Print to File Experience Report

4/22/2010

PRF Term Testing Project

By

Raymond Pierre-Louis
Sherene Campbell
Lakshmi Gandham
Dallas Brown

The testing of the Florida A&M University Print to File program was performed by four students of the Validation and Verification Spring 2010 class. Their experience lies within this document that shows in detail what they accomplished and learnt in this class project.
# Table of Contents

1. **STATEMENT OF WORK** ........................................................................................................ 3  
   1.1. System/Software under test .......................................................................................... 3  
   1.1.1. Updated System under test ................................................................................... 3  
   1.2. Scope of work.............................................................................................................. 4  
   1.3. Project Team and roles ............................................................................................... 5  

2. **TEST ANALYSIS AND PLANNING** ............................................................................... 5  
   2.1. Project Organization / Work Breakdown ................................................................. 5  
   2.2. Test Techniques and Rationale ................................................................................ 6  
   2.2.1. Black Box Testing ................................................................................................. 6  
   2.2.2. White Box Testing ............................................................................................... 7  
   2.2.3. Load Testing ......................................................................................................... 7  
   2.2.4. Inspection ............................................................................................................. 8  
   2.3. Lessons Learned ........................................................................................................ 8  
   2.3.1. Developer 1 .......................................................................................................... 8  
   2.3.2. Developer 2 .......................................................................................................... 8  
   2.3.3. Tester 1 ................................................................................................................ 8  
   2.3.4. Tester 2 ................................................................................................................ 9  

3. **TEST DESIGN** ............................................................................................................... 9  
   3.1. Design of test cases/Data ........................................................................................ 9  
   3.1.1. Black Box Testing ............................................................................................... 10  
   3.1.2. White Box Testing ............................................................................................... 11  
   3.1.3. Load Testing ....................................................................................................... 11  
   3.1.4. Inspection ........................................................................................................... 11  
   3.2. Test Design Artifact Management .......................................................................... 11  
   3.3. Lessons Learned ...................................................................................................... 12  
   3.3.1. Developer 1 ....................................................................................................... 12  
   3.3.2. Developer 2 ....................................................................................................... 12  
   3.3.3. Tester 1 .............................................................................................................. 12  
   3.3.4. Tester 2 .............................................................................................................. 12  

4. **TEST IMPLEMENTATION** ............................................................................................. 13  
   4.1. Test Harness ............................................................................................................. 13  
   4.1.1. File Comparison .................................................................................................. 13  
   4.1.2. White-box .......................................................................................................... 13  
   4.2. Artifact Management ............................................................................................... 14  
   4.3. Lessons Learned ...................................................................................................... 14  
   4.3.1. Developer 1 ....................................................................................................... 14  
   4.3.2. Developer 2 ....................................................................................................... 14  
   4.3.3. Tester 1 .............................................................................................................. 14  
   4.3.4. Tester 2 .............................................................................................................. 15  

5. **TEST EXECUTION** ...................................................................................................... 15
5.1. Test Automation ................................................................. 15
   5.1.1. Test Harness ................................................................. 15
   5.1.2. File Comparison .......................................................... 15
   5.1.3. White-box wbcA.run tool ........................................... 15
5.2. Artifact management .......................................................... 16
5.3. Process management .......................................................... 16
   5.3.1. Phase 1 ................................................................. 17
   5.3.2. Phase 2 ................................................................. 17
   5.3.3. Phase 3 ................................................................. 17
5.4. Lessons learned ................................................................. 17
   5.4.1. Tester 1 ................................................................. 17
   5.4.2. Tester 2 ................................................................. 17
   5.4.3. Developer 1 ........................................................... 18
   5.4.4. Developer 2 ........................................................... 18

6. TEST EVALUATION ............................................................ 18
   6.1. Artifact Management ....................................................... 18
   6.2. Summary of Test Results ............................................... 18
   6.3. Test Quality report ......................................................... 18
   6.4. Lessons Learned ............................................................ 18
       6.4.1. Tester 1 ................................................................. 18

7. LESSONS LEARNED ........................................................... 19
   7.1. Developer 1 ................................................................. 19
   7.2. Developer 2 ................................................................. 19
   7.3. Tester 1 ................................................................. 19
   7.4. Tester 2 ................................................................. 20

8. APPENDICES ......................................................................... 20
   8.1. All Artifacts (Links) ......................................................... 20
   8.2. Samples of our Test Automation ...................................... 20
   8.3. Sample Test Data ......................................................... 21
       8.3.1. Tester 1 ................................................................. 21
       8.3.2. Tester 2 ................................................................. 21
   8.4. Sample Test Results ....................................................... 21
       8.4.1. Tester 1 ................................................................. 21
       8.4.2. Tester 2 ................................................................. 21
   8.5. Sample Tools ............................................................... 22
       8.5.1. Google Docs .......................................................... 22
       8.5.2. wbcA.run tool ......................................................... 22
1. STATEMENT OF WORK

1.1. SYSTEM/SOFTWARE UNDER TEST

The Software under test is called Print to File whose function is to format text files. The original purpose was to give programs/code line numbers and a neat format in the UNIX environment. The program specification has grown from formatting only program/code files to everyday text files to give left margins, line numbers, pages, characters per line and header/footer.

The original PRF program and its documentation were incomplete. Documentation consisted of only the incomplete specification document. This specification vaguely describes the features of PRF and some of these features are not implemented in the software. As a result the specification and software were both under test along with their different versions due to the fact they need rework.

1.1.1. Updated System under test

The problems with the original version of PRF were that all arguments had to be mentioned in a precise order which increased the chance of user error.

- Old method of invocation: prf.run myProgram.cpp myProgram.lst 6 18 Y Y
- New Method: prf.run myProgram.cpp myProgram.lst lm=6 cl=18 hf=Y ln=Y

As displayed above the user might enter a number for left margin (lm) but it was in the position for characters per line in the old method of invocation. Tags (lm, cl, hf and ln) gives the user more flexibility by entering them in any order or not at all if they wish to use the default value. The features word wrapping and characters per line needed to be implemented since they were missing from the original version of the software. The software was edited to have a more user friendly interface and to implement characters per line function.

Currently the edited software is written in C++ and has 560 lines of code previously 310 lines of code. The current software is sectioned into six functions including the main function and does not take advantage of C++’s object oriented nature. It is a flat program meaning all the high level code in located in one file. The design complexity is high in nature due to the high number of loops and conditional statements needed to complete its task. As a result many test cases are need to fully test the program. Thus white-box testing has to be used on test cases suites to determine the coverage/effectiveness of them.

1.1.1.1. McCabe Cyclomatic complexity

\[
CC = E - N + p \\
= 202 - 110 + 1 \\
= 91
\]

Where:

CC - Cyclomatic complexity
E - Number of edges
N - Number of nodes
p - Number of components.

The table 1 below is to shows how the cyclomatic complexity affects the risk of faults in the program.

<table>
<thead>
<tr>
<th>Cyclomatic Complexity</th>
<th>Risk Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-10</td>
<td>Simple, low risk</td>
</tr>
<tr>
<td>11-20</td>
<td>Moderate complexity, medium risk</td>
</tr>
<tr>
<td>21-50</td>
<td>Complex, high risk</td>
</tr>
<tr>
<td>51+</td>
<td>Very high risk</td>
</tr>
</tbody>
</table>

Table 1 – Complexity risk (Cole, 2007)

We calculated PRF to have a cyclomatic complexity of ninety one and according to table 1 it’s very high in complexity. This is due the large number of decisions with many possible branches. They are a large number of IF statements “back-to-back” which creates a large number of paths for example we have numerous groups of 4 for more if statements.

1.2. SCOPE OF WORK

This term project contained development and testing of the Print to File software. Our scope of work contains development, documentation, testing and automation.

The development aspect consisted of a more intuitive way for the user to use the program using tags. Tags are a way to allow the user to give a descriptive way of providing arguments. For example if they are considering to specify a non-default value for characters per line to fifty characters per line a “cl” (characterline) tag is used in the form cl=50. They are four features with at least twenty four permutations of different orders to place the arguments in the invocation. Tags allow arguments to be placed in any order or any number of arguments.

Documentation consist all all work done on the object such as specification, fault logs, testing history and user manuals. The specification had to be revised since features were added and removed from the software. As a result the specification document is also under our scope of work.

Testing of PRF was performed by using testing different techniques on the different versions of the program; the original software and the revised software. Load and black-box testing are used to find faults in the system regardless of its version. White-box testing is used to determine the quality of the testing techniques. After faults are found they are fixed then regression testing occurs to ensure the faults fixed have not affected any other part of the system.

Automation of testing is needed to allow efficiency and reproducibility especially when regression testing occurs; it will make it go faster and easier.
1.3. PROJECT TEAM AND ROLES

The PRF team consisted of four graduate students with a flat structure. A flat structure means all students have equal responsibility even though we have a lead role. There was a project lead/developer, a developer and two testers. This was a small class project so it was difficult to have discrete hierarchical roles. At different points in the project team members had to switch into another role such as the developer turning into a tester. Each student had their own strength and weaknesses so they were grouped according to their strengths into their primary roles. Also, experience played a role in team assignments; certain members were unfamiliar with the programming language which created restrictions on the role they could play. This also meant they dedicated all their time to testing which results in an advantage because we will always have someone testing while development and documentation occurs.

Within this document names are not used but the role in which was played. The table below shows the names that relate to the roles.

![Diagram 1 – Structure of Team PRF primary roles]

2. TEST ANALYSIS AND PLANNING

2.1. PROJECT ORGANIZATION / WORK BREAKDOWN

The project is organized as planning, development, testing and regression testing. Planning and documentation is always occurring in parallel with the rest of the project. The other categories happen sometimes in parallel such as development (fixing of faults) and testing. Testing occurs by finding faults in the program. Development consists of making changes to the program code which is followed by testing again which is called regression testing. Regression testing ensures the developers changes did not affect other parts of the code by checking to make sure what worked before still works.

The work is broken down in following way as displayed in diagram two.
We have two main sections; documentation and software. Documentation dealt with recording testing procedure, history, faults, plans and specification. Software dealt with the actually execution and non-execution based testing and development.

### 2.2. TEST TECHNIQUES AND RATIONALE

#### 2.2.1. Black Box Testing

Black box testing treats the software as a "black box"—without any knowledge of internal implementation. For the testing of the PRF, black box testing methods include:

##### 2.2.1.1. Blind Testing

With this form of testing, the input file is given and the output is observed. This is also called ‘exploratory’ testing.

##### 2.2.1.1.1. Rational

This testing was performed when the software under test was given with limited specification to determine what features were present. Thus it was very helpful for the initial acceptance testing to determine functionality and what features were needed to be added to the development phase.

##### 2.2.1.2. Ad-hoc testing

Ad-hoc testing is a commonly used term for software testing performed without planning and documentation. The tests are intended to be run only once, unless a defect is
discovered. Ad hoc testing is a part of exploratory testing, being the least formal of test methods.

2.2.1.2.1. Rational

It is always helpful in determining fault using different approaches to testing. Ad hoc testing provides a random nature to find fault as if it were being used by the user. Since testing was performed by developers and testers that knew what they were looking for the ad hoc testing created test cases outside the known domain. This is the rational for the inclusion of ad hoc testing.

2.2.1.3. Boundary Value Testing

Boundary value analysis is software testing design technique in which tests are designed to include representatives of boundary values. The values on the edge of an equivalence partition or at the smallest value on either side of an edge are taken.

2.2.1.3.1. Rational

The values could be either input or output ranges of a software component. Since these boundaries are common locations for errors that result in software faults they are frequently exercised in test cases.

2.2.1.4. Functional Testing

Functional testing refers to tests that verify a specific action or function of the code. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work".

2.2.1.4.1. Rational

The purpose of this form of testing is to see if the program function as our decision table specifies. We use this form of testing by creating a test case (file and arguments) that meets a rule in the decision table and record whether the response meets the criteria of the decision table/specifications.

2.2.2. White Box Testing

White box testing uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software. The tester chooses test case inputs to exercise paths through the code and determines the appropriate output.

2.2.2.1. Rational

White box testing is used to ensure that test cases have a reasonable coverage of the program. If our test cases on test 10% of decision nodes this is deemed inadequate.

2.2.3. Load Testing

Load testing is the process of putting demand on a system or device and measuring its response. When the load placed on the system is raised beyond normal usage patterns, in order to test the system’s response at unusually high or peak loads, it is known as stress testing. The load is usually so great that error conditions are the expected result, although no clear boundary exists when an activity ceases to be a load test and becomes a stress test.
2.2.3.1. Rational
Since the program uses files as input stress testing is used to see how the program operates with large files. Also this is used to see if there is a limit on the size of the input files.

2.2.4. Inspection
This is a non-execution based type of testing. This technique requires a reader, inspector, moderator and recorder.

2.2.4.1. Rational
It helps to find faults against standards and logic. Inspections are really great since it helps everyone on team to better understand the code; to aid in better test case design for executable testing techniques. It also gives a different perspective on testing which allows to find different types of faults in the program.

2.3. LESSONS LEARNED

2.3.1. Developer 1
I gained valuable insight into how to tailor the various testing techniques that are available to suit a particular project. In addition, I learned that often times, a particular testing technique may not be suitable depending on the type and scope of the project.

2.3.2. Developer 2
I gained experience in choosing the appropriate types of testing for the project. It was difficult researching the purpose and nature of testing techniques we never heard about to see whether it was appropriate for our project. As a result I learnt about many different techniques such as stress, load and conformance testing. Load testing was new since it wasn't taught in class but I found it to be very appropriate for the project. If the program crashed with files over 1MB that would be a fault of high priority.
Since time was limited only testing techniques that highly suit a small program and the nature of our software were chosen. Also, techniques the team members were already familiar with was given priority so that we did not spend too much time learning how to perform that type of testing. I also learnt that black box testing is very standard testing due to it evaluates directly whether the program functions according to specification. This is what most clients focus on first rather than load or usability testing.

2.3.3. Tester 1
I have learned that for efficient testing, the specification is the major and the most important artifact. The clearer the specification is, the more it is easy for the tester to do the job. The more the test cases are run, the more are the chances for a complete verification of the code. As the tests are on similar files with various parameters or arguments, we can have a better visibility of the bugs. The comparisons become easy. Doing a wide range of test runs enabled me to find out more bugs. The testing range from simple tests with less size to big files and also with various extensions like .cpp, .ttx, .doc, .java etc. The most important of all is the usage of automation, which made things easy.
2.3.4. Tester 2

I have learned that with the lack of specification, that testing can be accomplished without being overly excessive. With the increase in number of tests ran, I came to realize that using a variety of different test files, that the problems that surfaced were very similar, some even identical to help verify major bugs in the program. My usage of the UNIX environment has increased my overall speed in the UNIX environment because of the many tests that had to be conducted. I also learned a better system on how to organize the different test cases and their output files when testing a lot of different test files. I named the output files based on the test number and the test file named (i.e. Test file name: testfile1 with the first test in the test cases. I would name the output file testfile1out1). The test numbers also where names based on the rules in the decision table. The negative aspect of the blind testing for the PRF code was that with the unlimited format and ways to arrange test in a file, it was nearly impossible to check to make sure that the code worked for every possible text file. I created 10 different test files of several files. I also added a .cpp files to test how the program behaved with using a computer code. The other file that I used contained the line .np and it was used to check how the page breaks works. Overall, the extensive testing helped me understand how the test harness worked. Many problems that were discovered during the functional testing reflected similar bugs for different files which showed that the test harness needed to be altered to work correctly.

3. TEST DESIGN

3.1. DESIGN OF TEST CASES/DATA

In the first draft of the project plan the test data was to be auto-generated. The idea was to have two extensive data banks; one with coded lines and the other with “regular” words. From these data banks a UNIX script would be used to generate the files depending on the arguments provided by the tester such as file extension, size of files and code or normal text file. This was not achieved so test cases were made manually. Unfortunately during development too much effort was need to complete this section and it would make the project late so it was abandoned. Manual test cases creation were used instead and proved to be effective.

PRF test cases comprised of two sections, the file and the arguments as shown in diagram 3. The file is the document that contains words or alphanumeric characters. The arguments are the input into the program for characters per line, line margin, line numbers and header/footer.
3.1.1. Black Box Testing

Black box test cases are created with the decision table and specification in mind. This is because the decision table and specifications tells how PRF is to operate. We create test cases for each rule to see if the response is what the decision table predicts.

3.1.1.1. Blind Testing

These test cases are created as if we did not know how the program operated. This form of testing was extremely helpful in our situation with lack of specifications. This allowed us to see what features the program currently has and what it does not do that we expect.

3.1.1.2. Functional Testing

These test cases are created with the decision in mind to tell us how the program should operate. When these test cases are created the tester has to check which rule it satisfies to ensure what they are testing for is in the specifications.

3.1.1.3. Ad-hoc Testing

Ad-hoc test cases consisted of taking random files and operate on the using any form of arguments. This involves invalid arguments; we treated this testing as a new user would. He/She may accidentally enter an invalid argument.

3.1.1.4. Boundary Value Testing

The test cases consisted of files which specific features and the boundary testing occurred on the arguments provided such as a file with eighty characters per line and the argument characters per line (cl) is set to cl=80, cl=81 and cl=79.
3.1.2. **White Box Testing**

The test cases used in black-box testing are used for the white-box test cases to see the coverage of the test suite.

3.1.3. **Load Testing**

This testing is done by creating files of a variety of sizes and see if the program behaves abnormally with the input.

3.1.4. **Inspection**

This is a non-execution based type of testing. This technique requires a reader, inspector, moderator and recorder.

3.2. **Test Design Artifact Management**

Each test case was managed by its creator. Since we are a small team, the different types of testing was given to each tester and he/she dealt with the revisions and versions of their test cases.

<table>
<thead>
<tr>
<th>Developer 1</th>
<th>Developer 2</th>
<th>Tester 1</th>
<th>Tester 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Adhoc Testing</td>
<td>• White Box Testing</td>
<td>• Boundary Testing • Load testing • Functional Testing</td>
<td>• Functional Testing • Blind Testing</td>
</tr>
</tbody>
</table>

Diagram 4 - Test Case Design Management

Each personnel of the team is responsible for their test cases' lifecycle. That is the specification (purpose), design, creation, execution, and maintenance/revision. The specification deals with the purpose of the test case and how it is meeting the overall objective. For example if Tester 1 is performing functional testing, the test case should reveal which rule of the decision table it is satisfying. When this test case fails for some reason for failure may have to found by editing the test cases to determine the exact issue. This results in maintaining the test case and thus creates different versions of the test case.

Diagram 5 - Test Case Life cycle
3.3. LESSONS LEARNED

3.3.1. Developer 1

It can be quite difficult, or almost unfeasible to account for the many different types of input that could be supplied to a program like PRF. In the end, I believe we made the right decisions in terms of revising how the program should handle a particular scenario. For instance, if the user entered an invalid argument, should the program automatically exit or should it continue to prompt for legal input. I also learned some clever techniques when dealing with string inputs.

3.3.2. Developer 2

Designing test cases can be difficult due to the fact that the program’s complexity high (large number of decision nodes). One has to find a way to cover as much of the code as possible. Test cases are manually created so it can be tedious.

Also managing these test cases are also difficult since the tester may intuitively think that changing a certain variable would be more effective to find faults after executing a certain test case. In this case changing at least one argument creates a total different result and test case. A process has to be defined to control adaptation. This is difficult because the time to document this change probably takes five second would take maybe three minutes to document. This would discourage tester in documenting this small change. If this could be automated to give the tester more room for maintenance and less time for documentation manual test case design and documentation would be more effective and easier. So, I learnt an easier way for documentation of manual test cases is needed to fully track the progression of testing. The utr (test log) and thx (test history) files helps to track the amount of test cases performed and their success. Tracking the maintained test cases is the issue.

3.3.3. Tester 1

Designing of the test cases are the artifacts which need a lot of care in creating them. The pass and failure of the test runs depend on the test cases being used. Utmost care is needed while designing them, as a single mistake in the test case will make the run fail and which might create an impression that the program is wrong though that is not the case. The creation of the test cases for the PRF took a lot of effort as the files are to be formatted accordingly and it is more pain for files of greater size. The most important aspect is the tester should have all the knowledge of how the output file is supposed to be, because without that there is no way of creating the test cases correctly. The test runs are automated using the test harness and which gives the pass/fail reports as utr and thx files.

3.3.4. Tester 2

In the beginning, my initial concept of creating the test cases were too general. Reading through the code and each conditional statement allowed me to see that I needed more test cases. The Actions were strictly based on a PASS/FAIL basis, but analyzing the code, I realized that there were more prompts and notices that the program would perform with the variety of different inputs. I have learned that there can always be a decision table editor created when dealing with any program and even with the lack of specification. The test cases that I designed, the first part of the decision table tested the accuracy of prompting the user and testing to make sure the correct output file was created. The next group of test cases were only testing for test cases
where the arguments where input using the labels. Dividing the two ways to input the arguments help with the organization when dealing with the many test cases and I was able to see and understand exactly where the errors occurred.

4. TEST IMPLEMENTATION

4.1. TEST HARNESS

A test harness execution script was developed to assist in automating the testing of prf. The test harness (run_prf.csh) contains an oracle which compares the actual result file and the expected result file (exp.out). The correct syntax for running the test harness is shown below:

% csh run_prf.csh [runid] [inputfile] [outputfile] [optional arguments]

The test results file (utr_prf.txt) contains a list of all the tests that have been executed and displays whether those executions passed or failed. At the start of each test run, a run id is written to the test results file. A sample portion of the test results file can be viewed in the appendices.

    echo "prf TEST EXECUTION $runid STARTED on `date`" >> $utr

The run id written at the start of each test run splits the test results file into sections, which helps to distinguish the results of each test run performed.

A test history file (thx_prf.txt) is automatically generated and updated which keeps a record of all the test runs performed as well as the final result of each individual run. A sample portion of the test history file can be viewed in the appendices.

    echo "#$runid $user `date` TEST EXECUTION: $testres" >> $thx

4.1.1. File Comparison

An oracle was created to help determine whether a test run passed or failed. The oracle does this by comparing the actual output file generated by prf for a specified test case input, to an expected output file (exp.out).

By making use of the powerful utilities native to UNIX (e.g. awk, sed, etc), the oracle is capable of testing for: match in line numbers, match in header/footer, match in characters per line, and match in margin length. If all pass then the test run is deemed a pass.

The reason for choosing this method to test is because it allows us to perform a more thorough test of files and offers more flexibility. However, there are two main drawbacks to this method of testing. First, the test is platform dependent (only works in UNIX) because of the tools used to compare the files. Second, the expected test results must be developed by hand or obtained from other sources.

4.1.2. White-box
4.1.2.1. **wbcA.run tool**  
This tool was created by Dr. Edward Jones that takes a control flow graph (cfg) file (containing the edges) along with a file showing the nodes traversed by a test case.

4.1.2.1.1. **Nodes hit file**  
The wbcA.run tool needed a file that has the nodes traversed by the test case. This was generated using an instrumented version of PRF. This is a file that has in each node a cout statement which is printed to an external file when the test case is run.

4.2. **ARTIFACT MANAGEMENT**

- **prf.cpp**: source code for prf program written in C++. This source code contains all the modifications made to allow the user to specify Header/Footer, Line Margin, Characters Per Line, and Line Numbers.

- **prf.run**: executable program. This is the output produced after successfully compiling the prf source code. Takes arguments via the command line and also has the option of allowing the user to enter arguments using the keyboard.

- **run_prf.csh**: test harness execution script. It includes the test oracle which determines whether a test run passed or failed. This script also produces a test result file as well as a test history file.

- **utr_prf.txt**: test results file. Automatically produced and appended when the test harness script is executed. Contains the results of all the test runs performed.

- **thx_prf.txt**: test history file. Also produced and appended automatically by the test harness execution script. Contains a log of all the test runs performed.

4.3. **LESSONS LEARNED**

4.3.1. **Developer 1**
Creating the oracle for comparing the expected output file to the actual output file was a challenging but rewarding experience. There were many factors that had to be taken into consideration, especially since the program is used to format text so the test harness must be able to contend with a wide variety of test input and output. In the end, it was worth it since I learned a lot more about various UNIX utilities and their attributes and I also learned more about shell scripts.

4.3.2. **Developer 2**
Editing and reviewing Control graphs is very tedious due to the various branches you have to track. One change would require changing a variety of different elements. A program to auto generate CFG would be more reliable and easier to update. Also for the wbcA.run tool a cfg file was need. The format of the file requires each edge typed which was long, time consuming and tedious. As a result it was not finished. With enough time this would have been to evaluate the coverage of test cases used. Every other feature was done such as cfg’s and the PRF instrumented program that generated a file with a list of nodes traversed by the program.

4.3.3. **Tester 1**
I had worked on testing the program using the test harness. The test harness developed for the prf is quite different from the others I worked as part of the assignments. This harness again re-runs the program using all the arguments and compares the so formed output file with the test case which is named as exp.run. Of all the assignments I worked using the test harness, the one used for this is of greater help because manual comparison of the files would have been impossible with this test harness. The files generated by running the test harness like the thx and utr files are of great help for getting the analyzed results.

4.3.4. Tester 2
I learned how to construct CFGs for white-testing. I have prior experience with creating CFGs for computer programs. The White-Box testing CFGs were different in how the conditional statements were handled. I was used to creating nodes outside of the conditional branches but I learned that I could not do it that way for the white-box testing. I was used to the traditional way of numbering every line of code and assigning a node to every declaration or close or open bracket. Constructing the CFGs also helped me with understanding how the code worked. I was a little confused with all of the while and for loops nested together, but with creating the CFGs, I was able to understand exactly where a conditional statement ended and where it started. Creating the CFGs also helped me with testing and understanding how the program behaved.

5. TEST EXECUTION

5.1. TEST AUTOMATION

5.1.1. Test Harness
Syntax:

% csh run_prf.csh [runid] [inputfile] [outputfile] [optional arguments]

Example on how to run it:

% csh run_prf.csh 1 testfile.txt test.out ln=y hf=y lm=7 cl=40

Expected results file should be called exp.out and should be made before running the harness.

Test results information is written to utr_prf.txt and thx_prf.txt

5.1.2. File Comparison
The file comparison is done automatically done in the test harness so no commands are necessary to perform this automation. All that is required is an expected output file and the execution of the test harness.

5.1.3. White-box wbcA.run tool
5.1.3.1. wbcA.run tool
This tool is executed in UNIX using the command:

%./wbcA.run node_out.txt wbt_results.txt cfg_prf.txt

node_out.txt: This file contains the nodes that are “hit” during the execution of the test case
wbtt_results.txt: This file contains the results of the coverage

5.1.3.1.1. **Nodes hit file**

When the white box instrumented file is run with test cases a node_out.txt file is produced.

5.2. **ARTIFACT MANAGEMENT**

The artifacts that need to be managed during test automation are the fault and test cases log. The fault log is used to log the different faults found in the program during testing. The test cases log is a log of the different test cases ran. The test cases log is to accompany the fault log and also to track the testing progress of the project.

The documents are done using Google Documents due to it being reliable and editable by all team member at the same time in real time. Since these are web-based documents they can be accessed anywhere without complication unlike other tools such as Bug tracker which has known accessibility issues. Remotely logging into a school server can be unreliable which results in being less efficient than the team should be.

The management of these files is assuring that each document is accurate and up to date with information.

5.3. **PROCESS MANAGEMENT**

The process used in this project is similar to other testing life-cycles. Similar in the way that each phase is not discrete but rework is employed. In diagram 6 it shows that we had three phases within this project.

![Diagram 6 – Testing Process](image-url)
5.3.1. Phase 1

This phase dealt with testing of the original PRF software along with its specification. We used blind testing and reviews in this phase. Only blind testing was used because we did not have a through specification to use black-box testing. We had to figure what were the current capabilities of the software and draft our own specification. We had to meet with the client (department chair) to develop a document that got his approval.

5.3.2. Phase 2

With our accurate specification we drafted our decision table to start the process for black-box testing. We also developed the necessary features that were needed for PRF to completely satisfy the requirement followed by testing of these requirements. An inspection was also done as another method testing to find faults with standards, syntax and logic.

5.3.3. Phase 3

This is close-out phase where all bugs are fixed and regression testing occurs. We wrap up the project to assess whether it has been a success and see method of improving the process for another team that would need to work on this software.

5.4. LESSONS LEARNED

5.4.1. Tester 1

The test execution is the actual process which can be called as the test run. All the previous are the preparations to make the test run efficiently. The important lesson is the usage of the automation of the process. I had used the test harness for testing the program and it is a gift for testing. It made testing quite easy where in this case manual comparison of the expected output files and the resulted output file is quite impossible. The efficient use of the automation made testing quite easy apart from the creation of the test cases.

5.4.2. Tester 2

I have learned about process management and the testing process. The testing process involved many times where I would have to have to repeat the process multiple times to make sure that the decisions and rules were being met. The specification revision was a part of the testing process that was constantly being performed as different bugs where found. One specification revision that I noticed that needed immediate change was the case in where the user inputs a wrong character or value for an argument, the program originally would just notify them that they put in a wrong argument and tell them what arguments where accepted. The specification revision was created so that if they put in a wrong value for an argument, it would keep prompting them until they put in a correct value. This helped me tremendously and also help with testing because it limited the user and program to only accept working values. The difficulty of the testing process was keeping up with the changes that were made by the developers and still testing using the older code trying to find new bugs. There were times that the testing had to wait for the developers to take out bugs. Overall, the testing process was a learning experience and I learned the importance of team work and how it can help with the testing process.
5.4.3. Developer 1

I learned a few of the advantages and disadvantages of automating the testing process. First, off automating the testing process can significantly reduce the time spent performing tests. It also ensures consistency because of its repeatability, especially since we developed a test harness that can easily be reran. As for the drawbacks, one major thing that I learned is that automated testing may be useful for certain testing activities, but not necessarily for all. Also, debugging the test harness itself can prove to be difficult. Errors in the script may lead to faulty test results.

5.4.4. Developer 2

Automation was difficult in the sense that if we waited for testing to start after automation started we would not have finished the project. Automation told a while to develop along with the actual development of the software. Time had to managed very well. I was implementing the auto file generator but too much time and effort was being placed in this process that manual test cases would be more efficient in this limited time for the project. Overall I learnt automation is not always the best answer.

6. TEST EVALUATION

6.1. ARTIFACT MANAGEMENT

After each test run the results are stored in a thx (test history) and utr (results file). Also the features of the test case is stored in a test case log (name of files, argument used etc.) for reference and also a fault log exists for recording bugs found.

6.2. SUMMARY OF TEST RESULTS

The test results were very successful since many faults were discovered considering the high complexity of the program. A full description of the test results is located in the project’s repository (link can be found in section 13.1) along with a test assessment.

The testing of the boundary and the load testing are done using the test harness, which made easy for comparing the results.

There are almost 100 test cases created for testing of the PRF for the load and boundary testing. There are 16 failed cases and 84 passed cases (The failed cases are less in number as the part of the project we are supposed to correct the bugs of the program too).The unit results (utr) file and the history (thx) file have the detailed results of the test cases run.

6.3. TEST QUALITY REPORT

6.4. LESSONS LEARNED

6.4.1. Tester 1

Testing is no easy task and there is a lot of patience to do the testing. The efficiency of the tester lies in the test cases selected for the testing the program. Especially for the testing of the PRF I have learnt that along with using good test cases it is also important to have the wide variety of test cases as input to the program. The preparation of the test cases are so important that slightest error can lead to the ambiguity
of the program. The test results are to be analyzed with great care that we know the exact functioning of the program being tested. The efficiently the results are analyzed, the more we can say that the program is bug free and running as stated in the specification.

7. LESSONS LEARNED

7.1. DEVELOPER 1

I enjoyed being a part of this project and being a part of the PRF team. We each complemented each other’s strengths and weaknesses. Furthermore, I was able to apply a great deal of the knowledge gained throughout the semester from the Verification and Validation into PRF. I learned how valuable it is to automate the testing process, and to keep track and manage the results of those tests. I also learned that in the end, users make the best testers. It is quite easy for the developer to overlook a bug, especially since they are focused on producing the code. By continually testing and re-testing we were able to discover a number of bugs that could have gone unnoticed otherwise. Consequently, I learned how beneficial it is for developers and testers to be in constant communication.

7.2. DEVELOPER 2

This project has taught me managerial, development and testing skills. Managing a project that was initially intended to be strictly a testing project proved difficult. This was that the first draft of the project had to thrown out to encompass development and heavy documentation. Also a team with members that were not familiar with C++ constricted how work was divided.

Development proved relatively easy; the difficult part was finalizing the specification to know what to develop. Fixing faults and developing features were on-time and was very successful.

Testing is not as easy as an average Joe may think. You may be re-testing the same sections of the code which results in wasted time and effort.

7.3. TESTER 1

Testing proved to be so important in any project and this project made me realize it. This project enabled me to experience the how the role of a tester would be in a real world. Team work is an important thing I have learned in the project. We had many group meetings which enabled us to discuss on how the work is to be done and what procedures needed to be followed. There was exchange of ideas and each and every aspect was analyzed by each of us working in a team. Communication was made using all means like emails, meetings, calls etc. Coming to the working on PRF, I had learned the efficient use of the decision table to test it. There are almost 54 rules created in the decision table and each rule is used to replicate the various test cases for the program. The preparation of the test cases was a real challenge for the PRF program as it deals with the editing of files and doing it annually is a tough job. The functional testing was done for all the rules in the decision table and then there was boundary testing on the various boundary cases and load testing is done on files of various sizes. The testing was automated using the test harness which is a great boon for testing especially for the programs like PRF while manual comparison is quite challenging. The so formed test results are analyzed to find the correct the testing of the program. There are many challenges all the way long and they are solved as a team.
7.4. **TESTER 2**

I have also found my weaknesses and strength of testing. My white-box testing has greatly improved as a result of the PRF file. My strength lies within Functional Testing. Helping to create rules and conditions for a decision table has been one of my favorite aspects of testing because I think that it helps create organization when dealing with complex programs. I have also learned how easily the decision table helps with testing in general. The amount of conditions and rules were very large and the decision table allowed me to see and limit making duplicate test cases which helps make sure that I am satisfying all rules. This program also has shown me my particular place of interest within the testing process. I was very comfortable with being a tester. The team working skills were also very good. Communication was good. There was many times where I had contact with each individual to help with a particular part of the assignment whether it was testing or developing. There are many times that I had to consult other testers just to verify that the problems that I was encountering are accurate to ensure that we didn’t make duplicate logs for bugs. I also had to communicate with the developers to stay keep up with the new changes and challenges that arose from testing. I also learned that I needed more experience with programming in C++. I have come to the understanding that I needed more practice. However, as the testing progressed, I learned more about how the code worked without understanding every detail in the code. I think that the group has done an exceptional job with the time and specification constraints given.

8. **APPENDICES**

8.1. **ALL ARTIFACTS (LINKS)**

8.2. **SAMPLES OF OUR TEST AUTOMATION**

---

**FIGURE 1 – PORTION OF TEST RESULTS FILE (UTR_PRF.TXT)**

---

**FIGURE 2 – PORTION OF TEST HISTORY FILE (THX_PRF.TXT)**
8.3. SAMPLE TEST DATA

8.3.1. Tester 1

1) Thx_prf.txt - It is the History of all the test runs
2) Utr_prf.txt - It is the unit test results file
3) The third file is the load and boundary test procedures
4) The decision table of the PRF.

8.3.2. Tester 2

<table>
<thead>
<tr>
<th>No</th>
<th>Test</th>
<th>Data File Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>testfile1</td>
<td>testfile1out1 lm=70 hf=y ln=y cl=45</td>
</tr>
<tr>
<td>8</td>
<td>testfile4</td>
<td>testfile4out8 lm=62 hf=y ln=y cl=-20</td>
</tr>
<tr>
<td>9</td>
<td>testfile1</td>
<td>testfile1out9 lm=73 hf=y ln=y cl=65</td>
</tr>
<tr>
<td>27</td>
<td>testfile2</td>
<td>testfile2out26 lm=84 hf=y ln=y cl=-5</td>
</tr>
<tr>
<td>28</td>
<td>testfile1</td>
<td>testfile1 testfile1 lm=22 hf=y ln=n cl=70</td>
</tr>
</tbody>
</table>

8.4. SAMPLE TEST RESULTS

8.4.1. Tester 1

The fourth file is a zipped folder of all the test cases

8.4.2. Tester 2

<table>
<thead>
<tr>
<th>No</th>
<th>Output File</th>
<th>Created</th>
<th>Pass/Fail</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>testfile1out1</td>
<td>Failed</td>
<td>667</td>
<td></td>
</tr>
</tbody>
</table>

The test failed as expected because the output file created is different from the test file which shows that this part of the test fails to match the output file.

The file did not match because of the header and footer and line numbering added on.

| 8  | testfile3out7| Failed  | 1544      |

The test failed as expected because the characters per line that was input by the user did not satisfy the condition of the characters per line being positive.
This caused the program to stop and the test harness to not create any output file.
8.5. **SAMPLE TOOLS**

8.5.1. **Google Docs**

8.5.2. **wbcA.run tool**

8.5.2.1. **Nodes traversed file**

```
1 2 4 5 7
1 2 4 5 6 7
1 2 3
```

8.5.2.2. **Control Flow graph file**

```
8 NODES
8 EDGES
1 2
2 3
2 4
4 5
4 8
5 6
5 7
6 7
```

8.5.2.3. **Report Generated**

UNIT NAME: [Name of unit]

#NODES = 8
#NODES COVERED = 7
COVERED NODES: 1 2 3 4 5 6 7
NODE COVERAGE = 7 / 8

#EDGES = 8
#EDGES COVERED = 7
COVERED EDGES: 1-2  2-3  2-4  4-5  5-6  5-7  6-7
EDGE COVERAGE = 7 / 8

#DECISIONS = 3
COVERED DECISIONS = 2 5
#DECISIONS COVERED = 2
DECISION COVERAGE = 2 / 3