Introduction to Software Testing

1 Software Faults

Every day we hear of serious problems caused by software. They have caused space missions to fail[1], aircraft to fall out of the sky[2], banks and commercial enterprises to expose millions of customers' personal data to theft[3], and much, much more. The causes of these problems can be put into two general categories:

- software that does exactly what an *incorrect* "specification" says it should do, and
- software that fails to do exactly what its "specification" states.

Before we go any further, we need to explain the concept of a ***software specification***. For the most part, software is written to do some particular "thing", such as editing photos, sending messages, or guiding an aircraft along a planned route. The "thing" it is supposed to do, however complex it might be, must be clearly defined and unambiguous. For most "real" software, the problem is so complex that this clear and unambiguous description of what it is supposed to do is written down and might be many hundreds of pages long. This written description of what the software is supposed to do under all possible circumstances is called the ***software requirements specification***, or the ***software spec***, or simply the "spec", as software professionals usually call it.

If you are a student learning programming, when you are given an assignment, that assignment's description is a software spec, or is supposed to be. Some teachers may not write down every detail, either because they forgot to, or because certain details do not matter to them, or because the details do matter but they want the students to fill in the missing logic and make the decisions. From this point forward we will assume that the specification is always a written document, and we elaborate on this twofold categorization of problems caused by software.

As noted above, one category are those problems caused by software that does exactly what it is supposed to do, but whose specification is at fault. An example, is when the spec says it should output the sum of values computed when it should have said to output the sum of the squares of those values.

The other category are those problems caused by software that does not do what its specification requires of it. For example the spec says to output the sum of squares of the values, but the software outputs the sum of the absolute values of the values. This does not rule out the possibility that the specification is also incorrect; it is possible that even if the software did what its specification required, a different problem might occur because the specification is wrong anyway. For example, the spec might say to output the sum of values when it should have said to output the sum of the squares of those values, but the software outputs the sum of the absolute values of the values. Even if it did what the spec said, it would be wrong.

When software does not "meet" its spec, we say it has a ***fault***, or a "***bug***". When that software is run and the fault causes the program to behave incorrectly, we say the software ***failed***. Yes, that sounds harsh, but that is the word we use!

A programmer cannot solve problems of the first category - if a spec is wrong, the problem was injected into the software development process before the programmer's job was started. (Sometimes the programmer might read a spec that has such a blatant mistake that it is obvious, but in this case the programmer is really wearing a different hat - that of a spec proofreader.) A programmer must prevent the second kind of problem from occurring, and one means of doing that is by testing.

[1] See the Washington Post, October 1, 1999 for details.
[2] See the article about the Airbus 440 failure in May 2009 at http://www.theguardian.com/technology/2015/may/20/airbus-issues-alert-software-bug-fatal-plane-crash
[3] See, for example, http://www.cbsnews.com/news/gsa-system-showed-ssns-for-183k-contractors.

2 The Mindset of a Software Tester

A programmer's job is to write code that has no faults. But programmers are human and they make mistakes. They write code that does have faults. Therefore, they need to put on a different hat every once in a while and take on the role of the person who will find their mistakes, the ***tester***. The tester is the person who tries to find all of the mistakes in the software before it is given to the users of that software.

The job of the tester is to find all of the mistakes. If the tester does not find mistakes he or she is not doing his or her job. The tester has to assume that mistakes exist but that they are hiding. The hardest mistakes to find are the ones that have been hidden so well that they are only exposed by the most clever of software tests. These are the big payoff in the world of software testers, because these types of software faults are the ones that might not be exposed in the beginning, but they may cause disaster later on, when it is too late. The software tester is doing a better job the more bugs he or she finds.

When you have to test your own code, it is hard to overcome a natural fear of finding your own mistakes. Many people do not test their code thoroughly because they have the wrong attitude - they fear finding mistakes and so devise tests that fail to find them. Your mission is to find your mistakes before anyone else (like your instructor) does!

3 Software Testing Strategies

There has been quite a bit of research about how to test software, including a great deal of theorizing and a great deal of experimentation. There are some pretty sophisticated concepts that underlie many strategies. We will not discuss these concepts now, choosing instead to start with simple, easy to understand, and easy to master strategies. They may not find all bugs, but they will get you thinking.

What is a test? The word "test" is mostly used as a shorthand for a "test case". A ***test case*** has two parts: the input to be given to the program, and the expected output of the program. This is a simplification, as the input includes a complete description of the state of the environment in which the program runs, and the output includes a description of the expected environment after a correct program processes the input. For now we stick to the easy-to-understand concept that a test case has a test input and a test output, and we use the word test interchangeably with the term test case.

A very basic strategy for devising tests is to read the software specification and build a set of tests based *almost* entirely on the information in the spec. When a set of tests is based only on the spec, it is called a ***black-box test set***. This term makes reference to the fact that the program is being treated like a box whose insides are invisible to the tester. The tester does not see the code. The code is a black box.

We say "almost" entirely because the design of the test set relies upon our understanding of human nature and our experience. For example, a specification might have a rule that states that when the value of some variable, say X, is less than 100, the program should output a message A, and when it is not, then the program should output the message B. It seems reasonable to conclude that any set of tests that finds all errors in a potential implementation of this spec should include a test input that outputs message A and one that outputs message B, so two tests seems like enough. The first would force X to be less than 100 and the second would force X to be greater than 100. The astute reader might observe that my description of the second input is incorrect: it should be a test input that forces X to be greater than or equal to 100, because this is the complement of the condition that X is less than 100. In any case, our reasoning leads us to conclude that two test inputs that force X to be 50 and 150 are enough.

But people often make the mistake of using the `<=` operator instead of the `<` operator when they code. Suppose we are testing such a program, which has the expression `X <= 100` instead of `X < 100`. For this program, when $X == 100$, the condition X `<=` 100 is true and message A will be output instead of message B. If the two test cases force X to be 50 and 150 respectively, they will not "expose" this potential error in coding. We need a third test input that forces X to be exactly 100 to make sure that message B is output. The fact that we need this third test stems from our experience with the kinds of mistakes people tend to make.

This discussion about the role that understanding human nature plays in selecting test cases is very important. It is telling you not to forget the way you and others think when you wear the software tester's hat.

The idea used in the example above is so common that it has been given various names. It is one requirement of a strategy called ***boundary-value testing***, and is also called ***fencepost testing***. We will discuss these ideas formally at a later time.

The intent of the previous discussion is to get you thinking a bit about methodical ways to test software and about how information in a program's written specification can be used to devise a set of test cases. In the next section, we will explore by way of example how to devise a set of black box tests that are "pretty good" for certain kinds of programs.