

Reflections on Scheduling Software Tests, the Project Life Cycle, and the Last Minute Bug

Originally published in "SQA Magazine" (July 1998)

Remember high school? Who was your favorite teacher?

My favorite high school teacher was Mrs. Kellerman. She taught U.S. history. Her most memorable lecture described why we studied history. Her belief was that from studying history, you could learn judgment. Her point was that in it's a fallacy to look back on mistakes you've made and lament, "If I'd only known" because typically, you'll won't have all the answers before presented with a decision. She included examples such as the Hessions celebrating on Christmas night secure in the belief that Washington's army was in retreat and the Delaware river was too choked with ice for any to cross. Washington knew different. Or the political bosses in 1900 believing that they had "buried" Theodore Roosevelt in the vice-presidency where they knew he could do no harm. A year later, however, McKinley was dead and "that damn cowboy" was in the White House. Or Richard Nixon starting his career attacking corruption in high places.

What's this got to do with life cycle of software development? Just this; when the time comes when you have to make a decision about whether the software is ready to move from one stage in its development to the next, you will never have all the information that you would like to have. Schedules will always change, the unexpected will always happen, and there will always be unknowns.

I like to think of decisions involving the readiness of software to move closer to its release as a type of algebraic expression composed of constants and variables. My goal is to always limit the number of variables. After all, which of the following expressions is easier to resolve? (I vote for the second one.)

This? $q = \sqrt{\frac{y*(z-x)}{a+b}}$

Or this? $q = \sqrt{y}$

How do you reduce the number of variables? Define quantifiable requirements that have to be met before the project can move from one stage to the next. The stages? Here's a summary:

Project Life Cycle Stage	Guidelines for Criteria to be met Before Entering Stage
Start of Debugging/Unit Testing	<ul style="list-style-type: none"> • Marketing requirements defined, agreed to by project team • Functional spec complete, reviewed, agreed to by project team • Design spec complete, reviewed, agreed to by project team • Test plan complete, reviewed by development, agreed to by development, customer support, and test groups
First Complete Handoff for Testing	<ul style="list-style-type: none"> • Coding, debugging, unit testing of major functions complete • Smoke test complete

Reflections on Scheduling Software Tests, the Project Life Cycle, and the Last Minute Bug

	<ul style="list-style-type: none"> • Release note describing code changes, bugs fixed, known problems, smoke test results complete
Subsequent Handoffs for Testing	<ul style="list-style-type: none"> • Bugs found in prior handoffs solved • Planned tests complete • Smoke test complete • Release note describing code changes, bugs fixed, known problems, smoke test results complete
Start of Alpha Test	<ul style="list-style-type: none"> • Planned tests complete • Alpha test plan complete, reviewed, agreed to by alpha test personnel, alpha test support personnel, development and test groups • Open bugs reviewed, severe bugs affecting alpha test tasks resolved, minor bugs (and workarounds) documented
Start of Beta Test	<ul style="list-style-type: none"> • Alpha test complete • Beta test plan complete, reviewed, agreed to by customer, customer support personnel, development and test groups • Open bugs reviewed, severe bugs affecting beta test tasks resolved, minor bugs (and workarounds) documented • Level of open bugs agreed to by customer
First Customer Ship	<ul style="list-style-type: none"> • Final bugs (from alpha and beta test) reviewed, all severe bugs solved, consensus reached by project team on number and severity of open minor bugs • Open minor bugs (and workarounds) documented

Project Stage Transition Criteria Guidelines

The key is that the requirements must be quantifiable. You have to be able to measure your progress against these requirements. You can't measure progress against subjective goals. If your requirements are that the product must be "good", it's bound to never be good enough.

You also have to be careful about falling into measurement traps. For example, there is a world of difference between "coding 90% complete" and "coding 100% complete." In the first case, you've probably only spent about one-half the total time that will ultimately be needed for coding. In the second case, you're done! Want another example? Debugging is nearly always "99% complete." It's the other 1% that seems to take forever![1] Using unclear milestones also make scheduling a project just that much harder. It's very hard to estimate how much time you'll need to accomplish an opened task!

How many open bugs are too many open bugs? How many minor bugs can be open when the product goes to alpha or beta test or release to customers? In an ideal world there would be none. In a less than ideal world, as few as possible. In the real world, this depends on the following factors:

- a) The Nature of the Bugs - All minor bugs should be reviewed by the project team to ensure that the impact to the user of each of them individually and all of them as a group to the user is truly "minor." In addition the expected frequency with which the bug will be encountered should be considered.
- b) The Risks of Fixing Them - A bug with negligible impact to the user may require either extensive coding changes or changes to complex (i.e., bug prone) code. In these cases, you're better off documenting the bug and its workaround and not changing the code.
- c) The Nature of the Bug "Workarounds" - A minor bug with a simple, elegant workaround (i.e., a way to avoid or recover from a bug) may not cause a hardship to the user. A workaround that requires the user to reboot his system every hour is not elegant and should be fixed.
- d) Contractual Agreements - In the case of custom developed software, the number of minor open bugs allowed is often decided for you by the customer.

Reflections on Scheduling Software Tests, the Project Life Cycle, and the Last Minute Bug

1.1 Why is Scheduling so Hard?

Most people, even pessimists, turn into Nellie Furbush[2] and become “cockeyed optimists” when they try to plan how long it will take to accomplish a task. So what happens? We fall into making the (sometimes unspoken, but always present) false assumption that this time, things will go right and only take as long as they should to accomplish. Part of the optimism is based on the materials used. Physical materials such as hardware have physical limitations. Software, however, is basically built from ideas. Our capacity for ideas and imagination are unlimited, and the software can easily be made to implement these ideas, so we’re optimistic about being able to create it![3]. This is the case even when we know from our first hand past experiences how very wrong things can go.

Part of this overly optimistic view of the future is a denial of the past. The attitude I’ve encountered runs something like “Hey, that was then, this is now!” The only problem is, I keep seeing people repeat the same mistakes. Why is this? Remember how I said it was important to have measurable requirements? It’s also important to have measurable improvement requirements. How do you define these improvement goals? You document the project as it progresses periodically (that’s right, not just at the end of the project) review what’s going right and what went wrong, and define what needs to be done to not duplicate the same mistakes over and over again.

There is another dangerous assumption that creeps into the scheduling process for software; that if you fall behind schedule, you can simply throw more people at the project in order to make up for lost time. This will work if the task can be divided among multiple workers without any communication between those workers. Frederick Brooks describes this type of task in his book The Mythical Man-Month as being “perfectly partitionable”[4]¹ and uses reaping wheat as an example. Developing and testing software, however, are tasks that require extensive communication between workers. This communication takes time. It can take so much time, in fact, that a point can easily be reached where adding more people to a project team will actually make a late project even later.[5] It’s a funny thing, but people seem to understand this fact in the abstract, but when it comes to “their” project, they still think that “throwing bodies” at the problem will make a bad situation better. It’s that same “this time it’ll work” behavior!

1.2 Why is Forecasting Bug Levels Even Harder Than Scheduling?

One of the difficulties in defining a test schedule is estimating the number of bugs that will be encountered. You may be able to define the specific tests that should be run, and estimate the time required to run the tests, but your test schedule will quickly become inadequate if you discover large numbers of serious bugs as you may have tests blocked by the bugs. In addition, every bug fix will require additional testing and regression testing. There’s no easy and guaranteed accurate method of forecasting the number (and seriousness) of bugs that you will discover testing a software product. However, an objective assessment of the project, coupled with the results of previous projects can be very helpful in forecasting bug levels. It’s easy to fall into the trap of creating an overly optimistic schedule and then find yourself falling further and further behind as you are unable to achieve milestones. It’s just as easy to fall into the trap of forecasting a low level of expected bugs and then having to scramble to find time to verify the bug fixes.

In assessing the current project to forecast the number of bugs that will be found, it’s important to involve the entire project team. This is the case because ultimately, the number of bugs found is not just a testing concern, it’s everyone’s concern. Everyone should understand the scope and complexity of the project, and the risks inherent in its design and development plan.

In addition to understanding the complexity of the software, you must use a standard method of measuring that complexity. Software metrics as a discipline endeavors to provide that standard method of

1

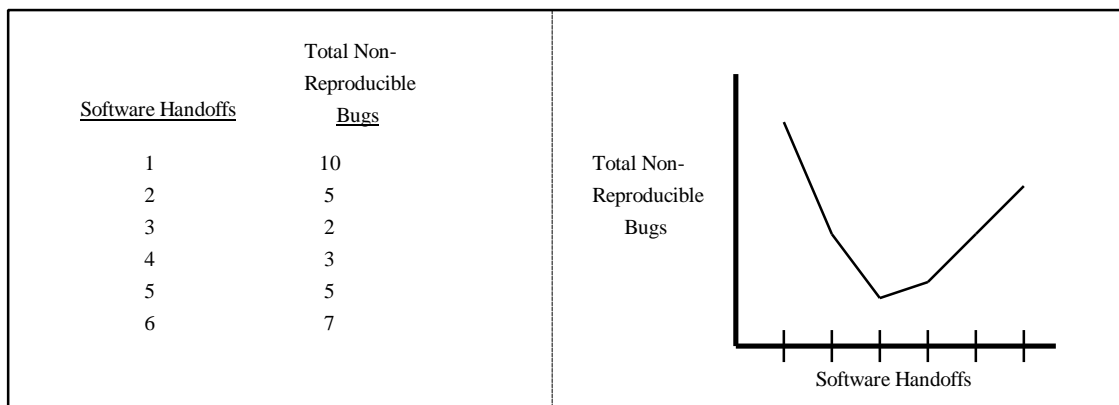
Reflections on Scheduling Software Tests, the Project Life Cycle, and the Last Minute Bug

complexity measurement. The most basic metric is to measure the number of bugs per line of code (LOC), or per thousand lines of code (KLOC). This metric is simple to apply, but it suffers from being tied to the format of the source code. For example, if you don't count comment lines, you get a more true count of bugs per line of "code," but you will miss bugs introduced by incorrect or misleading comments. Likewise, if you include white space or blank lines to make your program more readable, you will affect the lines of code total[6]. A more sophisticated metric is "McCabe's complexity metric[7] where you count the decision points in a program.

What's the best metric to use? How do you introduce the use of metrics into your software development process? First, pick a simple metric such as LOC, and start measuring all code consistently. (Either always count comments, or always don't count comments.) Second, find an automated means of applying more sophisticated metrics. It doesn't matter how careful you are, you will always make mistakes (and probably lose your mind) determining complex software metrics by following formulas in a book. Third, compare the metrics you find against the actual bug counts until you find a system that works for you. In my own case, we're currently in the LOC phase and are looking at systems to enable us to perform additional metrics.

In reviewing the results from past projects, you will be able to use the information in the bugs reported during those projects. The number of bugs reported for specific functions or subsystems that will be included in the project to be tested will give you an idea of the expected reliability of the software. Remember, you are more likely to find bugs in those areas where bugs have been found in the past. In addition, the root causes of bugs reported on past projects will help to identify procedural problems (e.g., ineffective code reviews) that, if they have not be repaired, will cause bugs in the software. In the course of reviewing past projects to forecast bug totals on the current project, don't forget to consider the number of bugs caused by the fixing of other bugs. Determine the "batting average" of past projects (e.g., one new bug introduced for every 5 bugs fixed) and take it into account when you estimate a bug total for the current project.

The information from these past projects is important, The manner in which the information from past projects is presented, is also important. For example, the following figure illustrates the same information, that being the number of non-reproducible bugs in each test handoff for a project, as a list of bug totals and a graph. The list and the chart both include the same information, but the chart more clearly dramatically highlights the problem of increasing numbers of non-reproducible bugs affecting the project. Note that ideally, fewer bugs such as these will be encountered later in a project as the software should become more and more stable during its development. To illustrate:



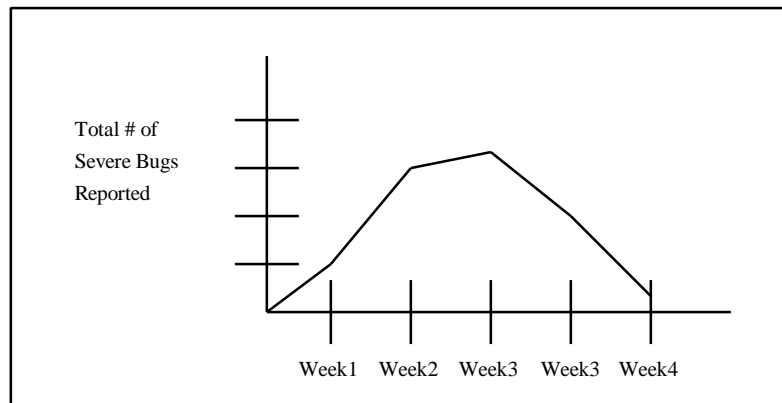
The Effectiveness of a Graph

Finally, it's important to not treat the original bug forecast as if its sealed in amber. Rather, the forecast should be revised periodically as the project progresses. Everyone will know more about the project after

Reflections on Scheduling Software Tests, the Project Life Cycle, and the Last Minute Bug

they've worked on it for a while. It may be that people's estimates and preconceptions will turn out to be grossly inaccurate after the project has started. You're always gaining knowledge in the course testing a product. That knowledge should be put to use.

Remember the old saying about how "statistics don't lie, but liars use statistics?" I was a witness to a great example of type of behavior a few years ago. I was working on a product that was in trouble. We had missed many major milestones, and were encountering large numbers of serious (system crashing) bugs. The project manager was very competent, but she was also a great politician. She was presenting a project status report, which included the following chart, to upper management.



A "Near Occasion of Sin"

At first glance, it looked like a bad situation was getting better. At this point, my manager poked me in the ribs. I raised my hand and explained that while the number of serious bugs found in the past two weeks was lower than previous weeks, this was caused by the fact that the system was crashing so often we couldn't run many tests. I'll say one thing for her, she could take a punch. She didn't bat an eye. She simply moved on to the next chart.

In order to use the bug levels reported for a given product as the basis for forecasting future bug levels, it's important that the bugs are periodically reviewed to ensure that all bugs logged are legitimate. A large number of bugs with a final disposition of "non-reproducible" may point to stability problems with either the product under test or the testing environment. Likewise, a large number of bugs with a final disposition of "not a bug" may point to a difficult user interface or a lack of understanding on the part of the test engineers. Neither of these types of bugs should be left out of a study for forecasting future bugs levels as the problems that caused the bugs may be encountered again.

Non-legitimate bugs, however, should not be considered in a study for forecasting future bugs levels. Running up the score with bugs such as these makes determining future bug levels difficult, if not impossible.

Some years ago, I encountered an especially memorable example of someone inadvertently running up the body-count with "cheese-cake" bugs: The test engineer in question was a gifted mathematician with an advanced degree from the University of Leningrad (aka St. Petersburg). She had done some programming in her career, but she had absolutely no prior software QA test experience before she was assigned to my software test team. Her first assignment was to write a series of BASIC compiler verification programs. She quickly discovered a bug in the SQRT (square root) function. Her initial reaction was embarrassment because her test program had failed. It took quite a while for us to explain to her that her program was actually a success because it had uncovered a bug. It was at this point we discovered a truly charming aspect of her personality. She was only about 4' 10", and insisted that anyone coming into her office should sit down. "Please to sit" was how she phrased it. After much effort, we got her to accept the

Reflections on Scheduling Software Tests, the Project Life Cycle, and the Last Minute Bug

premise that the more bugs she found and reported, the better. The next day, she logged dozens of new compiler bugs. At first, we thought she had found her true calling in life. In examining the bugs, however, we found that she had included a call to the broken SQRT function in each of her test programs. It subsequently turned out that software testing was not her true calling.

1.3 The Sequence of Tests

1.3.1 Smoke Test

The first part of the testing is a "smoke test" to determine if the software is sufficiently stable to permit complete testing. A smoke test is an initial test performed on a software release to confirm that the release is sufficiently stable to permit more extensive testing. Why is it called a smoke test? The name comes from hardware engineers who would *carefully* apply power to a newly developed (or patched) board or component and watch for smoke. Very exciting.

A smoke test should be performed each time a software release is presented for testing. The specific tests making up the smoke test will vary from release to release based on the performance of the previous release. For example, if a series of problems with sorting data in a database is encountered in release (X), then tests for sorting will be included in the smoke test for release (X+1).

Every smoke test starts with installing the software on a system that is either "clean" (i.e., with no previous version of the software installed) or has the same software installed as the customer's system. This is important to ensure that if a release is missing any files, that fact will not be masked by its running successfully with files installed in previous releases. After the installation, the checksums should be checked, and all bugs reported to be solved should be verified. Finally, a small number of tests, some from each functional area of the software should be performed to complete the smoke test.

Note that having development run the smoke test before the software is presented for test can greatly reduce instances of software releases turning out to be DOA when presented for test. Having the test department rerun the smoke test after the software is delivered for test can serve to verify the process by which the software is packaged on portable media.

1.3.2 Functional Test

After the smoke test is successfully completed, the testing of the core functions is performed, followed by the testing of the ancillary functions. The order in which the functional tests will be executed will be such that the "core" functions of the product under test will be verified before the ancillary (or "stand-alone") functions are verified. The rationale for this approach to testing is that the ancillary functions, in order to perform successfully, require the correct operation of the core functions.

The approach for functional testing should be to perform as many tests as possible on each release. If tests for one functional area of the software are blocked by bugs, tests for other functional areas should be performed. This approach to testing will "wring" as many bugs as possible out of each release in the shortest time.

After as many functional tests as can be performed on a release are completed, a new release should be accepted for testing. This process should be repeated until the functional testing is complete. The functional tests for later releases should include a subset of those test that have already been completed. This is called a "regression test" and serves to verify that functions that had performed correctly in the past were not inadvertently broken by subsequent development or the solving of bugs.

Reflections on Scheduling Software Tests, the Project Life Cycle, and the Last Minute Bug

1.3.3 System Test

After functional testing is complete, and the criteria for transition to system test (e.g., maximum number of open bugs) that were established earlier in the project are met, system testing can be performed. As was the case with functional testing, each release should undergo a smoke test, and as many tests as possible should be run on each releases. After system testing is complete, and a consensus is reached by the project team that the criteria established for the start of alpha test have been met, the software is delivered to alpha test.

1.3.4 Alpha and Beta Test

The pre-system test procedures are repeated, and after the increasing stringent criteria are met, the software is delivered for beta test. At the conclusion of beta test, a final regression test is performed to ensure that all bugs believed to be solved are actually solved, and no new bugs are present. At the conclusion of this final regression test, the product is released to the field.

A note on regression testing:

It's likely that a fair number (and sometimes a large number) of bugs will be in the process of being fixed at any point in time during a software project. Fixing each bug involves changing code, and this introduces risk as bugs may be inadvertently introduced while the existing bugs are solved. In order to ensure that new bugs are not introduced, a test of functions that had previously worked is performed. This test is called a regression test. The specific tests performed as the regression test will vary and expand throughout the stages of the project, as more and more of the products functions are implemented.

Automated tests are very helpful in regression testing as you may have to rerun the tests several times, each on multiple software handoffs. In addition automated tests lend themselves to regression testing because, the regression test will consist of tests for functions that have already been tested, and therefore exhibit known behavior. Therefore, once the product is fully functional, the results of each pass of the regression tests can simply be compared against the previous pass to locate any new bugs introduced.

Reflections on Scheduling Software Tests, the Project Life Cycle, and the Last Minute Bug

1.4 The Last Minute Bug and How to Deal With It

Have you ever heard of Harry Vardon? He was the greatest professional golfer of his day. (His day was up until World War I.) There's a story that someone once asked him what was the most important thing to remember in playing golf. His answer? "No matter what happens, keep on hitting the ball." It's like that with software sometimes.

It's frequently the case that a serious bug is found very late in the testing of a software project. This can be a stressful situation as the bug, by delaying the release of the product to the field can literally cost the company revenue on a daily basis. It's important to keep calm and stay focused on the goal of removing the bug from the software, while not introducing any new bugs. The following actions should be helpful in dealing with a last minute bug:

- Work with Development
- Stay in Control of the Test Environment
- Perform Regression Tests
- Keep the Focus on THE BUG

1.4.1 Work with Development

The developer working on solving the bug may only have in-depth experience with specific parts of the product. This may lead them to follow a design that will introduce bugs in related parts of the product. The test engineers will have experience with the entire product and should therefore review any design changes made to solve the bug. Remember that bug prevention is much more efficient than bug detection. The last thing you want to create a new bug in the process of fixing a last minute pressure to solve the bug, and may only concentrate on the specific part of the product

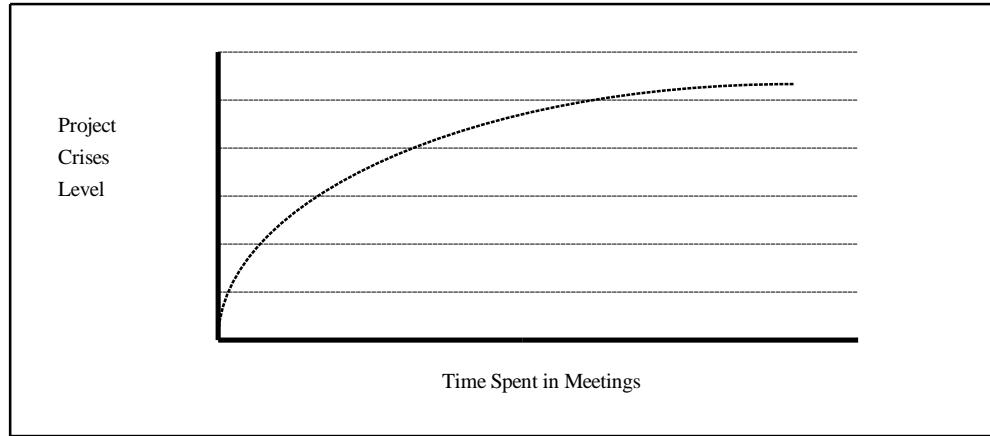
1.4.2 Stay in Control of the Test Environment

In order to be able to verify a bug solution incorporated into the final release of a product, you will need to test that solution with the correct version of the software. This may sound obvious, but a last minute bug brings with it schedule pressures that may cause multiple versions of a program to be created in rapid succession as the solution is debugged. It is imperative that the solution be verified with the same software with which it will be shipped. It does no good to verify a bug fix in Thursday's version of the code if Wednesday's ends up on the tape sent to the customer.

In addition to controlling the hardware and software in the test environment, the people performing the testing need to be controlled. It's often the case that, management will attempt to solve a problem by "throwing bodies" at it. Tasks that can be broken down into pieces and performed by people with no communication between them can be done quicker by adding staff. Testing software, especially complex software, however not only requires extensive communication between people (the more people, the more coordination is needed), it also requires people to be trained before they can make any contribution to the test effort.[8]

And, don't forget to control management's efforts to get personally involved! Frequently when a last minute bugs (or bugs) are delaying a project, higher and higher levels of management start noticing the project and start trying to help. All too often, this involvement results in test engineers being compelled to write status reports and attend frequent (even daily) status meetings. Let them know that you appreciate their interest, but also let them know that if you are spending all your time keeping them up-to-date, you'll have no time left to actually test the software. The following chart, while having a humorous aspect, is frequently only too true:

Reflections on Scheduling Software Tests, the Project Life Cycle, and the Last Minute Bug



Relationship Between Project Crises Level and Time Spent in Meetings

I recently had to deal with the "case of the helpful development manager." We were nearing the end of a project, time was very short, but we had a large number of tests that had never been run. Enter the helpful development manager. He appeared at my office door and announced, "I found someone to help you out. Me!" His heart was in the right place, but the amount of time I had to spend training, directing, and correcting his work ended up looking like a test case for proving Brooks' law that "Adding manpower to a late software project makes it later." [9]

1.4.3 Perform Regression Tests

After the bug fix has been verified, a regression test must be run to make sure that the bug fix has not introduced any new bugs. The design of the bug fix will point the way toward deciding which specific tests should be performed. It's ironic, but the shorter the schedule, the more requests you will receive for testing status. Quantifying the test status in a milestones file, and making the file available to everyone on the project team on an ongoing (sometimes even on a daily) basis will preempt you're being continually asked "how's the testing going?"

1.4.4 Keep the Focus on THE BUG

It is frequently the case that once the bug is fixed, the temptation to fix other bugs, or even add a new feature will be too strong to resist [10]. Once the code has been unfrozen some people will feel it's season for "just one more" change. You may not be able to stop this from happening, but if it does, quantify and publish the risk to the schedule that the additional testing will involve.

I encountered a situation like this some time ago. The project team had just completed what we all thought was the final bug review. We had all agreed that one last major bug was to be fixed, and that a small number of minor bugs would not be fixed as the risks in making additional last minute code changes outweighed the benefits of fixing the minor bugs. A certain development engineer had other plans. He not only fixed the final major bug, he also fixed one of the minor bugs. In the process, however, he introduced a new major bug. We were lucky to have had automated regression tests that this new major bug. It was sort of like a doctor deciding to remove a kidney during a lung operation (while the patient was already open) only to discover that the patient only had one kidney!

References:

- [1] Fredrick Brooks, The Mythical Man-Month (Reading, MA: Addison-Wesley, January 1982), p. 154.
- [2] As portrayed by Mary Martin in Richard Rodgers and Oscar Hammerstein's "South Pacific" (1949).

Reflections on Scheduling Software Tests, the Project Life Cycle, and the Last Minute Bug

- [3] Brooks, p. 14.
- [4] Brooks, p. 16.
- [5] Brooks, p. 18.
- [6] Boris Beizer, Software Testing Techniques (New York: Van Nostrand Reinhold, 1990), p. 280.
- [7] McCabe, T. J., "A Complexity Measure", IEEE Transactions on Software Engineering, SE-2, 1976, p. 308-320.
- [8] Brooks, p. 16.
- [9] Brooks, p. 25.
- [10] Walter S. Mossberg, "In the Garden They're Bugs; in PCs They're Defects," The Wall Street Journal, March 16, 1995, p. B1.

Len DiMaggio (ldimaggi@bbn.com) manages the Software Test and Quality Assurance Department for GTE Internetworking (formerly BBN, Cambridge, MA, 02140, <http://www.bbn.com>) and is writing a book on the implications of the internet and world wide web on software testing and quality assurance. GTE, through its Internetworking division, is a leading provider of Internet services for businesses and organizations. BBN, the company that created the ARPANET, the forerunner of the internet, was acquired by GTE in 1997.