Dalhousie University
Department of Electrical and Computer Engineering

ECED 3403 – Computer Architecture

Testing requirements with examples[1]

Larry Hughes, PhD
24 May 2021

1 Introduction

Testing shows the presence, not the absence, of bugs[2]

The computer engineer must thoroughly test, even with unlikely parameters, the hardware and software, and ultimately the system itself, to ensure that the system operates properly and reliably.[3]

Software testing is critical to the functioning of any system. It is the verification of the design. Handling common, and predictable cases is just the start of well-planned software. The solution must handle a reasonable range of inputs that may have unexpected behaviours.

A test may explore *edge cases*. An edge case represents a scenario caused within the code that may stretch a data type beyond its capacity such as an input string or location counter. How large can the location counter be? What happens when a counter is incremented past the largest possible value? Is this acceptable or predictable?

Certain *special cases* may produce unintended input; for example, escape characters such as '\n' or '\t' have different meanings in strings.

Testing should consider a reasonable subset of inputs, outputs, and ranges that would push the code to its limits. The tests must stretch the code and ensure desired functionality.

Design the tests carefully and decide what is to be tested. Broadly speaking, there are four levels of testing:[4]

Unit Testing: A level of the software testing process where individual units of a software are tested. The purpose is to validate that each unit of the software performs as designed.

Integration Testing: A level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.

---

[1] Original by Justin Lynch and Larry Hughes released 1 June 2018. This revision has new examples, corrections, and extensions.

[2] Dijkstra (1969) J.N. Buxton and B. Randell, eds, *Software Engineering Techniques*, April 1970, p. 16. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969.

[3] *Computer Engineering Curricula 2016*. Retrieved June 22, 2017, from Association for Computer Machinery - Curricula Recommendations: http://www.acm.org/binaries/content/assets/education/ce2016-final-report.pdf

[4] *Software Testing Levels*, Software Testing Fundamentals, http://softwaretestingfundamentals.com/software-testing-levels/. Accessed 1 June 2018

System Testing: A level of the software testing process where a complete, integrated system is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.

Acceptance Testing: A level of the software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

Regardless of the level, if a test fails, it is necessary to identify *why* it failed. Keep in mind that there are essentially three possible reasons for a test failing:

1. The design was incorrect (i.e., the designer did not understand the problem)
2. The implementation was incorrect (i.e., the coder did not understand the design)
3. The test was incorrect (the tester did not understand what was to be tested)

If a test fails, examine the software to determine why it failed and what corrections are necessary to have it to pass the tests. It is necessary to properly identify what is being tested, and ensure the test verifies it.

Does the test do what is expected of it? Maybe the input file has hidden characters that are not being accounting for? Or maybe the program was initialized in a state that is unable to transition to the state to be tested?

Sometimes the test is well thought out but there were problems with implementation. Perhaps the value of a pointer is printed which eventually gets set to a NULL value. On the last iteration, the code may throw a segmentation fault. In this case, the printing of the pointer within the test caused the fault, not the code itself.

If a test is to be changed, be sure to justify and understand what was wrong with it. Do not change tests just to make the code pass the test.

It is important to remember that when correcting a fault in the software, the overall structure of the software must be considered. A correction for one failed test may break another. Be sure to re-run existing tests and ensure the correction did not cause a fault elsewhere.

In major software systems, tests are not written and then disposed of, testing suites are designed, maintained, and expanded as the system evolves. Testing suites are automated, containing hundreds or thousands of tests that produce reports on the success of the software. "Old" tests that ensure unchanged software is not affected by any additions or corrections to the system.

Testing is essential.

2 Requirements

In ECED 3403, you are required to perform a number of system tests to demonstrate that your software meets the assignment's requirements. Each test is to have the following structure:

Name: Tests need an identifier. The name should give an indication as to what was tested. Reading the names of your test should give the reader a good idea of the capability of your software

Purpose/Objective: Be detailed. Identify the potential problem, the defined behavior when encountering this problem, the inputs to test the case, and the expected outputs. The exact scope of the test and what it is doing should be clear.

Test Configuration: Not all tests start from scratch. Sometimes, it is reasonable to assume a certain state or limited type of input to narrow the test focus. This section should include all the relevant inputs, settings, configurations, and assumptions made during the test.

Expected Results: The expected results of the test.

Actual Results: A capture of the output (i.e., actual results) of the program.

Pass/Fail: Indicate whether the test passed or failed.

By rights, the test requirements should be written independently of the software, often at the design stage. In other words, the programmer should not run the program and use the results as the actual results.

3 Examples

The following examples use the S-Record checking program, `Check SRecord.exe`.[5] You can get a copy of the executable modules and its source and header modules from the course Bright Space website.

Three tests are presented, you should come up with several more.

TEST 1: Missing file name test

Purpose: This test checks for missing S-Record input files.

Configuration: The program can be run in a command-line interpreter or using drag-and-drop.

- If using a CLI, the name of the executable module, `CSR`, should be entered after the prompt, followed by <Enter>.
- If using a windows-based system, there should be an executable module, `CSR`. Click on the name of the executable.

Expected results: In either mode, the program should display the message, `Missing file name`, and then pause, waiting for <Enter>. The program terminates after <Enter>.

Actual results: In both instances, the `Missing file name` message was produced, and the program paused, waiting for input.

The results of running the CSR module in a CLI:

```
C:\Users\larry\Desktop\2021 - 3403\2021 - Tutorials\Week 3\Examples - testing>CSR
Missing file name


C:\Users\larry\Desktop\2021 - 3403\2021 - Tutorials\Week 3\Examples - testing>
```

[5] Since CLIs do not recognize executable files with spaces in their name, a copy of `Check SRecord.exe`, `CSR.exe`, is used in these tests.

In the windows-based system:

```
Missing file name
```

In both cases, <Enter> caused the program to terminate.

Pass/Fail: The test was successful.

TEST 2: Input file does not exist

Purpose: In this test, a file name is supplied, but the file does not exist.

Configuration: This test can only be run using a command-line interpreter and requires the tester to supply a name of a file that does not exist in the directory.

Expected results: The tester should check the directory to ensure the file name chosen does not exist in the director. For example, assuming the file `abc.xyz` does not exist in the directory, the CSR module should display the message, `Error opening file >abc.xyz< - possibly missing`.

Actual results: When CSR is supplied with the name of a file that does not exist (allowing Test 1 to succeed because the file name has been supplied), it produces the following output (file `abc.xyz` does not exist):

```
C:\Users\larry\Desktop\2021 - 3403\2021 - Tutorials\Week 3\Examples - testing>csr abc.xyz
Error opening file >abc.xyz< - possibly missing


C:\Users\larry\Desktop\2021 - 3403\2021 - Tutorials\Week 3\Examples - testing>
```

The program terminated after the <Enter>.

Pass/Fail: The test was successful.

TEST 3: Invalid S-Record header

Purpose: The S-Record has an invalid header, that is, one other than ‘S’.

Test Configuration: An S-Record file with record headers starting with characters ‘A’, ‘X’, ‘s’, and ‘3’ is to be used:

```
S009000041312E61736D15
A10F1000006000790158094008 5CFF23DF
X10520001000CA
s10F1000006000790158094008 5CFF23DF
310520001000CA
S10F1000006000790158094008 5CFF23DF
S10520001000CA
S9031000EC
```

The test file can be obtained by modifying an existing S-Record (.xme) module.

Expected results: The CSR program should display the S-Record and the message `Invalid header: >>#<< - record ignored`, where '#' is the invalid header symbol. For example, a header starting with 'X' would generate the message `Invalid header: >>X<< - record ignored`.

Actual results: The output from the above input is:

```
S0090000041312E61736D15
Header: S Type: 0 Length: 9 Address: 0000
A1.asm§
Chksum: ff
Valid checksum

A10F1000006000790158094008 5CFF23DF
Invalid header: >>A<< - record ignored

X10520001000CA
Invalid header: >>X<< - record ignored

s10F1000006000790158094008 5CFF23DF
Invalid header: >>s<< - record ignored

310520001000CA
Invalid header: >>3<< - record ignored

S10F1000006000790158094008 5CFF23DF
Header: S Type: 1 Length: 15 Address: 1000
Address: 1000:
Chksum: ff
Valid checksum

S10520001000CA
Header: S Type: 1 Length: 5 Address: 2000
Address: 2000:
Chksum: ff
Valid checksum

S9031000EC
Header: S Type: 9 Length: 3 Address: 1000
Starting address: 1000
Chksum: ff
Valid checksum
```

The program terminates after <Enter>.

Pass/Fail: The test was successful.