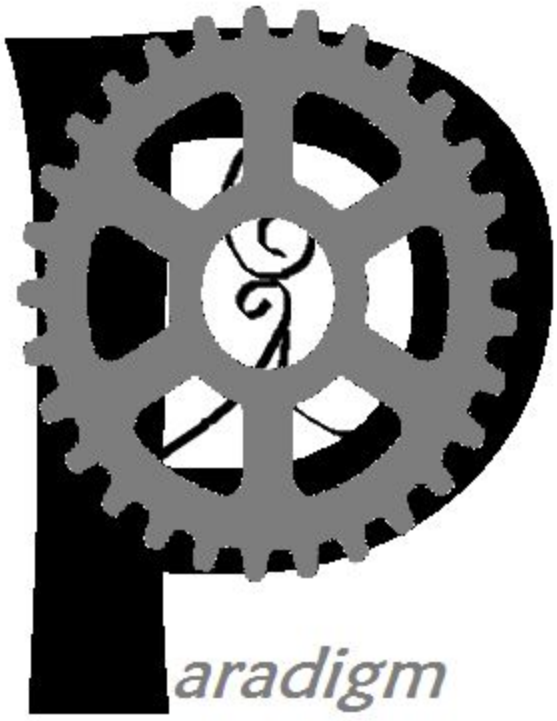
Test Suite
Test Report

v.3.0



Team Members:
Cavaughn Browne
Christian Norfleet
Damien Moeller
Aimee Phillips

Table of Contents

1. Introduction

The purpose of this document is to describe the testing for the Test Suite application. This document describes the scope of testing, activities to be completed, schedule and resources used for the testing the system before it is delivered.

1.1. Product Overview

The Test Suite creates appropriate test data for early undergraduate computer science student's programs. The test suite is to be run concurrently with the program being tested [the test suite does not run the program]. The Test Suite allows for a variety of test settings that can be saved for later use. After running the test suite can then save a report which can be reviewed by the student's professor to help validate the program. During testing, if the student hand-writes expected results, they may compare them to the results, which is given by the program, and analyzed within the Test Suite.

1.2. Test Types

The following are the types of testing our team will carry while developing the Test Suite system:

1.2.1. Unit Testing

Unit testing is a level of software testing where each component is isolated and tested to discover errors before integrating the component into the system.

1.2.2 Integration Testing

Integration testing is a level of software testing where the interaction between the previously tested individual components of the system will be examined. Therefore, the individual components are combined and tested as a group.

1.2.3 System Testing

System testing is a level of software testing where after a complete integration test, the software is then tested in its intended environment with actual user input and user interactions.

2. Scope and Objectives

2.1. Unit Testing

At regularly scheduled intervals and at the end of each module's creation, the module will be subjected to review and testing by the coder and the rest of the team. The purpose of these tests is to ensure each module is correct and free from as much errors as possible. The issues to consider are correctly reading inputs from a source file, generating appropriate test cases for those inputs, suggesting proper test drivers, GUI functions such as comparing test results, saving/loading configurations and saving the test report) are working as expected and time.

2.2. Integration Testing

Integration tests will be done to ensure that the modules interface well with one another and there is no break in the flow of functionality. The modules will be integrated one after another as their unit testing is completed and tested. So, when all the modules are integrated, the final tests will prove that the system works as a complete integrated unit.

2.3. System Testing

This tests ensures that the system meets the requirements as specified in the system requirements document. The complete and integrated software will be tested in the actual

environment where it will be used by the users of the software. Therefore, this assesses the software's quality ensuring any deficiencies are fixed before delivery. All possible inputs and activities are tested to make sure the software will perform as expected and the error count is miniscule when it is delivered.

3. Resources

Paradigm's Test Suite application will be tested using the following resources.

3.1. Human

The Paradigm development team is Cavaughn Browne, Damien Moeller, Christian Norfleet, and Aimee Phillips. The Test Suite is created with consulting with software engineering expert and customer Catherine Stringfellow.

3.2. Hardware and Software

The system will be tested using several computers. Each machine will have at least the following specifications:

- A 32 bit architecture.
- 1.6 GHz processor.
- At least 2 GBs of RAM.

The system will be tested on computers with at least the following software:

- 32 or 64 bit Windows Operating System.
- Visual Studio 2012 or later.

4. Test Schedule

Figure 1 shows the testing process that Paradigm will carry out during development of the system. Testing during each iteration will be scheduled as followed:

Unit testing: As needed during module creation and for each module immediately after it is created.

Integration testing: After the completion of multiple modules. The activity should be performed until it can be said with certainty that modules act in an expected manner with each other.

System testing: After implementation of each module is complete and at least one week before delivery, system testing will be done to further assure the system is as correct as possible and satisfies the requirements.

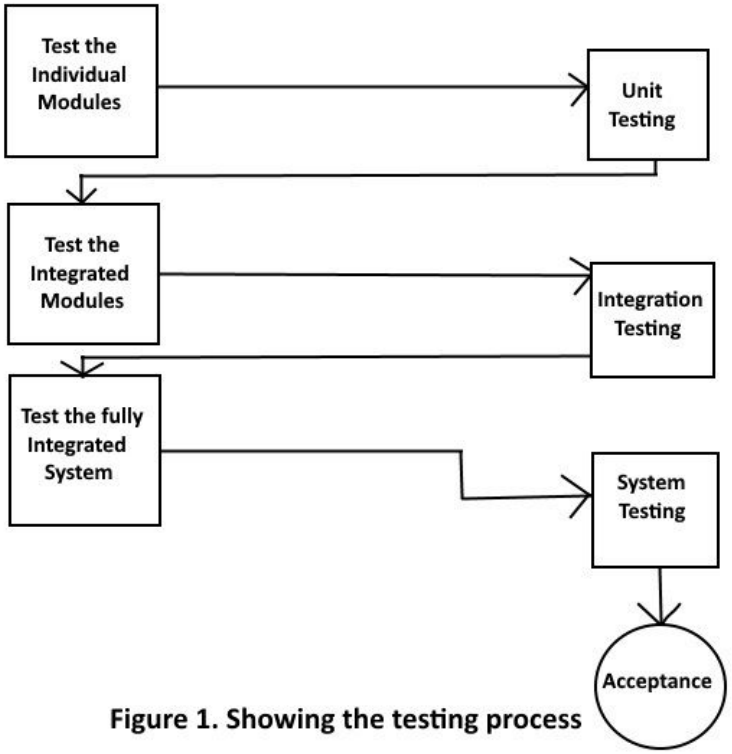


Figure 1. Showing the testing process

5. Test Cases

Table 1 Opening the file

#	Description	Input	Expected Output (EO)	Actual Output (AO)	Test Result
1	Test for opening the source file by clicking	Click on the open file button or file->openfile then double click C:\Users\Damien\Documents\test1.cpp	C:\Users\Damien\Documents\test1.cpp Variables: testInt, testChar, testString, testFloat	C:\Users\Damien\Documents\test1.cpp Variables: testInt, testChar, testString	Failed Inputs after >> on another line is missed
2	Test for opening file using pathname	Click on the open file button or file->openfile then typing "C:\Users\Damien\Documents\P2b.cpp"	C:\Users\Damien\Documents\test1.cpp Variables: int Adult, int Children, double Meal, double SingDessert	C:\Users\Damien\Documents\test1.cpp Variables: int Adult, int Children, double Meal, double SingDessert, double RoomFee, double TaxRate, double Deposit	Failed Doesnt all inputs with declaration on multiple lines
3	Test for opening file using pathname	Click on the open file button or file->openfile then typing "C:\Users\Damien\Documents\test2.cpp"	C:\Users\Damien\Documents\test1.cpp Variables: testInt, testChar, testString	C:\Users\Damien\Documents\test1.cpp Variables: testInt, testChar, testString	Passed
4	Test for opening file containing getline inputs	Click on the open file button or file->openfile then open "10405LCS.cpp"	Opened"10405LCS.cpp" Variables: string s1, string s2	File path of "10405LCS.cpp" Variables: string s1, string s2	Passed

Table 2 Test case generation

#	Description	Input	Expected Output (EO)	Actual Output (AO)	Test Result
1	Test case generation for empty file	C:\Users\Damien\Documents\Empty.cpp (empty file)	Empty list for test inputs and variables	Empty list for test inputs and variables	Passed
2	Random test case generation at test level 1	C:\Users\Damien\Documents\test2.cpp (program with multiple cin and file input statements)	Test Inputs: level 1 (random integer value) (random char value) (random string value)	Test Inputs: level 1 (random integer value) (random char value) (random string value	Passed
3	Random test case generation at test level 2	C:\Users\Damien\Documents\test2.cpp (program with multiple cin and file input statements)	Test Inputs: level 2 (random integer value) (random char value) (random string value)	Test Inputs: level 2(random integer value) (random char value) (random double value)	Passed
4	Random test case generation at test level 3	C:\Users\Damien\Documents\test2.cpp (program with multiple cin and file input statements)	Test Inputs: level 3 (random integer value) (random char value) (random string value)	Test Inputs: level 3(random integer value) (random char value) (random string value)	Passed
5	Boundary test case generation at test level 1	C:\Users\Damien\Documents\test.cpp (program with statements float testval; cin >> testval;) Lower limit: 1 Upper limit: 5	Variables: testval, testval Test Inputs: (0 < x < 1) 1, (1 < x < 2), (4 < x < 5), 5, (5 < x < 6), 3	Variables: testval, testval Test Inputs: (0 < x < 1) 1, (1 < x < 2), (4 < x < 5), 5, (5 < x < 6), 3	Passed
6	Boundary test case generation at test level 2	C:\Users\Damien\Documents\test.cpp (program with statements float testval; cin >> testval;) Lower limit: 1 Upper limit: 5	Variables: testval Test Inputs: (0 < x < 1) 1, (1 < x < 2), (4 < x < 5), 5, (5 < x < 6), 3, min float value, max float value	Variables: testval Test Inputs: (0 < x < 1) 1, (1 < x < 2), (4 < x < 5), 5, (5 < x < 6), 3, min float value, max float value	Passed

7	Boundary test case generation at test level 3	C:\Users\Damien\Documents\test.cpp (program with statements float testval; cin >> testval;) Lower limit: 1 Upper limit: 5	Variables: testval Test Inputs: (0 < x < 1) 1, (1 < x < 2), (4 < x < 5), 5, (5 < x < 6), 3, min float value, max float value	Variables: testval Test Inputs: (0 < x < 1) 1, (1 < x < 2), (4 < x < 5), 5, (5 < x < 6), 3, min float value, max float value	Passed
5	Loop test case generation at test level 1	C:\Users\Damien\Documents\test.cpp (program with statements int testval; cin >> testval; and while (testval >= 1)) cin >> testval;	Variables: testval, testval, testval, testval, testval Test Inputs: 0, 1, 0, 1, 1, 0	Not implemented	Failed
6	Loop test case generation at test level 2	C:\Users\Damien\Documents\test.cpp (program with statements int testval; cin >> testval; and while (testval >= 1)) cin >> testval;	Variables: testval, testval, testval, testval, testval, testval, testval Test Inputs: 0, a, 1, a, 0, 1, 1, 0	Not implemented	Failed
7	Loop test case generation at test level 3	C:\Users\Damien\Documents\test.cpp (program with statements int testval; cin >> testval; and while (testval >= 1)) cin >> testval;	Variables: testval, testval, testval, testval, testval, testval, testval, testval, testval Test Inputs: 0, a, 1, a, 0, 1, 1, 0, 1 0, 1 1 0	Not Implemented	Failed

Table 3 Result comparison

#	Description	Input	Expected Output (EO)	Actual Output (AO)	Test Result
1	Test for results comparison	“Test1” into HW result field, “Test1” into Program result field	√	√	Passed
2	Test for results comparison	“Test1” into HW result field, “1tseT” into Program result field	X	X	Passed

Table 4 Test driver generation

#	Description	Input	Expected Output (EO)	Actual Output (AO)	Test Result
1	Test for generating test driver	Click generate test driver after a header file is opened.	Files are produced which can run the input classes.	Files are produced which can run the input classes.	Passed

6. Analysis

The Test Suite adequately satisfies the demands of the customer. Some bugs were found and some were fixed. However, there are a few that still remain. The Test Suite does not accurately find inputs in c++ source code if the following conditions don't hold:

- Inputs must be declared on the same line as their type
- Inputs must use >> operator or the getline() function
- An input variable with the >> operator has to be on the same line as the operator
- Correct test drivers will only be generated if the class in the .h file has only public methods.

Other minor bugs found include:

- When tabs are switched, there is a "jerky" effect, however, that doesn't affect the rest of application.
- Corresponding menu strip items aren't being disabled with their buttons.
- The testing methods do not differentiate between .h files and .cpp files e.g. (O-O testing should only work on .h files and shouldn't with .cpp).

Testing has improved the quality of the product significantly.

7. Revision History

Date	Revision History	Comment
4/11/2017	v.1.0	Document Draft
4/19/2017	v.2.0	Revised draft document - complete test plan
5/3/2017	v.3.0	Test Report from Test plan

8. References

"Integration Testing." *Software Testing Fundamentals*. N.p., n.d. Web. 10 Apr. 2017. <http://softwaretestingfundamentals.com/integration-testing/>.

Kung, David Chenho. *Object-oriented Software Engineering: An Agile Unified Methodology*. New York, NY: McGraw-Hill, 2014. Print.