By Lars Mats, Telelogic Technologies Malmö AB

SOME COMMON SOFTWARE-TESTING PROBLEMS ARE NOT SEVERE, AND OTHERS ARE MORE COSTLY TO CORRECT. REAL-LIFE SCENARIOS PROVIDE HELPFUL GUIDELINES THAT CAN PREVENT THESE PROBLEMS IN THE FIRST PLACE.

The top five software-testing problems and how to avoid them

If you make a list of some of the most important traps in testing, you will realize that in many cases the problems are nontechnical. More often than not, they are consequences of the test process itself, including the overall composition of the test team and whether the company follows well-integrated processes for formal requirements handling and change management.

An informal survey of the relative cost of testing software compared with the overall cost of developing software gives a range of estimates, from 10% in smaller organizations to 70% in some larger and mature organizations. The results indicate the huge discrepancy in the level of importance that different organizations give to testing. Some of these problems are more common to younger organizations; others are pitfalls that anyone can encounter.

The following case stories and suggested remedies can help you overcome real-life software-testing problems.

PROBLEM 1: THE CONFUSED TEST TEAM

A project manager fears that his team will not complete the test activities by an aggressive deadline. You make a house call, and as you interview the test team, you find that the members have serious doubts about whether the chosen methods are appropriate and, in fact, whether they work at all. Your first feeling is that the team is upset over things other than work and is not focusing on how to perform its tasks. The project is on an aggressive delivery schedule, and, therefore, many of the available trained engineering resources are working 24 hours a day on the design team. In actuality, a lack of technical leadership leaves the test team flabbergasted and unable to complete the test work on time.

Test activity falling behind schedule and low morale often indicate that the test team does not have enough resources or is unqualified to perform the task at hand. This deficiency may be due to limited resources, a lack of training of the individual team members, or a leadership problem, or it could suggest that appropriate testing means are unavailable to the team.

This problem, although grave, is usually easy to fix. The difficult part is detecting the problem, because an unqualified team leader may hide or play down his or her shortcomings. A separate, working quality-assurance team should frequently review overall project progress.

Also, because it is often as difficult to test a system as it is to build one, do not make the mistake of putting all the best engineers in software development. Although they may invent brilliant software solutions, if an equally brilliant test team is not detecting problems, the solutions won't necessarily fly. This scenario is particularly true when you are building highly complex software within strict time limitations. In those instances, it is vital that the test team and engineers are equally qualified to test complex software requirements under similar time constraints.

After determining whether you have a leadership, resource, or training problem, or a combination of problems, you can take straightforward measures to remedy the situation. Remember that detecting and correcting problems at an early stage is crucial. The longer the problem continues, the more difficult it is to handle. In general, you need to balance the available design resources with the available testing resources.

PROBLEM 2: THE TEST-MAINTENANCE FAILURE

After 16 months of creating test specifications from a requirements specification, the requirements organization publishes a new version of the requirements specification. At this point, only very limited traceability exists from the written tests to the corresponding requirements specification. Consequently, locating the tests that you need to update according to the new requirements specifications requires another several months.

Requirements-specification changes lead to abnormally long turnaround times. At its worst, testing cannot cope with requirements changes. Either the traceability of requirements to test cases is inappropriate, or the test method in use does not localize the effects of changes to requirements or other artifacts that the test specifications depend upon.

Design the tests with maintainability in mind, just like you do when you design the system. Successful uses of this approach include grouping test cases by requirements, implementing traceability to and from requirements, and using abstractions in the chosen test description language. The added benefit is an ability to calculate requirements and test coverage based on this traceability.

Even better, using test-case-generation techniques to convert specifications into test suites makes test specification less dependent on changes to the specification. If you do use this method, store not only the test cases but also, and more importantly, the criterion that you use to generate the test cases from specification.

Automatically generating test cases from a formal requirements specification typically significantly reduces the turnaround time.

PROBLEM 3: MANUAL TESTING

A test team is spending most of its time running test cases but is executing the tests slowly. It takes as much as a day just to test one new feature of a system, and often the tests fail due to system timeouts. Executing full regression tests has been so expensive that the team avoids doing so whenever possible. Needless to say, the test execution is manual.

When applying manual testing, the team is frequently unsure about the repeatability of failing test cases. The turnaround time for releasing a new revision of software after it has been sufficiently tested is too long and seems to be ever increasing. The test team is busy doing manual testing instead of producing new test specifications, or updating old ones to match a new or changed requirement. Consequently, test documentation is lagging behind.

Manually testing a complex system with real-time requirements is, at best, unreliable, and at worst, impossible. Fortunately, significant research and world standardization has occurred in the last 10 years that makes possible reliable automated testing of this type of system. The ISO (International Standards Organization) has arrived at a standard (9646) on formalized and automated testing of communicating and real-time software, called the TTCN (Tree and Tabular Combined Notation).

FOR SOME APPLICATIONS THE BENEFITS OF STANDARDIZED TEST SUITES PROVIDE AN EXCELLENT STARTING POINT FOR FURTHER TESTING.

Make sure from the start that you can test your design using automated methods. It must have accessible interfaces and an architecture that permits possible overhead from test-related components. Draw benefits from the standard; tell your clients that you have tested your product according to the ISO 9646 standard. For some applications, primarily in the telecommunications domain, the benefits of standardized test suites provide an excellent starting point for further testing.

PROBLEM 4: THE UNCERTAINTY PRINCIPLE

Uncertainty introduced by the testing method is virtually unavoidable. The following C code example is a classic, although oversimplified, illustration of the problem:

```
if ( x != 0 )
y = x;
else
assert( 0 );
x+=2;
```

The above code behaves differently if compiled in debug mode than it behaves if you compile it in release mode. Because it is in C, the "assert" macro expands to nothing, and the x+=2 statement takes the place of the assertion after the "else" statement. Other, often more difficult, examples include optimizers being too aggressive when test code in a system otherwise affects its size, speed, or behavior, and when system or component simulators produce incorrect results. A problem that occurs during testing is not repeatable when running a non-test session, and a feature that worked just fine during testing fails in real life.

Under test, the system behaves differently than in its release version. This situation typically occurs when the test environment affects the behavior of the test subject. Examples of factors introducing this behavior are conditionally compiled testing and debugging code, as well as special interfaces for testing.

Minimize the differences within the system under test and the software in its final form. In many cases, you can eradicate the problem, although black-box testing through the interfaces that the software under test normally provides often minimizes the need for specialized testing or debugging versions of the software. Another approach to solving this problem is to deliver the system with the debug code still in it.

PROBLEM 5: SELECTING THE RIGHT TESTS

You work with a group that tests software for maintaining a communications network. The software you are testing comprises statistics-gathering nodes, an analysis module, and a user interface for connecting to other components. The system is distributed, and the user interface runs on a PC. The test group has started constructing test cases using an automated tool to test the system through the user interface.

After about four months of work, you present the first results—a set of test scripts that invoke the UI tool from Windows and open and close a few dialog boxes. The tests are built to search the screen for various icons, check the layout of the dialogs, and check for specific strings in the menus. After running all of these tests, hardly one line of code in the analysis module was tested. According to the architects of the system, 90% of the system complexity is in the analysis module. In this case, it is clear that the test method and test-case selection should be different.

The testing may not cover some of the important aspects of the application or system, for instance, by selecting only the expected interactions when testing a fault-tolerant application or testing only some subset of the required functions. Most of the time, it is more important to concentrate on determining the technical correctness of a system. You need to prioritize all requirements and the functions that are likely to contain critical problems. Most of today's applications are too complex to test in every possible way, through all possible paths and states. Prioritizing the paths and scenarios that you test first is a valuable, timesaving lesson for a test team, particularly when resources are limited.

MOST OF TODAY'S APPLICATIONS ARE TOO COMPLEX TO TEST IN EVERY POSSIBLE WAY; PRIORITIZING THE PATHS AND SCENARIOS YOU TEST FIRST IS VALUABLE.

BE PREPARED

Testing activities can fail in many ways, however, you can prevent most problems with the following practices:

- form a well-qualified test team with the appropriate means for performing the tests at hand,
- make testing an integral part of software development,
- employ change-management processes,
- ensure requirement traceability to and from tests,
- automate test specification and execution, and
- design for testability.

Given the complexity of current and anticipated software and communications systems, you should expect software testing to become even more complicated. Consequently, even more potent tools and methodologies will emerge over time. Manual testing is becoming a less viable alternative, and integration with the overall design processes and tools will prove necessary to keep pace in testing these complex current and future systems.□

AUTHOR'S BIOGRAPHY

Lars Mats is a design manager at Telelogic Technologies (Malmö, Sweden). He has worked as a developer, trainer, and consultant on tools and methodologies for major telecommunications companies in Europe, the United States, and Australia. He holds an MS from Uppsala University (Sweden).