Teaching guide: Testing

This resource will help with understanding of robust and secure programming through testing. It supports Section 3.2.12 of our current GCSE Computer Science specification (8520). The guide is designed to address the following learning aims:

- Understand the essential need to test programs.
- Identify test data that can be used to increase the effectiveness of test results (such as data at the boundaries of acceptance and erroneous data).
- Explore the benefits of unit testing when adopting a structured approach to programming (not on the specification).

Test plans

Developers, test programs all the time. Quite often this is informal testing – run the program and check that it works and, if not, find out where the errors are. Testing is, of course a vital step in ensuring that our programs are correct. It is, however, not always straightforward to know which tests to perform to accomplish this.

Take this example from an earlier teacher resource:

```
setters_number ← USERINPUT
guess ← USERINPUT
IF setters_number = guess THEN
   OUTPUT 'well done!'
ELSE
   OUTPUT 'bad luck!'
ENDIF
```

There are two possible pathways through this program – the path for which the Boolean condition is `True` and the one for which it is `False`. To ensure that this program is correct we will choose two sets of test data – one where `setters_number` and `guess` is equal and another where they are not. Test results should be logged and the obvious way to do this is by using a table. We can fill in most of the table before the tests are carried out:

Test Number	Description	Input Data	Expected Outcome	Result
1	Test for the correct outcome when setters_number and guess are the same value	setters_number: 1 guess: 1	output `'well done!'`	
2	Test for the correct outcome when setters_number and guess are not equal values	setters_number: 1 guess: 2	output `'bad luck!'`	

When the tests are run the last column can be filled in and, assuming the implementation of this program is correct, the completed test table will look like this:

Test Number	Description	Input Data	Expected Outcome	Result
1	Test for the correct outcome when setters_number and guess are the same value	setters_number: 1 guess: 1	output `'well done!'`	PASSED
2	Test for the correct outcome when setters_number and guess are not equal values	setters_number: 1 guess: 2	output `'bad luck!'`	PASSED

Two tests suffice for this program because there are only two paths through the program and, as the program is so small and so easy to follow, if the program passes these two tests we can be fairly confident that it will work in all other cases. More commonly though, testing is not such a straightforward exercise. We will look at two further examples which both require more substantial testing before lastly looking at a way to automate testing within our programs.

Choosing the Test Data

Consider the following example to allow user input of a number between certain boundaries:

```
TRY
   # type check (will go to straight to CATCH if it fails)
   guess ← STRING_TO_INT(guess_as_string)
   # range check
   IF guess < 1 OR guess > 100 THEN
      OUTPUT 'Must be between 1 and 100'
   ELSE
      # if all checks passed then input is valid
      valid ← True
   ENDIF
   CATCH
   OUTPUT 'Must enter an integer (e.g. 42)'
ENDTRY
```

This is a more complex example that validates a user's input and consequently the test plan requires some more thought. Firstly, let's create a list of the purpose of this program:

- the user should enter an integer
- the integer should be between 1 and 100 inclusive (i.e. 1 and 100 are permitted but 0 and 101 are not)

From here you can think about test data that would make you confident that this program is correct. We can't be exhaustive and test every single possible input in our testing for the simple reason that the input to this program are strings that represent integers and there are an infinite number of integers. As it is not possible to perform an infinite number of tests then we need to carefully select representative test data.

Programmers frequently make logical errors at the boundaries of what should and should not be accepted. Using the example above, this would mean at the lowest possible value that should be accepted and then the value below that, and the highest possible value and then the value above that. All of these items of data are called boundary data, the data that should not be accepted is additionally called erroneous and together this data forms a range check on our program. Populating a test table with this boundary data gives us:

Test Number	Description	Input Data	Expected Outcome	Result
1	Test for the highest acceptable integer value (boundary)	`'100'`	The value should be accepted with no output	
2	Test for the integer one above the top range (boundary, errorneous)	`'101'`	The value should not be accepted and the program will output `'Must be between 1 and 100'` and prompt the user to re-enter a value	
3	Test for the lowest acceptable integer value (boundary)	`'1'`	The value should be accepted with no output	
4	Test for the integer one below the bottom range (boundary, erroneous)	`'0'`	The value should not be accepted and the program will output `'Must be between 1 and 100'` and prompt the user to re-enter a value	

In addition to these values you also have to test for more obscure user input. The input must be a string that can be converted to an integer so you should perform a type check and see if the program performs correctly when a value such as '50.1' or 'Fifty' is entered. You could also check that the program rejects the empty string as input (i.e. when the user enters nothing) – this is known as a presence check.

Test Number	Description	Input Data	Expected Outcome	Result
5	Test that non-integer data is rejected (erroneous)	`'50.1'`	The value should not be accepted and the program will output `'Must enter an integer (e.g. 42) '`	
6	Test that non-integer data is rejected (erroneous)	`'Fifty'`	The value should not be accepted and the program will output `'Must enter an integer (e.g. 42) '`	
7	Test that empty user input is rejected (erroneous)	`''`	This should not be accepted and the program will output `'Must enter an integer (e.g. 42) '`	

Finally, we could enter some normal data to check that the program works with normal, typical data that should be accepted:

Test Number	Description	Input Data	Expected Outcome	Result
8	Test that normal data is accepted (normal)	`'50'`	The value should be accepted with no output	
9	Test that normal data is accepted (normal)	`'12'`	The value should be accepted with no output	

Nine separate tests have been defined to get to the point at which you can be highly confident that our program performs as expected. In another resource we covered implementing validation as extensible subroutines, hopefully now you can appreciate why this not only reduces the amount of written code in a program that contains significant amounts of similar validation but also greatly speeds up the testing process.

In summary, these are the types of test you have conducted:

Type of Test	Meaning
Range check	Test that the program performs differently depending on whether the input values are within a given range
Type check	Test that the program responds accordingly if the input is of an incorrect type
Presence check	Test that the program responds accordingly if the input is nothing

And this is the type of test data we have used:

Type of Test Data	Meaning
Boundary data	Values at the extremes of what should and shouldn’t be accepted
Erroneous data	Data that, for whatever reason, is incorrect
Normal data	Correct, typical data

There are many other types of testing that need to be performed, particularly as the programs you develop become larger, such as systems testing and integration testing, but these are beyond the requirements of the specification.

Unit testing (not in the specification)

This resource ends with a skill that pulls together elements of subroutines, structural programming and testing. It is highly dependent on the language and possibly the programming environment that you are using to develop your programs but familiarity with unit testing will greatly increase your speed and confidence in developing programs that work.

Previous resources have covered subroutines and the benefits of modularising program development; decomposing a problem into self-contained subroutines until each module is focused enough to become a solution in its own right.

Each of these subroutines can be tested individually using the type of tests and test data we have just encountered, however it is possible to write these tests directly into the program you are developing and have the tests run automatically when your program starts.

An example of this can be seen in the subroutine from the gardening example used in the Teaching guide - structured programming. This subroutine had the following skeleton program when the structure was being sketched out in code:

```
SUBROUTINE calculate_quote(dimensions, turf)
   RETURN 0.0
ENDSUBROUTINE
```

At this point in development, you could create a test plan for this subroutine that could look like this (note that the point here is not to test the format of the inputs, but to test that the calculation that the subroutine performs is correct):

Test Number	Description	Input Data	Expected Outcome	Result
1	Test that quote is calculated correctly using the formula area*turf value*1.5	dimensions has value [2.0, 3.0] turf has value Turf('x', 10.0)	2.0*3.0*10.0*1.5 = 90.0	
2	Test that quote is calculated correctly using the formula area*turf value*1.5	dimensions has value [1.1, 2.2] turf has value Turf('x', 3.3)	1.1*2.2*3.3*1.5 = 11.979	

You could call this subroutine twice with these parameters and output the result to standard output, or you could create a new subroutine that performs these two tests like so:

```
SUBROUTINE unit_tests()
   test_1 ← calculate_quote([2.0, 3.0], Turf('x', 10.0))
   IF test_1 ≠ 90.0 THEN
      OUTPUT 'calculate quote test 1 failed'
      QUIT()
   ENDIF
   test_2 ← calculate_quote([1.1, 2.2], Turf('x', 3.3))
   IF test_2 ≠ 11.979 THEN
      OUTPUT 'calculate quote test 2 failed'
      QUIT()
   ENDIF
ENDSUBROUTINE
```

Calling this subroutine at the start of the program will quietly perform the test and, if they both pass, will output nothing. However, if either of these tests fail then an output message will be displayed and the program will quit.

Apart from being a targeted way to test individual subroutines, this style of testing is embedded in your program throughout the development process and is called every time your program runs which avoids manually repeating tests.

A final observation on unit tests which is less obvious is ensuring that they do fail. As these tests are silent if successful you need to be convinced that they are actually working. To accomplish this, you could write your unit tests before you implement the subroutine that you are testing. If you run your program with only the skeleton of the subroutine written then the test should fail – if it doesn't then you have made an error in your test! A slimmed down version of the program in development could look like this:

```
# more subroutines as skeleton code

SUBROUTINE calculate_quote(dimensions, turf)
   RETURN 0.0
ENDSUBROUTINE

# more subroutines as skeleton code

SUBROUTINE unit_tests()
   test_1 ← calculate_quote([2.0, 3.0], Turf('x', 10.0))
   IF test_1 ≠ 90.0 THEN
      OUTPUT 'calculate quote test 1 failed'
      QUIT()
   ENDIF
   test_2 ← calculate_quote([1.1, 2.2], Turf('x', 3.3))
   IF test_2 ≠ 11.979 THEN
      OUTPUT 'calculate quote test 2 failed'
      QUIT()
   ENDIF
ENDSUBROUTINE
```

```
# call the test subroutine and expect it to fail
unit_tests()
```

Once you have checked that the unit tests work as expected, you can develop the `calculate_quote` subroutine, run the program again and – if you don’t receive any test failure messages – be confident that you have implemented it correctly.