TESTING COMPUTER PROGRAMS

THE NEED FOR TESTING

Assuming your program has compiled successfully with no warning messages, what comes next? The answer is testing, but this is not a simple as it seems. Any program needs testing in order to show that it works in the desired fashion. Most programs need some form of input, and most of these of these can accept a potentially infinite number of inputs. Even simple programs show this behavior. It is a good guess that we cannot test our programs on all inputs, and in fact it can proved that no amount of testing can prove, in general, that a program works, i.e. that it meets the requirements laid down in the specification, or problem description. We can, however, prove that certain kinds of programs are correct (they work correctly), but this is a hard job, involving complex mathematics and special-purpose computer tools. The rest of us need to test our programs by choosing appropriate inputs, observing the behavior of the program with these inputs, and comparing the outputs with the ones expected by examining the description.

STRATEGIES FOR TESTING

The idea of testing a program is twofold: the first is to see whether the program works in a manner that fits the problem description; the second is to "break" it, i.e. to find a set of circumstances in which it fails. Only when we know how a program fails can we improve it. The chapter on debugging discusses how programs can fail and what we do about it, but here we will concentrate on testing the program's desired behavior. A testing strategy is necessary because without a clear goal, testing can be a very hit-or-miss affair. Many programmers take this approach by default, and tie it in specifically to a debugging strategy, which actually is a different topic. A typical approach runs as follows (don't follow this if you want to be an efficient programmer!):

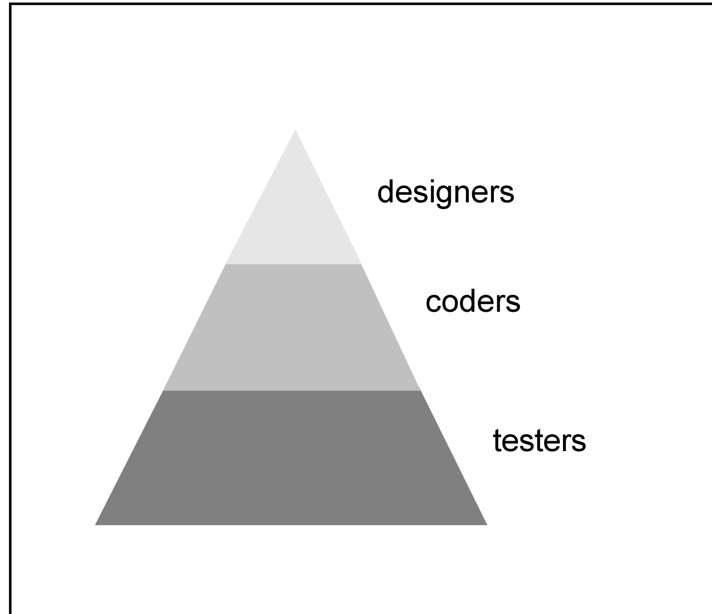
- Write the program, and compile it.
- Run it with either an arbitrary input, or one you are specially interested in.
- Watch the program fail.
- Find the problem and worry it to death until the program works on this one input case.
- Try another input case.
- Watch the program fail.

Etc. etc.

The reason this approach fails should be obvious. You may well be modifying the program for an input case that is not very typical (merely interesting), and you may well miss the important cases that, if the program were to handle them, would make the program robust, i.e. tolerant to all kinds of user input.

A better approach is to make a definite attempt at being *objective*. We often invest so much of ourselves in our programs that we are unable to see where the program's weaknesses are. In fact,

commercial software is always tested by people other than the programmers who write the code. This gives a good measure of objectivity, and they are likely to find more errors as a consequence. It is instructive to look at the development triangle which shows the relative size of the teams involved with software development. Even though the people higher up the triangle are more highly paid, it is still true that a large portion of the costs of developing complex software is spent on testing. There are two



main kinds of objectivity in testing, known as the black-box and white-box approaches. These names are often used by engineers testing hardware devices of various sorts. The device is an object, and testing is like doing *experiments* on the device.

THE BLACK-BOX APPROACH

A black box cannot be opened up to reveal its workings. All we can do is to "poke" it with an input and watch what behavior it exhibits. If it behaves in the expected manner for every input we give it (every experiment we run) then we say that the device works. If it produces unexpected, or even undesired behavior, then we give it back to the designers and tell them to fix it.

The Consumers Reports people follow this testing strategy with products that they evaluate. For instance, they might test bread-making machines by filling them up with the recommended ingredients and looking at the quality of bread that it produces. If the bread is light-colored and soggy in the middle we might want to say that the machine is no good (of course we might have operated it wrongly). However, if the bread is even colored and uniformly done, the we would want to say that the machine has passed the test. The testers are not concerned with how the machine works, or what set of procedures it goes through, just with the bread it produces. Similarly a program can be tested purely for its input-output behavior. This actually is easy for a program, because we cannot "open up" a program to see what makes it tick without special software tools called *disassemblers*. These tools can be used in real emergencies, but they are hard to use and good results are not guaranteed. More often, the black-box testers will simply hand the program back to the designers just as if it was a hardware device, and tell them how the program behaves and where it goes wrong.

THE WHITE-BOX APPROACH

Although good, objective black-box testing is extremely useful in software development, it if often used in conjunction with a more difficult, but ultimately more useful approach. It is very rare that we are handed a program to test without knowing who wrote it, without having access to the source code that was compiled into the running program. This knowledge can aid our testing in that the programmers can give the testers some hints as to how the program works, why it does what it does, and where to look for potential trouble spots. Thus the programmers and testers work as a team and can be more effective, especially with large program that have many different modes or features. Black-box testing would be aimless and patchy in these cases, whereas white-box approach can attempt to be comprehensive. The white-box approach is so-called (perhaps *transparent*-box would be better) because we are allowed to open up the box to examine its inner workings. The engineers who designed the bread-making machine would certainly follow this strategy. They would know that, for instance, that soggy bread might be caused by a faulty timer, or using too low a baking temperature. They would be able to test this by altering the timer or the heater, and "doing another experiment", i.e. running he machine again. In a similar fashion, software testers can focus their tests on specific portions of the code by choosing particular kinds of inputs.

The white-box approach is often used in conjunction with a design technique. The top-down stepwise refinement technique we studied earlier is well-suited to a top-down white-box testing strategy. The program may actually be written incrementally, level by level, where the lowest levels in the TDSR hierarchy are implemented as "stubs", which do nothing, or provide minimal behavior so that their interaction with the level above them in the hierarchy may be tested. Two other related techniques are the *walk-though*, which is an informal simulation of the program's execution, and the use of special-purpose *test code* that can test portions of the program without the necessity of having the whole program in a finished state.

A walk-through is essentially a paper-and-pencil exercise carried out in a group meeting. The designers present their design to representatives of the whole team, and guide everyone through it as if the program were actually in existence. The advantage of the approach is that weaknesses in the design can be caught before expensive coding and testing are carried out. These are typically caught when a team member who was not involved in the initial design process asks a "what-if" question. For example, "what if the user clicks this button before that one?", or "what if the user forgets to press return before returning to the main screen?", or "what if the size of this array is too small to hold all the values needed?" These questions, and the answers to them can benefit the design and improve it before coding begins.

Test code is an advanced technique that relies on good structured design principles allowing special code to be inserted into the application code to test specific parts of it. The advantage is that this may be the only way to test certain parts of the program which otherwise would be too remote from the input. In other words, it is very hard to choose input values that can pass appropriate values to the routine in question. This is the ultimate white-box technique, because test code can only be written with direct reference to application code that has already been written. Only coders should write this code, although testers can have input as to what is to be written and why. The big disadvantage of test code is clear: adding extra code to code that may already be wrong can merely add to the confusion. How do we know that the test code is correct? The answer is we don't, but as long as the code is relatively short and simple, this is a valuable technique.

BLACK OR WHITE-BOX?

The white-box approach is a better approach in the long, run, but we will reserve that until our discussion on debugging. However, the black-box approach is extremely valuable because of its insistence on objectivity. We will study this approach in some detail below.

THE BLACK-BOX TEST SUITE

Most practical programs have an infinite, or at least very large, number of possible inputs. For instance even a simple program that reads two integers can have up to ~10^{18} possible combinations. It is impossible to test them all, and to check that the output is correct in each case. Instead possible inputs must be split into categories, and this breakdown is the subject of a testing strategy. All strategies should follow the "divide and conquer" method. This assumes that an error is produced by a particular input or set of inputs, and that isolating this set will enable the error to be isolated. There are, however, additional methods that will help in this endeavor. Inputs can often be categorized as:

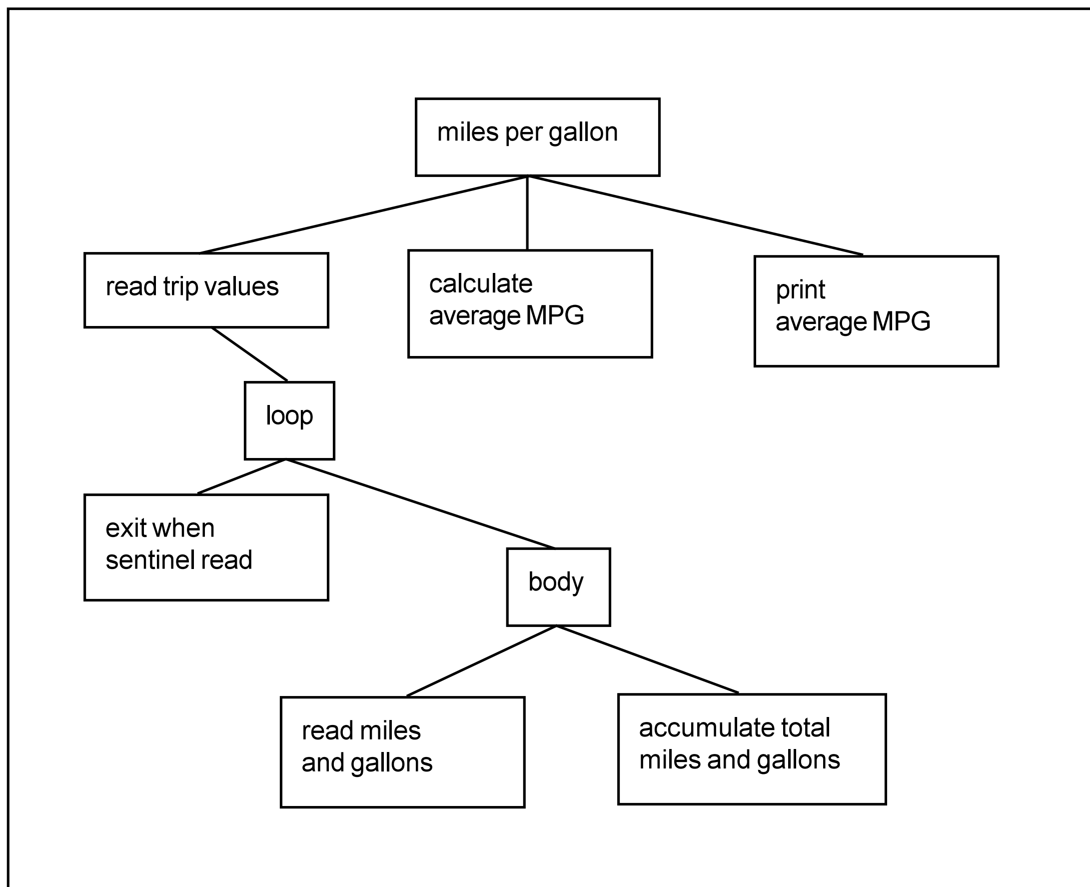
- Typical, expected values
- Special, unexpected values
- Invalid, or out-of-range values

TYPICAL INPUT

Most testing should be done using typical input values that the program design is supposed to cater for. The problem description will often contain ranges or even absolute values that the program should expect to handle. Of course, even these can be too many or too complex for exhaustive testing, so the tester can choose appropriate values, or generate them randomly, the latter with an eye to objectivity. Some programs do not place restrictions on their input values, but even if they do not, programs need to be tested on typical values. In some cases, these will be sets of typical inputs, not just one. The problem with testing on a single value is that the program may work perfectly on your chosen typical representative value, but fail on another because it handles the "typical" value specially without your knowledge. Choosing more than one typical value can minimize this problem. These typical inputs also become part of the test suite.

SPECIAL INPUT

Often programs will fail when presented with data that is zero or absent, or too small. It can also be that the data is too big, or too long. A good strategy in these cases is to present the program with the simplest possible case (typically zero or absent data) and then to present it with a complex case involving large values, long strings, or large numbers of items. If the program passes these tests, it may still fail on "typical" input data, and several tests should be run on data that represents the typical case. It is useless to rely on one representative input, and even when several tests have succeeded, suspicion is still appropriate, and more probing is often necessary before feeling happy about the program. For instance a program that expects to read a name might get an empty string; a program that expects a number of inputs of the same kind may get none; a program that stores a maximum of 100 names might get exactly 100. A list of values might be terminated by a special sentinel value, or perhaps a negative value where the norm is positive. The program should be able to cater for these, without failing, so they become part of the test suite



INVALID INPUT

Many inputs have range restrictions, so the extreme end of the range, and beyond, should be tested. Names of one character; ages less than zero, or greater than 150; car speeds over 500 m.p.h., and so on. These ranges should be tested for at their extremes, and beyond to be safe. Even more common is the program that tests for valid input but does not test for invalid input. If a number between 1 and 5 is supposed to be input, and the user puts in 6 (or 10000!) what happens? If a response should be 'y' or 'n', what happens with a 'z'? Programs that behave gracefully under this kind of fire are called *robust*. Programs that crash when presented with invalid input are just *useless*!.

Having assembled the test suite (and this could be as small as a handful of values for small programs) the program should be executed using the inputs and its behavior observed and recorded. This is rather like experimentation. If the program succeeds on all inputs, i.e. its observed behavior matches the expected behavior, as extracted from the specification, then you are done! Mostly, however, and especially for programs of any size, disasters of various sorts will occur. How to fix these disasters is a job for debugging. However, they would not have occurred if comprehensive testing had not been carried out.

EXAMPLE 1: THE MILES PER GALLON PROBLEM

Problem statement: Print the average gas consumption in miles per gallon of a series of trips, each consisting of the number of miles traveled and the number of gallons used on the trip.

A typical top-down design diagram shows a loop, terminated by a sentinel value, that accumulates total miles and total gallons (see diagram below). When the loop terminates the average is calculated and printed.

Testing this program might follow this sequence:

- Try the sentinel value alone, i.e. no trips at all. Since the totals will both be zero, a message should tell the user that no trips have been recorded.
- Try a trip with zero miles and zero gallons. Since this will produce an average of zero divided by zero, this could be a real problem for the program. At least the zeros could be caught (tested for) and rejected as "improper" values.
- Try one trip with easy numbers for miles and gallons. 100 miles and 10 gallons should give an average of 10 m.p.g.. Do this several times with different values, but ones that produce easily calculated averages. 1000 and 100; 5000 and 500 etc. You could try very large numbers, but internal restrictions in the size of integers could cause the program to produce strange results in these cases. These errors are typically not tested for, but in the case of a program that needs to be foolproof (banking, and financial programs in particular) they really have to take care of every possible eventuality.

EXAMPLE 2: THE GRADE BOOK

Problem statement: The program reads student names and grades from a file and prints three tables: 1. Names in alphabetical order, showing grades for each student. 2. Names in alphabetical order, with total points and average score for each student. 3. Averages in descending order, with student's name.

- Some questions to consider during analysis of the problem are:
- How many students?
- How many grades per student?
- Maximum and minimum grade?
- Length of a student's name?
- Spaces in names?
- File termination: sentinel or end-of-file?
- Duplicate names allowed?

The answers to these questions during the design phase can be used during testing. For instance, if it is decided there should be a maximum of 100 students, then this becomes a special value to be tested for. If it is decided that there should be no spaces in students' names, then a name *with* a space becomes an invalid input to be tested for. If the minimum grade is 0 and the maximum is 100, then these become range values, again to be incorporated into the test suite. The lesson here is that design requirements and testing are intimately involved.

EXAMPLE 3: THE BOOSTERS CLUB

Problem statement: The program creates a file of records for a high-school boosters club, in which each record contains the parents' name, the children's names (at most ten children per family), the names of the sports in which they have participated, and the amount contributed.

For this one, let us try to assemble the three main categories of input discussed above. A typical input might be:

```
Brown     Fred     Soccer        5.00
Goodyear  Sally    Volleyball    10.00
Brown     John     Volleyball    2.50
```

Note the choice of the same parents' name twice, and the same sport twice in different families. The expected output in this case might be:
For special values, we might try long names, and integers instead of decimal numbers for the money amounts:

Invalid inputs might be numbers instead of names, nonexistent sports and very large money amounts:

```
Parent                        Amount
____________________________________
Brown (Fred, John)             $7.50
Goodyear (Sally)              $10.00
____________________________________
Total                         $17.50

Sport                         Amount
____________________________________
Volleyball                    $10.00
Soccer                         $7.50
```

```
   AnExtremelyLongNameThatMightBreakTheProgram Sally     Volleyball
10.00
   Brown                             John    Volleyball     25
```

```
15.00     Fred      Soccer         5.00
Brown     Fred      Water-polo     200.00
Brown     Fred      Soccer         10000000.00
```

SAMPLE RUNS OF A PROGRAM FOR EXAMPLE 1

Here are several runs of a program that is a solution to the problem in example one: the MPG problem. It is not necessary to know what language the program was written in since we are going to use the black-box approach to testing. There are eight tests, following the strategy outlined in the discussion above. In each case, the user input is underlined.

TEST 1

The sentinel, as indicated in the instructions to the user, is -1 for a number of gallons. This results in a message that no gallons have been used, and an average MPG of 0.0.

```
Type any number of lines, each with a number of gallons
used and the miles traveled
Type -1 for gallons and 0 for miles when done
-1 0
No gallons used!
The average MPG for the whole trip is 0.0
```

TEST 2

Inputting zero gallons and zero miles produces another message that zero gallons and/or miles have been input.

```
Type any number of lines, each with a number of gallons
used and the miles traveled
Type -1 for gallons and 0 for miles when done
0 0
Zero gallons or miles: input ignored
-1 0
No gallons used!
The average MPG for the whole trip is 0.0
```

TEST 3

Simple numbers: 10 gallons and 100 miles, produces the correct results.

```
Type any number of lines, each with a number of gallons
used and the miles traveled
Type -1 for gallons and 0 for miles when done
10 100
-1 0
The average MPG for the whole trip is 10.0
```

TEST 4

More simple numbers, this time two sets, also produces the expected result.

```
Type any number of lines, each with a number of gallons
used and the miles traveled
Type -1 for gallons and 0 for miles when done
10 100
20 200
-1 0
The average MPG for the whole trip is 10.0
```

TEST 5

A variant on the trip MPG changes the resultant average MPG.

```
Type any number of lines, each with a number of gallons
used and the miles traveled
Type -1 for gallons and 0 for miles when done
10 100
20 200
10 180
-1 0
The average MPG for the whole trip is 12.0
```

TEST 6

Resonably big numbers produce the correct result.

```
Type any number of lines, each with a number of gallons
used and the miles traveled
Type -1 for gallons and 0 for miles when done
100 1000
-1 0
The average MPG for the whole trip is 10.0
```

TEST 7

And again.

```
Type any number of lines, each with a number of gallons
used and the miles traveled
Type -1 for gallons and 0 for miles when done
500 5500
-1 0
The average MPG for the whole trip is 11.0
```

TEST 8

However, very large numbers produce strange results becuase of the internal limits on the size of integers.

```
Type any number of lines, each with a number of gallons
used and the miles traveled
Type -1 for gallons and 0 for miles when done
100000 10000000000
-1 0
The average MPG for the whole trip is 21474.8
```

TEST 9

Obviously invalid input produces an infinite loop. Here the input was i0 (instead of 10).

```
Type any number of lines, each with a number of gallons
used and the miles traveled
Type -1 for gallons and 0 for miles when done
i0 100
0 6684120
Zero gallons or miles: input ignored
0 6684120
Zero gallons or miles: input ignored
0 6684120
Zero gallons or miles: input ignored
0 6684120
Zero gallons or miles: input ignored
0 6684120
etc. etc.
```

TEST 10

The last test is also for invalid input. Here a negative number of gallons was entered, giving an incorrect result.

```
Type any number of lines, each with a number of gallons
used and the miles traveled
Type -1 for gallons and 0 for miles when done
10 100
-5 50
-1 0
The average MPG for the whole trip is 30.0
```