deliver their products on time while software was almost always late, he responded:

"Heck, the hardware boys have been around since they designed and built the pyramid! We're brand new!"

## **The Problem**

I think we can all agree that "it is preferable to test and reject a system before legal title passes, than to accept the system and try to rescind the contract later because the systems 'fails'." That is why it continues to amaze me that many so-called experts are unable to create and/or execute successful acceptance criteria, acceptance tests and acceptance clauses.

As a consultant, expert witness, and special master, I have reviewed many testing and acceptance clauses that were:

- not documented
- conceptually ineffective in a business sense
- legally unenforceable
- incomplete

- inconclusive
- not specific, or
- haphazardly drafted so that the results of the tests could not be quantitatively or qualitatively evaluated.

With hindsight, such "ineffective and ignored clauses" were precursors to misunderstanding and invitations to future lawsuits.

Until the art of acceptance testing matures, how can we maintain and maximize an effective software development, acquisitions, and implementation program that achieves its goals of increasing company value, productivity, and competitive advantage? We will attempt to answer that question and the following in this article:

- What is a sane, efficient and effective approach to testing and acceptance
- How do you actually conduct and establish systems and acceptance tests?
- How do you memorialize your expectations and understandings in a meaningful acceptance test contract clause(s)?
- How do you go about selecting "a right system" in the first place-so that you only spend time testing systems that have the best chance of meeting your needs?

This article will discuss the past and present problems and myths associated with systems and acceptance testing. After, I will present an integrated set of realities, technologies, attitudes, and contract understandings/clauses. In a future article, I will talk about how to develop and negotiate effective and legally remedied Acceptance Testing clauses in your procurement contracts-clauses that are fair, provide incentives and put all parties on the alert (including your own company) as to what the responsibilities and remedies of each party are in developing tests, capturing/analyzing results, evaluating discrepancies, fixing problems and assuring successful implementation.

## **Why Today's Testing Practices Still Are Not Successful?**

In general software testing has not improved much during the last twenty years. Recent surveys conducted by the prestigious Software Engineering Institute of Carnegie Mellon University (SEI) note the following lingering and insidious quality and acceptance related problems and conclusions:

- companies have great difficulty in establishing system requirements (and requirements changes do not receive consistent and disciplined control)
- management does not understand software problems (and management tracking and control of the software engineering process was found to be lacking or non-existent)
- lack of an identifiable, supported, and documented software process
- lack of consistent standards for software

- lack of good analytical methods and hard empirical data for software (to enable project management to estimate the real time and resources required, to avoid overruns and last minute crises, to insure commitment at all appropriate levels).
- documentation was found to be the wrong type, poorly written, incorrect, or inadequate, and
- inadequate test coverage is provided; testing is promptly suspended when time or money runs out.

Specifically regarding testing, the SEI report concluded based upon its own observations:

- lack of adequate unit testing (testing the basic logic within each module) deferred problems to the integration, systems test, acceptance tests, and operational phases
- lack of regression testing (the ability to go back and retest all of your criteria and test cases after a correction or change is made to the programs) had serious negative impact on the integrity of the system, and
- lack of stress testing (testing with realistic but peak volumes of transactions and data) increased the likelihood that the user will encounter program failures when the system is heavily loaded.

### **Those Magnificent Men In Their Flying Machines**

Size does not give a company immunity from the impact of inadequate and ineffective testing. Everyone is affected. I have dealt with these problems with organizations whose sales were as low as one half million dollars per year to those as large as fifty billion dollars per year.

Each year in total, billions of dollars are lost in time, efficiency, profits and opportunity, incorrect decision-making, goodwill and legal judgments because the companies that purchase, develop, distribute, test and use systems do not know how to adequately and properly test systems. Given the importance of information to the lifeblood of today's corporation, such ineptness, lack of commitment to protecting the information resources, and unsystematic testing and acceptance policies are unconscionable.

Below are some examples of systems that failed and resulted in large monetary damages because they were not tested properly. However, bear in mind that much smaller systems and acceptance failures have even more severely crippled, and in some cases bankrupted, smaller companies.

The AEGIS system was built to help manage the decision on whether approaching aircraft is friendly or not. It was tested in the open seas. However, it was used in a narrow body of water and shot down a non-military, non-hostile Iranian Airbus.

- Bank of America's new system, in 1988, lost control over \$8 billion in trust accounts. As a result angry grantors transferred their accounts to competing institutions. The bank was embarrassed, lost huge profits and fees, and was forced to institute massive layoffs ranging from the Chief Information Officer and his management team to the entire Trust Department.
- In 1985, the IRS in Atlanta created a new system that failed to process tax returns in a timely manner. Because of delayed processing, the IRS paid out \$80 million in interest to taxpayers. (The new IRS systems "facelift" is estimated to cost \$8 billion.)
- Several years ago, the new Federal Reserve System had a bug that mis-posted \$20 billion on the first day of operation. This enabled the wrong banks to get interest for these amounts. (Only \$18 billion was reported by the member banks as inaccurate!)
- The Department of Motor Vehicles in New Jersey put in a new system in 1986 that caused them to lose control over the status of several million drivers, their registration, renewals, tickets, etc., for days.

With all of these bad experiences noted, the good news is that: "Acceptance testing is not magic!" It can be done and done well. By using the methods presented here, many procurement professionals will dramatically enhance the success of systems introduced into their companies.

## **A Working Philosophy And Some Essential Definitions**

### ***What is a "good test?"***

When a new system is developed, it is a given that there will be problems, "pains" and bugs. Using a medical analogy, a patient who "doesn't feel well," i.e., knows s/he has pains and problems, goes to the doctor for tests. If the doctor tells the patient that all the tests are negative (no problems found), the patient really has reason to become fearful. That is because the testing process has failed. The patient knows that there is a problem but the state of the art of the tests just could not uncover it. In reality, these tests could be looked upon as money wasted and failed medical science.

With information systems, riot unlike human systems, it seems that the only good test is one that discovers or uncovers bugs. A test that does not do so is a waste of time and energy (except for the last time it is performed which proves that already identified bugs have been cured).

So then, we can define a good test as one that reveals bugs. The philosophy of the testers and the developers of the tests should be to see how they can make the system fail-how many times, how badly, how many ways, etc. The more bugs found, the merrier.

### ***What is Acceptance Testing?***

There is more to acceptance testing than just testing hardware and software. Acceptance Testing criteria include:

1. Hands-on testing and acceptance must be performed by the actual user's and their assistants including consultants, auditors, internal systems staff, etc.
2. The actual software and hardware configuration (or equivalent configuration) must be tested, and
3. The actual policies, procedures, manuals, operations, organization structure, and controls that will be in place and required to meet the stated objectives of the systems must be used in the test.

### ***What is Power Testing?***

'Power Testing' is a term I coined that has three components:

1. a "find those bugs" philosophy
2. a "risk-driven" philosophy-an approach that pre-determines where to test for and find the most critical bugs; i.e., where the greatest probability of error and greatest impact caused by a bug on your organization intersect, and
3. a series of specific commandments - rules, assumptions, and realities (some learned the hard way), based upon years of experience, research, and discussion with others.

Here are my 16 Commandments of Power Testing.

### **The 16 Commandments Of Systems/Acceptance Power Testing:**

#### **1. The Realities**

- 1.1. Thou cannot test everything.
- 1.2. Thou shalt let risk point you to the most important errors.
- 1.3. Thou shalt not confuse 1000 tests with testing 1000 functions.
- 1.4. Thou shalt hunt where the elephants drink-error guessing.

#### **2. The Attitudes**

- 2.1. Thou shalt not build on bad specifications and unclear interpretations.
- 2.2. Good tests need good code and good design.
- 2.3. Let thy enemy design and perform your tests.
- 2.4. Honor thy reference checks.

#### **3. The Technologies**

- 3.1. Thou shalt control changes.
- 3.2. Thou shalt commit to Regression Testing.

- 3.3. Test thy documentation.
- 3.4. Test for performance or perish.

#### **4. The Cornerstones**

- 4.1. Accept running programs-not working programs (unit testing).
- 4.2. Bless applications software more carefully than custom software.
- 4.3. Thou shalt cultivate and win management and user commitment and understanding.
- 4.4. Thou shalt measure progress-and know when to stop.

### **THE REALITIES: IF YOU CANNOT TEST FOR EVERYTHING, THEN WHAT DO YOU DO?**

The preceding material discussed why system *SI* acceptance practices today are still inadequate and developed some important working definitions and working philosophies in general. It also introduced the "16 Commandments for Acceptance Power Testing."

The following sections will show how these methods and the remaining 12 (altogether known as "Reid's Remedies") produce dramatic improvements in system quality, usefulness, user acceptance, and testing effectiveness and efficiency, while materially lowering testing costs.

#### **1. Thou cannot test everything**

Let us start with a given-that all newly developed and implemented non-trivial systems will have bugs. There are virtually no exceptions to this rule. The reasons include:

- complexity and sheer number of logic paths in most code,
- limited resources typically allotted to testing,
- lack of awareness of how to properly test programs, and
- the amount of human interaction and near perfect communication required by all of the stakeholders to "cure" a new system (i.e., management, analysts, designers, vendors of the CASE and other software engineering tools, programmers, users, other systems and developers to which the new system is being interfaced, consultants, testers, documentation specialists, contingency planners, computer operators, trainers, supervisors, systems maintenance staff, etc.)

Note that I use the word *cure* instead of *debug*. Debug, with respects to Grace Hopper (who invented the term), typically means correcting the logic of the programs; cure, as I use it, means testing and correcting all those factors that make the system actually perform differently than planned or expected - a much broader, but more practical objective.

While one may argue that the following story is not about a bug, it shows how other difficulties, such as the non-compatibility of various software, can cause debilitating

problems that often have greater impact than bugs, and must be cured.

My uncle in Florida recently bought a copy of Microsoft Windows after the salesman promised that implementing and using Windows was "as easy as pie." As you may have already guessed, Windows destroyed his QEMM 386 Expanded Memory Manager (with no warning or word of this beforehand) and proceeded to make his 4 MB memory run out of capacity when he tried to print his letterhead which uses Wordperfect 5.1 graphics. "It never fails," he told me. I responded, "On the contrary, it almost always fails!" He subsequently unloaded Windows and went back to his old, working, tested configuration. I cautioned him to not be one of the first to load MS-DOS 5.0 either!

Ultimately, he bought Windows and MS-DOS 5.0, using the latter's Extended Memory Manager, to cohabit his system (it turned out that QEMM 386 would not work with Windows until the next QEMM release). By my definition, Windows may be considered debugged by Microsoft engineers, but certainly it cannot be considered "cured." It did not perform as expected because it was unable to work with the other programs on the system; it did not alert the implementer/user of this detail clearly beforehand - one had to experiment with it and figure it out. If the whole integrated system does not work, it does not work!

My uncle does not really care what caused it. It could/should have been detected in testing and either corrected outright or made known to salespeople and prospective users in the documentation, the warnings and messages provided.

Let us look a little further at why you cannot test everything. First, you must insure that the system does what it is supposed to do. Next, you must test the system such that it does not do what it is not supposed to do. This requires testing all of the specifications affirmatively, and testing that any and all unplanned inputs and errors do not get processed in a way that appears as if there were no errors. This creates a very large number of testing possibilities.

Second, a system that is logically correct, but is structured in a non-standard, clever, non-maintainable way is really a time bomb waiting to explode. Explosions of this type typically happen when it is least convenient. Usually just after you have integrated the system into your operations and realize that the planned changes that you need to make to really make the system most valuable to your company are riddled with irrational logic from the base system that make it non-workable and non-acceptable.

So by definition, testing a system must include testing:

- the overall design, coding standards and the adherence thereto
- the quality, helpfulness and accuracy of the documentation, etc.

A system that cannot be easily, or at least reasonably, maintained should not pass the Acceptance Tests.

A simple program with just 60 decisions, either IF statements or loops (which one can consider as implied IF statements) with binary outcomes will have  $2^{60}$  possible logic paths. This is a non-testable and astronomical number of combinations. It is more than the grains of sand on all of the world's beaches. Still, this is far less than the number of decision trees found in any non-trivial business application.

Whether you wanted to test all of the logic paths using the Glass Box Approach (looking at all of the distinct combinations of the logic paths) or the Black Box Approach (where you care nothing for the logic and just create a correct number of tests to match all of the mathematical possibilities, i.e.,  $2^n$  tests) you can see that the volume of test conditions is hopelessly insurmountable without some sort of methodology to reduce the number of test cases to a manageable size.

Theoretically, "complete testing" would have you testing every possible input and combinations thereof. You would have to test all valid inputs, all invalid inputs, all edited inputs (test inputs after the program allows you to edit them), all variations on input timing, and so on.

It becomes even worse when you consider that for every test case you must:

- prepare the expected input, the required master file data, the expected output
- run the tests
- compare the results, and
- fix the program or files and rerun the tests as necessary.

There is a lot of work associated with each and every test case.

So, why test? Because the purpose of testing a system is to find as many of the most troublesome problems and get them fixed (cured) before the system is released. By beating up on the system it actually becomes stronger and your destructive approach will be most constructive in the long run.

Given that we cannot test for everything *and* that your new system most assuredly will have bugs and other fundamental problems, how can you reduce the number of places to test, kinds of tests, and test cases themselves to a manageable total?

*Answer: By understanding the next three realities and following their directives.*

## 2. Thou shalt let risk point you to the most important errors

Earlier, we discussed how you should recognize that a "great" testing strategy is one that is driven by risk. This concept is so important that it is worth summarizing and repeating again.

Risk is defined in a 2 x 2 matrix: Probability (low or high) by Impact (low or high), where *Probability* is "the likelihood that a particular piece of code will fail" and *Impact* is "the negative consequences of the failure."

Probability factors include: complexity of the logic; telecommunications sophistication; newness of the technology to the industry or to the company experience level of the stakeholders, etc.

Impact factors are company specific, and therefore, subjective. For instance, what will happen if a certain part of the program fails? Will a missile land ten miles off target? A wrong vendor be paid? A hacker gain access? A management report be incorrect, late, or sloppy? Will users lose time? Will the database be corrupted or the system locked up? Will a function be difficult to use and thus avoided?

Letting *Risk* drive your test priorities (which dramatically reduces the number of areas you must test and the time and cost to test) means you are focusing on the "high probability-high impact" quadrant of the risk matrix. That is where a systems problem can cause the most damage and a cure will provide the most benefit.

Designers and users must identify the high-high areas early so that these areas can be programmed and tested first. Otherwise, these areas will not be adequately tested. These tests can then be followed by the high-low or the low-high quadrants, and then lastly, if time permits, the low-low quadrant.

### 3. Thou shalt not confuse 1000 tests with testing 1000 functions

Often a test team will be tempted into generating random test data. Frequently, it may be due to a lack of training or an attempt to complete the project early. Other reasons may include being out of time, money, energy or budget, or because one is lazy and does not have the energy to go through the mental exercise of developing distinct and individual tests that rigorously test the entire system adequately.

The trouble with this ineffective approach is that it does not insure the exercising of different paths in the program. Indeed most such tests exercise the same paths in the program and are redundant and provide no useful additional information, i.e., they are a waste of time.

To help solve this problem and multiply the effectiveness and efficiency of your testing, try using the concept of *Equivalence Classes*. Items in the same Equivalence Class exercise the same code. Even if you do not know or understand the particular code being tested, the concept is easy to follow.

Consider that you must test a program that adds a series of two digit numbers and prints the results. It would be a waste of time and effort to test 21, 44, 68, 87 and other valid two digit numbers because they are all in the same Equivalence Class and with a very high probability exercise the same program logic. If one works, there is no reason to suspect that the others will not. It is best to immediately test other Equivalence Classes such as a single digit number, 00, the boundary numbers of 10 and 100, negative numbers, and non- numerics.

Note that there are two types of Equivalence Classes: valid and invalid. The distinction is important because each type needs to be tested in a different manner. You can test different valid Equivalence Classes for each field on the same test run since most test runs require or permit several input fields. If one does not work, it is easy to isolate that data field and the resulting code that caused the error.

However, when you are testing invalid Equivalence Classes you expect and desire to get appropriate error conditions and messages. Often, entering multiple invalid conditions will mask the true identity of the data, code, and source of the problem that generated the error because the first error will veil the others.

Equivalence Classes are one of the most important and basic tools that are often overlooked in building valuable test cases. My own experience and empirical evidence shows that even though you might think these are only educated guesses, they really do express the basic underpinnings of how programs are built and how program logic works. Accordingly, they are remarkably effective as a tool to practically and exponentially reduce the number of test runs that must be conducted and they create time for you to work on testing the complicated areas.

#### 4. Thou shall hunt where the elephants drink - error guessing

Given that we cannot test for everything, it is important to focus on the highest risk test cases. The use of *error guessing* accomplishes this.

Based upon years of programming and testing experience, one begins to know where problems lurk and are more or most likely to occur:

- at boundaries
- in loops (infinite loops, and "off-by-one," and "loop-not-entered" errors) and counters (especially when the end-points are not specific in the specifications)

- in arrays (defined too small; subscripts contain non-integer values; out of boundary violations)
- with passed data, parameters and subroutines (i.e., variables passed are not identically defined; different number of parameters passed than received; parameters not passed and received in same order; sub-routine changes parameters, but main routine assumes that parameter only used for input to subroutine)
- in "IF" logic and in "CASE" logic (i.e., code incorrectly assumes that all possible values have been tested when setting up alternative branches for execution; code does not have a "catch all" case to capture other invalid data possibilities; incorrectly nested "IFs" incorrectly used Boolean operators (AND, OR, NOT, etc.)
- exception processing
- with illiterate, poorly specified calculations
- at the value "0"
- variable problems ("initialization"), variables defined too small causing truncation; not resetting to zero at "later" state; variable type incorrectly declared)
- year end and period calculations that must zero out fields
- the paths that end users are particularly likely to follow
- calculation problems (dividing by zero; overlooking for the possibility of negative numbers; processing complex calculations in the incorrect order)
- files not opened and closed as needed; processing goes past end-of-file, and
- the failure to execute planned tests because the testers are bored and/or
- disorganized).

While the last two areas are different in nature from the others, I have included them because they too are places where errors are likely to occur.

What the above says, for instance, in terms of the simple 2-digit adding program discussed earlier, is that the boundary cases of 10 and 100 are more likely to cause problems than the numbers between them.

The value "0" is another high yield case and is problematic especially if it ends up as a divisor, as well as causing problems because it is overlooked as possible data in "IF" and "CASE" and initialization logic.

Exception logic also provides relatively high yield cases that are incidentally easy to prepare because they are frequently overlooked after valid cases are tested successfully. Exception logic includes:

- adding records that already exist

- deleting records that do not yet exist, and
- changing records that do not yet exist, and the rest.

Also remember that "the cockroach likes to drink at the elephant's watering hole." The Cockroach Theory of Debugging says that if you find a bug in a program, there is a strong likelihood that you will find others in the same area. This has been proven time and again. Perhaps it is because the programmer did not understand the specifications, or the specifications were wrong, or the programmer was new to that kind of logic or structure, or he/she was tired, or a number of others reasons that might explain why errors cluster (and bugs pack). Indeed the probability of finding a bug in a particular piece of code is directly proportional to the number of hugs already found. So, if you find some problems, fight the urge to move on, and continue testing (spraying) until the code becomes very clean.

## **THE ATTITUDES**

We have discussed why systems/acceptance practices today are still deficient, introduced the "16 Commandments for Acceptance Power Testing," and discussed the First Four Commandments-The Realities - in detail. In this section, we will discuss and further develop the Second Four Commandments - The Attitudes. The methodologies and benefits that these Commandments produce will also be discussed.

### **5. Thou shalt not build on bad specifications and unclear interpretations**

Boris Beizer of Data Systems Analysts, Inc., aptly defines a *bug* as "program behavior that does not meet design specifications." He further continues: "This definition presumes that design specifications exist and that they are written before the program. With no specifications, any kind of program behavior could be correct."

Indeed you may surmise that a report showing sixteen divisional profit and loss centers is incorrect because you thought (or know) that there were only eight divisions. Without specifications, who knows for sure?

Many programmers cannot wait to jump in and wildly begin coding before clear and total specifications have been developed and communicated. Indeed, it is one way that programmers believe that they can impress others and show how "good and fast" they are. In such circumstances, programmers often turn out reasonably good code, but only for a part of the required solution. Developing a total solution first would have had a dramatic effect on the overall design of the system and program. Accomplishing a total programming solution after the fact often requires that all of the "premature" coding done so far to be scrapped and reprogrammed causing embarrassment, additional costs, missed deadlines and heated tempers.

These "wild and woolly" programmers often develop specifications after the fact that, not surprisingly, match the code that has been developed. However, it does not meet the needs, expectations, or original intentions of the requesting organization. Indeed, if the programmers knew what was required up front, they would not have done it wrong in the first place. I have often given the following advice to computer professionals: "When you get the urge to begin programming before the specifications are completed - lie down until the feeling passes."

For those of you that think this does not apply to you - think again. This applies to everyone and is true for those who believe that the problems and processes they are programming are relatively straightforward.

Let us discuss a rather famous test case that has been given to college students majoring in computer science for the last twenty years. The results are profoundly revealing. I have been offering this problem to systems professionals at many of my lectures on this subject.

The students are each asked to develop the test conditions to thoroughly test and accept a program that performs the following:

- accepts three data entries from the keyboard
- interprets the data as the three sides of a triangle, and
- prints out a statement concluding whether the triangle is equilateral, isosceles, or scalene.

This apparent trivial problem has perplexed computer science students for years. Why? Well, before answering that question, try to solve the problem on a separate piece of paper. Remember, try to identify all of the necessary test situations without repeating yourself by giving cases from the same Equivalence Class.

Now that you have developed your own approach and test cases solution to this problem, let us find out why this problem is so difficult for students and professionals. The key is that almost all of the programmers do not test for a valid triangle. Sure they test to see that the numbers are truly numeric (although most students do not check for zeroes and negative numbers). But only a very dismal 16% know to test whether the three sides can physically be connected into a triangle (tested by insuring that every combination of two sides of the triangle - data elements - is greater than the third side - the third data element).

An adequate specification would have alerted the programmer to test this consideration. So, if your solution allows (or even worse, if you gave these as valid test cases) 1,2, 3 or 0, 0, 0 or -2, 2, 2 or 1, 1, 2 to be processed as valid triangles you are dead wrong-the casualty of poor specifications. These and many other examples of three sided non-triangles should be rejected with a message indicating that "These inputs cannot create a

valid triangle."

This essential part of the program is missed even after I ask the failing programmers to develop a specification "after the fact." These programmers tend to work back from what they have already so proudly programmed and "tested" and make the same mistake in their specification. Of course, one would expect the backwards specification to confirm the already deficient and prematurely developed program logic. After all, if the programmer had all the information beforehand and/or really thought the problem through, she or he would not have made the programming error in the first place. So then, their logic is consistently incomplete.

This is why Roy Carlson of the University of Wisconsin concludes, "The sooner you start to code, the longer the program will take."

## 6. Good tests need good code and good design

Getting a system to acceptable system performance involves detecting, locating, analyzing, correcting and insuring that programming and functional deficiencies are solved. While the process can be painstaking at best, it can become unfeasible if the system itself was poorly designed or coded.

Let me give you an analogy. Suppose you are in the middle of a war and are trying to take a town from the enemy. You send in a squadron and find that they have become trapped because of a flaw in your plan (logic) or because the plan was not executed correctly by the squad. Now, if you have no maps, or worse, inaccurate maps (i.e., no or bad specifications, program design charts, and programming standards), you have little way of knowing when, how and where to deploy more troops (the fixes).

Indeed, trying to rush in head-on may exacerbate the problem if you send too many men in without adequate intelligence and understanding of the environment. The maps, the position, strength, and power of the enemy and your own forces are all critical for your understanding what has caused the problem, how best to fix it, what other problems the fix will bring, and what you should be careful of when implementing the new solution.

Just like in war, the management team is shooting in the dark (and maybe shooting themselves in their collective "feet") without the key maps, blueprints, plans and understanding of how the systems have been designed, organized, fortified (unit tested), standardized and constructed. As a tester who must iteratively debug the system to reach acceptance, you cannot be sure whether your fix will do the job, create another equivalent problem, or create or permit a catastrophe to occur (a behind schedule, over budget, or lost user confidence and business interruption situation).

Without a good underlying infrastructure and adequate technical underpinnings, an identified problem resulting from testing may be difficult to locate and interpret. The

impact of the change may not be discernible or predictable, and regression testing (discussed in an upcoming section) will have to go on *ad infinitum* to be sure the fixes were implemented correctly.

Even if the specifications are adequate, programs that are unstructured, spaghetti-like, poorly documented, do not use CASE tools, do not use data dictionaries or common objects or routines (where appropriate) can become a nightmare to fix. I am reminded of the old cartoon where a data processing manager sees a little cockroach scurrying out from under his computer. He responds by using the tested debugger "RAID" and sprays it on the bug as it returns under the computer. He is rewarded for his efforts by an army of angry bugs racing towards him looking for new cover.

Often it is better to simply throw out the entire offending code and re-develop the system with the appropriate design and programming standards, than to fix the discovered errors. This is especially true given that it will most likely cost more to maintain the system over its useful life than to develop it from scratch, even if the programs were constructed and functioning well in the first place. Accordingly, even though you get through acceptance testing, you will overpay for poorly designed and programmed logic forever.

Lastly, if the design and coding are poor, it will be more difficult to develop meaningful and efficient test cases. In the "black box" testing approach, one develops test cases based on what the system is supposed to accomplish. In the "glass box" testing approach, test cases are developed based upon the actual design and coding structure of the systems to see that important branches and sophisticated nested IF statements, etc., are being processed and properly functioning.

Good test data development should include some of both approaches. Such efforts can be hampered or paralyzed without proper and adhered-to documentation and standards. Accordingly, without the benefit of good design your test cases will be incomplete. They will not be driven by risk, nor will they be designed to help you isolate the causes and fixes for the problem.

## 7. Let thy enemy design and perform your tests

Just as too many cooks in the kitchen can spoil the broth, one cook testing his own recipe can be one too many.

I have found it very difficult for professionals who have designed systems and developed code to thoroughly, effectively, and indifferently test and validate their own work, for several reasons:

- they have a vested interest in their work being correct

- they have a tendency to *not* test simpler but critical conditions and logic, as they know (or believe they know) intimately how the system works and assume that it need not be tested in these areas
- they cannot see the trees for the forest, or the forest for the trees. Designers and programmers cannot effectively test their own systems because they are too close to the problem themselves
- if programmers misunderstand the objectives, functionality, and specifications of the system, they will design, program, and test the system incorrectly and never spot the error anyway
- if they are late or over-budget, they have a tendency to overlook certain tests and rationalize them away
- they may be skilled in design and programming, but that does not make them apt "power-testers" (following our Commandments)
- good power-testing requires that test conditions, expected results, before- and after-databases be maintained, and discrepancies be logged and accounted for. People are less likely to judge themselves harshly or admit errors if they know that an audit trail will be left that can/may be used in evaluating them in the future. In this case, not only is the system inadequately tested, but the company loses the ability to know how to better train, supervise, and develop tools and programming aids in the future if the testing documentation is lost, incomplete, or inaccurate, and
- developers may tend to fix and cover up last minute errors in non-standard ways which can hamper maintenance coders in the future if development-testers are not reviewed or required to communicate with independent testers.

For these reasons, you need an independent, trained person with a perspective and desire to test for the most numerous and most risky/offensive errors - which are the primary objectives of testing and acceptance in the first place. Who is better able and vested in finding and correcting the errors before the system is put into production than the developer? Who will be as fussy and careless and idiosyncratic as users and does not see things your way all the time? That person is the person I sarcastically refer to as your enemy. In fact, he or she is really doing you, the users, and the company the services of a best friend.

This is a great philosophy because users (although not enemies by definition) in production environments, working the systems over time, will put the systems through the same grueling tests as professionally trained enemies and unbiased experts, albeit inadvertently.

Users will sometime during the early life of the new system:

- put in garbage, incomplete data, invalid data, too much or too little data; overload the systems and kill response time

- put in out-of-sequence transactions
- backup, recover, and restart incorrectly and not able to recover from errors
- mis-schedule systems usage
- under- or over-staff
- are not ready to use the system
- corrupt master and data files
- become disenchanted when the system performs a function adequately but in an unexpected manner
- sign off from the system incorrectly
- force inefficient or incorrect systems interfaces
- not be able to use or do not use the user manuals
- try to force the systems to work as they believe things ought to work, and
- on and on.

The "enemy" philosophy best simulates actual user operations and turns out to be the most successful kind of Acceptance Tests for uncovering all the types of

- procedural
- operational
- structural
- logic
- coding
- standards
- telecommunications and
- human communications

problems that really prevent our systems from being successful (as useful, efficient, and effective as they should and must be to sustain competitive advantage and maximize value).

## 8. Honor thy reference checks

One technique that must not be overlooked in your selection, implementation and testing processes is really talking to and learning from other users. After all, these peers can be your guinea pigs, your beta testers - those brave souls who have learned the surprises, and learned how to fix, work around,

or live with them. They know first hand about the strengths, weaknesses. opportunities, and threats of the system. They know invaluable information and answers to questions

about how the system really works and what assumptions about the system are wrong.

They know secrets that you might otherwise have to learn the hard way about:

- the vendor(s), users, and management commitment required
- MIS Group resources needed
- telecommunications interfaces dictated
- security levels employed
- quality levels of vendor's modifications usefulness of documentation
- complexity of file conversion efforts
- adequacy of training
- quality of and level of installation support dictated
- sensitivity of monthly/periodic close schedules
- impact of peak transaction volumes on performance
- impact of hardware and operating system upgrades
- quality of the vendors' "hotline" services
- vendor performance on remedying problems, and
- vendors' willingness to negotiate, etc.

All of these items are key components to testing and accepting systems, and otherwise predicting an increasing the chances of the long term success of the system in your own environment - and usually provide added critical insights to supplement your own internal acceptance tests.

Learning from the references' mistakes and experiences seems to me to be so valuable, and yet so few procurers do an adequate job of getting the information before signing the contract and planning the acceptance testing. The best approach I have found to getting the reference to open up and tell you what you need to know begins with setting up rapport. After all, no one wants to naturally say something negative about a vendor (especially if it can get back to the vendor and upset relationships) and no one wants to admit that they made a mistake, or did not do a thorough job, or were unprofessional in going about buying a specific system and/or set of services.

Additionally, if the vendor gives you a reference, it usually means that the reference will be somewhat positive, although it never fails to amaze me that many direct vendor references will give honest and unflattering information because of a combination of the techniques I will outline here and the fact that vendors are not as in touch with what users really feel and believe as they think they are.

How, then, do you establish rapport? First, ask open ended questions to get the other person to talk. Second, ask the right questions. Open-ended questions are those that ask "why," "how," and "what" and not ones that can be answered by a "yes" or "no."

Avoid:

- Is the system a good system?
- Does it meet your needs?
- Was the vendor responsive?
- Did the vendor bargain?

Begin with first establishing your reason for calling with something like "I am considering purchasing the XYZ system. I will either know a lot about the system 60 days before I buy it - or 60 days after I buy it. I would prefer the former. XYZ gave me your name so that I can learn a bit about your experiences, and better plan the appropriate resources required to get the system up and running in my environment. If this is a good time, would you mind if I asked you a few questions?"

Your next series of questions should be aimed at finding out about the nature of the company and person being interviewed, to get the other person to talk, and to see just how similar your situations might be. For instance:

- What is the size of the system you installed? What modules?
- How long did it take? Was this shorter or longer than you expected?
- What was your role?
- What objectives or problems was this system supposed to meet?
- What experience levels did your company have and bring to bear on the implementation?
- What was your company's previous experience and track record with systems like this one?
- What system did you replace?

Because people like to talk about themselves, I usually have no problem getting answers to most of the above questions. They establish rapport and a basis for determining how relevant the reference's experience will be to your own company situation.

Now you want to get to the real "meat" of the reference check with these killer questions:

- Regarding the new system, what one thing would make your job easier?
- What would you do differently if you had to do it all over again?
- What improvements do you wish the vendor would make to his system, and why?

- How would you change your contract if you could do it over again?
- Even with all of your planning, what surprised you the most?
- If you could snap your fingers and change two aspects of your relationship with the vendor, what would those changes be?
- What key things did you learn from this installation?
- How did you decide on this system?
- Who else did you look at?
- What references did you check?
- Any advice or last words of wisdom for me?

Only a fool, or a completely happy user (very rare indeed), could answer these questions with "I am ecstatic about the system. We made no mistakes. We would change nothing."

Learning who else the reference reviewed and why they selected XYZ over the others can provide useful insight that can help you finalize your selection. Checking with earlier installation references, who by now the vendor may be neglecting, will give you insight that the later references will not have. If possible, find other users who were not referred by the vendor, through user groups, vendors' partial client lists, industry associations, etc.

Then call a reference for the vendor you did not select (i.e., the next best alternative in your mind so far) and ask that reference whether they reviewed XYZ and why they did not select XYZ. Correlate the results of your other reference checks and put these experiences and results into your selection analysis and acceptance test planning process. Knowing where and what are the problems likely to occur, and their solutions based upon others' experiences, can be a powerful weapon in appropriately testing and accepting and training your people in a winning systems solution.

## **THE TECHNOLOGIES**

According to IBM and AT&T (who periodically publish the results of tests that they consistently and independently perform) "it takes three to five times the cost to fix errors once the system has been programmed as compared to catching the error in the design phases - and 10-30 times the cost to repair a bug, problem, or logic misunderstanding once the product is released."

In my experience, if the program erodes users confidence in the program's ability and dependability, it can cost up to one hundred times the original fixed cost as users prepare and keep other records, systems, and analyses surreptitiously and delay responsible decision making in order to ensure that the new system is in fact performing as expected.

The ironic title of this section belies the seriousness of the statement: "An interrupting of your program(s)" can mean a debilitating interruption in your business; and looking at acceptance and acceptable performance as "only a test" can mean the difference between profit and loss and bankruptcy to a business, or life and death to a plane full of people relying on the heavily tested local air-traffic control programs.

And while no software is entirely free from bugs, "power tested" software can be expected to have none that will seriously effect its use or its accuracy.

There is much that can be done by management, information systems professionals, vendors, and users alike to help assure that systems will be free from serious bugs, properly documented, performing adequately, and able to be changed and controlled over future releases.

This section presents the Technologies. Change control, regression testing, documentation accuracy, performance evaluation, and a discussion on how these methods, produce dramatic benefits in your systems quality and usefulness by increased user acceptance and improved testing cost effectiveness and efficiency.

## 9. Thou shalt control changes

Nothing is more frustrating and doomed to failure than trying to test a system that is still in flux and constantly changing in scope. Once an error is found, not only is the error fixed, but the designers and programmers typically use this as an opportunity to add in additional features and functions. The test cases which have been so meticulously planned and documented no longer are relevant because of the additional interrelationships between the original and the new functions. And errors cannot be easily traced to either the original specification or the new changes.

Foregoing quality because of changing and creeping scope will have tremendous ongoing costs to the developer and users alike - costs which no company can voluntarily afford to give away.

There is a formula that holds true in systems building as well as in any manufacturing process (and for services too):

**QUALITY + SCOPE = TIME + DOLLARS**

This means that if you are going to change the SCOPE of the system (features and functions of the system) AND you do not increase either the DOLLARS or the TIME to get the job done, then the QUALITY will suffer. Ever creeping and changing scope (i.e., systems features and functions, inputs, reports, interfaces, transaction types, formulas,

controls, documentation, uses of the system, etc.) is natural within any system and within any healthy company. However, to build and properly test and accept a system, your target must be fixed - if only for that release.

You must determine, but not too prematurely, what the new system will contain. Then you must agree to plan all other additions to the system as prioritized future enhancements and releases. Users will always want more once they get to actually use a system. And why not? Once they use it they get smarter as to what they really need. Is that not what prototyping is all about?

Only when the system scope is fixed can a reasonable budget be estimated and the required resources allocated to design, program, test and accept the system with the given level of quality and functionality.

Identifying future releases as anticipated feature and function upgrades allows you to stay on budget, remain on schedule, keep the quality up, while also enhancing your credibility with users. It also permits you to adhere to a Regression Test policy which is the subject of the next commandment.

## 10. Thou shalt commit to regression testing

Whenever I think of regression testing, I recall how skyscrapers are designed and built. Once the base and or lower levels are designed, built, tested and approved, the next progressive levels are built. If the higher levels "do not work" for some reason (e.g., the lead tenant wants a different look, the higher level frames cannot support the weight of the newly decided Italian marble), the builders can rest assured that the lower levels do not have to be rebuilt (for the most part).

Regression Tests give assurance to systems management, developers, testers and users that new system development, enhancements, releases, and acceptance will be based upon a bedrock of proven code. Regression testing usually refers to two kinds of tests that both involve using "old tests."

In the first instance, if a bug is identified, the corrections are made and the program is run against the same test that highlighted the error. This is a regression test. Often additional tests are added to insure that the fix is even more thoroughly tested. This, too, is part of the battery of acceptance tests.

In addition, regression tests are performed to assure that the fixes did not disturb any other parts of the program that were previously tested and deemed successful.

Just like the scientist who keeps notes of his successful and unsuccessful tests, the programmer can also use these regression tests to assure that the system is progressing

forward and no new errors are being introduced as others are being repaired.

You should run regression tests every time the program changes, whether it be for major fixes, enhancements, or new releases of the software or hardware. If a particular piece of code executed correctly before, in all probability it will work correctly this time. However this cannot be assumed, and while it may appear like a waste of time, you must run these regression tests "just in case."

In part, the acceptance tests themselves can rely on users running and reviewing the results of the regression tests - especially if those tests are complete and cover the types of tests and philosophies covered in the other discussed Commandments (i.e., error guessing, driven by risk factors, eliminating equivalence classes, etc.)

The good news is that such tests can be automated. All that is necessary is to teach the computer to run these same tests, capture the input, process the results, compare the new results to a data base of expected "correct" results, and give the results to you. Just have the computer print the discrepancies, so that if there are no errors, you will not have to invest any time.

Even in those situations where developing an automated regression testing system is difficult and expensive to develop (i.e., for on-line systems where special "keyboard capture programs" and "standard input device interfaces" using disks to simulate keyboard entry or screen memory and/or for systems where complex data base programs are required to maintain correct output for comparison to test results) they more than offset the cost of poor quality and user avoided systems.

In some complicated environments you may be forced to use only partially automated testing with some manual regression tests. That is okay, too. Especially if tests will drag on for a few dozen cycles, your initial investment will pay for itself many times over.

Going back for a moment to the previous Commandment, another difficulty with regression tests is developing them for a system that is changing as it is being programmed. This was true when I was engaged to work with the State Controller's Office to co-oversee the testing, acceptance and signing for the completion of the California State Lottery's LOTTO games.

The Governor's charge to us that the "Lottery pay the winners, the whole winners, and nothing but the winners," albeit cute, was a difficult and relevant statement of our responsibilities. This was especially relevant during an election year in a state where the Governor first vetoed a Lottery and where charges of abuse and fraud were being lodged against the Scratch Off Games as an election issues (no such abuses were ever found). Systems bugs especially those that would make LOTTO become LOOTO, would have the

potential to wreck the funding of the entire already economically strapped public school system in California.

Note that it was expected that the Lottery would lose \$3,000,000 each day that it was delayed. The fund would never be recouped because lotteries theoretically go on forever. Delays incur tremendous costs.

While the financial and general processing requirements for the LOTTO Games stayed pretty much constant, the marketing whizzes kept coming up with new "sizzling" ideas to make the LOTTO more marketable and attractive to the buying public. Although their ideas made the California LOTTO the largest and most successful beginning Lottery in history, they posed challenges as changes were made, sometimes at the last minute, with little time available to thoroughly analyze and desk check the impact of such changes on the logic and performance of the rest of the system.

Maintaining a set of rigorous Regression Tests (mostly automated and using them systematically enabled us to identify real and potential problems quickly. It also allowed us to fix and retest these changes and bugs before the fierce and virtually immovable deadlines were upon us.

## 11. Test thy documentation

Ah, documentation. What is it? Why is it always the last thing to get done and only if there is time?

Why do users need it so?

First, what is it? It is all of the technical, programming, systems analysis and design (including users requested reports, screens, logic, features, functions, controls, and interfaces, and deferred items with appropriate signatures). It also includes maintenance, reference, training, demonstration, help screens, users, systems operators, control (including backup, recovery and restart, hot line, diagnostic materials, etc.) Other parts include installation and interface, and contract materials (including Requests for Proposals, Proposal, selection guidelines, attachments to the contracts, etc.), regression tests and expected results, acceptance test and signoff documentation, correspondence, guidelines and materials surrounding the planning development, purchase, implementation, testing, acceptance, performance, operation and maintenance of the system(s).

Why is it always the last thing to get done, and only if budget exists? Developers and programmers

for the most part enjoy getting on with building the product and not playing, correcting, and sanitizing the supporting and underlying paperwork. First, they do not understand its importance in allowing the system to be successful and cost beneficial over the long term

(in terms of reduced maintenance and life cycle costs, and in terms of user acceptance).

Second, they are not trained in developing and maintaining documentation. Last of all, they believe that they are being paid and rewarded by the number of lines of clean code that they produce, or the number of screens or features and functions that they deliver.

Why do users need documentation? Even for the most trivial of games, unless you are as clever as my six year old son, you will need to refer to the user documentation to review how to process periodic and less frequently processed transactions, respond to error and/or warning messages, and learn how to prepare special one time reports.

Even a perfectly functioning system loses credibility if the manuals say one thing and the systems perform in another way. A third party will find that during systems and acceptance testing is a perfect time to test the user documentation and insure that a harmonious package is delivered to users. And the perfect tester is an intended user.

A technical person is likely to look at user documentation, which often is prepared by another technical professional, and overlook many of the omissions that the drafter has made because they are from the same mindset. A user's thought process is likely to find different mistakes and will probably note the deficiencies right away.

This technique is something even the very savvy Lotus Corporation continues to exploit. The new

Lotus 1-2-3 for Windows product asks its customers/users to complete a Reader Comment Form in its "Before You Begin" manual "to help us improve our documentation" by questioning:

"What sections do you like? What topics need more explanation or were left out? What errors did you find and where? What part did you most use to learn the system? What books or on-line documentation would you add and why to this set? Other comments?"

On the other hand, technical people should review the technical documents of others. Users may make technical documents grammatically perfect, but technically deficient in terms of the accuracy and completeness of documentation needed to support the ongoing maintenance and development of the system over its life cycle.

Additionally, during the Acceptance Tests, I require/strongly recommend that the users first try to work out problems and potential bugs that may be caused by user/operator error by referring to the manuals and either resolving the problems or at least documenting the problems for the technical people. I repeat, because this is important, simply do not let problems go directly to the technical people without user-led analysis and research in the

user manuals first.

Indeed, the systems and acceptance tests are probably the last chances to thoroughly debug the user manuals. With employee turnover and mobility the way it has been for some years in industry, these manuals will be significant training tools and time and cost savers for years to come.

## 12. Test for performance or perish

Almost all of the acceptance testing I encounter tends to focus on feature, function, documentation and support testing, as opposed to performance testing. Performance testing includes those tests intended to establish how your systems and resources will actually execute in your intended environment in terms of speed, uptime, downtime, throughput capability, response times, portability, recovery and restart and backup performance. Testing also includes how the systems balances and prioritizes batch and on-line real time transaction processing, job mix constraints, etc. during normal, peak, at year-end and special periodic processing volumes.

I believe performance testing is not adequately performed for several reasons:

1. Users are unable to project just how the new system will be used in terms of volumes, file sizes, simultaneous transaction, peak periods, etc. This is especially true if the new system avails new capabilities to the users and if the systems processes transactions differently than the system being replaced.
2. Vendors are unable to assume the risk of guaranteeing performance when the user is unable to identify just how the system will be used at the customer company after the system is implemented.
3. Users are lulled into the belief that they cannot understand the technical formulas that vendors give them, which relate to some "magical" and not understood mix of transactions and processes for the average company - not their company.
4. New users inadequately interview and rely on reference checks regarding performance from otherwise happy vendor reference checks without really taking the time to understand the different processing requirements between the two organizations.

Lee Gruenfeld, formerly with a big six accounting firm and now a "fiction writer," establishes a very nice solution that both the user and the vendor can agree to without taking on incalculable risks. Note, that while in other Commandments the focus was very strongly biased toward what users must achieve to improve testing quality and results, performance testing involves a partnership between the user and the vendor. A very specific meeting of the minds between the parties that must be reached before such testing can be adequately approached.

The strategy is for the user and the vendor to agree upon a fixed "best guess" set of volumes, files, simultaneous activities, transaction mixes, and operations that will best

represent how the user is likely to use the system in the future. This documented series of parameters and the required results in terms of response times, throughput, etc., becomes the definitive performance acceptance testing criteria. Because it is fixed and can be systematically measured, vendors are comforted that they will not be going after and guaranteeing an undefined and moving target.

Even before such tests begin, I often see this process flush out the "truth" in terms of the power and performance that the vendor will guarantee for the system. When you ask for a transaction response time (given a particular specified load) of two to three seconds, 98% of the time, with no transaction taking more than ten seconds, the vendor will often confide during the negotiation and before the testing commences that a four to five second response is what is more likely and can be guaranteed.

Contractually, the user can bargain for a specific set of system resources at a given price to obtain the performance criteria promised with the "best guess" database. If the hardware/software is unable to perform, the vendor at its own expense and at the users' discretion will be required to supply the resources, including hardware, software, and human resources, to make the system perform as promised. If the user decides that he/she must actually use the system differently or in a more powerful manner than contracted for in the performance test, those expenses are borne by the user. Each side pays for mistakes that it alone is responsible for. Of course, you may bargain for other dollar arrangements as to who should bear the cost of mistakes, paying only the vendor's cost without markup, etc.

You must make sure in the aforementioned scenario that there is room in the product line to grow in the event that more resources are indeed required. Also, it is typically wise to buy a little less than the vendor recommends in this scenario because the smart vendor will add a little extra power to assure that his/her company will not be on the hook for more resources at no cost or a deeply reduced price. If the slightly smaller system indeed performs well, you have saved your company valuable dollars. If you must in fact configure your system to the one that was originally proposed by the vendor or the one you tested, you can do so at the originally negotiated price with no real penalty or additional cost to your company.

## **EVERY GREAT SYSTEM MUST BE BUILT ON A STRONG FOUNDATION: ITS CORNERSTONES**

In this section, we will present the Cornerstones: developing running programs; testing applications software; cultivating management commitment; and knowing when to stop. We will discuss how these cornerstone philosophies produce dramatic benefits in your systems quality and usefulness in increased user acceptance, and in improved testing cost effectiveness and efficiency.

### 13. Accept running programs - Not working programs (Unit Testing)

Let us define working programs as those programs that work for the first few months after installation, but then fail to operate on a consistent basis because of "invalid data somehow creeping into the system." Developers who hide behind the "garbage in-garbage out" motto are the ones that most frequently develop working, but not running programs.

Running programs are programs that are developed and tested to run under the worst but invariably, the actual conditions that programs must run under. Conditions such as frazzled and frenzied operators, under-trained users who are moving from job to job, poorly controlled data captured on source documents, malicious hackers, disgruntled employees, under-tested operating systems and hardware, etc. Running programs are ones that have been thoroughly "unit tested" and are able to provide reliable results throughout the life of the system.

Some years ago, the accounting firm of Main Hurdman gave some of the most concise and astute definitions of the objectives of different levels of testing that I have seen:

- Unit Testing-Does the module function at a detailed level as designed?
- Interface Testing-Do the modules interface with others?
- Integration Testing-Do the application programs function as specified in program design?
- System Testing-Does the system meet its initial objectives specified in the user requirements documents?
- Acceptance Testing-Does the application meet the user's needs while executing in his environment?

Unit testing is the key to developing running programs and is the responsibility of designers and programmers with the objective of finding discrepancies between the module's logic and the module's external specifications at the most basic level. The criteria for the test include:

- Execute main logic path(s) and error handling at least twice
- Test boundary limitations
- Test all limiting values at their limit, e.g., field size (number of characters)
- Test initialization
- Check the first and last record, and
- Check data structures.

Unit tests generally use embedded switches to check logic paths instead of "live" data.

Simply testing that the system works correctly for anticipated (and hoped for) good data and "reasonable" errors will not provide a long term running solution. Systems must be developed to capture all erroneous data. Just as it was important to develop such "early warning tests to capture bad data" in the programs, it is equally critical to test that each works during final unit testing.

Otherwise, once an error gets past the point it was first introduced into the system, it is unlikely - even unreasonable- to expect that it will be tested again later for validity and resolution. This is especially true when data is further combined with other data later in the processing flow, say into difficult formulas. Such errors are difficult, if not impossible, to pinpoint, and create loss of confidence in systems and wasted time and expense for users and management.

Unit tests should examine such things as what happens when the user puts one billion dollars into a system only able to handle up to \$999,999,999? What happens when the systems expects an answer of M-male or F-female and receives something else? What happens when there are zero records, or 0, 1, 2 records in a file that is to be sorted? What happens when the report queues are exceeded? Such boundaries are the most frequently neglected by programmers during unit testing, yet are the most frequently and first to be breached by innocent, or sometimes savvy, users.

I recall a major systems development project that involved several hundred programmers and testers. I was brought onto the project relatively late to oversee the development of the acceptance and rollout of the systems. Although the team consisted of a sophisticated group of systems professionals, and the first year's revenues from the systems would exceed one billion dollars, the systems/acceptance test had to be halted as the team had to revert back to unit testing the basic elements of each program simply because we could not figure out what was causing the errors we were uncovering at the acceptance test level. (Does the application meet the user's needs while executing in his environment?)

Unit testing, assumed to be "not that critical," had been sacrificed to time and money pressures. Indeed, this attitude, and the team's willingness to produce working programs that would not run reliably under the most real and demanding conditions, was primarily responsible for approximately fifty days delay, which cost the vendor penalties in the seven figures and the company itself tens of millions of dollars.

Clichés live on because of the elements of truth contained within them. "Crawl before you walk, and walk before you run" has been around for ages. Perform your basic unit test in order to produce the appropriate foundation for long-term successful running programs. Otherwise, you may have to go all the way back to the beginning.

#### 14. Bless applications software more mindfully than custom software

There are some myths associated with applications packages that are particularly dangerous to new users and testers alike:

- Applications systems have been debugged by other users
- Since the package is not custom written, nothing can go wrong
- Because the system is used at dozens of other locations, a systems test and acceptance test are not really required, and
- An applications acceptance test is not needed once a Systems test has been successfully performed with vendor data.

The belief that applications systems have been debugged because they have been around for awhile is false, but for counter-intuitive reasons Applications are developed to sell to a mass market, i.e., please most of the users most of the time. Therefore, the system may not meet your expectations or your particular way of doing business.

For instance, how does the system handle a transaction that reduces your unit inventory to less than zero? When this happens, what do you want the computer to do? Keep the inventory to only zero? Hold the transaction in suspense until it can be resolved? Go to negative inventory? Value the inventory category as a minus? Calculate the inventory turnover using a zero, a negative number, etc.?

Remember, the computer is really a very stupid machine. A human must program it in the minutest details to tell it what to do. When one develops a custom system from scratch, all of the parameters and permutations, and the decision rules must be considered and decided upon affirmatively up front. In a purchased application package, the user is allowed to be sloppy and not consider all the combinations and preferences because the system will default to its own previously developed logic based upon the procedures, policies and needs of the developer and/ or other users and companies. This dereliction of affirmative user decision-making has been responsible for hundreds of failed installations and dozens of lawsuits and swift career changes.

Planning acceptance tests for application packages, therefore, must include testing features and functions at the most detailed level. This will assure that the logic of the systems is or can be made consistent with your needs based upon the user establishing and changing available systems options and parameters.

It is also important to remember that although there may be dozens, hundreds, or even more users out there, your process of setting the myriad of parameters, switches, tables, and codes essentially customizes" the system for you and may contain a combination of such parameters that has never been used or tested. If you do not test to determine what the cumulative effects are, this combination can have catastrophic effects on your particular installation.

Such parameters can include:

- report formats
- audit trails
- month end, quarterly, and periodic reporting
- classification schemes
- hierarchy schemes
- various formula usage
- security schemes
- module integration requirements (those within and without the page)
- processing sequences
- backup procedures
- security levels and codes
- operating systems and utility program levels, and
- data communications protocols, etc.

In addition, every system is part of a larger "enterprise" system that includes policies, procedures, people, management style and objectives. An application system that works well in one company environment may not be acceptable in another. Initializing and/or converting your old files into new formats often can be a major effort that introduces errors into the system. Furthermore, new procedure manuals may have to be developed which integrate the new applications with the functions that take place before data is captured/ entered into the system and which discuss how the information generated by the system should be used by your company.

Appropriate systems/acceptance tests for applications software must assure that the system is operating as the users expect, and that the users are operating as the system expects. The purpose is to flex the system to test the "nuances" (i.e., standards, defaults, logic structures and logic handling) that the original designers preferred. There is no one right way-but there may be your way and their way (and wouldn't it be nice to know before you implement the system, rather than to learn it by surprise later as the system gives you results that are not what you 'need or expect?')

Assuring that the nuances in your particular installations are understood accordingly requires much more than simply exercising the system with only vendor-prepared data. Vendor data typically is developed to show a system in its best light, including how quickly the system responds to inquiries. The purpose of acceptance testing is not to say that the system is fine, but to find any problems and bugs in the systems before accepting the system as is, before going live with production, and before fully paying the vendor.

## 15. Thou shalt cultivate and win management commitment and understanding

You have probably heard the riddle: "Regarding a bacon and eggs breakfast, what's the difference between the chicken and the pig?" Answer: "The chicken was involved - but the pig was committed!"

So, too, must top management be committed to assure that appropriate testing policies, procedures, budgets, and authority are provide to guarantee your company will put quality software into the field. Why is commitment so necessary? Because testing is difficult, looked upon as destructive, and very political.

In testing California's Lottery Systems I was fortunate enough to be assigned to the State Controller's office to assist in overseeing the testing, acceptance, and sign-off of the systems to go live. Among the players and forces that were affecting the development and testing teams were the following:

- Outside consultants looking for a large fee
- The press, who wanted to announce the news of the upcoming lottery and related stories
- The vendors, who faced a \$50,000 per day penalty for each day the system was behind the contracted start date
- Public relations agencies, who needed to know the exact start date so they could plan big kick-off events and get free publicity
- The Governor, who vetoed the Lottery in the first place and who needs the system to go up on time, absolutely free of bugs or any taint that the systems could be tampered with, rigged, or infiltrated
- The schools of California, who received over \$350,000,000 the first full year the games went live
- Retailers (approximately 12,500 on day one), who wanted the increased revenues, cash flow, and foot traffic in their stores that the Lottery would bring
- The lottery security police, who needed certain policies and automated controls in place before they could adequately control the games
- The voting public, who demanded the games in the first place, and
- The telephone company, who would make a fortune from the new switches and traffic carried and helped control the tickets purchased from the thousands of Lottery terminals.

Because it was an election year, there was also a special panel investigating fraud and abuse in the scratch-off games. Although none was found it made the development of the new system even more political (as if that were possible).

I have found that at virtually all installation sites there are a great many politics. For instance, many reputations are on the line because systems and acceptance testing are not

well understood and almost always underestimated. The vendor has promised a delivery date to the user and his own management. The users are expecting a cutover to relieve them from the servitude of an already inadequate current system. Developers and programmers want to move on and get rewarded for the good job they believe they have done on the new system.

The systems/acceptance testers are viewed as a destructive bunch - out to nitpick piddling errors found in the to-be-released system. AU of the powers look to the tester to highlight small quickly - fixable errors, more concrete problems "should" be documented for a future release. After all, there will be time to fix it later - and much more time for finger pointing. It is only management's commitment to understanding the importance and costs associated with delivering faulty systems, and management's dictate that a system will not be delivered until it has passed quality testing standards, that will cause "Baldrige Award" quality systems to become a reality.

Another aspect to commitment is getting all of the technology parties to commit to the same goal/ definition of "testing errors." "What are these errors that the testers are looking for so eagerly anyway?" query the developer/programmers. One easily remedied error may cause a room full of "System Error Reports" because of the way that it formats reports. Another error, however, may clobber your file indexes and make your database unusable. Isn't one more disastrous than the other? Absolutely!

Yet testers are motivated to find the greatest number of systems discrepancies, while programmers are evaluated/motivated to make the least number of programming errors- regardless of the impact on the company. A better approach, put forth by Bob Stahl, President of Interface Design Group, is to evaluate performance by the cost and amount of time it takes to fix an error and the impact of the error on operations. This would hopefully center programmers, testers, developers, and managers alike to focus on those items and codes that really have the most potential impact on the company.

## 16. Thou shalt measure progress - and know when to stop

Quick as you can-take the following multiple choice test. Ready, set, go! "What is the definition of a completely debugged program?"

1. We ran out of time
2. We ran out of money
3. We are not sure how to test such a difficult online program
4. The users will find the rest of the problems and bugs anyway, and
5. All of the above.

The incorrect but most frequently given answer is "5. All of the above." Testing is still misunderstood, undervalued, too lightly budgeted and under planned – although it doesn't

have to be. It is no wonder that testing is the first area sacrificed when systems, monetary, and human resources get low. It is also the psychology behind answer "4. The users will ultimately discover the problems ..." albeit at the worst possible moment that has caused the current torrent of users and management alike to demand a more rational and practical testing methodology for creating and maintaining quality systems.

One of the weapons that must be included in each tester's arsenal is "Completion Criteria." These are a set of prioritized tests that must be accomplished in order for users and analysts to sign off on a program/system. Since it is impossible to completely test everything (our very first Commandment), the completion point must by necessity allow some low return testing to go uncompleted or abridged.

Clearly, the relative importance of the functions play a major role in determining the degree of correctness the program/system requires. The completion criteria for the new Federal Bureau of Investigation Criminal System is different from the appropriate completion point for testing the number of cars that pass a deserted desert signpost between midnight and 6:00 a.m. Criteria for calling a halt in testing must insure that the system has been thoroughly tested and that the tests do not cost more than the benefits provided by the additional quality. The completion point is identified before testing begins and must be reached before testing can be halted and acceptance considered.

The testing priorities assigned are based upon the functional equivalence, error guessing, cockroach theory, and risk driven commandments that we have already discussed. Testing results are monitored continually to insure that the number of bugs discovered for a given amount of effort is diminishing dramatically. Thus, the completion criteria approach assures that:

- You are focusing on the most critical features and functions of your system
- You are focusing on those programming areas that have proven to be the most problematic historically, and
- You have developed a measuring stick that tells you when the testers have reached diminishing testing returns.

Completion criteria define that point beyond which you must stop testing-because you are assured to have found and corrected the most serious bugs. You cannot beat that.

\*\*\*\*\*

The 16 Commandments discussed above will provide you with a methodology, checklist, and hierarchy for planning your systems and acceptance policies, processes, and procedures. If followed religiously, the commandments will allow and cause developers, users, testers, vendors and installation managers alike to create, install and accept systems better able to handle company/user needs and objectives at a minimum of cost. The price

of a failed installation, given the mission critical use of computers and information resources in today's businesses, simply cannot be tolerated. The Commandments are here as a road map to your success!

While the Commandments help you find/correct design and programming, functionality, and performance deficiencies, they do little to prevent you from selecting a "wrong" software solution for our needs in the first place (i.e., the preliminary selection process comes before the testing process).

A future article will overcome this problem through a unique and masterful way to request, evaluate, select and contract for the "best systems" to meet your needs.

Note also that the testing arena (strategies, methodologies, training, tools, IV&V, the conduct and use of "static" tests, and more) continues to change and improve. As these constructs become available, savvy customers, vendors, users, integrators, and consultants must embrace them and make them a part of their own testing regimen.

*Warren S. Reid*