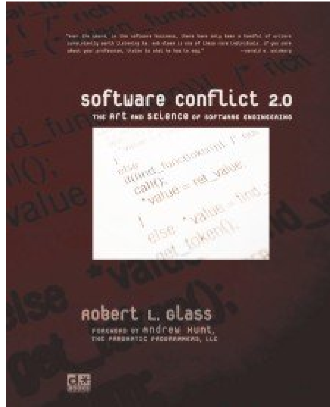
devel oper. *

Book Excerpt



Software Conflict 2.0:
The Art and Science of Software Engineering

by Robert L. Glass

ISBN 0977213307; US $29.99 / UK £22.99; 308 pages

The Many Flavors of Testing

There is evidence that testing is still vitally important to software development, and that it probably always will be. Reviews may be more cost effective, according to recent studies, and proof of correctness (if it ever scales up to larger problems) may be more rigorous, but neither can take the place of taking the software into a near-real environment and trying it out.

Once we realize that we are committed to a future full of testing, it is worth exploring what testing really means. I would assert that there are several flavors of testing, and that all too often when we speak of testing we consider far too few of those flavors.

Here are the flavors I see:

1) First of all, there is goal-driven testing. It is in goal-driven testing that the reason for testing drives the tests to be run. There are roughly four goals for testing:

   a) Requirements-driven testing. Here, enough test cases are constructed to demonstrate that all of the requirements for the product have been tested at least once. Typically, a requirements-test case matrix is constructed to ensure that there is at least one test for every requirement. Tools are now available to support this process; 100 percent requirements-driven testing is essential for all software products.

Reprint or distribute only with written permission from developer. Books.*

b) Structure-driven testing. Here, test cases are constructed to exercise as much of the logical structure of the software as makes sense. Structure-driven testing must supplement (but never replace) requirements-driven testing because all-requirements testing is simply too coarse a level to insure that sufficient tests have been run. "Good" testing usually tests about 60-70 percent of the logic structure of a program; for critical software, closer to 95 percent should be tested. Testedness may be measured by a tool called a test coverage analyzer. Such tools are now available in the marketplace.

c) Statistics-driven testing. Here, enough tests are run to convince a customer or user that adequate testing has been done. The test cases are constructed from a typical usage profile, so that following testing, a statement of the form "the program can be expected to run successfully 96 percent of the time based on normal usage" can be made. Statistics-driven testing should supplement (not replace) requirements-driven and structure-driven testing when customers or users want assurance in terms they can understand that the software is ready for reliable use.

d) Risk-driven testing. Here, enough tests are run to give confidence that the software can pass all worst-case failure scenarios. An analysis of high-risk occurrences is made; the software is then examined to determine where it might contribute to those risks. Extra thorough testing of those portions is then conducted. Risk-driven testing is typically used only for critical software; once again, it should supplement, but not replace, requirements-driven and structure-driven tests.

2) In addition to goal-driven testing, there is phase-driven testing. Phase-driven testing changes in nature as software development proceeds. Typically software must be tested in small component form as well as total system form. In so-called bottom-up testing, we see the three kinds of phase testing discussed later. In top-down testing the software is gradually integrated into a growing whole, so that unit testing is bypassed in favor of continual and expanding integration testing.

a) Unit testing is the process of testing the smallest components in the total system before they are put together to form a software whole.

b) Integration testing is the process of testing the joined units to see if the software plays together as a whole.

c) System testing is the process of testing the integrated software in the context of the total system that it supports.

Reprint or distribute only with written permission from developer. Books.*

It is the manner of intersecting goal-driven testing and phase-driven testing that begins to tax the tester's knowledge and common sense.

For example, do we perform structure-driven testing during unit test, or integration test, or system test? What I would like to present here are thoughts on how to begin to merge these many flavors. Let us take a goal-driven approach first, and work that into the various phases.

Requirements-driven testing means different things in different phases. During unit testing, it means testing those requirements that pertain to the unit under test. During integration testing, it means testing all software requirements at the requirements specification level. During system testing, it means repeating the integration test but in a new setting.

Structure-driven testing also means different things in different phases. During unit testing, it means testing each of the lowest-level structural elements of the software, usually logic branches (for reasons that we do not go into here, testing all branches is more rigorous than testing all statements). During integration testing, it means testing all units. During system testing, it means testing all components of the system, with the integrated software simply being one or more components.

Statistics-driven testing is only meaningful at the integrated-product or the system-test level. The choice of which is application dependent; normally, the system level will be more meaningful to the customer or user.

Risk-driven testing may be conducted at any of the levels, depending on the degree of criticality of the system, but it is probably most meaningful at the system level.

There is one other consideration in testing: Who does the testing? Usually, all unit-level testing is done by the software developer; integration testing is done by a mix of developers and independent testers; and system testing is done by independent testers and perhaps system engineers. Notice, however, that whereas requirements-driven and statistics-driven testing require little knowledge of the internal workings of the software or system under test, structure-driven testing and risk-driven testing require software-intimate knowledge. Therefore, developer involvement in testing may need to be pervasive.

It is popular in some circles to declare testing to be nonrigorous, a concept on its way out. My view of testing is entirely the opposite. Properly done, testing can be rigorous and thorough. It is a matter of knowing how, and doing it. I hope this short discussion may set some testing concepts into proper perspective.

###

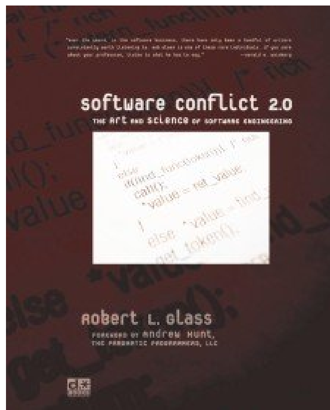
 ©2006 by Robert L. Glass

Reprint or distribute only with written permission from developer. Books.*

About the Author

Robert L. Glass held his first job in computing in 1954. Author of over 25 books, he is one of the true pioneers of the software field. He is the editor and publisher of The Software Practitioner, and also writes regular columns for Communications of the ACM and IEEE Software. In 1995 he was awarded an honorary Ph.D. from Linkoping University of Sweden, and in 1999 he was named a Fellow of the ACM professional society. His unique viewpoint and timeless writings have for decades offered insights to practitioners, managers, professors, entrepreneurs, researchers, and students alike.

About the Book



Title	Software Conflict 2.0: The Art and Science of Software Engineering
Author	Robert L. Glass
Foreword	Andrew Hunt, Pragmatic Programmers, LLC First edition Foreword by Donald J. Reifer, Reifer Consultants Inc.
ISBN	0977213307
Pages	308
Price	$29.95 U.S. / UK£22.99

As loyal Robert Glass readers have come to expect, Software Conflict 2.0 takes up large themes and important questions, never shying away from controversy. Robert Glass has a unique perspective, owing partly to his longevity in the field, partly to his breadth and depth of experience as a practitioner, and partly to his experiences on multiple continents crossing back and forth between the worlds of the university and the professional programming shop.

No matter what unique corner of the software engineering world you call home—be it aerospace or e-commerce—whether you are a researcher, hardcore coder, consultant, or manager, Software Conflict 2.0 tackles questions and conflicts that you will recognize. Bob Glass's wide and deep perspective on the art and science of software engineering will widen and deepen your own perspective.

The first edition of Software Conflict was published circa 1990 and, until now, has been out of print for some time. Why? Mainly because that's the normal pattern for software books: a new book is hot when it hits the streets, but then trends change,

Reprint or distribute only with written permission from developer. Books.*

paradigms shift, and eventually the publisher stops placing orders with the printer. As hundreds of new books are published every year, a real treasure can be buried in the shifting sands.

Sometimes the significance of a software book transcends the endless cycle of trends and revolutions. In fact, some of the great software books continue to be discussed even decades after their original publication. Why do people keep reading these "dated" software engineering books?

Because the insights of these great books are timeless, as valid today as they were yesterday. Because these insights help us become better software professionals, better researchers, better managers. And because the writings of a computing pioneer like Robert L. Glass might just reveal something about where we are today and where we’re headed.

Software Conflict 2.0 features six new essays by Robert Glass and a new Foreword by Andrew Hunt of the Pragmatic Programmers.

Order now at `www.DeveloperDotStar.com` or wherever books are sold.

Reprint or distribute only with written permission from developer. Books.*