Editor: Robert L. Glass ■ Computing Trends ■ rlglass@acm.org

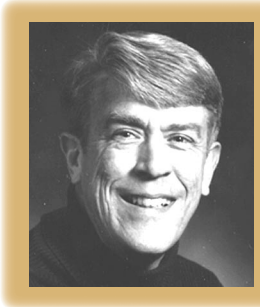
A Classification System for Testing, Part 2

Robert L. Glass

… in which I present a simple but often-overlooked view of testing across the software life cycle.

My last column described a software-testing classification system included in a book I wrote (*Building Quality Software*) over 17 years ago. I bragged so much about that system that I overran my column's expected length and committed myself to a Part 2. So, here it is.

In that previous column, I explained why you might care about testing classification (boring!) as described in that ancient book (double boring!) I argued that it offered insight on a topic of major importance that no one before or since has offered (see why I called it bragging?).



I also described the vertical axis of a 2D testing classification matrix. The axis listed four goal-driven approaches:

1. *Requirements-driven testing* determines whether the requirements for the software artifact under test have been satisfied. It's the most essential level of testing.
2. *Structure-driven testing* determines whether the as-built software product's pieces function as they should. This level is also essential.
3. *Statistics-driven testing* determines how well the software product satisfies the user's or customer's needs for trustworthiness. It's for customers who don't rely on requirements- and structure-driven findings, and it involves test cases drawn from typical usage profiles.
4. *Risk-driven testing* determines whether the software product is vulnerable to its most important risks. It's vital in high-reliability settings.

Here, I describe how those goal-driven approaches play across the software life cycle.

Testing Phases

Software testing has three phases:

- *Unit testing* involves the lowest-level components of the evolving software product. Typically, the unit being tested is a module or collection of modules.
- *Integration testing* involves the intermediate level of software production. Here, the artifacts under test are integrated clusters of units, often a partial or complete set of software modules.
- *System testing* involves the final level of software production. Many software systems fit into a larger system of some sort—for example, an airframe with software parts or a payroll system in an integrated business package.

Testing at each of these phases is very different. In unit testing, the software product is far from complete; you usually must build a framework so that you can test the unit. In integration testing, the glued-together software units provide the necessary framework, and you can test the software as if it's a finished product. In system testing, you test the entire system—usually something much larger than the software system—to determine whether the software plays satisfactorily with the system's different pieces.

So far, nothing about these phases is surprising. Any software-testing book written since the

Continued on p. 103

 0740-7459/09/$25.00 © 2009 IEEE

Continued from p. 104

beginning of software time (the early 1950s) could have contained this discussion. If that's all I had to say here, you could have happily skipped this column.

Combining the Axes

Things become interesting when you combine the two axes of the testing classification matrix. How do you combine the four goal-driven approaches with the three phase-driven approaches?

Back when I wrote *Building Quality Software*, I was surprised at how difficult it was to merge those notions. After all, most authors of software-testing books thought that after they discussed goals (however they identified them) and phases (which almost everyone agreed on), there was nothing left to say. I didn't realize that another discussion step was missing until I taught a class of my software colleagues at Boeing, who began asking questions I had never thought about.

For example, what does it mean to do requirements-driven testing at the unit-test level? Well, that one was fairly easy. The unit will have some kind of documented requirements, perhaps an informal discussion about what it should do (the formal requirements document or the user manual is unlikely to say anything about the requirements for any particular unit). Those informal requirements form the basis for unit requirements-driven testing.

Also, what does it mean to do structure-driven testing at the integration-test level? (Most discussions of structure-driven testing assume we're dealing with a unit). Here, I concluded, we must treat the constituent modules of the integrated whole to be the structure under test. But this gives a whole new meaning, and requires a whole new approach, to structure-driven testing.

Out of all this thinking, a picture began to emerge of how to classify (and organize) software testing. That final matrix looked like Table 1.

It's a nice summary, I immodestly say, of a topic that's surprisingly complex. I hope you find it as interesting as I did. After all, that would be my only excuse for dredging out an obscure discussion from that ancient book.

Table 1

A matrix for classifying and organizing software testing*

Testing approach	Testing phase		
	Unit testing	Integration testing	System testing
Requirements-driven	100% unit requirements	100% product requirements	100% system requirements
Structure-driven	85% logic paths	100% modules	100% components
Statistics-driven	—	—	90–100% of usage profiles if required
Risk-driven	As required	As required	100% if required

* *These figures represent the degree of each kind of testing that should occur in each phase. Where the figure isn't 100 percent, it indicates the degree of testing that's reasonably practical.*

That first part of this column appeared in *Software*'s focus-on-the-past 25th-anniversary issue. It seems appropriate to include this second part in *Software*'s first issue of its next 25 years. I believe this matrix still has something useful to say about testing in the future.

Robert L. Glass is editor emeritus of Elsevier's *Journal of Systems and Software*, the publisher/editor of the *Software Practitioner* newsletter, and a visiting professor at Griffith University. He likes to tell people that his head is in the academic end of computing but his heart is in its practice. Contact him at rlglass@acm.org; he'd be pleased to hear from you.

IEEE Software (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1314; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscription rates: IEEE Computer Society members get the lowest rate of US$51 per year, which includes printed issues plus online access to all issues published since 1988. Go to www.computer.org/subscribe to order and for more information on other subscription prices. Back issues: $20 for members, $163 for nonmembers (plus shipping and handling).

Postmaster: Send undelivered copies and address changes to *IEEE Software*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08855-1331. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.