Systematic software test

Peter Sestoft, Department of Mathematics and Physics
Royal Veterinary and Agricultural University, Denmark

English version 1, 1998-07-31

This note introduces techniques for systematic test of software.

Contents

1 Software test

Programs often contain errors (so-called bugs), even though the compiler accepts the program as well-formed: the compiler can detect only errors of form, not of meaning. Many errors and inconveniences in programs are discovered only by accident when the program is being used. However, errors can be found in more systematic and effective ways than by random experimentation. This is the goal of *software test*.

You may think, why don't we just fix errors when they are discovered? After all, what harm can a program do? Consider some effects of software errors:

- In the Gulf war (1991), some Patriot missiles failed to hit incoming Iraqi Scud missiles (which subsequently killed people on the ground). Accumulated rounding errors in the control software's clocks caused large navigation errors.
- Errors in the software controlling the baggage handling system of Denver International Airport delayed its opening by a year (1995), causing losses of around 360 million dollars.
- The first launch of the European Ariane 5 rocket failed (1996), causing losses of hundreds of million dollars. The problem was a buffer overflow in control software taken over from Ariane 4. The software had not been re-tested (to save money).
- Errors in a new train control system deployed in Berlin (1998) caused train cancellations and delays for weeks.
- Errors in poorly designed control software in the Therac-25 radio-therapy equipment (1987) exposed several cancer patients to heavy doses of radiation, killing some.

1.1 Syntax errors, type errors, and semantic errors

A Java program may contain several kinds of errors:

- *syntax errors*: the program may be syntactically ill-formed (e.g. contain `while x {}`, where there are no parentheses around `x`), so that strictly speaking it is not a Java program at all;
- *type errors*: the program may be syntactically well-formed, but attempt to access non-existing fields of objects, or apply operators to the wrong type of arguments (as in `true * 2`, which attempts to multiply a logical value by a number);
- *semantic errors*: the program may be syntactically well-formed and type-correct, but compute the wrong answer anyway.

Errors of the two former kinds are relatively trivial: the Java compiler `javac` will automatically discover them and tell us about them. Semantic errors (the third kind) are harder to deal with: they cannot be found automatically, and it is our own responsibility to find them, or even better, to convince ourselves that there are none.

In these notes we shall assume that all errors discovered by the compiler have been fixed. We present simple systematic techniques for finding semantic errors and thereby making it plausible that the program works as intended (when we can find no more errors).

1.2 Structural test and functional test

Two important techniques for software testing are *structural test* and *functional test.*

Structural test, sometimes called white-box test or internal test, focuses on the *text* of the program. The tester constructs a test suite (a collection of inputs and corresponding expected outputs) which demonstrates that all branches of the program's choice and loop constructs — `if`, `while`, `switch`, and so on — can be executed. The test suite is said to *cover* the statements and branches of the program.

Functional test, sometimes called black-box or external test, focuses on the *problem* that the program is supposed to solve. The tester constructs a test data set (inputs and corresponding expected outputs) which includes 'typical' as well as 'extreme' input data. In particular, one must include inputs that are described as exceptional or erroneous in the problem description.

Structural and functional test are complementary. Structural test does not focus on the the problem area, and therefore cannot discover whether some subproblems are left unsolved by the program, whereas functional test can. Functional test does not focus on the program text, and therefore cannot discover that some parts of the program are completely useless or have an illogical structure, whereas structural test can.

Software testing can never *prove* that a program contains no errors, but it can strengthen one's faith in the program. Systematic software test is necessary if the program will be used by others, or if one plans to base scientific conclusions on the result of the program.

2 Structural test

The goal of structural test is to make sure that all parts of the program have been executed. The test suite must include enough input data sets to make sure that all methods have been called, that both the true and false branches have been executed in `if` statements, that every loop has been executed zero, one, and more times, that all branches of every `switch` statement have been executed, and so on. For every input data set, the expected output must be specified also. Then the program is run with all the input data sets, and the actual outputs are compared to the expected outputs.

Structural test cannot demonstrate that the program works in all cases, but it is a surprisingly efficient (fast), effective (thorough), and systematic way to discover errors in the program. In particular it is a good way to find errors in programs with a (too) complicated logic, and to find variables which are initialized with the wrong values.

2.1 Example 1 of structural test

The program below receives some integers as argument, and is expected to print out the smallest and the greatest of these numbers. We shall see how one performs a structural test of the program. (We should warn that the program is actually erroneous; is this obvious?)

```
public static void main ( String[] args )
{
  int mi, ma;
  if (args.length == 0)                               /* 1 */
    System.out.println("No numbers");
  else
    {
      mi = ma = Integer.parseInt(args[0]);
      for (int i = 1; i < args.length; i++)           /* 2 */
        {
          int obs = Integer.parseInt(args[i]);
          if (obs > ma) ma = obs;                     /* 3 */
          else if (mi < obs) mi = obs;                /* 4 */
        }
      System.out.println("Minimum = " + mi + "; maximum = " + ma);
    }
}
```

The choice statements are numbered 1–4 in the margin. Number 2 is the `for` statement. First we construct a table which shows, for every choice statement and every possible outcome, which input data set covers that choice and outcome:

Choice	Input data set	Input property
1 true	A	No numbers
1 false	B	At least one number
2 zero times	B	Exactly one number
2 once	C	Exactly two numbers
2 more than once	E	At least three numbers
3 true	C	Number $>$ current maximum
3 false	D	Number $\leq$ current maximum
4 true	E number 3	Number $\leq$ current maximum and $>$ current minimum
4 false	E number 2	Number $\leq$ current maximum and $\leq$ current minimum

While constructing the above table, we construct also a table of the input data sets:

Input data set	Contents	Expected output
A	(no numbers)	'No numbers'
B	17	17 17
C	27 29	27 29
D	39 37	37 39
E	49 47 48	47 49

When running the above program on the input data sets, one sees that the outputs are wrong — they disagree with the expected outputs — for input data sets D and E. Now one may run the program manually on e.g. input data set D, which will lead one to discover that the condition in the program's choice 4 is wrong. When we receive a number which is less than the current minimum, then the variable `mi` is not updated correctly. The statement should be:

```
else if (obs < mi) mi = obs;              /* 4a */
```

After correcting the program, it may be necessary to reconstruct the structural test. It may be very time consuming to go through several rounds of modification and re-testing, so it pays off to make the program correct from the outset! In the present case it suffices to change the comments in the last two lines of the table of choices and outcomes, because all we did was to invert the condition in choice 4:

Choice	Input data set	Input property
1 true	A	No numbers
1 false	B	At least one number
2 zero times	B	Exactly one number
2 once	C	Exactly two numbers
2 more than once	E	At least three numbers
3 true	C	Number $>$ current maximum
3 false	D	Number $\leq$ current maximum
4a true	E number 2	Number $\leq$ current maximum and $<$ current minimum
4a false	E number 3	Number $\leq$ current maximum and $\geq$ current minimum

The input data sets remain the same. The corrected program produced the expected output for all input data sets A–E.

2.2 Example 2 of structural test

The program below receives some numbers as input, and is expected to print out the two smallest of these numbers, or the smallest, in case there is only one. (Is this problem description unambiguous?). This program, too, is erroneous; can you find the problem?

```
public static void main ( String[] args )
{
  int mi1 = 0, mi2 = 0;
  if (args.length == 0)                                 /* 1 */
    System.out.println("No numbers");
  else
    {
      mi1 = Integer.parseInt(args[0]);
      if (args.length == 1)                             /* 2 */
        System.out.println("Smallest = " + mi1);
      else
        {
          int obs = Integer.parseInt(args[1]);
          if (obs < mi1)                                /* 3 */
            { mi2 = mi1; mi1 = obs; }
          for (int i = 2; i < args.length; i++)         /* 4 */
            {
              obs = Integer.parseInt(args[i]);
              if (obs < mi1)                            /* 5 */
                { mi2 = mi1; mi1 = obs; }
              else if (obs < mi2)                       /* 6 */
                mi2 = obs;
            }
          System.out.println("The two smallest are " + mi1 + " and " + mi2);
        }
    }
}
```

As before we tabulate the program's choices 1–6 and their possible outcomes:

Choice	Input data set	Input property
1 true	A	No numbers
1 false	B	At least one number
2 true	B	Exactly one number
2 false	C	At least two numbers
3 false	C	Second number $\geq$ first number
3 true	D	Second number $<$ first number
4 zero time	D	Exactly two numbers
4 once	E	Exactly three numbers
4 more than once	H	At least four numbers
5 true	E	Third number $<$ current minimum
5 false	F	Third number $\geq$ current minimum
6 true	F	Third number $\geq$ current minimum and $<$ second least
6 false	G	Third number $\geq$ current minimum and $\geq$ second least

The corresponding input data sets might be:

Input data set	Contents	Expected output
A	(no numbers)	No numbers
B	17	17
C	27 29	27 29
D	39 37	37 39
E	49 48 47	47 48
F	59 57 58	57 58
G	67 68 69	67 68
H	77 78 79 76	76 77

Running the program with these test data, it turns out that data set C produces wrong results: 27 and 0. Looking at the program text, we see that this is because variable `mi2` retains its initial value, namely, 0. The program must be fixed by inserting an assignment `mi2 = obs` just before the line labelled 3. We do not need to change the structural test, because no choice statements were added or changed. The corrected program produces the expected output for all input data sets A–H.

Note that if the variable declaration had not been initialized with `mi2 = 0`, the Java compiler would have complained that `mi2` might be used before its first assignment. If so, the error would have been detected even without the test.

This is *not* the case in many other current programming languages (e.g. C, C++, Pascal), where one may well use an uninitialized variable — its value is just whatever happens to be at that location in the computer's memory. The error may even go undetected by the test, when the value of `mi2` equals the expected answer by accident. This is more likely than it may sound, if one runs the same (C, C++, Pascal) program on several input data sets, and the same data values are used in several data sets. Therefore it is a good idea to choose different data values in the data sets, as done above.

2.3 Summary, structural test

Program statements should be tested as follows:

Statement	Cases to test
`if`	Condition false and true
`for`	Zero, one, and more than one iterations
`while`	Zero, one, and more than one iterations
`do-while`	One, and more than one iterations
`switch`	Every branch must be executed

Composite logical expressions such as `(x != 0) && (1000/x > y)`, must be tested for all possible combinations of the truth values of the terms. That is,

`(x != 0)`	`&&`	`(1000/x > y)`
false		
true		false
true		true

Note that the second term in a conjunction will be computed only if the first term is true (in Java, C, and C++). This is important if the condition is e.g. `(x != 0) && (1000/x > y)`, where the second term cannot be computed if the first one is false, that is, if `x == 0`. Therefore it makes no sense to require that the combinations (false, false) and (false, true) be tested.

In a disjunction `(x == 0) || (1000/x > y)` it holds, dually, that the second term is computed only if the first one is false. If `(x == 0)` is true, then the second term makes no sense. Therefore, in this case too there are only three possible combinations:

`(x == 0)`	`||`	`(1000/x > y)`
true		
false		false
false		true

Methods The test must makes sure that all methods have been executed. For recursive methods one must test also the case where the method calls itself.

The test data sets are presented conveniently by two tables, as demonstrated in this section. One table presents, for each statement, what data sets are used, and which property of the input is demonstrated by the test. The other table presents the actual contents of the data sets, and the corresponding expected output.

3 Functional test

The goal of functional test is to make sure that the program solves the problem it is supposed to solve; to make sure that it works. Thus one must have a fairly precise idea of the *problem* that the program must solve, but in principle one does not need the program text when designing a functional test. Test data sets (with corresponding expected outputs) must be created to cover 'typical' as well as 'extreme' input values, and also inputs that are described as exceptional cases in the problem statement. Examples:

- In a program to compute the sum of a sequence of numbers, the empty sequence will be an extreme, but legal, input (with sum 0).
- In a program to compute the average of a sequence of numbers, the empty sequence will be an extreme, and illegal, input. The program should give an error message for this input, as one cannot compute the average of no numbers.

One should avoid creating a large collection of input data sets, 'just in case'. Instead, one must carefully consider what inputs might reveal problems in the program, and use exactly those. When preparing a functional test, the task is to find errors in the program; thus destructive thinking is required. As we shall see below, this is just as demanding as programming, that is, as constructive thinking.

3.1 Example 1 of functional test

Problem: Given a (possibly empty) sequence of numbers, find the smallest and the greatest of these numbers.

This is the same problem as in Section 2.1, but now the point of departure is the above problem statement, not any particular program which claims to solve the problem.

First we consider the problem statement. We note that an empty sequence does not contain a smallest or greatest number. Presumably, the program must give an error message if presented with an empty sequence of numbers.

The functional test might consist of the following input data sets: An empty sequence (A). A non-empty sequence can have one element (B), or two or more elements. In a sequence with two elements, the elements can be equal (C1), or different, the smallest one first (C2) or the greatest one first (C3). If there are more than two elements, they may appear in increasing order (D1), decreasing order (D2), with the greatest element in the middle (D3), or with the smallest element in the middle (D4). All in all we have these cases:

Input data set	Input property
A	No numbers
B	One number
C1	Two numbers, equal
C2	Two numbers, increasing
C3	Two numbers, decreasing
D1	Three numbers, increasing
D2	Three numbers, decreasing
D3	Three numbers, greatest in the middle
D4	Three numbers, smallest in the middle

The choice of these input data sets is not arbitrary. It is influenced by our own ideas about how the problem might be solved by a program, and in particular *how it might be solved the wrong way.* For instance, the programmer might have forgotten that the sequence could be empty, or that the smallest number equals the greatest number if there is only one number, etc.

The choice of input data sets may be criticized. For instance, it is not obvious that data set C1 is needed. Could the problem really be solved (wrongly) in a way that would be discovered by C1, but not by any of the other input data sets?

The data sets C2 and C3 check that the program does not just answer by returning the first (or last) number from the input sequence; this is a relevant check. The data sets D3 and D4 check that the program does not just compare that first and the last number; it is less clear that this is relevant.

Input data set	Contents	Expected output
A	(no numbers)	Error message
B	17	`17 17`
C1	27 27	`27 27`
C2	35 36	`35 36`
C3	46 45	`45 46`
D1	53 55 57	`53 57`
D2	67 65 63	`63 67`
D3	73 77 75	`73 77`
D4	89 83 85	`83 89`

3.2 Example 2 of functional test

Problem: Given a (possibly empty) sequence of numbers, find the greatest difference between two consecutive numbers.

We shall design a functional test for this problem. First we note that if there is only zero or one number, then there are no two consecutive numbers, and the greatest difference cannot be computed. Presumably, an error message must be given in this case. Furthermore, it is unclear whether the 'difference' is signed (possibly negative) or absolute (always non-negative). Here we assume that only the absolute difference should be taken into account, so that the difference between 23 and 29 is the same as that between 29 and 23.

This gives rise to at least the following input data sets: no numbers (A), exactly one number (B), exactly two numbers. Two numbers may be equal (C1), or different, in increasing order (C2) or decreasing order (C3). When there are three numbers, the *difference* may be increasing (D1) or decreasing (D2). That is:

Input data set	Input property
A	No numbers
B	One number
C1	Two numbers, equal
C2	Two numbers, increasing
C3	Two numbers, decreasing
D1	Three numbers, increasing difference
D2	Three numbers, decreasing difference

The data sets and their expected outputs might be:

Input data set	Contents	Expected outputs
A	(no numbers)	Error message
B	17	Error message
C1	27 27	`0`
C2	36 37	`1`
C3	48 46	`2`
D1	57 56 59	`3`
D2	69 65 67	`4`

One might consider whether there should be more variants of each of D1 and D2, in which the three numbers would appear in increasing order `(56,57,59)`, or decreasing `(59,58,56)`, or increasing and then decreasing `(56,57,55)`, or decreasing and then increasing `(56,57,59)`. Although these data sets might reveal errors that the above data sets would not, they do appear more contrived. However, this shows that functional testing may be carried on indefinitely: you will never be *sure* that all possible errors have been detected.

3.3 Example 3 of functional test

Problem: Given a day of the month *day* and a month *mth*, decide whether they determine a legal date in a non-leap year. For instance, 31/12 (the 31st day of the 12th month) and 31/8 are both legal, whereas 29/2 and 1/13 are not. The day and month are given as integers, and the program must respond with `Legal` or `Illegal`.

To simplify the test suite, one may assume that if the program classifies e.g. 1/4 and 30/4 as legal dates, then it will consider 17/4 and 29/4 legal, too. Correspondingly, one may assume that if the program classifies 31/4 as illegal, then also 32/4, 33/4, and so on. There is no guarantee that the these assumptions actually hold; the program may be written in a contorted and silly way. Assumptions such as these should be written down along with the test suite.

Under those assumptions one may test only ‘extreme’ cases, such as 0/4, 1/4, 30/4, and 31/4, for which the expected outputs are `Illegal`, `Legal`, `Legal`, and `Illegal`.

Input data set	Contents	Expected output
A	0 1	Illegal
	1 0	Illegal
	1 1	Legal
	31 1	Legal
	32 1	Illegal
	28 2	Legal
	29 2	Illegal
	31 3	Legal
	32 3	Illegal
	30 4	Legal
	31 4	Illegal
	31 5	Legal
	32 5	Illegal
	30 6	Legal
	31 6	Illegal
	31 7	Legal
	32 7	Illegal
	31 8	Legal
	32 8	Illegal
	30 9	Legal
	31 9	Illegal
	31 10	Legal
	32 10	Illegal
	30 11	Legal
	31 11	Illegal
	31 12	Legal
	32 12	Illegal
	1 13	Illegal

It is clear that the functional test becomes rather large and cumbersome. In fact it is just as long as a program that solves the problem! To reduce the number of data sets, one might consider just some *extreme* values, such as 0/1, 1/0, 1/1, 31/12 and 32/12; some *exceptional* values around February, such as 28/2, 29/2 and 1/3, and a few *typical* cases, such as 30/4, 31/4, 31/8 and 32/8. But that would weaken the test a little: it would not discover whether the program mistakenly believes that June (not July) has 31 days.

4 Practical hints about testing

- Avoid test cases where the expected output is zero. In Java variables with class scope (not inside methods) automatically get initialized to 0. The actual output may therefore equal the expected output by coincidence.
- In languages such as C, C++, Pascal, where variables are not initialized automatically, testing will not necessarily reveal uninitialized variables. The accidental value of an uninitialized variable may happen to equal the expected output. This is not unlikely, if one uses the same input data in several test cases. Therefore, choose different input data in different test cases (as done in the preceding sections).
- Automate the test, if possible. Then it can conveniently be rerun whenever the program has been modified.
- When testing programs that have graphical user interfaces (mouse, menus, etc.) one must describe carefully step by step what actions (menu choices, mouse clicks, etc.) the user must perform, and what the program's expected reactions are. Clearly, it is cumbersome (and expensive) to carry out such manual tests, so professional software houses use various tools to simulate user actions.

5 Test in perspective

- Testing can never prove that a program has no errors, but it can considerably improve the confidence one has in its results.
- Often it is easier to design a structural test suite than a functional one, because one can proceed systematically on the basis of the program text. Functional test requires more guesswork (about the possible workings of the program), but can make sure that the program does what is required by the problem statement.
- It is a good idea to design a functional test at the same time you write the program. This reveals unclarities and subtle points in the problem statement, so that you can take them into account while writing the program — instead of having to fix the program later.
- From the tester's point of view, a test is successful if it *does* find errors in the program; in this case it was clearly not a waste of time to do the test. From the programmer's point of view the opposite holds: hopefully the test will *not* find errors in the program. When the tester and the programmer are one and the same person, then there is a (psychological) conflict: one does not want to admit to making mistakes, neither when programming nor when designing test suites.
- It is a useful exercise to design a test suite for a program written by someone else. This is a kind of game: the goal of the programmer is to write a program that contains no errors; the goal of the tester is to find the errors in the program anyway.
- It takes much time to design a test suite. One learns to avoid needless choice statements when programming, because this reduces the number of test cases in the structural test. It also leads to simpler programs that usually are more general and easier to understand.[1]
- The effort spent on testing should be correlated with the consequences of possible program errors. A program used just once for computing one's taxes need no testing. However, a program *must* be tested if errors could affect the safety of people or animals, or could cause considerable economic losses. If scientific conclusions will be drawn from the outputs of a program, then it must be tested too.

[1]The absence of choice statements is no guarantee that the program is easily understandable, though. See Exercises 10 and 11.

6 Exercises

1. Problem: Given a sequence of integers, find their average.
   Design a functional test for this problem.
2. Write a program to solve the problem from Exercise 1. The program should take the input from the command line. Run the functional test.
3. Design a structural test for the program written in Exercise 2, and run it.
4. Problem: Given a sequence of numbers, decide whether they are sorted in increasing order. For instance, 17 18 18 22 is sorted, but 17 18 19 18 is not. The result must be `Sorted` or `Not sorted`.
   Design a functional test for this problem.
5. Write a program that solves the problem from Exercise 4. Run the functional test.
6. Design a structural test for the program written in Exercise 5, and run it.
7. Write a program to decide whether a given (day, month) pair in a non-leap year is legal, as discussed in Section 3.3. Run your program with the functional test given there.
8. Design a structural test for the program written in Exercise 7. Run it.
9. Problem: Given a day of the month *day* and a month *mth*, compute the number of the day in a non-leap year. For instance, 1/1 is number 1, 1/2 is number 32, 1/3 is number 60, and 31/12 is number 365. This is useful for computing the distance between two dates, e.g. from sowing to harvest. The date and month can be assumed legal for a non-leap year.
   Design a functional test for this problem.
10. We claim that this Java method solves the problem from Exercise 9.

    ```
    static int dayno(int day, int mth)
    {
      int m = (mth+9)%12;
      return (m/5*153+m%5*30+(m%5+1)/2+59)%365+day;
    }
    ```

    Test this method with the functional test designed above.
11. Design a structural test for the method shown in Exercise 10. This appears trivial and useless, since there are no choice statements in the program at all. Instead one may consider jumps (discontinuities) in the processing of data. In particular, integer division (`/`) and remainder (`%`) produce jumps of this sort. For $mth < 3$ we have $m = (mth + 9)\ mod\ 12 = mth + 9$, and for $mth \geq 3$ we have $m = (mth + 9)\ mod\ 12 = mth - 3$. Thus there is a kind of hidden choice when going from $mth = 2$ to $mth = 3$. Correspondingly for `m / 5` and `(m % 5 + 1) / 2`. This can be used for choosing test cases for structural test. Do that.