# 1 A Self-Assessment Test



Since this book was first published over 30 years ago, software testing has become more difficult and easier than ever.

Software testing is more difficult because of the vast array of programming languages, operating systems, and hardware platforms that have evolved in the intervening decades. And while relatively few people used computers in the 1970s, today virtually no one can complete a day's work without using a computer. Not only do computers exist on your desk, but a "computer," and consequently software, is present in almost every device we use. Just try to think of the devices today that society relies on that are *not* software driven. Sure there are some—hammers and wheelbarrows come to mind—but the vast majority use some form of software to operate. Software is pervasive, which raises the value of testing it. The machines themselves are hundreds of times more powerful, and smaller, than those early devices, and today's concept of "computer" is much broader and more difficult to define. Televisions, telephones, gaming systems, and automobiles all contain computers and computer software, and in some cases can even be considered computers themselves.

Therefore, the software we write today potentially touches millions of people, either enabling them to do their jobs effectively and efficiently, or causing them untold frustration and costing them in the form of lost work or lost business. This is not to say that software is more important today than it was when the first edition of this book was published, but it is safe to say that computers—and the software that drives them—certainly affect more people and more businesses now than ever before.

Software testing is easier, too, in some ways, because the array of software and operating systems is much more sophisticated than in the past, providing intrinsic, well-tested routines that can be incorporated into applications without the need for a programmer to develop them from scratch. Graphical User Interfaces (GUIs), for example, can be built from a development language's libraries, and since they are preprogrammed objects that have been debugged and tested previously, the need for testing them as part of a custom application is much reduced.

And, despite the plethora of software testing tomes available on the market today, many developers seem to have an attitude that is counter to extensive testing. Better development tools, pretested GUIs, and the pressure of tight deadlines in an ever more complex development environment can lead to avoidance of all but the most obvious testing protocols. Whereas low-level impacts of bugs may only inconvenience the end user, the worst impacts can result in large financial loses, or even cause harm to people. The procedures in this book can help designers, developers, and project managers better understand the value of comprehensive testing, and provide guidelines to help them achieve required testing goals.

Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and, conversely, that it does not do anything unintended. Software should be predictable and consistent, presenting no surprises to users. In this book, we will look at many approaches to achieving this goal.

Now, before we start the book, we'd like you to take a short exam. We want you to write a set of test cases—specific sets of data—to test properly a relatively simple program. Create a set of test data for the program—data the program must handle correctly to be considered a successful program. Here's a description of the program:

> **The program reads three integer values from an input dialog. The three values represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles, or equilateral.**

Remember that a scalene triangle is one where no two sides are equal, whereas an isosceles triangle has two equal sides, and an equilateral triangle has three sides of equal length. Moreover, the angles opposite the equal sides in an isosceles triangle also are equal (it also follows that the sides opposite equal angles in a triangle are equal), and all angles in an equilateral triangle are equal.

Evaluate your set of test cases by using it to answer the following questions. Give yourself one point for each yes answer.

1. Do you have a test case that represents a *valid* scalene triangle? (Note that test cases such as 1, 2, 3 and 2, 5, 10 do not warrant a yes answer because a triangle having these dimensions is not valid.)
2. Do you have a test case that represents a valid equilateral triangle?
3. Do you have a test case that represents a valid isosceles triangle? (Note that a test case representing 2, 2, 4 would not count because it is not a valid triangle.)
4. Do you have at least three test cases that represent valid isosceles triangles such that you have tried all three permutations of two equal sides (such as, 3, 3, 4; 3, 4, 3; and 4, 3, 3)?
5. Do you have a test case in which one side has a zero value?
6. Do you have a test case in which one side has a negative value?
7. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is equal to the third? (That is, if the program said that 1, 2, 3 represents a scalene triangle, it would contain a bug.)
8. Do you have at least three test cases in category 7 such that you have tried all three permutations where the length of one side is equal to the sum of the lengths of the other two sides (e.g., 1, 2, 3; 1, 3, 2; and 3, 1, 2)?
9. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third (such as 1, 2, 4 or 12, 15, 30)?
10. Do you have at least three test cases in category 9 such that you have tried all three permutations (e.g., 1, 2, 4; 1, 4, 2; and 4, 1, 2)?
11. Do you have a test case in which all sides are zero (0, 0, 0)?
12. Do you have at least one test case specifying noninteger values (such as 2.5, 3.5, 5.5)?
13. Do you have at least one test case specifying the wrong number of values (two rather than three integers, for example)?
14. For each test case did you specify the expected output from the program in addition to the input values?

Of course, a set of test cases that satisfies these conditions does not guarantee that you will find all possible errors, but since questions 1 through 13 represent errors that actually have occurred in different versions of this program, an adequate test of this program should expose at least these errors.

Now, before you become concerned about your score, consider this: In our experience, highly qualified professional programmers score, on the average, only 7.8 out of a possible 14. If you've done better, congratulations; if not, we're here to help.

The point of the exercise is to illustrate that the testing of even a trivial program such as this is not an easy task. Given this is true, consider the difficulty of testing a 100,000-statement air traffic control system, a compiler, or even a mundane payroll program. Testing also becomes more difficult with the object-oriented languages, such as Java and C++. For example, your test cases for applications built with these languages must expose errors associated with object instantiation and memory management.

It might seem from working with this example that thoroughly testing a complex, real-world program would be impossible. Not so! Although the task can be daunting, adequate program testing is a very necessary—and achievable—part of software development, as you will learn in this book.