Automated Tester

Software Engineering Final Documentation

Original Team: White Space Cowboys

Ryan Brown Kelsey Bellew Ryan Feather

March 22, 2014

Contents

List of Figures

List of Tables

List of Algorithms

Mission

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.

Document Preparation and Updates

Current Version [1.0.1]

Prepared By:
Kelsey Bellew
Ryan Brown
Ryan Feather

Revision History

Date	***Author***	***Version***	***Comments***
2/14/14	*Ryan Brown*	*1.0.0*	*Initial version*
2/17/14	*Ryan Brown*	*1.0.1*	*Fixed Directory Name Bug*

1 Overview and concept of operations

This program is a utility for the testing of student software. In version 1, it took a student's program and checks if it conforms to a suite of test cases and presently, in version 2, looks through a directory heirarchy for all student programs and checks it's conformation to 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 run input through a student's code and test if the output is what was expected by the instructor or as created by the generator. A summary file will be generated for each run of the program, detailing which tests passed or failed.

1.2.1 Compiling

The students' program will be supplied as source code, so the Auto Tester must compile it using the GNU C compiler.

1.2.2 Identifying Test Cases and Inputting

For each student program is compiled, the Auto Tester uses searched 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 test case encountered. A final sumary of performance is printed to a final log after all programs have been executed

1.3 Systems Goals

The goal of this system is to provide an environment for test-case grading. This is implemented through the use of the Auto Tester executable, a golden program to test generated test files on (located at the directory of execution) and special test files (placed in any sub-directory or the directory of execution).

The system should be accurate in its output matching and able to do many tests very quickly.

1.4 System Overview and Diagram

The Auto Tester is a highly independent system. It takes as input a students source code, several test case files, and outputs the results of the tests. Externally it does not require much components. A gcc compiler is a must on the target system, but other than that the program relies on the filesystem to fetch its test cases.

Internally, the Auto Tester has 3 major components:

1. A function that fetches the filenames of each test case found in the programs directories and subdirectories.
2. A function that opens each input file and runs it through the student program, capturing the output.
3. A function that takes each test case output and checks it against the expected answer. This records the results in a log file.

These components combine to test the student program with multiple test cases and output valuable information to be used for grading.

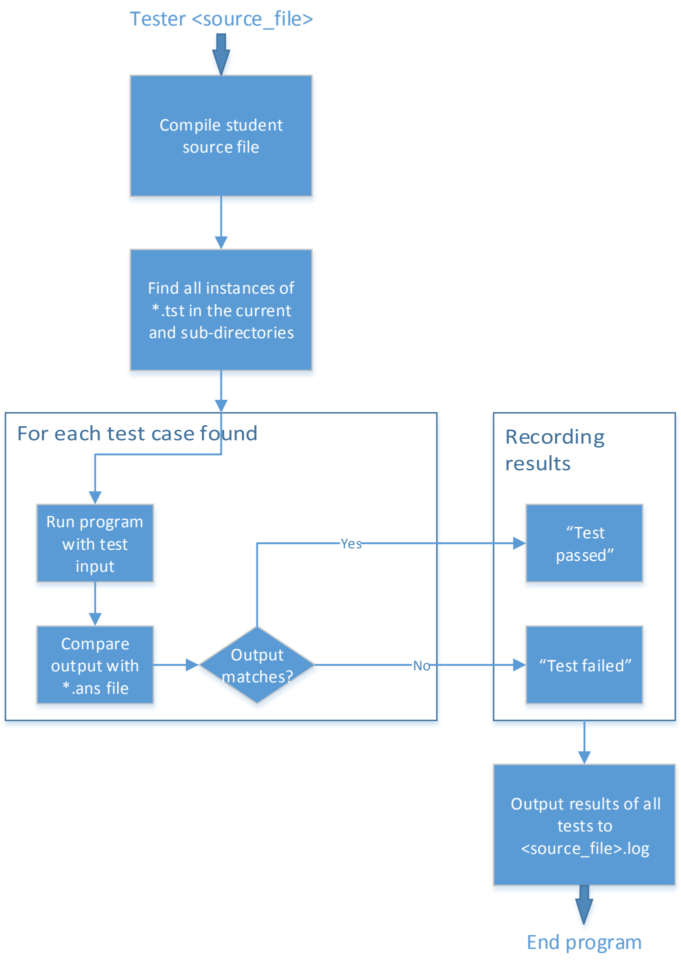
Tester <source_file>
Compile student source file
Find all instances of *.tst in the current and sub-directories
For each test case found
Run program with test input
Compare output with *.ans file
Output matches?
Yes
No
Recording results
"Test passed"
"Test failed"
Output results of all tests to <source_file>.log
End program

Figure 1.1: The program flow of the Tester

1.5 Technologies Overview

See Table 1.1. All technology and programs used to develop and run the Auto Tester can be found on any modern GNU\Linux distribution.

GNU C Compiler	For compiling the students program
GNU find	A command in GNU tools used to search directories
popen	A function from stdio.h, used to fork a process and supply it with input
C++ Template Libraries	Specifically the vector container

Table 1.1: Technologies used in Auto Tester

2

Project Overview

2.1 Team Members and Roles

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.

The team members of Lounge Against The Machine were Adam Meaney, Joseph Manke, and Alex Wulff.

- Adam Meaney was the Scrum Master.
- Joseph Manke was the Product Owner.
- Alex Wulff was the Technical Lead and Co-Scrum Master and Co-Product Owner.

2.2 Project Management Approach

This project uses the Agile methodology. The sprints are three weeks in length, and the product backlog is hosted on Trello. Bug or trouble tickets are also to be posted on Trello, along with initial user stories.

2.3 Phase Overview

The initial phase that this project went through took us from conception to version 1.0.0 at which point the project changed to Lounge Against The Machine. The current version, V2.0.0 has extended features such as a test generator and multiple test program support. It started with several planning talks and research into the technical challenges this project presented. Once decisions were made about the technical design of the system, development began.

In addition to technical challenges it took some time to clarify requirements and turn the provided user stories into the design of the current product.

Once the working version was established and most of the development in this sprint was done, the system and code were tested. A code review was done and several fixes were implemented accordingly, bringing us to a stable release of 1.0.0. Version 2.0.0 was widely devised as a joke by Alex Wulff, but has since been modified into a respectable extension of the original. Extensive testing on the test case generator has been completed in the attempt to provide the most detailed and customizable generator a customer could desire.

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.

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 either pre-written tests or custom generated tests on a master directory containing the programs of their choosing, and to provide a log to the user of which tests passed and which tests failed, as well as a percentage reflecting the results of the test. Any tests labled as "..._crit.tst" will be evaluated first and if any such criticals fail, the current program will achieve 0 percent and will be reported as FAILED in the final log.

3.2 Stakeholder Information

There are three main stakeholders who have an interest in the completion of this project; this would be Dr. Logar, the Customer, the members of the original and subesquent development teams (Kelsey Bellew, Ryan Brown, Ryan Feather, Joseph Manke, Adam Meaney, and Alex Wulff).

3.2.1 Customer or End User (Product Owner)

The Product Owner was Ryan Brown and currently is Joseph Manke. 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.

3.2.2 Management or Instructor (Scrum Master)

The Scrum Master was Kelsey Bellew and is currently Adam Meaney, 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.

3.2.3 Investors

The investors in this case, will be the members of other teams. If this project is not complete, not well written, or has any obvious or unobvious flaws at the end of the firstsprint, the members of the team who continue on with this project will be forced to handlethe project as is.

3.2.4 Developers –Testers

The Developers of this project are the members of the development team. The development team will also be the testers for this project.

3.3 Business Need

This software enables an automated way to test a user's program. All programs need to be tested before they are presented to a manager, or before they are shipped out, or even before or after they are turned in as homework. This software gives the user an easier, faster way to run tests on their program.

3.4 Requirements and Design Constraints

There are a several requirements and design constraints within this project. This section will discuss System Requirements, Network Requirements, Development Environment Requirements, and Project Management Methodology.

3.4.1 System Requirements

The Linux operating system is a constraint of this project. This is because of the use of the system() function within the code, which uses specifically linux command line commands to achieve several outcomes, such as compiling and running the program. This project could have been written for Windows, but would require an implimentation of a different set of system commands or a series of #define statements for function and include aliases.

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

There should not be any development environment requirements, other than a Linux operating system. However, this project has been tested on both Fedora, Arch, Gentoo and Ubuntu, on a number of different text and code editors. The only other requirement would be the avalibility of gcc.

3.4.4 Project Management Methodology

The stakeholders might restrict how the project implementation will be managed. There may be constraints on when design meetings will take place. There might be restrictions on how often progress reports need to be provided and to whom.

- Trello will be used to keep track of the backlogs and sprint status.
- The members of the team will have access to Sprint and Product Backlogs, as will the Customer.
- There will be three sprints encompassing this project.
- Sprint Cycles are three weeks.

- The source control used will be GitHub, and all members of the team and the Customer will have access to the GitHub directory.
- Code shall not be updated unless it is compiling and completes a function.

3.5 User Stories

This section is the result of discussions with the stakeholders with regard to the actual functional requirements of the software. The user stories will be used in the work breakdown structure to build tasks to fill the product backlog for implementation throught the sprints.

This section will contain sub-sections to define and potentially provide a breakdown of the larger user stories into smaller user stories.

3.5.1 User Story #1

(Sprint 1) The user wants to be able to enter their program into the testing program through command line.

3.5.1.a User Story #1 Breakdown

The program the user wants to be able to run will come in the form of code as opposed to an executable; this means that the program needs to be able to take this user given code and compile it. It also means that the user should not have to enter in any additional information or text past the initial run of the program.

3.5.2 User Story #2

(Sprint 1) User wants to be able to include multiple test cases in multiple directories.

3.5.2.a User Story #2 Breakdown

The program must be able to run if the user wants to only include test cases in a single directory, but also if they want to include test cases in multiple directories. The user also wants to see the results of the test cases in a specific place in the directory, namely the same directory the main program is in.

3.5.3 User Story #3

(Sprint 1) User wants to be able to see the percentage of tests passed.

3.5.3.a User Story #3 Breakdown

In order to tell at a glance, how well a program did at passing the tests, a percentage is calcualted and appended to the results displayed.

3.5.4 User Story #4

(Sprint 2) User wants to be able to test multiple programs as a group.

3.5.4.a User Story #4 Breakdown

This will facilitate a class like evaluation for easy grading and comparison of performance relative to the rest of the group.

3.5.5 User Story #5

(Sprint 2) User wants to be able to set minimum performance requirements with ”critical tests”.

3.5.5.a User Story #5 Breakdown

Such a provision enables a minimum performance standard to be achievable by each targeted program.

3.5.6 User Story #6

(Sprint 2) User wants to generate custom and random test cases and use a standard ”correct” program to provide truth to compare other program’s output to.

3.5.6.a User Story #6 Breakdown

The program needs a truth generating program to set the standard for all other programs. Truth will be made by all new tests as generated by the tester.

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 appropite 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

This section is used to describe the design details for each of the major components in the system.

4.1 Finding Test Cases (find tsts)

4.1.1 Technologies Used

- popen
- GNU find
- C++ vectors

4.1.2 Component Overview

This component, uses the GNU find program to recursively find all of the filenames of the form *.tst. This includes all subdirectories. popen is used to capturethe output, and that ouput is placed into a vector of filnames.

4.1.3 Phase Overview

This component was designed and developed in the initial phase and was an integral part 1.0.0 functionality.

4.1.4 Design Details

One small detail is that this function needs to deal with a troublesome character encoutnered at the end of the filenames. The code below demonstrates how this is done.

```
//removing that frustrating invisible character at the end of the strings
for(int i=0;i<tstfilelist.size();i++)
{
    tstfilelist.at(i).replace(tstfilelist.at(i).end()-1,
    tstfilelist.at(i).end(),"");
}
```

4.2 Running the Student Program (runtests)

4.2.1 Technologies Used

- C++ vectors

4.2.2 Component Overview

The basic features of this component were designed in V1.0.0 to run a specified test case and check its output against the expected output. In V2.0.0 this component has been modified to compile and test a specific program and all specified tests.

4.2.3 Phase Overview

This component was designed and developed in the initial phase and was an integral part 1.0.0 functionality. This was later modified for multiple programs in V2.0.0.

4.2.4 Design Details

In order to run the student program, a string is constructed and passed to the shell through the system() function. The results are stored in a temp file. The temp file and *.ans file are then passed to the component that compares them to see if the outputs match.

4.3 Checking the Output (filesequal)

4.3.1 Technologies Used

- C++ vectors

4.3.2 Component Overview

This component opens both files passed to it and checks if they are equivalent. It returns an answer of yes or no.

4.3.3 Phase Overview

This component was designed and developed in the initial phase and was an integral part 1.0.0 functionality.

4.3.4 Design Details

This component checks the two files by reading their contents into vectors. First the length of the two vectors are compared, then if it passes that test, the lines are compared against each other. If both tests pass, the files are equivalent.

4.4 Finding Student Code (find_students)

4.4.1 Technologies Used

- C++ vectors
- Dirent directory file descriptors

4.4.2 Component Overview

This component searches for every student source file and adds it to the list of the programs to be compiled and executed

4.4.3 Phase Overview

This component was designed and developed second phase as an integral part of 2.0.0 functionality.

4.4.4 Design Details

This component is designed to recursively scan all sub-directories for any source files.

```
while (entry = readdir(dir))
  {
    temp = entry->d_name;
    if ( temp != "." && temp != ".." )
    {
      if ( temp[temp.size() - 1] != '~' )
      {
        int length = temp.length();
        if ( (length > 4 && (temp.substr(length-4) == ".cpp")
             || temp.substr(length-2) == ".C")
            && level > 0 )
        {
          STUDENTVECTOR.push_back(directory + '/' + temp);
        }
        else if ( (length > 4 && (temp.substr(length-4) == ".cpp")
             || temp.substr(length-2) == ".C")
            && level == 0 )
        {
          if (GOLDCPP.empty())
          {
            GOLDCPP = directory + '/' + temp;
          }
        }
        else
        {
          find_students(directory + '/' + temp, level + 1);
        }
      }
    }
  }
```

4.5 Individual Test Reports(writeindividualreport)

4.5.1 Technologies Used

- C++ vectors

4.5.2 Component Overview

This component writes an individual report regarding each test as they happen in the current program's directory.

4.5.3 Phase Overview

This component was designed and developed second phase as an integral part of 2.0.0 functionality.

4.5.4 Design Details

This component will simply advise if a program has passed or failed a specific test. After a critical fail, the remaining tests will not be ran or logged.

4.6 Grouped Test Reports(writefinaloutfile)

4.6.1 Technologies Used

- C++ vectors

4.6.2 Component Overview

This component writes a final report regarding each program and their test successes as a percent or "FAILED" if a critical test was not passed.

4.6.3 Phase Overview

This component was designed and developed second phase as an integral part of 2.0.0 functionality.

4.6.4 Design Details

```
timeinfo = localtime (&rawtime);
  strftime (buffer,16,"%m_%d_%H:%M:%S",timeinfo);
  //string to hold the final output file name
  string outfilename (buffer); //QQQ!!! Alex : just log and time //+ logprogname+"_"+buffe

  outfilename = "log " + outfilename;

  //opening final output file
  ofstream fout;
  fout.open(outfilename.c_str());
```

4.7 Generated Test Case Files(generatetestcases)

4.7.1 Technologies Used

4.7.2 Component Overview

This component uses a text based menu to identify the user's requirements for a series of test files. These are then tested against the truth generating program at the level of execution to derive all acceptable outputs.

4.7.3 Phase Overview

This component was designed and developed second phase as an integral part of 2.0.0 functionality.

4.7.4 Design Details

```
for (int j = 0; j < inserts; j +=1)
      {
        // threshold  offset to min   by random fraction   times the difference
        //int inValue = (int) (min + ((double) rand() / RAND_MAX) * (max - min));
        if (max != 1 && min != 0)
        {
          inValue = (int (min) + rand()) % int(max) + 1;
        }
        else
        {
          inValue = MININT + rand() + rand(); // Max at randmax, min at LONG_MIN
        }

        fout << (int) inValue << endl;
      }
```

5

System and Unit Testing

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

5.1 Overview

This chapter will provide a breif overview of the testing approach, testing frameworks, and how testing will be done to provide a measure of success for the system.

5.2 Dependencies

This program was written with the assumption that all test files to run against the user program are in the directory with the user program, or in a child directory of the initial directory. With revision two, the assumption is that the truth generating program will be at the same directory level as execution and all programs to be tested will be in some sub-directory here under. It also only processes tests with the extention .tst, and only runs and compiles programs written in c++.

5.3 Test Setup and Execution

The majority of the test cases were developed by the Customer, Dr. Logar. They included several simple programs that took input and produced an output, with test cases for desired input and correct output. However, as a test development system, additional test had to be made locally to test for failures outside of the scope of the customer as limits needed to be verified against values and data types. All tests that caused any exception or segmentation fault were deemed unsatisfactory and were removed from further testing

Several of the programs were designed to fail in certian cases so that the development team could be sure that a program would display the correct passed/failed ratio. The team also tested different forms of the user program, and tested running the program from different directories. The point of this specific test was to make sure only tests contained within the directory or subdirectories where the user program resided would be run.

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

The specific tools used to develop this project were Gedit, Vim, Putty, and g++. Except for g++, all of these tools are not strictly necessary to the development environment. This project could easily be continued in any text or code editor, with presumably, any form of Linux. For easy construction, use the provided make file.

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 of git to keep code and documentation updated.

6.3 Dependencies

There are no specific dependencies with developing the system, other than a Linux operating system and gcc version three or higher.

6.4 Build Environment

Packages are built using a general g++ command in command line Linux. A make file is also provided for easy construction.

6.5 Development Machine Setup

There are no specific steps associated with setting up a macine for use by a developer.

7

Release – Setup – Deployment

This section will contain any specific subsection regarding specifics in releasing, setup, and/or deployment of the system.

7.1 Deployment Information and Dependencies

- Validate your linux installation has a current install of gcc (yum -install gcc / apt-get install gcc / emerge install gcc / pacman -S gcc)
- (Optional / Troubleshooting) validate linux headers are installed (above package manager with the arguments: linux-headers-$(uname -r))

7.2 Setup Information

To setup this project, g++ is needed. The project is built with:
g++ -o tester tester.c
The project is run by:
./tester

7.3 System Versioning Information

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, placement of truth program and programs to test. Test case files must be located in the exeution directory or below. The truth program needs to be placed at the level of execution and all programs to test must be stored in sub-directories.

8.1.1 Test Case Files

Files that contain the test cases themselves need to be given a .tst extension. The filname proceeding the extension will be used as the name of that test in both the local and summary log files. 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 Results

The results will be placed in a file with a .log extension. The filename will be timestamped and contain the name of the student program. A final log will be written to the execution directory with the name log and the current date / time.

8.2 Installation Guide

If running from the .cpp file, the system will first need to be compiled. The user will need to be in the same directory as the .cpp file with a Linux command line terminal, and use the following command:
g++ -o tester tester.cpp or make

After this is completed, the user should have an executable file they can use by invoking the following:
./tester

8.3 Programmer Manual

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

9

Class Index

9.1 Class List

Not Applicable

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

9.1 Sprint Report #1

This sprint lasted from 2/4/14 from 2/19/14. The members of the Whitespace Cowboys setup Github accounts as well as repositories on both personal Linux and Windows machines. Two offical scrum meetings were held, the first of which was to set up Github and the second of which was to assign different members to the writing of sections of documentation and hold a code review.

The coding section of the project was officially done on 2/14/14, when the code review was held. The team members made sure the coding and the coding standard were up to requirements, as well as checked the state of the current documentation in the .cpp file.

9.2 Sprint Report #2

This particular sprint lasted from 2/21/14 to 3/23/14. The members of Lounge Against The Machine reworked existing code to facilitate multiple programs of execution and a new generator system for test and answer file manufacture. The coding started shortly after the beginning of the sprint, but was delayed over spring break for a vacation of the Tech Lead. The project was finished and submitted on 3/23/14 where this document was updated with all relevant changes. All make and execution variables were checked before submission.

9.3 Sprint Report #3

Industrial Experience

9.4 Resumes

9.5 Industrial Experience Reports

9.5.1 Name1

9.5.2 Name2

9.5.3 Name3

Bibliography