Automatic Program Tester

Software Engineering Spring 2014 Documentation

Sprint 1: We Can't Follow Directions
Sprint 2: Software Engineering Adventure Line
Sprint 3: Lounge Against the Machine

Sam Carroll Colter Assman Shaun Gruening Erik Hattervig Andrew Koc
Jonathan Tomes Joe Manke Adam Meaney

April 27, 2014

Contents

Mission **xiii**

Document Preparation and Updates **xv**

1 Overview and concept of operations **1**
1.1 Scope . . . 1
1.2 Purpose . . . 1
1.2.1 Normal Run . . . 1
1.2.2 Generate Tests . . . 1
1.3 Systems Goals . . . 2
1.4 System Overview and Diagram . . . 2
1.5 Technologies Overview . . . 2

2 Project Overview **3**
2.1 Team Members and Roles . . . 3
2.2 Project Management Approach . . . 3
2.3 Phase Overview . . . 3
2.4 Terminology and Acronyms . . . 4

3 User Stories, Backlog and Requirements **5**
3.1 Overview . . . 5
3.1.1 Scope . . . 5
3.1.2 Purpose of the System . . . 5
3.2 Stakeholder Information . . . 5
3.2.1 Customer or End User (Product Owner) . . . 5
3.2.2 Management or Instructor (Scrum Master) . . . 5
3.2.3 Investors . . . 6
3.2.4 Developers –Testers . . . 6
3.3 Business Need . . . 6
3.4 Requirements and Design Constraints . . . 6
3.4.1 System Requirements . . . 6
3.4.2 Development Environment Requirements . . . 6
3.4.3 Project Management Methodology . . . 6
3.5 User Stories . . . 7
3.5.1 Sprint 1 . . . 7
3.5.2 Sprint 2 . . . 7
3.5.3 Sprint 3 . . . 7
3.6 User Story Breakdowns . . . 8
3.6.1 Sprint 1 . . . 8
3.6.2 Sprint 2 . . . 8
3.6.3 Sprint 3 . . . 9
3.7 Research or Proof of Concept Results . . . 10
3.8 Supporting Material . . . 10

4 Design and Implementation **11**
4.1 Student Directory Crawl . . . 11
4.1.1 Technologies Used . . . 11
4.1.2 Component Overview . . . 11
4.2 Compile a Program . . . 11
4.2.1 Technologies Used . . . 11
4.2.2 Component Overview . . . 11
4.3 Test Directory Crawl . . . 11
4.3.1 Technologies Used . . . 11
4.3.2 Component Overview . . . 11
4.4 Run Test Case . . . 12
4.4.1 Technologies Used . . . 12
4.4.2 PComponent Overview . . . 12
4.5 Run Difference Function . . . 12
4.5.1 Technologies Used . . . 12
4.5.2 Component Overview . . . 12
4.5.3 Phase Overview . . . 12
4.6 Student Log Write . . . 12
4.6.1 Technologies Used . . . 12
4.6.2 Component Overview . . . 12
4.7 Final Log Write . . . 12
4.7.1 Technologies Used . . . 12
4.7.2 Component Overview . . . 13
4.8 Test Generation . . . 13
4.8.1 Technologies Used . . . 13
4.8.2 Component Overview . . . 13
4.9 Code Performance and Coverage . . . 13
4.9.1 Technologies Used . . . 13
4.9.2 Component Overview . . . 13

5 System and Unit Testing **15**
5.1 Overview . . . 15
5.2 Dependencies . . . 15
5.3 Test Setup and Execution . . . 15
5.4 Example Test Case . . . 15

6 Development Environment **19**
6.1 Development IDE and Tools . . . 19
6.2 Source Control . . . 19
6.3 Dependencies . . . 19
6.4 Build Environment . . . 19
6.5 Development Machine Setup . . . 20

7 Release – Setup – Deployment **21**
7.1 Deployment Information and Dependencies . . . 21
7.2 Setup Information . . . 21
7.3 System Versioning Information . . . 21

8 User Documentation **23**
8.1 User Guide . . . 23
8.2 Installation Guide . . . 23
8.3 Programmer Manual . . . 23

Acknowledgement **25**

Supporting Materials **27**

Sprint Reports **29**
8.1 Sprint Report #1 29
8.2 Sprint Report #2 29
8.3 Sprint Report #3 29

Industrial Experience **31**
8.4 Resumes 31
8.5 Industrial Experience Reports 31
8.5.1 Colter Assman 31
8.5.2 Samuel Carroll 31
8.5.3 Shaun Greunig 31

List of Figures

1.1 System Diagram 2

5.1 Before test execution 16
5.2 After test execution 17

List of Tables

List of Algorithms

Mission

To lounge as much as possible while producing quality software.

Document Preparation and Updates

Current Version [3.0.0]

Prepared By:
Sam Carroll
Colter Assman
Shaun Gruenig
Erik Hattervig
Andrew Koc
Jonathan Tomes
Joe Manke
Adam Meaney

Revision History

Date	***Author***	***Version***	***Comments***
2/19/14	*Samuel Carroll*	*1.0.0*	*Initial version*
3/23/14	*Jonathan Tomesl*	*2.0.0*	*Sprint 2 Version*
4/27/14	*Joe Manke*	*3.0.0*	*Sprint 3 Version*

1

Overview and concept of operations

This document will look at the Lounge Against the Machine team's third sprint for building a program tester. Looking at the team members and their roles, the project management that we used, the sprint retrospective, any terminology or acronyms that we use. Next we will look at the user stories, backlog and requirements of the program. Then we will look at the design and implementation of the program focusing on major pieces of the code. The next section will look at system and unit testing of our program, followed by development environment, and release, setup and deployment of our program. We will finish this document with a look at a user documentation (including a user guide, installation guide and programmer manual) class index and class documnetation.

1.1 Scope

This document will cover the third sprint of our program tester built for Dr. Logar's software engineering class in Spring 2014 .

1.2 Purpose

The purpose of this program is to compile, run and test the simple programs created by others. The program willl also offer to generate additonal random tests.

1.2.1 Normal Run

The programs it tests are guaranteed to compile and run correctly. It will then search through a root directory, find each student's subdirectory, compile their code into the root directory, open log files for the student and the class as a whole. Student logs will go into the student's directory and the class log will be in the root. The student logs will have the results of each test, a final score, and results from running GNU tools gprof and gcov on the program. The final score will be based on the precent of the tests passed. A program may be marked as a fail if it fails a critical test, or times out. Critical tests are labeled as "crit_(something).tst".

1.2.2 Generate Tests

The program will ask for a number tests to generate, the number of inputs for each test, and the data type for the tests. The program will then generate a "GeneratedTests" directory in the test directory of the root folder. At which point, the program will generate the number of tests specified. If the program has previously generated tests, it will then remove the past generated tests and create new ones.

1.3 Systems Goals

1) Find student programs to compile.
2) Find the tests and use them to test the found programs.
3) Generate new random test cases.
4) Log information about the test runs, including pass percentage, code coverage, and code performance.

1.4 System Overview and Diagram

The program will be started on the command line. You may specify a testing directory. If a testing directory is not supplied, the program will assume its current directory is the testing directory. It will then display a simple menu for executing existing test cases, generated new test cases, or exiting. See Figure 1.1.

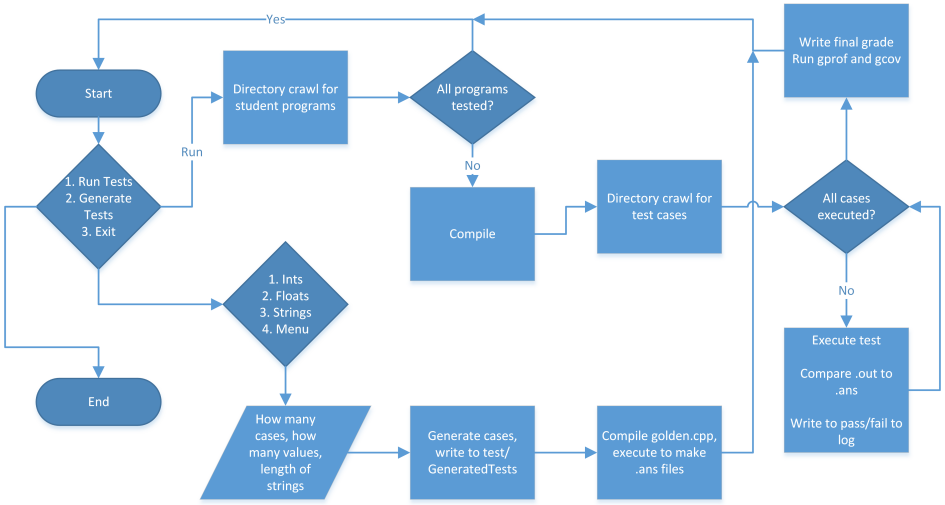

Figure 1.1: System Diagram

1.5 Technologies Overview

The program was written in C++, and was compiled with g++ on a linux environment. The program also requires use of GNU tools gprof and gcov. Project management was done through Trello. Source control was done through GitHub. Documentation was created using LaTeX.

2

Project Overview

This section provides information regarding the team and their roles, how the project was managed, and any relatively unkown terminology or acronyms.

2.1 Team Members and Roles

For Sprint 1, the team members were:
Product Owner: Sam Carroll
Scrum Master: Colter Assman
Technical Lead: Shaun Gruenig

For Sprint 2, the team members were:
Product Owner: Erik Hattervig
Scrum Master: Jonathan Tomes
Technical Lead: Andrew Koc

For Sprint 3, the team members were:
Product Owner: Joe Manke
Scrum Master: Adam Meaney
The team members shared duties as Co-Techical Leads.

2.2 Project Management Approach

The program was created using the Scrum Agile Approach. The sprint length for this project was 2 weeks. We began with a meeting to decide the user needs and split the program accordingly. Each of us would code different parts of the program and then we would all test and re-code as needed.

Cards were created on Trello to keep track of tasks, and a repository was created on GitHub for the code.

2.3 Phase Overview

Sprint 1 (We Can't Follow Directions):

This program started in a gathering information phase, the first thing we had to do as a team was generate several questions to ask the customer (Dr. Logar) about the software that she wanted. The next phase was asking if it could be done, as most of the code had been written at one point or another for our team, we decided that it was a program that could be written. The next phase was breaking the program into smaller milestones, we decided that the program would need to compile source code, search for test cases, run compiled code with test case, output test case results to a file, compare that output file to an expected output file, and then write to a file if the two files match or not, if they do we write the text case then passed to a log file, if they don't we write the text case and failed to the log file. Once we are done traversing the

directory we write the total number of passed, failed and percentages of each.

Sprint 2 (The Software Engineering Adventure Line):

First we began with an information gathering session by Dr. Logar and the product owners. This is where we found out the goals of the program and the desired results. After this we were assigned a team's work from the previous sprint, and evaluted it. Unfortunately we had to do a lot of revising and creating new functions because the old cold was not very modular. So we divided up the work of reworking the code and did our parts. After that we gathered and came up with how to meet the new requirements and worked out what we needed to code, ask the user for, and finished coding the program.

Sprint 3 (Lounge Against the Machine):

Information gathering was done in-class with Dr. Logar. During this class session, we were assigned the code from the team Adventure Line. A short code review ensued where we initially had issues with the directory setup. We were also given non-compiling LaTeXfiles instead of a PDF, which made documentation review difficult. After these small issues were straightened out, we divided up tasks for the sprint. Then we began adding to the existing code base to add the new functionality. Since we did not perform a very thorough code review, we did not discover a few issues of code efficiency until too late in the development cycle to fix them. Specifically, we realized that the program re-searches for all test cases for every tested program, instead of finding test cases once and storing their paths.

2.4 Terminology and Acronyms

N/A

3

User Stories, Backlog and Requirements

3.1 Overview

This section will look at the userstories of the development of the program, the requirements of the program, the proof of concept results, and research task results. It will also look at the reason for developing this software.

3.1.1 Scope

This document will contain stakeholder information, initial user stories, requirements, proof of concept results, and various research task results.

3.1.2 Purpose of the System

To test a set of basic computer programs written in the C++ language so a grade can quickly be assigned to a class of students. The system will also be able to generate random test cases from user input.

3.2 Stakeholder Information

The people most interested in this project is Dr. Logar and perhaps other computer science faculty, who are looking to quickly and easily, grade programs that are turned in by CSC 150 students.

3.2.1 Customer or End User (Product Owner)

The product owner on the first sprint was Samuel Carroll, he created a list with his teammates and on a day, selected by Dr. Logar, met with her and the other teams' product owner to determine exatly what Dr. Logar wanted. Samuel was also the team member most involved in keeping the Trello board up to date.

The product owner on the second sprint was Erik Hattervig, he gathered the necessary requirments of the second sprint from Dr Logar, relaied the information to his teammates, and set up the product backlog on the Trello board.

The product owner on the third sprint was Joe Manke. He gathered requirements from Dr. Logar, created the cards on Trello, and set up the GitHub repository for the sprint.

3.2.2 Management or Instructor (Scrum Master)

The scrum master for the first sprint was Colter Assman, his duties were to ensure that the project stayed on track and if any team member ran into some issues he would help them get back on track. Colter also was responsible for the running of the daily scrum.

The scrum master for the second sprint was Jonathon Tomes, he was in charge of managing the scheduling for the team members, creating spring schedules, and moving tasks from the product backlog to the sprint backlog. He also lead the scrum meetings.

The scrum master for the thrid sprint was Adam Meaney. He assigned tasks and managed the Trello board.

3.2.3 Investors

Our sole investor is Brian Butterfield, who will be reviewing all teams' products and awarding one team the Butterfield Cup for Excellence in Software Engineering, also colloquially known as the Buttercup.

3.2.4 Developers –Testers

Shaun Gruenig was the biggest tester for our program. As the team technical lead he kept us updated on if the project was running as we expected it to, and would often debug the issues our code had.

For the second sprint, all three members of the team tested each other's code and gave feed back on bugs and code quality via Github.

For the third sprint, both members of the team shared development and testing responsibilities, though Adam Meaney did more of the testing.

3.3 Business Need

Currently many computer science teachers have to write each test case out by hand. This is a very time consuming endeavor (especially considering how many students each one has), so this program would enable them to write a test cases which will then be input to our program. This program's purpose is to help teachers quickly and accurately assign grades to students.

3.4 Requirements and Design Constraints

The requirements are that the program be written in C++ and work in a Linux environment. Dr. Logar also required the use of Trello and GitHub for product management, and for documentation to be written with LaTeX.

3.4.1 System Requirements

The program must be able to run on a Linux machine, using the GNU operating system. therefore the code must able to compile using the GNU compiler. This means all of our code must be executable on Linux machines.

3.4.2 Development Environment Requirements

Linux/GNU system should be able to run our tester.

3.4.3 Project Management Methodology

The stakeholders had several requests on how the project was implemented. Including what to use to keep track of backlogs and sprint status, which parties had access to the sprint and produt backlogs, how many sprints will be used for this project, and restrictions on the source control.

- Trello was used to keep track of the backlogs and sprint status
- All parties will have access to the Sprint and Product Backlogs
- Three sprints will encompass this project
- The sprints will vary in length a little bit but be about 2-3 weeks in general
- Github was used for source control

3.5 User Stories

3.5.1 Sprint 1

3.5.1.a Compile and Run Source Code

The program must be able to compile and run source code found in the directory.

3.5.1.b Write Pass/Fail and Percentages to Log File

This program must be able to write output to a log file and to keep track of the total number passed cases and the total number of failed cases.

3.5.1.c Compare Output with Expected Output

The program must be able to compare the output that we get after running a test case to the output that we expect to get from the test case. The expected output will be found when searching the directory.

3.5.1.d Searching/Traversing the Directory

The program must be able to search through all the files and sub-directories of the directory that we are currently in.

3.5.1.e Invoking the Program

The user must be able to run our program by typing "./test directoryName" from the terminal.

3.5.2 Sprint 2

3.5.2.a Class Testing

The user should be able to run tests against an entire class's programs at once.

3.5.2.b Test Case Generation

The user should be able to generate test cases with randomly generated integers or floating-point numbers.

3.5.2.c Acceptance Testing

The user should be able to designate test cases as critical, and mark a program as a failure if it does not pass these tests.

3.5.3 Sprint 3

3.5.3.a Infinite Loop Detection

The program should be able to detect an infinite loop and halt program execution.

3.5.3.b Presentation Errors

The program should be more lax in output comparison, and allow a pass with presentation errors.

3.5.3.c Expanded Test Generation

The user should be able to generate test cases for strings and menu-driven programs.

3.5.3.d Performance Testing

The program should log performance statistics for tested programs using the GNU tool gprof.

3.5.3.e Code Coverage

The program should log code coverage statistics for tested programs using the GNU tool gcov.

3.6 User Story Breakdowns

3.6.1 Sprint 1

3.6.1.a Compile and Run Source Code

The program compiles tested programs using the GNU g++ compiler. Compilation and execution are achieved using system commands.

3.6.1.b Write Pass/Fail and Percentages to Log File

Each test case is given a pass/fail grade per program, based on output comparison. The final grade for a program is the percentage of test cases passed. In Sprint 2, critical failures were introduced. In Sprint 3, passes with presentation errors were introduced.

3.6.1.c Compare Output with Expected Output

For each test case, output is redirected to a .out file. This file is compared to a .ans using the diff command. For Sprints 1 and 2, the .out file must be identical to the .ans file for a program to pass a test case. In Sprint 3, looser requirements were introduced allowing a pass with presentation errors.

3.6.1.d Searching/Traversing the Directory

The program performs recursive directory crawls starting in the given directory to find test cases, notated by a .tst extension, and test source code with .cpp extensions. In Sprint 2, this was modified so all test cases should be in a directory named "test", but may be located in child directories of that. Also as of Sprint 2, test source code for each student should be in a directory with the same name as the .cpp file (i.e. directory "student1" contains "student1.cpp").

3.6.1.e Invoking the Program

The user must be able to run our program by typing "./test directoryName" from the terminal. This executable name is guaranteed by compiling the program using the provided makefile.

3.6.2 Sprint 2

3.6.2.a Class Testing

The executable should be located in a class directory. This directory should contain a directory named "test" where all test cases are located, and individual directories for each student. When tests are executed, a class log file sharing the name of the directory is written in the top level directory, and each student has an individual log file placed in their directory. The class log contains the final grade for each student.

3.6.2.b Test Case Generation

When the program is started, the user is presented with a menu to either run existing test cases or generate new test cases. When generation is selected, the user is prompted for what type of data to generate. In Sprint 2, this was limited to integers and floats. Then, the user is asked how many cases to generate, and how many values to put in each test case. With this information, the program randomly generates values between 0 and 1000 and writes them to files named Test_X.ans, located in the directory "test/GeneratedTests". This directory will be created anew every time test cases are generated.

After the test cases have been written, .ans files are created using a .cpp file located in the starting directory. This program is compiled and tested against each generated test case, with the .ans files also stored in "test/GeneratedTests". The user is then returned to the starting menu, allowing them to run their newly made test cases.

In Sprint 3, this was expanded to include strings and menus.

3.6.2.c Acceptance Testing

Tests with filenames of "x_crit.tst" are considered critical or acceptance tests. If a program does not pass all critical test cases, it is considered a failure regardless of the percentage of non-critical tests it passes. Critical tests are not included in the final grade percentage.

3.6.3 Sprint 3

3.6.3.a Infinite Loop Detection

This is not an attempt to solve the halting problem. Tested programs are now run in a forked child process, and killed if they do not complete within a given time. The user is asked if they want to adjust the timeout limit from its default of 60 seconds when they choose to run tests. If the program does not finish on its own before the timeout, it is considered a critical fail.

3.6.3.b Presentation Errors

Tested programs are now allowed to pass without their .out file being exactly identical to the .ans file for the test case. There are five allowed presentation errors:

- Whitespace differences
- Case sensitivity
- Correct first and last letters, incorrect internals (e.g., "Definitely" vs "Definently")
- Letters in incorrect order (e.g., "Option" vs "Optoin")
- Floating point values: If the .out file has more precision than the .ans file, but rounds up to the number in the .ans file, it is correct. If the .out file is less precise than the .ans file, it is incorrect.

Test cases which differ only on presentation are notated as "Passed with presentation errors" in the log file.

3.6.3.c Expanded Test Generation

The program can now generate test cases using strings or designed for menu-driven programs.

For strings, selecting the number of cases and values to generate is the same as for integers and floating-point values. In addition, the user selects if the strings should all be the same length or variable. The user then specifies a (maximum) length for the strings, capped at 80. All strings consistent only of lowercase letters encoded in ASCII.

For menus, the user only specifies how many test cases to create. The number of values per test case is determined a .spec file, located in the root directory. The format of each line of the .spec file is the menu option (an integer) then either "int" or "double" for every value to be generated. These values will not be bounded.

Example: The line "1 int int double" in a .spec file couldl generate the line "1 4095 728294 8374.837" in a .tst file.

3.6.3.d Performance Testing

The test programs are compiled with the -pg flag so that they may be profiled using gprof. After the program is tested, the command "gprof studentName ¿ studentName.gprof" is executed, writing the full flat profile to a file named studentName.gprof in the student's directory. The name of each function and its percentage of runtime are also written into the student's log file.

3.6.3.e Code Coverage

The test programs are compiled with the flags "-fprofile-arcs -ftest-coverage." After the program is tested, the command "gcov studentName" is executed, creating a file named "studentName.gcov" in the student's directory. The code coverage is also reported in the student's log file.

3.7 Research or Proof of Concept Results

Most of the code had been written by our team before. We knew how to run the system function in C++ to invoke a system command. We had built a directory crawler in an earlier class (though in Windows so some modification had to take place). All in all starting the program we knew we could complete it.

For Sprint 2 much of the same concepts applied for the new features that were added such as test case generation.

For Sprint 3, gprof and gcov had to be researched, but the rest of the new features were rather simple.

3.8 Supporting Material

In the man pages for the diff function it shows us that it returns one of three values and the case those values are returned, a zero if there is no difference between the two files, a one if there is a difference between the two files, or a two if something went wrong (doesn't happen often)

4

Design and Implementation

This section will describe the design details for each of the major components in the system.

4.1 Student Directory Crawl

4.1.1 Technologies Used

This uses the C++ standard library namely the dirent.h library.

4.1.2 Component Overview

This function creates a log file for the entire class in the root. Then it searches for subdirectories other than the Test directory, which contains only the .tst and .ans files. When it finds an applicable subdirectory it changes in, creates a log file for that student, and compiles the .cpp file found within. Then it runs the test directory crawl on the compiled program. After it has returned from the test crawl it does the final log write for the student and writes to the class log as well.

4.2 Compile a Program

4.2.1 Technologies Used

This uses the C++ standard library, system calls, and the g++ compiler.

4.2.2 Component Overview

This program recives a string which determines the name of the executable to be produced. Then from the current directory finds and compiles the first .cpp file found.

4.3 Test Directory Crawl

4.3.1 Technologies Used

This uses the C++ standard library namely the dirent.h library.

4.3.2 Component Overview

This function recusivly searches the test directory and upon finding a .tst file will run the function to test the executable against it. When the function finds a subdirectory it calls itself on that subdirectory, allowing it to fully search for all of the test files.

4.4 Run Test Case

4.4.1 Technologies Used

This uses the C++ standard library and system calls to execute the program.

4.4.2 PComponent Overview

This function takes the name of the .tst file and generates the name of the .ans and .out files. The .out file being created in the same directory as the one the .cpp file was found in, and the .ans in the same one as the .tst file was found in. After the names are generated, using the system command the executable is run with the .tst file used as input and the output being piped to the .out file. Once it has completed the RunDiff function is called to determine if the program executed properly. The log file and record are updated accordingly.

4.5 Run Difference Function

4.5.1 Technologies Used

This uses the C++, and system call function, as well as the diff function in Linux/GNU

4.5.2 Component Overview

This function will create a function that will run the diff function in Linux/GNU via the system call. First however it must create the string that will invoke the call, and take in the file names. Then it will pass inform others if the test case passed or failed so they can handle the information accourdingly. Also this doesn't print anything to the screen if there is a difference between the two files.

4.5.3 Phase Overview

This function can create the command string that will compare two files. This function will call the diff function. This function will not output anything to the screen if there is a difference between the two files. This function informs if there is, or is not, a difference between the two files.

4.6 Student Log Write

4.6.1 Technologies Used

This uses the C++ standard library, namely the fstream library.

4.6.2 Component Overview

This function takes an ofstream, the name of the test, and the success or failure of that test, and writes the formated results to the ofstream.

4.7 Final Log Write

4.7.1 Technologies Used

This uses the C++ standard library, namely the fstream library.

4.7.2 Component Overview

This function takes an ofstream, the name of the student, and a record of the pass and failure of all of the test cases. This function writes the percent of the tests passed, assuming that no critical tests were failed. If a critical test was failed then only "FAILED" is written, rather than the percentage. This function is used to write the last line of each student log file, as well as each line in the class log file.

4.8 Test Generation

4.8.1 Technologies Used

This uses the C++ standard library, namely the fstream library and the function rand() from the cstdlib.

4.8.2 Component Overview

These function use an ofstream to write numbers generated using rand() to files named Test_X.tst. Then the golden.cpp located in the parent directory is compiled and executing against these tests to make Test_X.ans files. For menu generation, another file located in the parent directory with a .spec extension is also used.

4.9 Code Performance and Coverage

4.9.1 Technologies Used

These uses the GNU tools gprof and gcov, and the fstream library.

4.9.2 Component Overview

After running test cases, the tested programs are run against the gprof and gcov tools to determine performance and coverage stats. Information from these tools is also written to the student log using an ofstream.

5

System and Unit Testing

There was no strict testing format for this project.

5.1 Overview

Each programmer tested their code before pushing it up to the git repository. After compatible parts were pushed up, they were combined and tested, to ensure they worked correctly together.

5.2 Dependencies

All of the tests depend on the g++ compiler, and linux platform. The code needs to compile a C++ file inorder to run and test that file.

5.3 Test Setup and Execution

The test cases were provided by Dr. Logar. These test cases were used to test the program. Then after we were satisfied with the programs performance on the test case, the file of the test cases was changed to ensure it still ran, more or less, the same, differences in the order the test cases were found being unimportant. Several different test sets were used to test the program.

5.4 Example Test Case

The following is an example test case for a run of the program. In this example, two menu-driven test cases are being generated and executed. The steps to this test case are as follows:

- Start the program from the terminal with ./test
- Enter "2" to generate test cases
- Enter "4" to select menus
- Enter "2" to generate 2 test cases
- Enter "1" to run test cases
- Enter "3" to exit the program

Figure 5.1 below shows the layout of the file directory before the program is run. Our program, the executable named test, is located in the class directory name CSC150. Also in the parent directory are golden.cpp, to be used to generate .ans files for the generated tests, and menu.spec, to be used when generating the test cases. There is one test case already in the test directory, named menu.tst, and its respective solution menu.ans. There are two student directories, student1 and student2, each containing a .cpp file

with the same name.

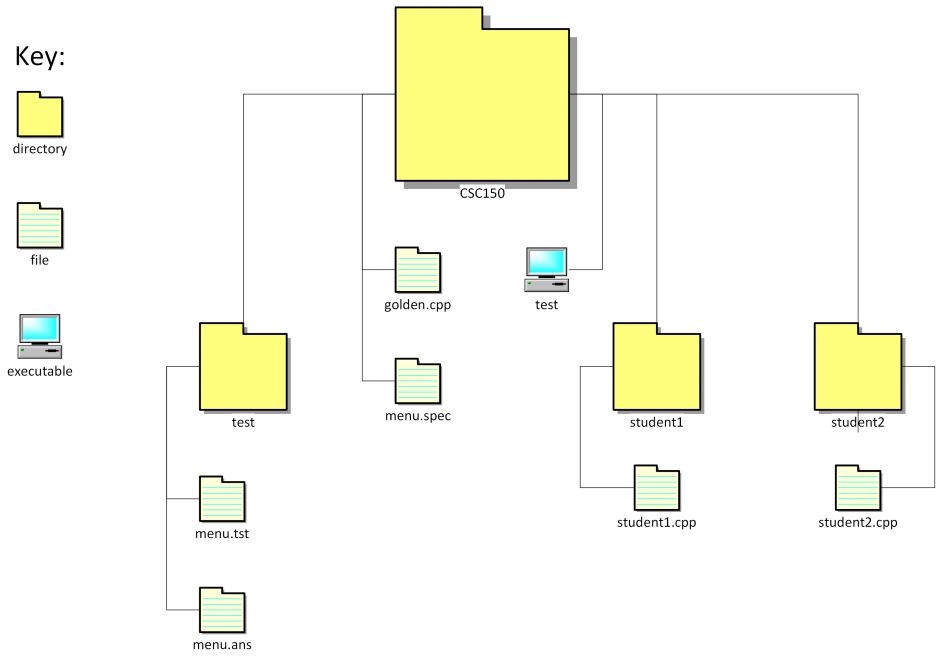

Figure 5.1: Before test execution

After the program, several more files have been added. The GeneratedTests directory has been created inside of test, and contains two .tst files and corresponding .ans files. In each student directory, an executable, a log file, .gprof and .gcov files, and gmon.out (created by gprof) have been added. In the parent directory, a class log has been created sharing the name of the directory. Figure 5.2 shows the ending directory structure.

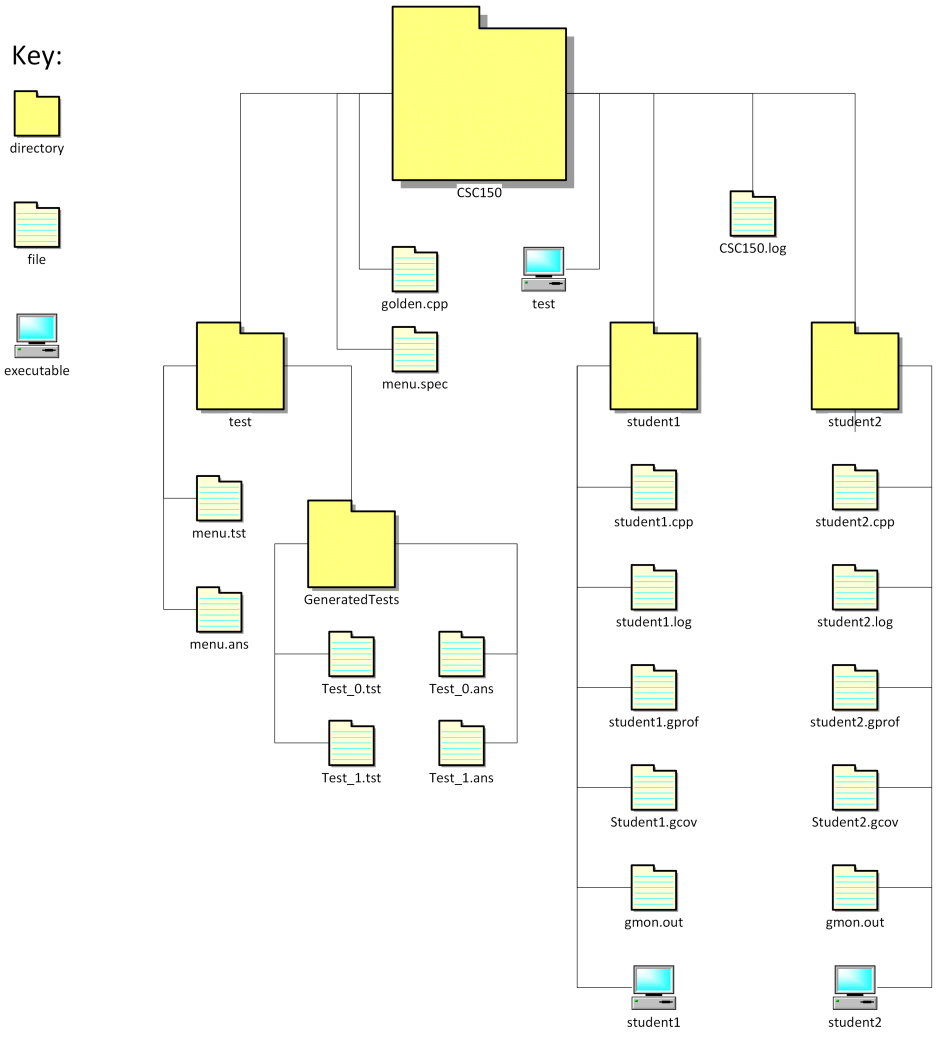

Figure 5.2: After test execution

6

Development Environment

This section will provide all the neccessary information to run, test and develop our source code.

6.1 Development IDE and Tools

The code was written in simple text editors, and requires no special IDE's or tools. All the code was tested on a Linux machine using the g++ compiler. The debug tool gdb was used to debug the code when necessary.

6.2 Source Control

We used GitHub for source control. The private repository can be found at this url:
https://github.com/jcmanke/SoftEng/Sprint3.git

6.3 Dependencies

This program is dependent on the C++ Standard Library as well as the g++ compiler on a Linux system.

6.4 Build Environment

The execuatble is built by the g++ compiler. You can either compile it manually, using the command:

```
g++ -o test ProgramTester.cpp
```

Or by using the following make file:

```
# compiler
CC = g++

# compiler options
CFLAGS = -c

#OBJECT FILES
OBJS = ProgramTester.cpp

tester: ProgramTester.cpp

   ${CC} -g -lm ${OBJS} -o test
```

6.5 Development Machine Setup

For the Development Machine set up any Linux machine with the GNU compiler should work.

7

Release – Setup – Deployment

The source code will be provided to Dr. Logar using email or the Submit It! page of the SDSMT Math and Computer Science webpage.

7.1 Deployment Information and Dependencies

Will need g++ and a linux like environment to run our program.

7.2 Setup Information

By navigating to the directory and running make, the program will be installed.

7.3 System Versioning Information

The system is in 3.0.0 stage, or second release.

8

User Documentation

8.1 User Guide

To use our program simply run it on the command line. You may specify a starting directory as a command line argument, or the program will assume to run in the directory it already is in. Next a menu will apear and ask for one of the options.

Run will let the program run normaly, it will search through the directory and find student directories containing the student source code. It will compile the code to the root directory. It will then search through the test directory in the root to find test casses to run on the student programs. It will log the result of each test to a student log file located in the student directory, and the final result will be repeated in an overal class log located in the starting directory. If the student fails a test labeled "crit_(something).tst" the student will immediately fail and no more testing will be done.

Generate will then prompt for a number of test cases, the number of inputs per test, and finally a data type to use for each test. It will then create a sub-directory called "Generated" in the test sub-directory of the root where the program was prompted to go or started in. If it had previously created tests, it will clear the directory and create a new one and all new test cases.

Exit will exit the program.

8.2 Installation Guide

1) Open the terminal and navigate to the directory containing our program source code and the makefile.
2) Run make.

8.3 Programmer Manual

Acknowledgement

Thanks to Dr. Logar, for her help, knowledge and guidance.

Supporting Materials

There are, for this sprint, no supporting materials.

Still no supporing materials for sprint 2

Sprint Reports

8.1 Sprint Report #1

The testing function can traverse a directory looking for source code. Compile that code, then look for test cases and run the compiled code with the given test case outputting the result in a new document. It can then compare that document with the expected results document and write to the log file if it passed or failed. Then it will continue looking for test cases until all the test cases in the root folder have been found. The tester will then write the total number of test cases passed, total failed and the percentage passed and failed.

8.2 Sprint Report #2

Had to redo a lot of the code. Most of it wasn't split into seperate functions and some of it was misdocumented. After getting around that, we got it to travese the root directory, finding the student directories and testing them against the tests located in the test directory. It can generate random tests (the number of and data types specified by the user). Introduced a simple menu system to make it easier to generate then run tests. It will log the results of the testing each student into a student log located in their directory, and to a class log located in the root directory.

8.3 Sprint Report #3

This section required a long time before we were even able to make it run. We used the test directory supplied on the website, and the program would register a segmentation fault. It turns out that the previous group had decided that all tests that were to be run would only be in a directory called tests, that the student directories would only have one .cpp file and no files that contained cpp, such as a cpp.log. This also applied to the golden.cpp that was expected to be the only cpp in the root testing directory.

After fighting through these issues, we managed to apply the new functionalities in short order. We implemented a customized diff function for the new cases, added string generation to the menu, and implemented performance testing and code coverage statistics through Gcov and Gprof. We also added a way to test if something was looping for too long, with an option to change the default timeout after selecting to run tests.

Industrial Experience

8.4 Resumes

8.5 Industrial Experience Reports

8.5.1 Colter Assman

8.5.2 Samuel Carroll

8.5.3 Shaun Greunig