Automatic Program Tester

Software Engineering Documentation

Lounge Against the Machine

Joe Manke Alex Wulff Adam Meaney

February 19, 2014

Contents

List of Figures

List of Algorithms

Mission

To create a program capable of automatically grading programs written for lower-level programming courses such as CSC150 and CSC250. Given the name of a source code file, it will compile the code and find all test cases (notated by a ".tst" extension) in the parent and any child directories of where this program's executable is located. The target program will be run against each test case and be recorded as a pass or fail based on the difference between expected and actual output. After all the tests are executed, the results, including statistics, will be written to a log file.

Document Preparation and Updates

Current Version [1.0.0]

Prepared By:
Joe Manke
Alex Wulff
Adam Meaney

Revision History

Date	***Author***	***Version***	***Comments***
2/18/14	*Joe Manke*	*1.0.0*	*Initial version*

1

Overview and concept of operations

The overview should take the form of an executive summary. Give the reader a feel for the purpose of the document, what is contained in the document, and an idea of the purpose for the system or product.

1.1 Scope

This document covers the design and implementation of the program.

1.2 Purpose

The purpose of this program is to assign pass/fail grades to simple programs run against a number of test cases.

1.2.1 Compilation

In order to test a program, it must be compiled into an executable.

1.2.2 Directory Crawl

This component finds all of the test cases.

1.2.3 Test Execution

The target program will be executed against a number of test cases.

1.2.4 Result Logging

Results of the tests must be recorded for the user to read.

1.3 Systems Goals

The goal of this system is to perform automated testing and grading for C++ programs.

1.4 System Overview and Diagram

The program begins by finding all test files. Then, the target program is compiled. Next, all of the test cases are executed. Finally, the results are written to the log file.

1.5 Technologies Overview

Developed in Linux in C++, using the g++ compiler.
Documentation created using TexWorks and MikTex.
Flowchart created using Gliffy.

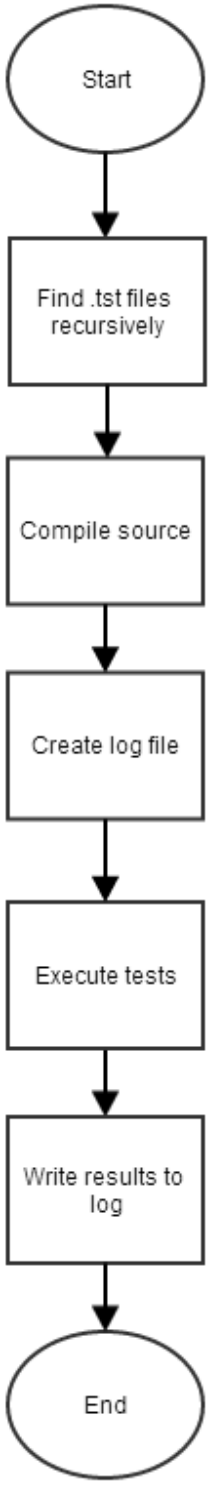

Figure 1.1: Program flowchart

2

Project Overview

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

2.1 Team Members and Roles

Joe Manke - Scrum Master
Alex Wulff - Technical Lead
Adam Meaney - Product Owner

2.2 Project Management Approach

This project was completed in 2 one-week sprints, managed on Trello.

Source control was done through GitHub.

Sprint 1 consisted of establishing our source control repository and implementing the functions necessary prior to actual test execution. This included directory traversal to find .tst files, compiling the source code, and creating the log file.

Sprint 2 consisted of combination of these functions and execution of tests.

2.3 Phase Overview

2.3.1 Design

The design phase consisted of determining specifications from the client, Dr. Logar. With the specifications determined, we devised a flowchart.

2.3.2 Implementation

The implementation phase was to write the code to fit the specifications of the program.
This phase was split over two sprints, as described in section 2.2 above.

2.3.3 Testing

After the code was written, it was tested to ensure proper functionality.

3

User Stories, Backlog and Requirements

3.1 Overview

3.2 Stakeholder Information

The main stakeholder for this program is Dr. Logar, but other stakeholders may include other computer science professors and TAs.

3.2.1 Customer or End User (Product Owner)

In his role as Product Owner, Adam Meaney was the liason between our team and Dr. Logar.

3.2.2 Management or Instructor (Scrum Master)

As Scrum Master, Joe Manke assigned tasks and managed the Trello board. He also wrote documentation.

3.2.3 Developers –Testers

Alex Wulff was the Technical Lead, and shared development duties with Adam. All three members of the team participated in code review and testing.

3.3 Business Need

The program must be able to compile and execute another program given only the name of the source code file.

The program must be able to find all test cases in the parent directory where the program is executed, and all of its child directories.

The program must be able to determine success or failure of individual tests and record results.

This program meets all of these needs. Program compilation and execution are handled using system calls. Test cases are found through a recursive directory crawl. Test success is determined by performing a diff between expected and actual output and recorded in a log file.

3.4 Requirements and Design Constraints

3.4.1 System/Development Environment Requirements

The program must be written in C++ and is required to work on the MCS department's Linux boxes, using Fedora. This necessitated that the program be developed and tested in Linux.

3.4.2 Project Management Methodology

- Trello will be used to keep track of the backlogs and sprint status.
- The development team and the primary stakeholder (Dr. Logar) will have access to the Sprint and Product Backlogs.
- The number of sprints and sprint length were left to the development team. We decided upon two one-week sprints.
- GitHub is the recommended source control.

3.5 User Stories

3.5.1 User Story #1

As a user, I should be able to compile and execute a program by providing the source code.

3.5.2 User Story #2

As a user, I should be able to locate test cases in child directories.

3.5.3 User Story #3

As a user, I should be able to see individual and aggregate results of test cases.

3.5.4 User Story #4

As a user, I should be able to run tests against a program multiple times without overwriting previous test results in the log.

4

Design and Implementation

This section is used to describe the design details for each of the major components in the system. This section is not brief and requires the necessary detail that can be used by the reader to truly understand the architecture and implementation details without having to dig into the code. Sample algorithm:

4.1 Compilation

4.1.1 Technologies Used

Uses the stdlib.h library to make a system call.

4.1.2 Design Details

- Find the core name of the file name (trim extension)
- Construct command: g++ -o corename filename -g
- Make system call

4.2 Directory Crawl

4.2.1 Technologies Used

Uses the dirent.h library.

4.2.2 Design Details

```
#include<dirent.h>
#include <string>

void ParseDirectory(string root)
{
    string temp;

    DIR *dir = opendir(root.c_str()); // open the current directory
    struct dirent *entry;

    if (!dir)
    {
        // not a directory
        return;
```

```
    }

    while (entry = readdir(dir)) // notice the single '='
    {
        temp = entry->d_name;
        if ( temp != "." )
        {
            if ( temp != ".." )
            {
                if ( temp[temp.size() - 1] != '~' )
                {
                    int length = temp.length();
                    if ( length > 4 && temp[length-1] == 't' && temp[length-2]
                        == 's' && temp[length-3] == 't' && temp[length-4] == '.' )
                    {
                        TESTVECTOR.push_back(root+'/'+temp);
                    }
                    else
                    {
                        ParseDirectory(root+'/'+temp);
                    }
                }
            }
        }
    }

    closedir(dir);

}
```

4.3 Test Execution

4.3.1 Technologies Used

Uses the stdlib.h library to make system calls.
Uses a vector to hold test cases and their results.

4.3.2 Design Details

For each test case:

- Get input and output file names
- Execute program against test case
- Diff output file and expected answer file
- Determine pass/fail, add to vector

4.4 Result Logging

4.4.1 Technologies Used

Uses the ctime library to generate test run timestamps.
Uses a vector to store test results.

4.4.2 Design Details

```
#include <ctime>
#include <vector>

void WriteLog(string prog)
{
    // make name / date log.
    time_t now;
    time(&now);
    string currTime = ctime(&now);
    string name = prog + " " + currTime.substr(0,currTime.length() - 2) + ".log";
    ofstream log(name.c_str());

    for (int i = 0; i < TESTVECTOR.size(); i+=1)
    {
        log << TESTVECTOR[i] << endl; // flush buffer as long strings
    }

    // now push stats
    log << "             |" << endl;
    log << "Bottom Line V" << endl;
    log << "-------------------------------------------------------------------------------

    log << "Correct: " << CORRECTTESTS << "\nFailed: " << TESTVECTOR.size() - CORRECTTESTS
    log << "Success Rate: " << CORRECTTESTS/ TESTVECTOR.size() * 100 << "%" << endl;
}
```

5

Development Environment

5.1 Development IDE and Tools

Need a Linux distro with a g++ compiler to build and run the program. Editing the program can be done with a text editor such as Gedit, VIM, or Emacs.

5.2 Source Control

Source control was done through GitHub.

5.3 Build Environment

Program is compiled using a makefile.

6

Release – Setup – Deployment

6.1 Setup Information

Copy grade.C and the Makefile into the parent directory containing your target program's source code and test cases. Run make to compile the program. Then run the program with ./grade sourcefile

Acknowledgement

Thanks to Dr. Logar for outlining requirements for the program and providing sample documentation.

Supporting Materials

Sprint Reports

6.1 Sprint Report #1

Sprint 1 consisted of program design, establishing source control, and implementing the major features. All of these tasks were accomplished on time.

6.2 Sprint Report #2

Sprint 2 consisted of integrating the major components, testing, and documentation.