Loughborough University

ELD522 Task 1 Software Quality Report

Joshua Smith B221322, Igor Spirin B020777

3/7/2017

Contents

Introduction

The purpose of the report is to document the findings of the investigation into the quality of code written for texture assessment.

The texture assessment is carried out by processing data collected by a Talysurf surface texture instrument in order to generate statistical or frequency spectrum of the data collected.

The program calculates parameters and spectral data from the above instrument. In particular:

Load data from file – The data is stored in a text file. It is assumed that it was captured using a texture assessment device

Calculate the frequency spectrum data – The algorithm performs calculations of the Fourier transform

Save the spectral data – The data is stored in text format file

Compute parameters – The parameters for the surface texture are computed

The report will focus on the accuracy and quality of algorithms for delivery of these parameters.

Quality Requirements

The quality of software depends on how well it meets the requirements. There are two types of requirements, functional and non-functional.

Functional requirements detail the operations that the software needs to perform. Non-functional parameters would include things that support delivery of functional requirements.

Functional:

- The program must load data from file
  Handled by load.c
- Calculate frequency spectrum of data in form of a Fourier transform
  Handled by fourier.
- Save the data in .txt format
  Handled by load.c
- Calculate surface texture parameters
  Performed in fourier.c and fft.c, however the function blocks overlap in functionality

Non-functional:

- Maintainability of code
  Some variables are defined in a way that can cause errors down the line
- Reusability of code
  The code is generally well commented
- End user support
  No evidence available for this
- Compatibility with the gathered data
  The code uses doubles, the data provided is type float

There is a degree to which quality is necessary that depends on the use case. Only the minimum standard of quality is <u>required</u> but it is sensible to exceed this standard. However there is a limit to how much quality is feasible to achieve given time and resource constraints. Because the standards to which the software in Task 1 is held are unknown

Testing Requirements

Testing is a practice of running a program with direct intention of finding a previously undiscovered fault. Depending on the complexity of the code and availability of resources, it may be difficult to carry out a full test of the code.

The aim of testing is to indicate presence of errors and reduce defects. In order to deliver a robust piece of code, many tests may be required so from cost/time investment perspective, it makes sense to automate testing as much as possible.

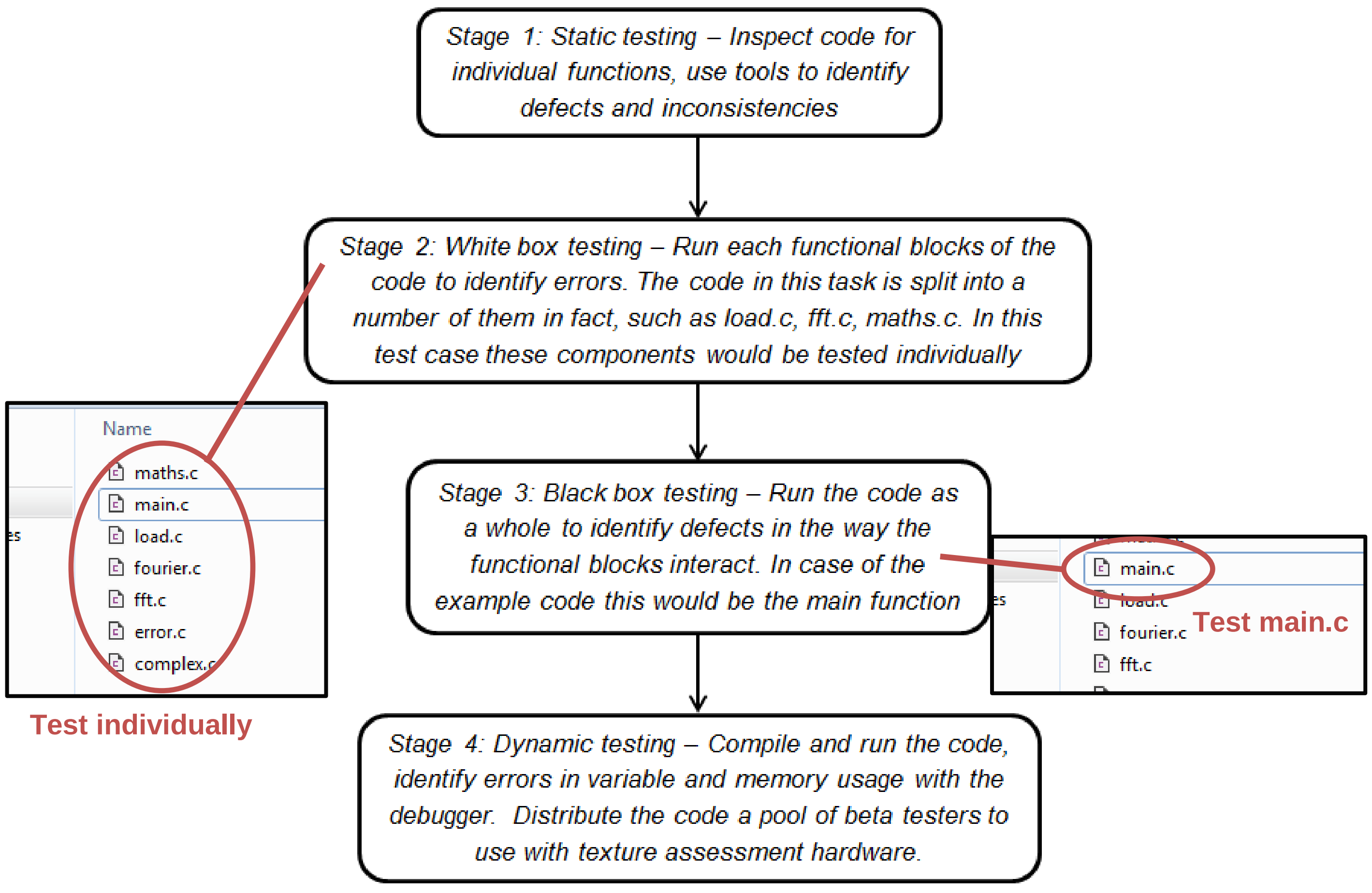
There are two types of testing, static and dynamic. Static testing focuses mainly on error prevention through code inspection. The code is not compiled or run during static testing.

Dynamic testing is different from static testing in requiring the code to be compiled and run. Manual testing would include use by 'beta' testers or end users. Code can also be tested automatically.

Furthermore there is white box testing which focuses on individual functional blocks and black box testing which focuses on the operation of the code as a whole on the interface level.

The task does not involve compiling or running the code so dynamic, white box and black box testing are not possible. Therefore, only technique available is static testing through code inspection.

In industry an example testing process may look like as follows:



Stage 1: Static testing – Inspect code for individual functions, use tools to identify defects and inconsistencies

Stage 2: White box testing – Run each functional blocks of the code to identify errors. The code in this task is split into a number of them in fact, such as load.c, fft.c, maths.c. In this test case these components would be tested individually

Stage 3: Black box testing – Run the code as a whole to identify defects in the way the functional blocks interact. In case of the example code this would be the main function

Stage 4: Dynamic testing – Compile and run the code, identify errors in variable and memory usage with the debugger. Distribute the code a pool of beta testers to use with texture assessment hardware.

Figure 1

The success of testing is measured in the number of defects found relative to the time put into testing. More defect detection can be achieved by doing broader tests first, then focusing on the specifics.

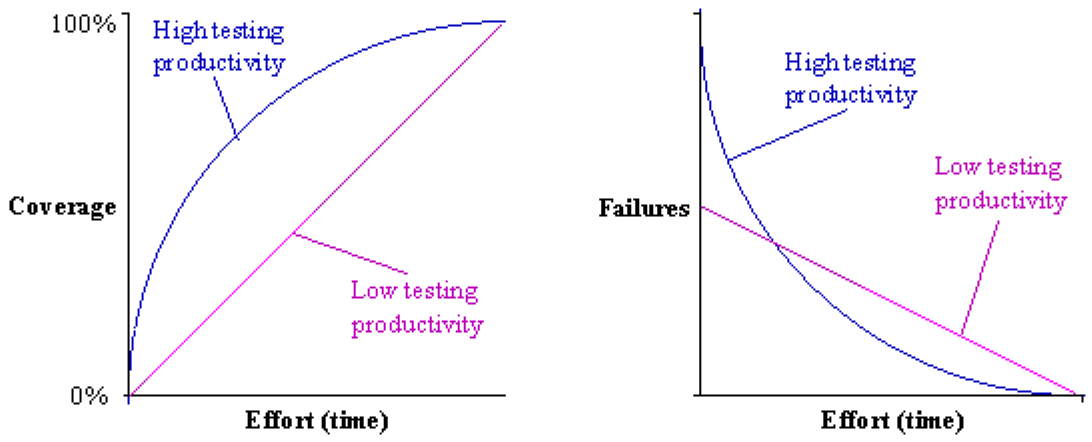


Figure 2

Figure 2 shows the relationships between detected failures and effort. Ideally the relationship should resemble an exponential curve

Simple Linear Reliability model shows how software become more robust with revisions.

$$\lambda = \lambda_0 \left(1 - \frac{\mu}{\nu_0}\right)$$

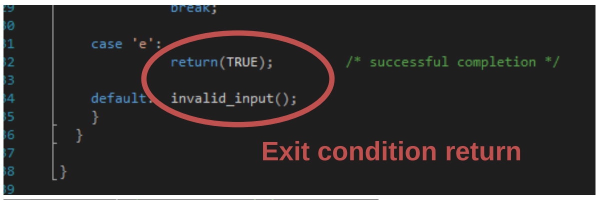
Where λ is failures this week, λ_0 is initial rate of failure and μ is failures recorded thus far. Total number of predicted failures is given by ν_0.

Reliability

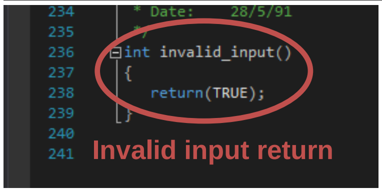
Reliability is defined as the probability that the software does not fail in a specified time frame and environment. Physical objects lose reliability over time but software becomes more refined over time with each revision that corrects the known defects.

Some things that can affect reliability in the example code:

1. *main.c (line 132) will return TRUE and exit from 'e' command. If a wrong input is entered, the code will return a TRUE as well (lines 236-239). The code should output an error message upon detection of a wrong user input instead of exiting.*

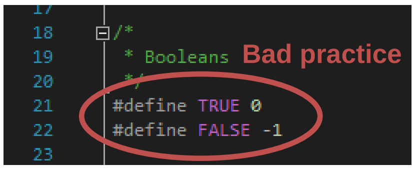


```
            break;

        case 'e':
            return(TRUE);       /* successful completion */

        default:  invalid_input();
        }
    }
}
```



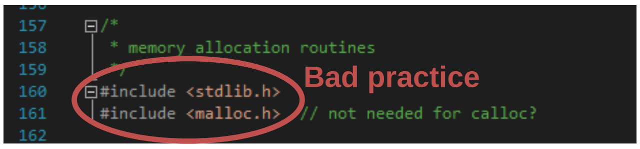
```
 * Date:    28/5/91
 */
int invalid_input()
{
    return(TRUE);
}
```

2. *In Global.h (lines 21, 22), TRUE and FALSE conditions are redefined as 0 and -1 respectively. From future development perspective it can be a cause of severe issues as these conditions are generally expected to be 1 and 0 respectively. For example, the next developer can create a while(TRUE) loop, but it will not work because in context of the code it means while(0).*



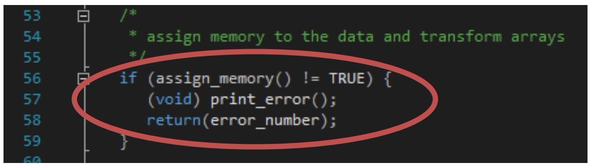
```
18  /*
19   * Booleans
20   */
21  #define TRUE 0
22  #define FALSE -1
23
```

3. *#include for <stdlib.h> and <malloc.h> in main.c (lines 160 & 161)hh are contained in the middle of the code instead of being included at the start with the other #includes. This is not an issue that can affect performance but is not considered to be good practice.*



```
157  /*
158   * memory allocation routines
159   */
160  #include <stdlib.h>
161  #include <malloc.h>  // not needed for calloc?
162
```

4. *main.c allocates memory for the texture assessment (lines 56-58). This way of memory allocation can produce errors when the allocated memory amount is exceeded.*



```
53      /*
54       * assign memory to the data and transform arrays
55       */
56      if (assign_memory() != TRUE) {
57          (void) print_error();
58          return(error_number);
59      }
```

Since memory isn't deallocated afterwards, this method can produce errors when the functionality of the code expands.

5. *The file containing the code did not appear to contain a README file of any sort. Meaning that beyond the comments in the code, there was little in form of a description about the functionality of each function block.*

Conclusion

An example code for Talysurf texture assessment device was examined in context of software testing, quality and reliability. It was discovered that the code does meet the functional requirements and is acceptable in terms of non-functional requirements.

Some potential sources of faults were discovered stemming from the way some variables were declared, as well as handling of user input and allocating memory.