Systematic software testing

Peter Sestoft
IT University of Copenhagen, Denmark[1]
Version 2, 2008-02-25

This note introduces techniques for systematic functionality testing of software.

Contents

1 Why software testing?

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 testing*.

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 1991 Gulf war, some Patriot missiles failed to hit incoming Iraqi Scud missiles, which therefore 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 the entire airport's opening by a year (1994–1995), causing losses of around 360 million dollars. Since September 2005 the computer-controlled baggage system has not been used; manual baggage handling saves one million dollars a month.
- 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.

[1]Original 1998 version written for the Royal Veterinary and Agricultural University, Denmark.

- 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.

A large number of other software-related problems and risks have been recorded by the RISKS digest since 1985, see the archive at http://catless.ncl.ac.uk/risks.

1.1 Syntax errors, semantic errors, and logic errors

A program in Java, or C# or any other language, 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;
- *semantic errors*: the program may be syntactically well-formed, but attempt to access non-existing local variables or non-existing fields of an object, or apply operators to the wrong type of arguments (as in `true * 2`, which attempts to multiply a logical value by a number);
- *logical 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. Logical 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 Quality assurance and different kinds of testing

Testing fits into the more general context of software quality assurance; but what is software quality? ISO Standard 9126 (2001) distinguishes six quality characteristics of software:

- *functionality*: does this software do what it is supposed to do; does it work as intended?
- *usability*: is this software easy to learn and convenient to use?
- *efficiency*: how much time, memory, and network bandwidth does this software consume?
- *reliability*: how well does this software deal with wrong inputs, external problems such as network failures, and so on?
- *maintainability*: how easy is it to find and fix errors in this software?
- *portability*: how easy is it to adapt this software to changes in its operating environment, and how easy is it to add new functionality?

The present note is concerned only with *functionality testing*, but note that *usability testing* and *performance testing* address quality characteristics number two and three. Reliability can be addressed by so-called stress testing, whereas maintainability and portability are rarely systematically tested.

1.3 Debugging versus functionality testing

The purpose of testing is very different from that of *debugging.* It is tempting to confuse the two, especially if one mistakenly believes that the purpose of debugging is to remove the last bug from the program. In reality, debugging rarely achieves this.

The real purpose of debugging is *diagnosis.* After we have observed that the program does not work as intended, we debug it to answer the question: why doesn't this program work? When we have found out, we modify the program to (hopefully) work as intended.

By contrast, the purpose of functionality testing is to strengthen our belief that the program works as intended. To do this, we *systematically* try to show that it does *not* work. If our best efforts fail to show that the program does not work, then we have strengthened our belief that it *does* work.

Using systematic functionality testing we might find some cases where the program does not work. Then we use debugging to find out why. Then we fix the problem. And then we test again to make sure we fixed the problem without introducing new ones.

1.4 Profiling versus performance testing

The distinction between functionality testing and debugging has a parallel in the distinction between *performance testing* and *profiling.* Namely, the purpose of profiling is *diagnosis.* After we have observed that the program is too slow or uses too much memory, we use profiling to answer the question: why is this program so slow, why does it use so much memory? When we have found out, we modify the program to (hopefully) use less time and memory.

By contrast, the purpose of performance testing is to strengthen our belief that the program is efficient enough. To do this, we *systematically* measure how much time and memory it uses on different kinds and sizes of inputs. If the measurements show that it is efficient enough for those inputs, then we have strengthened our belief that the program is efficient enough for all relevant inputs.

Using systematic performance testing we might find some cases where the program is too slow. Then we use profiling to find out why. Then we fix the problem. And then we test again to make sure we fixed the problem without introducing new ones.

Schematically, we have:

Purpose \ Quality	Functionality	Efficiency
Diagnosis	Debugging	Profiling
Quality assurance	Functionality testing	Performance testing

1.5 White-box testing versus black-box testing

Two important techniques for functionality testing are *white-box testing* and *black-box testing.*

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

Black-box testing, sometimes called external testing, focuses on the *problem* that the program is supposed to solve; or more precisely, the *problem statement* or *specification* for the program. The tester constructs a test data set (inputs and corresponding expected outputs) that 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.

White-box testing and black-box testing are complementary approaches to test case generation. White-box testing does not focus on the problem area, and therefore may not discover that some subproblem is left unsolved by the program, whereas black-box testing should. Black-box testing does not focus on the program text, and therefore may not discover that some parts of the program are completely useless or have an illogical structure, whereas white-box testing should.

Software testing can never *prove* that a program contains no errors, but it can strengthen one's faith in the program. Systematic software testing is necessary if the program will be used by others, if the welfare of humans or animals depends on it (so-called safety-critical software), or if one wants to base scientific conclusions on the program's results.

1.6 Test coverage

Given that we cannot make a perfect test suite, how do we know when we have a reasonably good one? A standard measure of a test suite's comprehensiveness is *coverage.* Here are some notions of coverage, in increasing order of strictness:

- *method coverage*: does the test suite make sure that every method (including function, procedure, constructor, property, indexer, action listener) gets executed at least once?
- *statement coverage*: does the test suite make sure that every statement of every method gets executed at least once?
- *branch coverage*: does the test suite make sure that every transfer of control gets executed at least once?
- *path coverage*: does the test suite make sure that every execution path through the program gets executed at least once?

Method coverage is the minimum one should expect from a test suite; in principle we know nothing at all about a method that has not been executed by the test suite.

Statement coverage is achieved by the white-box technique described in Section 2, and is often the best coverage one can achieve in practice.

Branch coverage is more demanding, especially in relation to virtual method calls (so-called virtual dispatch) and exception throwing. Namely, consider a single method call statement `a.m()` where expression `a` has type `A`, and class `A` has many subclasses `A1`, `A2` and so on, that *override* method `m()`. Then to achieve branch coverage, the test suite must make sure that `a.m()` gets executed for `a` being an object classs `A1`, an object of class `A2`, and so on. Similarly, there is a transfer of control from an exception-throwing statement `throw exn` to the corresponding exception handler, if any, so to achieve branch coverage, the test suite must make sure that each such statement gets executed in the context of every relevant exception handler.

Path coverage is usually impossible to achieve in practice, because any program that contains a loop will usually have an infinite number of possible execution paths.

2 White-box testing

The goal of white-box testing is to make sure that all parts of the program have been executed, for some notion of part, as described in Section 1.6 on test coverage. The approach described in this section gives statement coverage. The resulting test suite includes 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.

White-box testing 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 complicated logic, and to find variables that are initialized with the wrong values.

2.1 Example 1 of white-box testing

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 white-box test of the program. (Be forewarned 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 that shows, for every choice statement and every possible outcome, which input data set covers that choice and outcome:

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

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

Input data set	Input contents	Expected output	Actual output
A	(no numbers)	No numbers	*No numbers*
B	17	17 17	*17 17*
C	27 29	27 29	*27 29*
D	39 37	37 39	*39 39*
E	49 47 48	47 49	*49 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 white-box 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 property	Input data set
1 true	No numbers	A
1 false	At least one number	B
2 zero times	Exactly one number	B
2 once	Exactly two numbers	C
2 more than once	At least three numbers	E
3 true	Number > current maximum	C
3 false	Number ≤ current maximum	D
4a true	Number ≤ current maximum and < current minimum	E, 2nd number
4a false	Number ≤ current maximum and ≥ current minimum	E, 3rd number

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 white-box testing

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

The corresponding input data sets might be:

Input data set	Contents	Expected output	Actual output
A	(no numbers)	No numbers	*No numbers*
B	17	17	*17*
C	27 29	27 29	*27 0*
D	39 37	37 39	*37 39*
E	49 48 47	47 48	*47 48*
F	59 57 58	57 58	*57 58*
G	67 68 69	67 68	*67 0*
H	77 78 79 76	76 77	*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 white-box 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 testing.

This is *not* the case in many other current programming languages (e.g. C, C++, Fortran), 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 testing, 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++, Fortran) 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, white-box testing

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 `case` and `default` branch must be executed
`try-catch-finally`	The `try` clause, every `catch` clause, and the `finally` clause must be executed

A conditional expression such as `(x != 0 ? 1000/x : 1)` must be tested for the condition `(x != 0)` being true and being false, so that both alternatives have been evaluated.

Short-cut logical operators such as `(x != 0) && (1000/x > y)` must be tested for all possible combinations of the truth values of the operands. That is,

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

Note that the second operand in a short-cut (lazy) conjunction will be computed only if the first operand is true (in Java, C#, C, and C++). This is important, for instance, when the condition is `(x != 0) && (1000/x > y)`, where the second operand 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 short-cut disjunction `(x == 0) || (1000/x > y)` it holds, dually, that the second operand is computed only if the first one is false. Therefore, in this case too there are only three possible combinations:

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

Methods The test suite must make sure that all methods have been executed. For recursive methods one should 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 Black-box testing

The goal of black-box testing 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 black-box 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 or illegal 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 to be on the safe side'. Instead, one must carefully consider what inputs might reveal problems in the program, and use exactly those. When preparing a black-box 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 black-box testing

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

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	Actual 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 black-box testing

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

We shall design a black-box 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 property	Input data set
No numbers	A
One number	B
Two numbers, equal	C1
Two numbers, increasing	C2
Two numbers, decreasing	C3
Three numbers, increasing difference	D1
Three numbers, decreasing difference	D2

The data sets and their expected outputs might be:

Input data set	Contents	Expected output	Actual output
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 black-box testing may be carried on indefinitely: you will never be *sure* that all possible errors have been detected.

3.3 Example 3 of black-box testing

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`.

Contents	Expected output	Actual output
`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 black-box 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 and C#, static and non-static fields in classes automatically get initialized to 0. The actual output may therefore equal the expected output by accident.
- In languages such as C, C++ and Fortran, 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 at all possible. Then it can conveniently be rerun whenever the program has been modified. This is usually done as so-called unit tests. For Java, the JUnit framework from `www.junit.org` is a widely used tool, well supported by integrated development environments such as BlueJ and Eclipse. For C#, the NUnit framework from `www.nunit.org` is widely used. Microsoft's Visual Studio Team System also contains unit test facilities.
- As mentioned in Section 3 one should avoid creating an excessively large test suite that has redundant test cases. Software evolves over time, and the test suite must evolve together with the software. For instance, if you decide to change a method in your software so that it returns a different result for certain inputs, then you must look at all test cases for that method to see whether they are still relevant and correct; in that situation it is unpleasant to discover that the same functionality is tested by 13 different test cases. A test suite is a piece of software too, and should have no superfluous parts.
- When testing programs that have graphical user interfaces with menus, buttons, and so on, one must describe carefully step by step what actions — menu choices, mouse clicks, and so on — the tester must perform, and what the program's expected reactions are. Clearly, this is cumbersome and expensive to carry out manually, so professional software houses use various tools to simulate user actions.

5 Testing 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 white-box test suite than a black-box one, because one can proceed systematically on the basis of the program text. Black-box testing 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 black-box 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.
- Writing the test cases and the documentation at the same time is also valuable. When attempting to write a test case, one often realizes what information users of a method or class will be looking for in the documentation. Conversely, when one makes a claim ('when `n+i>arr.length`, then FooException is thrown') about the behaviour of a class or method in the documentation, that should lead to one or more test cases that check this claim.
- If you further use unit test tools to automate the test, you can actually implement the tests before you implement the corresponding functionality. Then you can more confidently implement the functionality and measure your implementation progress by the number of test cases that succeed. This is called test-driven development.
- From the tester's point of view, testing 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 white-box test. It also leads to simpler programs that usually are more general and easier to understand.[2]
- It is not unusual for a test suite to be as large as the software it tests. The C5 Generic Collection Library for C#/.NET (http://www.itu.dk/research/c5) implementation has 27,000 lines of code, and its unit test has 28,000 lines.
- How much testing is needed? 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.

[2]A program may be hard to understand even when it has no choice statements; see Exercises 10 and 11.

6 Exercises

1. Problem: Given a sequence of integers, find their average.
   Use black-box techniques to construct a test suite for this problem.
2. Write a program to solve the problem from Exercise 1. The program should take its input from the command line. Run the test suite you made.
3. Use white-box techniques to construct a test suite 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`.
   Use black-box techniques to construct a test suite for this problem.
5. Write a program that solves the problem from Exercise 4. Run the test suite you made.
6. Use white-box techniques to construct a test suite for the program written in Exercise 5. 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 (black-box) test suite given there.
8. Use white-box techniques to construct a test suite for the program written in Exercise 7. Run it.
9. Problem: Given a (day, month) pair, compute the number of the day in a non-leap year. For instance, (1, 1) is number 1; (1,2), which means 1 February, 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. the length of a course, the duration of a bank deposit, or the time from sowing to harvest. The date and month can be assumed legal for a non-leap year.
   Use black-box techniques to construct a test suite 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 black-box test suite you made above.
11. Use white-box techniques to construct a test suite 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 white-box test. Do that.
12. Consider a method `String toRoman(int n)` that is supposed to convert a positive integer to the Roman numeral representing that integer, using the symbols `I` = 1, `V` = 5, `X` = 10, `L` = 50, `C` = 100, `D` = 500 and `M`= 1000. The following rules determine the Roman numeral corresponding to a positive number:

- In general, the symbols of a Roman numeral are added together from left to right, so `II` = 2, `XX` = 20, `XXXI` = 31, and `MMVIII` = 2008.
- The symbols `I`, `X` and `C` may appear up to three times in a row; the symbol `M` may appear any number of times; and the symbols `V`, `L` and `D` cannot be repeated.
- When a lesser symbol appears before a greater one, the lesser symbol is subtracted, not added. So `IV` = 4, `IX` = 9, `XL` = 40 and `CM` = 900.
  The symbol `I` may appear once before `V` and `X`; the symbol `X` may appear once before `L` and `C`; the symbol `C` may appear once before `D` and `M`; and the symbols `V`, `L` and `D` cannot appear before a greater symbol.
  So 45 is written `XLV`, not `VL`; and 49 is written `XLIX`, not `IL`; and 1998 is written `MCMXCVIII`, not `IIMM`.

Exercise: use black-box techniques to construct a test suite for the method `toRoman`. This can be done in two ways. The simplest way is to call `toRoman(n)` for suitably chosen numbers `n` and checking that it returns the expected string. The more ambitious way is to implement (and test!) the method `fromRoman` described in Exercise 12 below, and use that to check `Roman`.

13. Consider a method `int fromRoman(String s)` with this specification: The method checks that string `s` is a well-formed Roman numeral according to the rules in Exercise 12, and if so, returns the corresponding number; otherwise throws an exception.
    Use black-box techniques to construct a test suite for this method. Remember to include also some ill-formed Roman numerals.