Product Title

Testing System Project

LaTex Samurai

Julain Brackins Jonathan Dixon Hafiza Farzami

February 20, 2014

Contents

List of Figures

List of Tables

List of Algorithms

Mission

The mission statement for this project is to create a test suite designed to compile and run C++ projects with various test cases.

Document Preparation and Updates

Current Version [1.0.0]

Prepared By:
Hafiza Farzami

Revision History

Date	***Author***	***Version***	***Comments***
2/17/14	*Hafiza Farzami*	*1.0.0*	*Initial version*

1

Overview and concept of operations

This report covers the project overview, user stories, backlog, design and implementation, development environment, deployment, and documentation for the testing project.

1.1 Scope

This section gives a brief overview of the system.

1.2 Purpose

The purpose of this program is to run `.cpp` files with given test files, and grade them.

1.2.1 Traversing Subdirectories

Traversing subdirectories is one of the main components of this system. The program runs a `.cpp` file using test files, and the test files are stored in the current and all the subdirectories.

1.2.2 Running the Program Using Test Cases

The software was designed in the Linux environment provided to the group by the university.

1.3 Systems Goals

The goal of this system is to grade a `.cpp` file just by typing `grade <filename>.cpp`. The product is built to test the `.cpp` file with all the given `.tst` test files in the current directory and all the subdirectories, and compare the results to the corresponding `.ans` files.

1.4 System Overview and Diagram

Here is a flow diagram showing the implementation process:

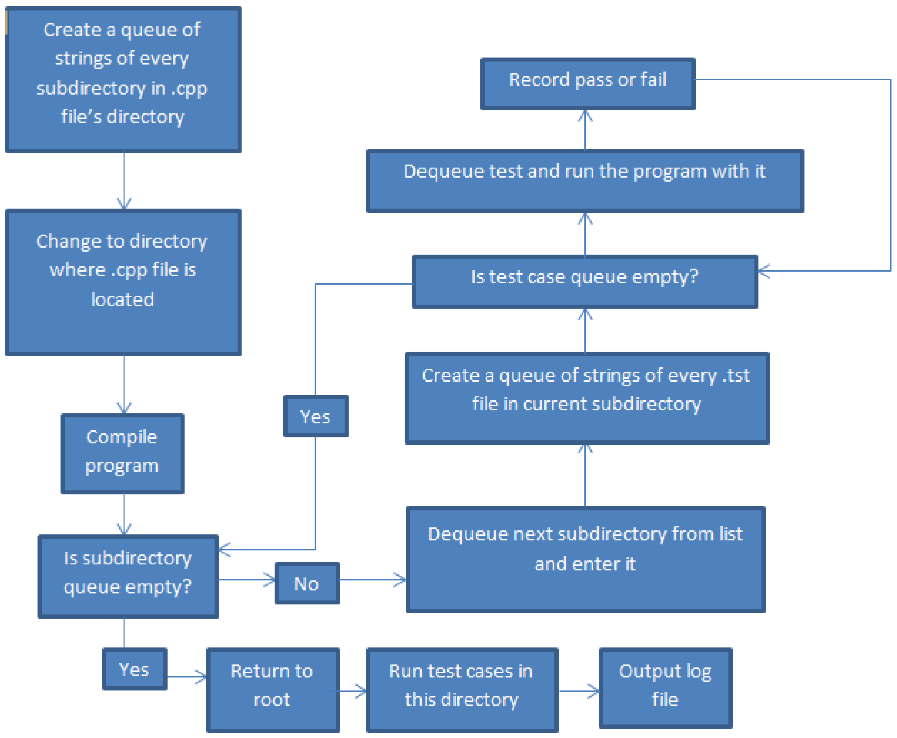
Create a queue of strings of every subdirectory in .cpp file's directory
Change to directory where .cpp file is located
Compile program
Is subdirectory queue empty?
No
Dequeue next subdirectory from list and enter it
Create a queue of strings of every .tst file in current subdirectory
Is test case queue empty?
Dequeue test and run the program with it
Record pass or fail
Yes
Yes
Return to root
Run test cases in this directory
Output log file

Figure 1.1: System Diagram

2

Project Overview

2.1 Team Members and Roles

- Jonathan Dixon - Product Owner
- Hafiza Farzami - Scrum Master
- Julian Brackins - Technical Lead

2.2 Project Management Approach

The aproach taken to manage this project is `scrum`. The project is broken into tasks to be completed over two weeks sprint. The tasks are listed in `Spring Backlog` in trello. During the sprint, the team meets for ten minutes scrum meetings to explain their progress, next steps, and impediments.

2.3 Phase Overview

Once a team member starts a given task, then the task is moved from `Spring Backlog` to `In Progress`. A done task is then moved to `Ready for Testing` tab. After a completed task is test, then it is stamped as `Complete`. Then the member moves to the next task of higher priority.

3

User Stories, Backlog and Requirements

3.1 Overview

This section covers user stories, backlog and requirements for the system.

3.1.1 Scope

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

3.1.2 Purpose of the System

The purpose of the product is to grade a `<filename>.cpp` file by running test files and comparing the results to answer files, and assigning percentage grade.

3.2 Stakeholder Information

This section would provide the basic description of all of the stakeholders for the project.

3.2.1 Customer or End User (Product Owner)

Jonathan Dixon is the product owner in this project, who is in contact with the scrum master and technical lead regarding the backlog.

3.2.2 Management or Instructor (Scrum Master)

Hafiza Farzami is the scrum master, who breaks the project into smaller tasks, and is in touch with both product owner and technical lead.

3.2.3 Developers –Testers

Julian Brackins is the technical lead for Sprint 1, and is in contact with both Dixon and Farzami regarding the requirements during scrum meetings and through trello notes.

3.3 Business Need

This product is essential for grading computer science programs focused on numerics. All the user have to do is have test cases and expected results in the directory that the `<filename>.cpp` file is in and any of the subdirectories, and run the `grade.cpp` program. It saves a lot of time, and is efficient.

3.4 Requirements and Design Constraints

Use this section to discuss what requirements exist that deal with meeting the business need. These requirements might equate to design constraints which can take the form of system, network, and/or user constraints. Examples: Windows Server only, iOS only, slow network constraints, or no offline, local storage capabilities.

3.4.1 System Requirements

This product runs on Linux machines.

3.4.2 Network Requirements

This software does not require internet connection.

3.4.3 Development Environment Requirements

There are not any development environment requirements.

3.4.4 Project Management Methodology

The method used to manage this project is `scrum`. The scrum master met with the product owner, and broke the tasks down to the technical lead. The team meets for ten minutes long scrum meetings to go over the progress, next steps, and impediments.

- Trello is used to keep track of the backlogs and sprint status
- Everyone has access to the Sprint and Product Backlogs
- This project will take three Sprints
- Each Sprint is two weeks long
- There are no restrictions on source control

3.5 User Stories

This section contains the user stories regarding functional requirements and how the team broke them down.

3.5.1 User Story

Write an automatic testing system. By Wednesday, your team should have a list of questions for me (the customer) on exactly what I require. Think CS150/250/300 programs - not major software.

3.5.1.a User Story Breakdown #1

The purpose of the program is to grade a student's file, comparing it to multiple test case files. It'll be called like this: `grade <filename>.cpp`. It will grade the file, looking through the current directory for any `.tst` files, which will contain the test cases. We must compile the program, which is in C++.

It is understood that all inputs will be valid (no error checking, program won't crash, etc...). The program will be tested on numeric computations. If it uses files, they'll be `input.txt` and `output.txt`. `<filename>.log` will contain the results of each test case.

The idea is to grab the test case, copy it to input.txt, run the program, compare outputs, and put results into the log file. `.tst` files will have a test case followed by a blank line, followed by expected results.

3.5.1.b User Story Breakdown #2

A student submits a program. That program is placed into a directory that forms the root of the directory tree related to that program. All the instructors and teacher's assistant will have the ability to write test cases (called case#.tst and the accompanying file case#.ans). There are no restrictions on where those files can be located except that they will be at the level of the .cpp file or below. For example. Manes might have a subdirectory where he puts his test cases. His TAs might have subdirectories under the Manes subdirectory where they put their test cases. Manes might just make the directory but not put any test cases in it just so his TAs have a place to put theirs (in subdirectories). I want to say test quadratic and have your program find all the applicable test cases (with the accompanying answer files), run the tests, log the results, and provide a summary. I want to be able to fix problems and rerun the test without losing the original log file. Append the date so I can tell them apart.

4

Design and Implementation

This section describes the design details for the overall system as well as individual major components. As a user, you will have a good understanding of the implemetation details without having to look into the code. Here is the

- Create a queue of every subdirectory in program folder
- Change to directory where program is located
- Compile program
- While subdirectory queue is not empty:
- Dequeue first subdirectory in queue
- Change into that subdirectory
- Create a queue of every .tst file in current directory
- While test case queue is not empty:
- Dequeue first test case in queue
- Run program using that test case
- Count whether the program passed or failed test case
- Change back to home directory (where program is located)
- Create a queue of every .tst file in home directory
- While test case queue is not empty:
- Dequeue first test case in queue
- Run program using that test case
- Count whether the program passed or failed test case
- Write log file containing percentage of tests passed and final grade

4.1 Traversing Subdirectories

4.1.1 Technologies Used

The dirent.h library is used for traversing subdirectories.

4.1.2 Design Details

```
bool change_dir(string dir_name)
{
    string path;
    if(chdir(dir_name.c_str()) == 0)
    {
        path = get_pathname();
        return true;
    }
    return false;
}

bool is_dir(string dir)
{
    struct stat file_info;
    stat(dir.c_str(), &file_info);
    if ( S_ISDIR(file_info.st_mode) )
        return true;
    else
        return false;
}
```

4.2 Running the Program Using Test Cases

4.2.1 Technologies Used

The software was designed in the Linux Environment provided to the group by the University.

4.2.2 Design Details

```
int run_file(string cpp_file, string test_case) //case_num
{
    //create .out file name
    string case_out(case_name(test_case, "out"));

    //set up piping buffers
    string buffer1("");
    string buffer2(" &>/dev/null < ");
    string buffer3(" > ");

    // "try using | "
    //construct run command, then send to system
    //./<filename> &> /dev/null  < case_x.tst > case_x.out
    buffer1 += cpp_file + buffer2 + test_case + buffer3 + case_out;
    system(buffer1.c_str());

    //0 = Fail, 1 = Pass
    return result_compare(test_case);
}
```

5

System and Unit Testing

Testing was first done on small parts of the program, and as we added more, we tested larger parts of the program.

5.1 Overview

In general, we began testing on a lower level by testing our capabilities for rerouting input and output. Once we managed that, we moved on to comparing output files to the test case files. Once Dr. Logar released the test cases, we created a function to confirm directory traversals. Then, when we combined the two, our program functioned correctly and we were able to add the finishing touches.

5.2 Dependencies

No dependencies.

5.3 Test Setup and Execution

In earlier phases of the sprint, a simple “Hello World!” .cpp file was created to test the usage of system commands to compile and run a program within our testing software.

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

Code was modified through the use of gedit and the text editor available on the git website.

6.2 Source Control

Git was used for source control. It was set up on GitHub, and can be accessed via the command
git connect https://github.com/jbrackins/CSC470.git

6.3 Dependencies

An internet connection to access the Git will be necessary.

6.4 Build Environment

The program may be built using the g++ compiler on linux.

7

Release – Setup – Deployment

7.1 Deployment Information and Dependencies

There are no specific dependencies for this program.

7.2 Setup Information

The program is built with the g++ compiler on linux.

7.3 System Versioning Information

The program will be versioned depending on the current Agile sprint.

8

User Documentation

This section contains the user guide, user documentation.

8.1 User Guide

This product is to make it easy for you to grade a computational computer program written in C++ language. In order to benefit from this product, the test.cpp file and Makefile for the testing program must be in the same directory. Located in the same directory as the test.cpp file should be a folder containing a .cpp file to be tested. The directory and .cpp names should match. In the .cpp file's directory, you should have:

- The `.cpp` file to be tested
- Test files (with `.tst` extensions)
- Corresponding solution files (with `.ans` extensions)

As a user, all you need to do is type `grade <filename>.cpp` in the terminal. It is assumed that you are using this product on `Windows` or `Linux` operating system. The program will run through all the test cases and compare the results from the `.cpp` file with the answer in `.ans` file. If the answers were similar, the `.cpp` file will get 100% credit for that partiular case, else the percentage will be zero. The program will run and return you a `.log` file containing the grade for a given `.cpp` file.

9

Class Index

9.1 Class List

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

10

Class Documentation

10.1 Poly Class Reference

Public Member Functions

- Poly ()
- ∼Poly ()
- int myfunction (int)

10.1.1 Constructor & Destructor Documentation

10.1.1.a Poly::Poly ( )

My constructor

10.1.1.b Poly::∼Poly ( )

My destructor

10.1.2 Member Function Documentation

10.1.2.a int Poly::myfunction (int *a*)

my own example function fancy new function

new variable

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

- hello.cpp

Acknowledgement

Thanks

Supporting Materials

This document will contain several appendices used as a way to separate out major component details, logic details, or tables of information. Use of this structure will help keep the document clean, readable, and organized.

Sprint Reports

10.1 Sprint Report #1

10.2 Sprint Report #2

10.3 Sprint Report #3

Industrial Experience

10.4 Resumes

10.5 Industrial Experience Reports

10.5.1 Julian Brackins

10.5.2 Jonathan Dixon

10.5.3 Hafiza Farzami

Appendix

Latex sample file:

10.1 Introduction

This is a sample input file. Comparing it with the output it generates can show you how to produce a simple document of your own.

10.2 Ordinary Text

The ends of words and sentences are marked by spaces. It doesn't matter how many spaces you type; one is as good as 100. The end of a line counts as a space.

One or more blank lines denote the end of a paragraph.

Since any number of consecutive spaces are treated like a single one, the formatting of the input file makes no difference to TEX, but it makes a difference to you. When you use LATEX, making your input file as easy to read as possible will be a great help as you write your document and when you change it. This sample file shows how you can add comments to your own input file.

Because printing is different from typewriting, there are a number of things that you have to do differently when preparing an input file than if you were just typing the document directly. Quotation marks like "this" have to be handled specially, as do quotes within quotes: "'this' is what I just wrote, not 'that'".

Dashes come in three sizes: an intra-word dash, a medium dash for number ranges like 1–2, and a punctuation dash—like this.

A sentence-ending space should be larger than the space between words within a sentence. You sometimes have to type special commands in conjunction with punctuation characters to get this right, as in the following sentence. Gnats, gnus, etc. all begin with G. You should check the spaces after periods when reading your output to make sure you haven't forgotten any special cases. Generating an ellipsis . . . with the right spacing around the periods requires a special command.

TEX interprets some common characters as commands, so you must type special commands to generate them. These characters include the following: $ & % # { and }.

In printing, text is emphasized by using an *italic* type style.

A long segment of text can also be emphasized in this way. Text within such a segment given additional emphasis with Roman *type. Italic type loses its ability to emphasize and become simply distracting when used excessively.*

It is sometimes necessary to prevent TEX from breaking a line where it might otherwise do so. This may be at a space, as between the "Mr." and "Jones" in "Mr. Jones", or within a word—especially when the word is a symbol like *itemnum* that makes little sense when hyphenated across lines.

Footnotes[1] pose no problem.

TEX is good at typesetting mathematical formulas like $x - 3y = 7$ or $a_1 > x^{2n}/y^{2n} > x'$. Remember that a letter like x is a formula when it denotes a mathematical symbol, and should be treated as one.

[1] This is an example of a footnote.

10.3 Displayed Text

Text is displayed by indenting it from the left margin. Quotations are commonly displayed. There are short quotations

> This is a short a quotation. It consists of a single paragraph of text. There is no paragraph indentation.

and longer ones.

> This is a longer quotation. It consists of two paragraphs of text. The beginning of each paragraph is indicated by an extra indentation.
>
> This is the second paragarph of the quotation. It is just as dull as the first paragraph.

Another frequently-displayed structure is a list. The following is an example of an *itemized* list.

- This is the first item of an itemized list. Each item in the list is marked with a "tick". The document style determines what kind of tick mark is used.
- This is the second item of the list. It contains another list nested inside it. The inner list is an *enumerated* list.

    1. This is the first item of an enumerated list that is nested within the itemized list.
    2. This is the second item of the inner list. LaTeX allows you to nest lists deeper than you really should.

  This is the rest of the second item of the outer list. It is no more interesting than any other part of the item.
- This is the third item of the list.

You can even display poetry.

> There is an environment for verse
> Whose features some poets will curse.
>
> For instead of making
> Them do *all* line breaking,
> It allows them to put too many words on a line when they'd rather be forced to be terse.

Mathematical formulas may also be displayed. A displayed formula is one-line long; multiline formulas require special formatting instructions.

$$x' + y^2 = z_i^2$$

Don't start a paragraph with a displayed equation, nor make one a paragraph by itself.

10.4 Build process

To build LaTeX documents you need the latex program. It is free and available on all operating systems. Download and install. Many of us use the TexLive distribution and are very happy with it. You can use a editor and command line or use an IDE. To build this document via command line:

```
alta>  pdflatex SystemTemplate
```

If you change the bib entries, then you need to update the bib files:

```
alta>  pdflatex SystemTemplate
alta>  bibtex SystemTemplate
alta>  pdflatex SystemTemplate
alta>  pdflatex SystemTemplate
```

Acknowledgement

Thanks to Leslie Lamport

Bibliography

[1] R. Arkin. *Governing Lethal Behavior in Autonomous Robots.* Taylor & Francis, 2009.

[2] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George A Kantor, Wolfram Burgard, Lydia E. Kavraki, and Sebastian Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations.* MIT Press, Cambridge, MA, June 2005.

[3] S. M. LaValle. *Planning Algorithms.* Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[4] V. Lumelsky and A. Stepanov. Path planning strategies for point mobile automation moving amidst unknown obstacles of arbirary shape. *Algorithmica*, pages 403–430, 1987.

[5] S.A. NOLFI and D.A. FLOREANO. *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines.* A Bradford book. A BRADFORD BOOK/THE MIT PRESS, 2000.

[6] Wikipedia. Asimo — Wikipedia, the free encyclopedia. `http://upload.wikimedia.org/wikipedia/commons/thumb/0/05/HONDA_ASIMO.jpg/450px-HONDA_ASIMO.jpg`, 2013. [Online; accessed June 23, 2013].