Autotester

Senior Design Final Documentation

Kernel_Panic

Anthony Morast Benjamin Sherman James Tillma
(Inherited from)The Obfuscators , Whitespace Cowboys

April 27, 2014

Contents

List of Figures

List of Tables

List of Algorithms

Mission

0.1 Obfuscators

To design and implement an automatic testing system that will completely and effectively meet all of the needs of our client

0.2 Whitespace Cowboys

Our mission is to provide reliable software for institutional use. We strive to improve the services and software that we provide to our customers.

Our belief is that in order to further this mission we must carefully apply the principles of sound engineering. To provide quality products we must hold ourselves to a high standard and thrive on quality work.

Our goal is the satisfaction of our customers. But beyond that is a need to satisfy the curiosity and ingenuity that drives us.

Our promise is the delivery of high quality software products.

0.3 Kernel_Panic

We make it our goal to provide high quality products that meet out customers needs. We strive to develop code that is reliable, maintainable, and dynamic.

We believe that we have done a good job when the customer gets more than expected. While we develop great products we are always striving to improve our process and ultimately our products.

Document Preparation and Updates

Current Version [2.0.1]

Prepared By:
Joseph Lillo
Daniel Nix
Elizabeth Woody

Kelsey Bellew
Ryan Brown
Ryan Feather

Anthony Morast
James Tillma
Benjamin Sherman

Revision History

Date	***Author***	***Version***	***Comments***
2/6/14	*Elizabeth Woody*	*1.0.0*	*Initial version*
2/13/14	*Obfuscators*	*1.0.1*	*Edited version*
2/16/14	*Lillo, Woody*	*1.0.2*	*Edited version*
2/17/14	*Daniel Nix*	*1.0.3*	*Edited version*
2/18/14	*Obfuscators*	*1.0.4*	*Final version*
3/21/14	*Ryan Feather*	*2.0.0*	*Initial running product*
3/23/14	*Ryan Feather*	*2.0.1*	*Finished untested product*
4/15/14	*Benjamin Sherman*	*3.0.1*	*Implemented gcov, gprof*
4/20/14	*James Tilma*	*3.0.2*	*String test cases and presentation errors*
4/21/14	*Anthony Morast*	*3.0.3*	*Menu driven testing and infinite loop testing*
4/26/14	*Kernel_Panic*	*3.0.4*	*Final version running*

1

Overview and concept of operations

This program is a utility for the testing of student software. It takes a students program and checks if it conforms to a suite of test cases. This document is included on the submission in order to facilitate use and maintenance of the utility. It covers the design, implementation, and usage of the provided software.

1.1 Scope

The scope of this document is meant to be comprehensive, giving users and managers the information necessary to use the product.

1.2 Purpose

The purpose of the Auto Tester is to allow batch-grading. Specifically, this program will traverse through a given directory, and run input through any found .cpps and test if the output is what was expected by the instructor. A summary file will be generated for each program found, and for each run of the Auto Tester, detailing which tests passed or failed, the percentage each program achieved overall, and/or a PASS or FAIL to dictate whether the student passed or failed the program. As well as the percentage of code that was tested by the test cases.

In addition to this, the Auto Tester will allow the user to easily generate random .tst files with the assosiated .ans file, given that they provide the Auto Tester with a 'golden cpp'.

1.2.1 Compiling

The product will be compiled using the g++ GNU linux compiler version 4.8.2. Provided is a Makefile that allows *make* to be typed into a terminal. After this is completed, the product will be ready for use.

1.2.2 Identifying Test Cases and Inputting

Once the student program is compiled, the Auto Tester searches the current directory and all sub-directories for files labeled as test cases (a .tst extension). Each test file is then used as input for the student program.

1.2.3 Checking Output

The Auto Tester will re-direct the programs output and evaluate it against the supplied test case. The results of the comparison (pass/fail) are recorded in a log file for each student, for each test case encountered.

1.2.4 Major System Component #1

Location of existing, applicable test cases for the program needing to be tested.

1.2.5 Major System Component #2

Run and test of a single program against located test cases

1.2.6 Major System Component #3

Record and summary of test results

1.3 Systems Goals

The goal of this system is to provide an automated testing application, designed specifically for professors testing submitted student programs. A user will be able to use the application to test a desired program against all applicable test cases the application can find in the directory tree related to that program. A time-stamped record will be created for each program found, to summarize the output of each test and to provide a general summary of the results as well as the percentage of code covered by the test cases. In addition, a summary log file will be generated including the names of the students, and what percent they got on the assignment, or if the simply failed the assignment.

The user (professor) will be prompted for *allowable time* which is the amount of time that a student program will be allowed to run before it is considered as in an infinite loop. When a student program is classified as being in an infinite loop it will be marked as failed.

Also, the use (professor) can choose to generate profiling information. When this option is enabled, the product will keep track of the amount of time spent in sub functions of student programs and supply this information in a file called *profile<timestamp>.out* which will be located in the same directory as a student source code.

1.4 System Overview and Diagram

The major system components listed above will, upon completion, combine to create this testing application. Upon running the application and providing it with the name of the desired target program, the application will complete the desired system goals via its major components.
Initially options will be gathered from the user. These options are the *allowable time*, *profiling enabled*, and whether or not they wish to *generate test cases*.
First, existing, applicable test cases will be found for the program needing to be tested.
Second, the program will be run against each test case input, and the resulting output will be compared to each desired test case output.
Last, the program will create a time-stamped record for each program tested, providing a reference of the output results and a numerical summary of the overall success rate, the percentage of the code tested by the test cases (*code coverage*), and, if enabled, run time statistics (*profiling*).
See Figure 1.1.

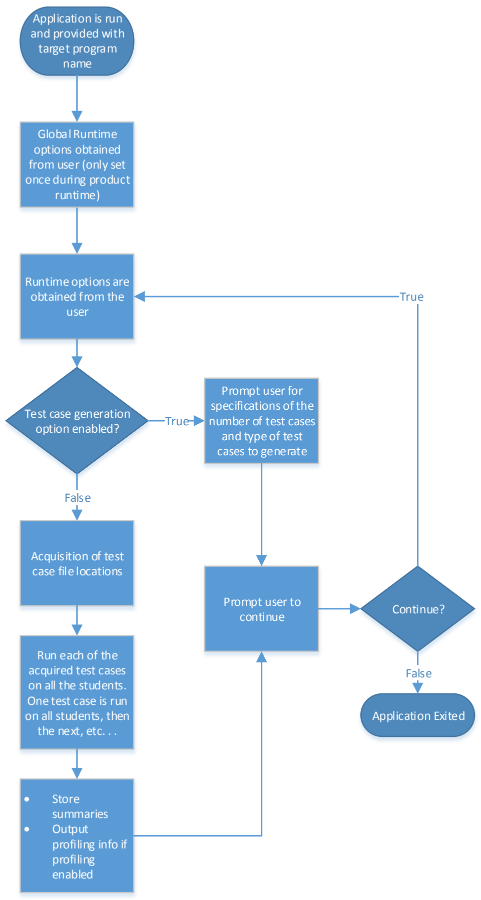
Application is run and provided with target program name
Global Runtime options obtained from user (only set once during product runtime)
Runtime options are obtained from the user
True
Test case generation option enabled?
True
Prompt user for specifications of the number of test cases and type of test cases to generate
False
Acquisition of test case file locations
Prompt user to continue
Continue?
False
Application Exited
Run each of the acquired test cases on all the students. One test case is run on all students, then the next, etc. . .
Store summaries
Output profiling info if profiling enabled

Figure 1.1: The Testing System Diagram

1.5 Technologies Overview

The development team used the Agile Software Development Method, via the Scrum framework, to develop this system. This incremental development method was the required development method for this project. Since the system was expected was built for a Linux environment, it was written and tested on a Linux Platform using generic text editors and the G++ compiler.

Integrated into the product was GCOV. GCOV was used to obtain the amount of student source code that was exercised by the test cases.

An optional technology used in the product was GPROF which is implemented when the user enables code profiling.

See Table 1.1.

Software Development Method	Agile Software Development Method
Planning and Organization	Trello Project Management Application
Platform	Linux
Language	C++
Code Coverage	GCOV
Code Profiling	GPROF
IDE	none (generic text editors)
Version Control	Git

Table 1.1: The Development Methods and Technologies Table

2

Project Overview

This section provides some housekeeping type of information with regard to the team, project, etc.

2.1 Team Members and Roles

2.1.1 Obfuscators

- Joseph Lillo - Technical Lead
- Daniel Nix - Product Owner
- Lisa Woody - Scrum Master

2.1.2 Whitespace Cowboys

The team members of the Whitespace Cowboys were Kelsey Bellew, Ryan Brown, and Ryan Feather.

- Kelsey Bellew was the Scrum Master.
- Ryan Brown was the Product Owner.
- Ryan Feather was the Technical Lead.

2.1.3 Kernel_Panic

The team members of Kernel_Panic were Anthony Morast, Benjamin Sherman, and James Tillma.

- Anthony Morast - Technical Lead
- Benjamin Sherman - Product Owner
- James Tillma - Scrum Master

2.2 Project Management Approach

This project was managed using the Agile Software Development Method, via the Scrum framework. Trello, a web-based project management application, was used to assign tasks and keep track of the product backlog. For version control, Git was used.

The sprint length for the development of this system was three weeks, with only one sprint being necessary for completion. The user stories were provided by the product owner, Benjamin Sherman, who actively communicated with the future application user. These were used to create the product backlog and to determine the requirements and components of the system.

2.3 Phase Overview

The program was developed through four different phases:

1. Implement a directory crawl to find all of the test cases.

2. Run the program to be tested, using the input from the test case. Store the result in an output file.

3. For each test case found, compare the tested program's output file to the desired output file (included with each test case).

4. Create a time-stamped log file to retain the output files and results of the test. This will include a record of the success and failure rate of the test.

5. Generate test files and create answer files. Answer files will be generated when the test files are piped into the golden cpp file.

6. Add profiling and code coverage functionality.

7. Modify *diff* to the effect that student solutions need not be exact matches to the answer.

8. Limit runtime of a student program. Consider the student program to be in an infinite loop if it runs too long.

2.4 Terminology and Acronyms

See Table 2.1

GNU and GNU Tools	The tools provided in most GNU\Linux environments
Agile Methodology	An approach to project management and software development `agilemanifesto.org`
C++ Template Libraries	A built-in set of classes used for storing data.
GCOV	GNU C++ code coverage utility
GPROF	GNU C++ code profiling utility

Table 2.1: Defining Some Important Terms

3

User Stories, Backlog and Requirements

3.1 Overview

This section contains several user stories, a backlog, and a list of requirements, of both the project and the user, and the user's equiptment for using the project. This chapter will contain details about each of the requirements and how the requirements are or will be satisfied in the design and implementation of the system.

The user stories are provided by the stakeholders.

3.1.1 Scope

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

3.1.2 Purpose of the System

The purpose of the product is to allow the user to run pre-written and product generated auto tests on a directory or program of their choosing, and to provide a log to the user of which tests passed and which tests failed for each program that is found within the given directory. A PASS, FAIL, and or percentage is written to a log placed within a found program's directory.

In addition, the purpose of the product is to allow the user to generate a given number of random tests with user given parameters.

3.2 Stakeholder Information

3.2.1 Customer or End User (Product Owner)

Whitespace Cowboys: The Product Owner is Ryan Brown. The Product Owner in this case is responsible for getting project specifications from the Customer and keeping the team members up to date with the Customer's user stories. The Product Owner is also responsible for managing and prioritizing the product backlog.

Obfuscators: Daniel Nix is the Product Owner of this project. He will clarify and define the End Users' needs and requirements for this product, as well as establish and prioritize the product backlog.

Kernel_Panic: The Product Owner is Benjamin Sherman. He will relay the customer specifications and requirements.

3.2.2 Management or Instructor (Scrum Master)

Whitespace Cowboys: The Scrum Master is Kelsey Bellew, and will drive the Sprint Meetings, keep a log of meetings and schedules. She will also deal with any and all unforseen issues causing dilemmas to the completion of the project.

Obfuscators: Lisa Woody is the Scrum Master for the project. She is responsible for scheduling the project meetings, as well as determining and assigning the tasks necessary to deliver the required product.

Kernel_Panic: The Scrum Master is James Tillma. He is in charge of driving the sprint meetings, setting deadlines, and handling the unforeseen obstacles that may be encountered.

3.2.3 Developers –Testers

Whitespace Cowboys: The Technical Lead for our team is Ryan Feather. He coordinates the merging of user stories and ensures program stability throughout development.

Obfuscators: Joseph Lillo is the Technical Lead and for the project. He will be responsible for the high level design and final testing of the program.

Kernel_Panic: Anthony Morast is the teams Technical Lead. He will be responsible for the maintaining product stability throughout the development process.

3.3 Business Need

This software must simplify and automate the grading process. The product will meet that need and enable the end user to not only see the immediate results of a test, but also to maintain a dated record of each test and its detailed output.

3.4 Requirements and Design Constraints

3.4.1 System Requirements

- The program must build and run in a Linux environment.
- The source code for programs to be ”tested” will be in C++.
- Source code will be in the ”root” directory and its subdirectories.
- A bash shell will be used to run the program

3.4.2 Network Requirements

There are no network requirements. This project does not use the internet unless the program it is testing uses the internet.

3.4.3 Development Environment Requirements

- The application must be written in C++
- All work must be done in Linux
- System calls (gcc, etc.) may be used in the application.
- The test case input will be stored in a .tst file. The accompanying desired output will be stored in a .ans file.
- Testing output should be in one log file which contains:
  Test output results (i.e., 52 different tests will produce 52 lines in the .log file)
  Number passed, Number failed, Percentage of success
- Code coverage percentage will be each student detailed log file.
- Code profile information from a student program will be located in the same directory as a students source code.

3.4.4 Project Management Methodology

There is only one customer for this application. This customer may place constraints on meeting times and frequency of required progress reports. Aside from customer requests meeting times and reports will be managed by the scrum master. For the first iteration, we need to compile and test only one program.

- Trello, a free web-based project management application, will be used to keep track of the backlogs and sprint status.
- All parties have access to the Sprint and Product Backlogs, via Trello.
- This particular project will be encompassed by only one Sprint.
- The Sprint Cycle of this project is three weeks.
- There are no restrictions on source control.

3.5 User Stories

3.5.1 User Story #1

As a user of the program, I would like to be able to specify a program to grade that will test the program and create a record of easy to understand output.

3.5.1.a User Story #1 Breakdown

This application will be targeted towards instructors needing to test submitted student programs against applicable test cases. The application will be run from the command line, using the name of the program to be tested as an inital parameter. For each existing test case, the program will be run using that test case's .tst file as input. The output will be recorded and compared to the accompanying answer file for that test case. A summary of the results must accompany the recorded output in the log file created each time the application is run.

3.5.2 User Story #2

As a user of the program, I want to be able to test the program against test cases located in the directory tree of that program.

3.5.2.a User Story #2 Breakdown

Each program to be tested will be placed into a directory that forms the root of the directory tree related to that program. The user will have the ability to add and remove test cases (called case#.tst) and their accompanying desired output file for comparison (called case#.ans). The application must find all of the applicable test cases and accompanying answer files, and test the specified program against all the test cases found.

3.5.3 User Story #3

As a user of the program, I want to be able to fix the problems in the program I am testing and rerun the test without losing the previously created log file.

3.5.3.a User Story #3 Breakdown

A new log file containing the tested program's outputs and summary must be created each time the application is run. This will be an important feature, enabling the user to alter the program and visualize the effects of the alterations on the program's output and testing summary. Each log file will be date-stamped to achieve this result.

3.5.4 User Story #4

User wants to be able to run multiple tests on multiple programs.

3.5.5 User Story #4 Breakdown

The Auto Tester must be able to take a directory, and do a directory crawl while finding all .tst and .ans files contained within the given directory, and then do a second directory crawl to find all programs and run all tests agains all found programs.

3.5.6 User Story #5

User wants to be able to run the Auto Tester without giving it specific program(s) to test.

3.5.7 User Story #5 Breakdown

The Auto Tester must be able to find a program or programs just given a directory, and must be able to differenciate between test files, answer files, student programs, and the golden cpp.

3.5.8 User Story #6

User wants to be able to have the option to have the tester auto generate tests.

3.5.9 User Story #6 Breakdown

The Auto Tester will prompt the user if they want to auto generate a test; they will be prompted to enter different parameters which will be used to generate a random .tst, the contents of which will be run against the golden cpp to generate a .ans file.

3.5.10 User Story #7

User wants to be able to set certian tests as 'must pass' tests; and if these tests are not passed, then the student is not given a percentage, but rather a FAIL.

3.5.11 User Story #7 Breakdown

The product must be able to detect if a test is marked as a 'must pass' test, and when the Auto Tester runs that test against a student's program and that program happens to fail that test, the Auto Tester must be able to recognize the failure of a critical test and cease the running of any further tests. It must then not give the program a percentage of passed tests as it might for other student programs, or as it did in the previous itteration of this product, and only output 'FAIL'.

3.5.12 User Story #8

As a user of the product I want to see the percentage of a students code that was exercised (tested) by the test cases in a students log file.

3.5.13 User Story #8 Breakdown

The product will use GCOV to monitor and retrieve code coverage information. This will not be an option but will be stored in each students log file.

3.5.14 User Story #9

As a user I want to be able to allow leniency on how close a students program output and the answer must match.

3.5.15 User Story #9 Breakdown

The user of the product can choose whether to allow slight formatting errors on a students program output. There can be minor differences between a test case answer and a student program answer.

3.5.16 User Story #10

As a user I want to be able to fail a student on a test case if their code goes into an infinite loop.

3.5.17 User Story #10 Breakdown

That this appears to be the unsolvable halting problem is not the case. Instead the user of the product will prompted for an allowable time period that a student program will be given to run one test case. If a students program doesn't complete in the specified time, then it will fail the test case under an assumed infinite loop.

3.6 Research or Proof of Concept Results

This section is reserved for the discussion centered on any research that needed to take place before full system design. The research efforts may have led to the need to actually provide a proof of concept for approval by the stakeholders. The proof of concept might even go to the extent of a user interface design or mockups.

3.7 Supporting Material

This document might contain references or supporting material which should be documented and discussed either here if approprite or more often in the appendices at the end. This material may have been provided by the stakeholders or it may be material garnered from research tasks.

4

Design and Implementation

The testing program has three main features which needed to be included. These were finding the location of files containing existing test cases for the tested student programs to run, running the single student programs against the located test cases, and recording and summarizing the test results. C++ was used to write all features.
See Figure 1 below.

Algorithm 1 Test C++ Program

```
Require: Class directory name from command line
Ensure: Test cases are found
  if not in lowest level directory then
    Find .tst files in current directory
    Add file paths to .tst file to vector
    Drop into each subdirectory
  else
    Return to parent directory
  end if
Ensure: Test cases are run
  Compile C++ source code
  while not all test cases have been run on all students do
    Run program against next test case
    Record if test passed/failed or infinite loop
    Record percentage of code coverage
    if Code profiling enabled then
      Record code profiling information
    end if
  end while
  Output .log file with test statistics
```

4.1 Find .spec file, generate cases

4.1.1 Technologies Used

The technologies used in the component is basic C++.

4.1.2 Component Overview

This component will find a .spec file in any of the directories. If the file exists the user is given the oprion to generate test files to test menus.

4.1.3 Phase Overview

This phase will add .tst files to the directory to test the students source code.

4.1.4 Design Details

This component finds a .spec file. If it exists, the user will be asked if they want to generate test files to test menus. If the user chooses yes, they will be asked how many files to generate, the max value, and the min value. Then test files are generate so that the values are piped in as input and the student's menu is tested.

4.2 Find .tst Files in Subdirectories

4.2.1 Technologies Used

The technology used in this component is basic C++.

4.2.2 Component Overview

This component uses a simple recursive directory crawl to find all .tst files in all subdirectories in the directory passed in via command line. These files will be used to produce output to test student code.

4.2.3 Phase Overview

This phase will find and queue the .tst files to be used later in testing the students source code.

4.2.4 Architecture Diagram

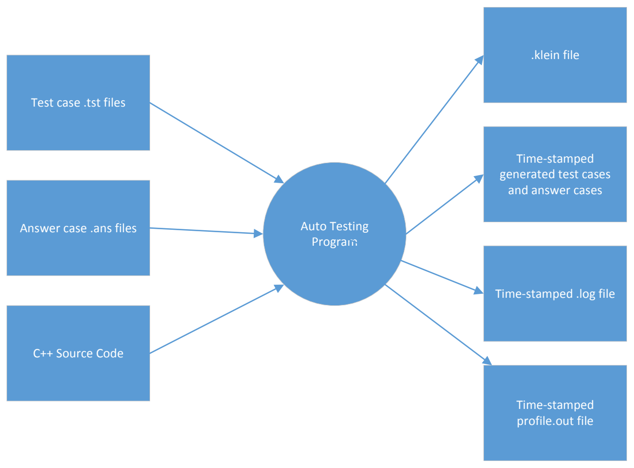

Figure 4.1: Architecture Diagram for the Tester Application

4.2.5 Data Flow Diagram

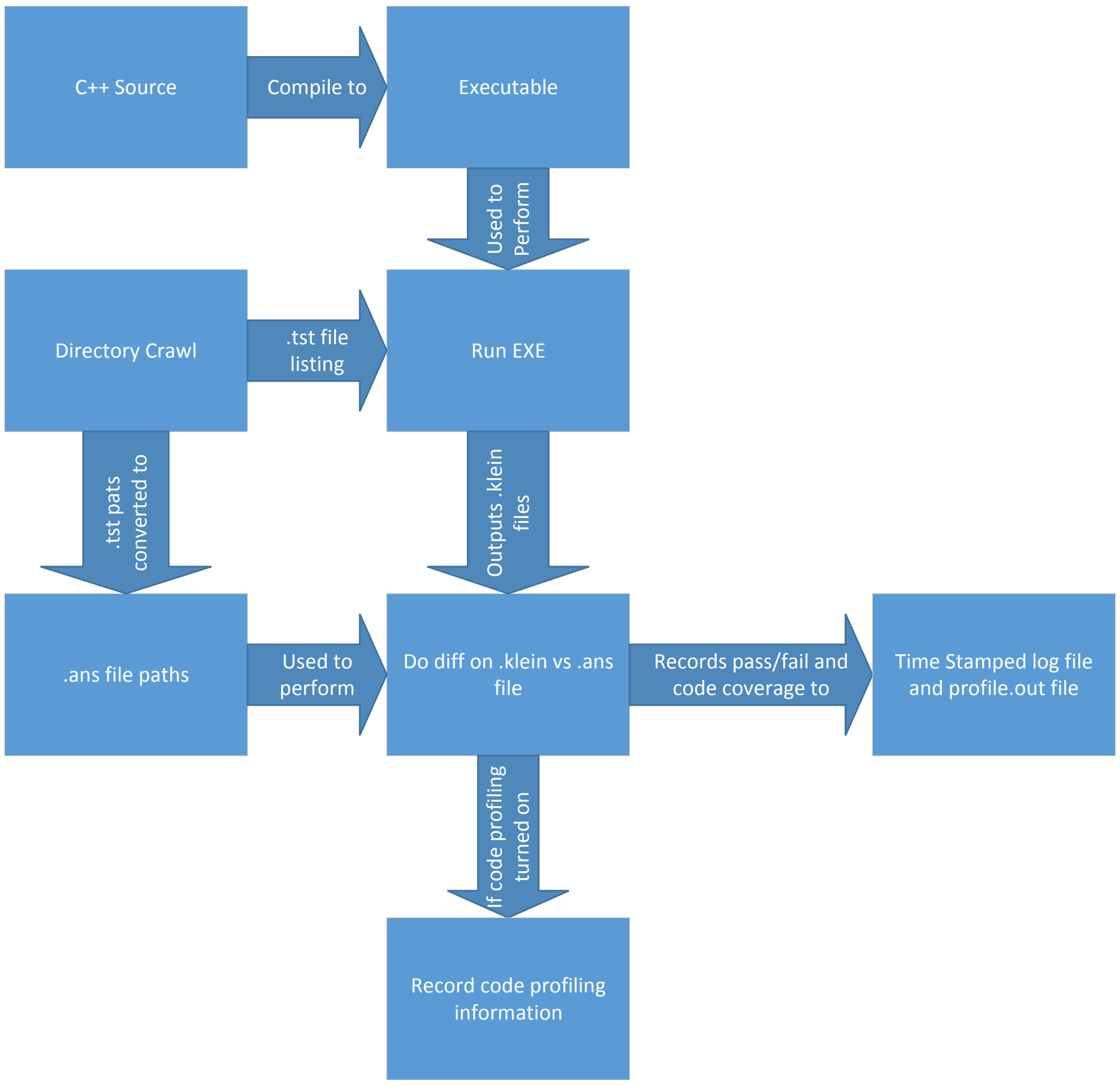
C++ Source
Compile to
Executable
Used to Perform
Directory Crawl
.tst file listing
Run EXE
.tst pats converted to
Outputs .klein files
.ans file paths
Used to perform
Do diff on .klein vs .ans file
Records pass/fail and code coverage to
Time Stamped log file and profile.out file
If code profiling turned on
Record code profiling information

Figure 4.2: Data Flow Diagram for the Tester Application

4.2.6 Design Details

The following function does the directory crawl and stores the relative path to .tst file in the ”dest” vector which is passed by reference. This is the recursive function that searches all sub-directories of the directory root which contains the executable program:

```
// Recursive Directory Crawl
void TestSuite::dirCrawl(string targetExt, string dir, vector<string> &dest)
{
    // Open current directory.
    DIR * proc = opendir( dir.c_str() );

    if (NULL == proc)
    {
        return;
    }

    // Read current directory.
    dirent * entry = readdir(proc);

    do
    {
        // Make recursive calls to sub directories
        if(DT_DIR == entry->d_type)
        {
            string name = entry->d_name;
            if ( "." != name && ".." != name )
            {
                string newDir = dir + "/" + entry->d_name;
                dirCrawl( targetExt, newDir, dest );
            }
        }
        // Watch for files with .tst extension
        else if ( DT_REG == entry->d_type )
        {
            string fileName = entry->d_name;
            unsigned int extPos = fileName.rfind(".");
            if ( extPos != string::npos )
            {
                string ext = fileName.substr( extPos );
                if ( ".tst" == ext )
                {
                    fileName = dir + "/" + fileName;
                    cout << fileName << endl;
                    dest.push_back(fileName);
                }
            }
        }
    }while(entry=readdir(proc));

    closedir(proc);
}
```

Once this function has finished executing the destination vector contains a listing of all paths to .tst files to be run by the program being tested.

4.3 Test Single Program Against Found Test Cases

4.3.1 Technologies Used

The only technology used in this section is C++ code.

4.3.2 Component Overview

The purpose for this section is to compile the C++ code, loop through the list of .tst files in the directory, and check if the program output matches the .ans file output.

4.3.3 Phase Overview

Testing a program against located test cases is necessary in the second and third phases because we must be able to check the program's output against the .ans file to see if each test was a success or failure before program output can be prepared.

4.3.4 Architecture Diagram

Due to the small size of the application, a single diagram describing the architecture of the program is provided in Figure 4.1

4.3.5 Data Flow Diagram

Due to the small size of the application, a single diagram describing the data flow for the program is provided in Figure 4.2.

4.3.6 Design Details

Testing the program may be broken down into three sections:

- Compiling source C++ code
- Running C++ code against test case
- Determining if program output was correct for test case

The following function compiles the program using a system call to g++:

```
//Function to compile c++ source code based on filename
bool TestSuite::compile_code( string filename )
{   string compile_instruction = "g++ "; // Form the string for the system call
    compile_instruction += filename;
    compile_instruction += " -o test_prog";

    system( compile_instruction.c_str() );

    return true;
}
```

To run the executable against a test case another system call is made redirecting input from the test file and output to the ".klein" file:

```
//Function to run c++ souce with redirected input/output
bool TestSuite::run_code( string test_file )
{   string run_instruction = "./test_prog < ";  //Form the string for the system call
    run_instruction += test_file;
    run_instruction += " > test_out.klein";

    system( run_instruction.c_str() );
    return true;
}
```

Finally, the program output file needs to be compared to the .ans file. This is also done with a system call to diff:

```
//Function to do diff on answer file and test program output file
bool TestSuite::correct_answer( string ans_file )
{  string diff_instruction = "diff test_out.klein ";
    diff_instruction += ans_file;

   return (! system( diff_instruction.c_str() ) );
}
```

The only change to compare all the test files is to place the preceding functions in a loop and pass each new test file as a parameter to run_code() which is done in run_test() as follows:

```
// Runs program with input from test files in testFiles vector.
void TestSuite::runTests()
{  numCorrect = numWrong = 0;
    pair<string, bool> result;

    vector<string>::iterator it;           // Iterate over test files.
    for ( it = testFiles.begin(); it != testFiles.end() ; it++ )
    {
        // Run program with given test file.
        run_code(*it);

        // Determine corresponding answer file.
        string ans = *it;
        ans.replace(ans.end()-4, ans.end(),answerExtension);

        // Populate results vector and counters.
        result.first = *it;
        if (correct_answer(ans))
        {
            result.second = true;
            numCorrect++;
        }
        else
        {
            result.second = false;
            numWrong++;
        }
        // Add results to vector.
        results.push_back(result);
    }
}
```

Upon completion, the TestSuite object contains counters for the number of correct and incorrect test cases to be used when outputting the log file.

4.4 Record and Summarize Test Results

4.4.1 Technologies Used

The only technology used in this section is C++ code.

4.4.2 Component Overview

After the test cases have been run a timestamped .log file needs to be produced. It should include if each test case passed or failed as well as the percentage of correct files along with the total number of correct and incorrect test cases.

4.4.3 Phase Overview

This component is included in the fourth phase of production. It was done last because the full output file could not be produced without knowing the results of all the test cases.

4.4.4 Architecture Diagram

Due to the small size of the application, a single diagram describing the architecture of the program is provided in Figure 4.1

4.4.5 Data Flow Diagram

Due to the small size of the application, a single diagram describing the data flow for the program is provided in Figure 4.2.

4.4.6 Design Details

The formatted output is produced by the outputLogFile() function. It lists if each test case was a pass or failure along with overall test statistics. In each student directory a gcov and gprof file will be generate to analyze performance and code coverage.

5

System and Unit Testing

This section describes the approach taken with regard to system and unit testing.

5.1 Overview

A file containing test case files and program files was provided by the client. However these directories contained no .spec files, no prgograms with infinite loops, and no programs that take string input. Therefore, files to test these parts of the program were created.

5.2 Dependencies

There are no external dependencies or frameworks for the program. Unit testing was carried out individually because the program was written without a predefined framework on which to test.

5.3 Test Setup and Execution

The test cases listed below were ran against the program to ensure proper functionality. Edge cases were considered and tested for each individual test as applicable.

- Program successfully executes given parent directory via command line
- Finds .spec file and asks to generate test cases if one exists
- Checks for proper menu driven testing user input and .tst file output
- Ensure the program will find infinite loops, break out of the loop, and adjust the student score.
- Checks that random strings are generated properly for .tst files.
- Produces gcov and gprof files for each of the students and places them in the correct directory.
- Properly display menu and error-check user input
- Compiles all .cpp files, to include the "golden .cpp"
- Properly crawls through the directory finding .tst files
- Runs student's code and properly tests the testcases
- Generates random numerical (both floats and ints) test cases
- Runs the generated test cases against the "golden.cpp" and successfully generates accompanying .ans files
- Generates each individual student's grade file, to include a class summary grade file

6

Development Environment

The basic purpose for this section is to give a developer all of the necessary information to setup their development environment to run, test, and/or develop.

6.1 Development IDE and Tools

Development for this project was done in Qt Creator and other simple text editors such as VIM and Gedit. Two linux command line tools, gcov and gprof, were used in the project. They were used to create files the user can analyze and determine a student's codes performance and what percent of the code was covered.

In Sprint 2, the Qt Creator environment was dropped. In sprint 3 however, it was once again used by part of the group.

6.2 Source Control

The source control used in this project was Github for both Windows and Linux. A developer could connect to it by several ways; the first of which is to go to the Github website, where they were included as contributors to the repository where the code and documentation was stored. The second was to use either Linux or Windows to checkout the repository and use the push and pull functions off git to keep code and documentation updated.

6.3 Dependencies

Dependencies for this program include the c standard library and the POSIX API. The program also takes advantage of several Linux system calls. Therefore, it is assumed that it will be run in a Linux environment.

6.4 Build Environment

The program compiles with GCC into a single executable. This can either be done by using the Makefile provided with the source code.

6.5 Development Machine Setup

This program is designed to run on a Linux based machine. It was developed on machines running Fedora 20 and Fedora 19. On these developement machines we ensured the GCC compiler was up to date as to not cause any issues when compiling and running the program.

7

Release – Setup – Deployment

This section contains subsections regarding specifics in releasing, setup, and/or deployment of the system.

7.1 Deployment Information and Dependencies

Dependencies for this application include the c standard library and the POSIX API, both used for the file system access in the test case finding portion of this program. This application also relies on Gcov and Gprof. Lastly, it relies on many Linux system calls.

7.2 Setup Information

A makefile will be provided with the application, which can be run using the command 'make' within the directory in which the makefile exists. To ensure a full rebuild of the system, run "make" with the "-B" argument.

The project is run by:
./tester [-g or -r] ⟨ directory ⟩ [-p or empty]

The directory is the place where the test files, student programs, and golden source are stored. Student code is in subdirectories, and test files can be placed anywhere. Any test files generated by the program will be stored in the "tests" subdirectory.

7.3 System Versioning Information

The system will be versioned with each Sprint of the class, incrementing the first decimal.

In addition to this, when a working version of the project was developed, that version was put into a branch with a time stamp. Additionally, every time a working bit of code was developed, it was put into the current branch with a description of what was currently working.

8

User Documentation

This section will contain the basis for any end user documentation for the system, and will cover the basic steps for the setup and use of the system.

8.1 User Guide

Usage of the Auto Tester is primarily concerned with the format and placement of the test case files. Test case files must be located in the same directory as the students program source, or in a subdirectory there of. The golden cpp, if present, must be given in the commandline, or be located in the first directory given. The .spec file must be present to enable menu driven testing.

8.1.1 Test Case Files

Files that contain the test cases themselves need to be given a .tst extension. The filename proceeding the extension will be used as the name of that test in the log file. The contents of the file will be the raw input to the student's program.

8.1.2 Expected Output Files

For each test case file, there needs to be a corresponding file that contains the expected output. This file must have the same name as the test case, but instead have a .ans extension.

8.1.3 Auto Generated Tests

The option to auto generate tests with a given 'golden cpp' will be available to the user. The user will run the Auto Tester with a given program, and then will go through a simple text based menu that will determine the parameters for the auto generated tests. The code will then place the .tst and .ans files in a subdirectory called "tests". The answer files will be generated by running the test files through the golden cpp. This product features menu driven testing, however, a .spec file must be present to enable this feature.

8.1.4 Results

The results for each student will be placed in a file with a .log extension, within the given student's directory. In addition, a complete log with the names of all the students and their pass/fail status and/or grade will be placed in the head directory. The filenames will be timestamped. Gcov information is put in each student log file. Gprof information, if enabled, will also be appended to the student log file.

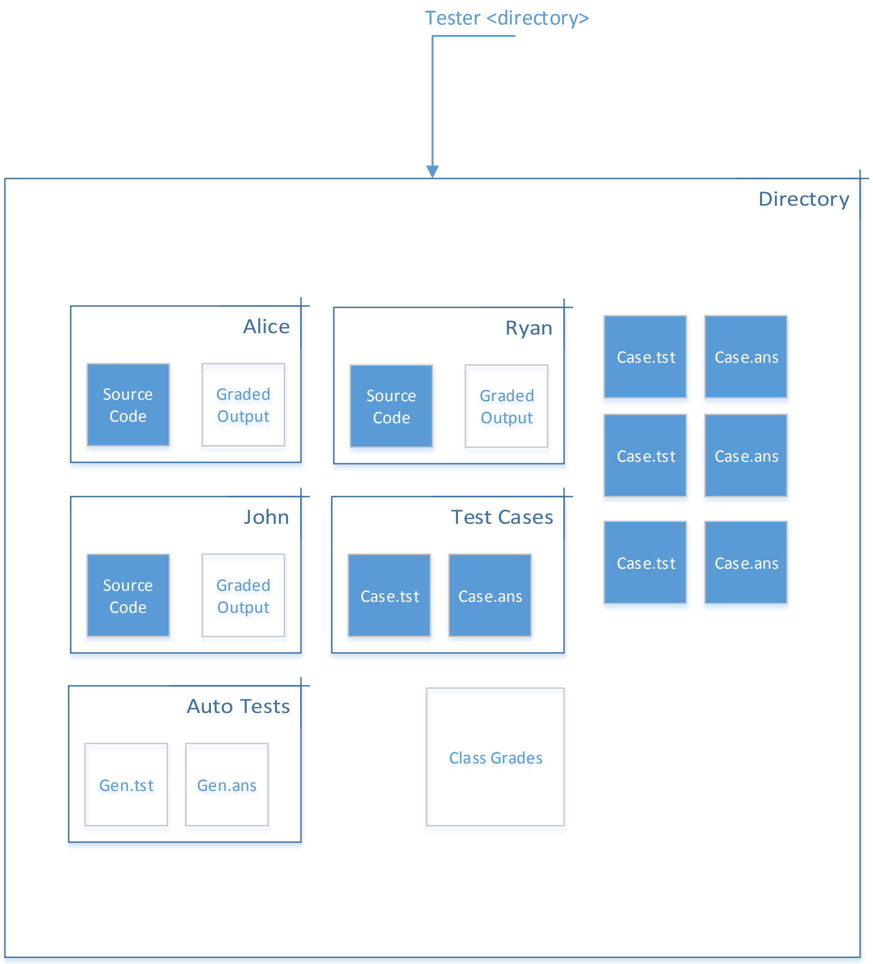

Figure 8.1: Example directory structure for runnning the tester

8.2 Installation Guide

To use the application, compile the source code using the supplied makefile.

After this is completed, the user should have an executable file they can use by invoking the following: ./tester [-g or -r] ⟨ directory ⟩ [-p or empty]

8.3 Programmer Manual

The code contained in the .cpp file is written in c++ and is to be compiled with g++.

9

Class Index

9.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

10

Class Documentation

10.1 TestSuite Class Reference

This class provides an interface for compiling and testing c++ programs against given test files.

`#include <testsuite.h>`

Public Member Functions

- TestSuite ()
- bool initTest (string program, string tstExt, string ansExt)
  Initialize a testing session. Compiles the given program and locates all test and answer files.
- void runTests() *Run the test.*
- void outputLogFile()
  Output a log file with results of test.
- bool menu_tests(string spec_file_path)
  generate menu driven test cases.
- void dirCrawl(string targetExt, string dir, vector<string>& dest)
  get all files of target extension in a directory.
- void helperfunc()
  Run non menu drive test case generation.
- void createSummary()
  Record class test summary.
- void presentatioMenu()
  Run menu that gets options from user for either ignoring or not ignoring presentation errors.

10.1.1 Detailed Description

This class provides an interface for compiling and testing c++ programs against given test files.

10.1.2 Member Function Documentation

10.1.2.a string TestSuite::getTestProgram ( ) const [inline]

Returns the name of the program that will be tested.

Returns

Name of program.

10.1.2.b bool TestSuite::initTest (string *program*, string *tstExt*, string *ansExt*)

Initialize a testing session. Compiles the given program and locates all test and answer files.

Parameters

program	Name of program to test.
tstExt	File extension of test files to search for.
ansExt	File extension of answer files to search for.

Returns

Success or Failure

10.1.2.c bool TestSuite::reset ( )

Reset private member data after completing a test session.

Returns

Success or Failure

10.1.2.d void TestSuite::setProgram (string *program*) `[inline]`

Set the program to compile and test.
Parameters

program	The name of the program.

The documentation for this class was generated from the following files:

- testsuite.h
- testsuite.cpp

Acknowledgement

Thank you Dr. Logar for being acting as our customer.

Sprint Reports

10.1 Sprint Report #1

As the development of this system was planned to be completed in one sprint, this is the final sprint review for the project. The review was conducted with the product owner, the technical lead, and the scrum master.

10.1.1 Code Development

The following were designed and written by Joseph Lillo:

TestSuite Class Declaration - This class provides an interface for compiling and testing c++ programs against given test files.

dirCrawl - a function used to perform a recursive Directory Crawl

The following were designed and written by Daniel Nix:

compile_code - a function used to compile c++ source code based on filename

run_code - a function used to run c++ souce with redirected input/output

correct_answer - a function used to do diff on answer file and test program output file

The following were designed and written by Lisa Woody:

Software Documentation and formatting - the final software write up for the project

Log file output functions - functions later integrated into other functions upon team review of the program.

The following were designed and written by the Obfuscators as a team:

runTests - a function used to run a program with input from test files in the testFiles vector

reset - a function used to clear member data associated with testing session.

10.1.2 Final Product Review

The final application, having passed the final testing stages, was demonstrated.

- Has every phase of the project been completed?

Yes, each of the four phases of the project have been completed.

- Have the phases been successfully integrated into the final product?

Yes, the phases are completely integrated into the final product.

- Can the application run the client-provided test files without error?

Yes, the application compiles and no errors were found.

- Does the final output match the intended and requested output?

Yes, the outputted log file matches the requested format.

Next, the documentation was reviewed.

- Has the documentation been completely filled out and reviewed by each team member?

Yes, the documentation has been completed and reviewed.

- Does the product meet the requirements and needs of each user story?

Yes, the product fulfills the needs of each user story laid out in the documentation.

- Does the product satisfy the product backlog and is it deemed deliverable by the product owner?

Yes, the product owner is deemed complete and deliverable by the product owner.

The team is happy with this sprint, and feels that next time it will be prepared for the newly learned Agile methodology and documentation style.

10.2 Sprint Report #2

This sprint lasted from 2/21 to 3/22. The members of the Whitespace Cowboys recived the Obfuscator's first sprint code, analyzed it, and turned in a report to Dr. Logar conserning the workability and documentation of the given materials. One official scrum meeting was held to hold a planning poker and to assign coding tasks. These are were reflected using Trello cards.

Several unofficial meetings were held, the last of which was used to go over what had not yet been completed and how the last steps would be taken to tie up the second sprint. During this last meeting, an unofficial code review was held concerning the completed functions, and github issues were worked out.

The code for the second sprint was officially completed on 3/23, with several last minute revisions made. Documentation was double checked, and the team made an effort to make sure the code was well documented and at least similar in coding standard.

10.3 Sprint Report #3

This sprint lasted from 3/24 to 4/25. The members of Kernel_Panic received program code and documentation shortly after the beginning of this sprint. We turned in an assessment of the materials we received.

Two meetings were held ass soon as the details of the sprint were on back to back class days. During these meetings we discussed the new features for this sprint and divided up responsibilities.

On April 23, after Easter, Kernel_Panic met to discuss the progress that had been made. Then again on April 25 to manage the merging of branches and confirm everything went smoothly. Lastly on April 26 to finalize our Sprint 3 project and submit.