Testing System Project

The Adventure Line

Erik Hattervig Andrew Koc Jonathan Tomes

April 27, 2014

Contents

Mission **xiii**

Document Preparation and Updates **xv**

1 Overview and concept of operations **1**
1.1 Scope . . . 1
1.2 Purpose . . . 1
1.2.1 Traversing Subdirectories . . . 1
1.2.2 Running the Program Using Test Cases . . . 1
1.2.3 Test Case Generation . . . 1
1.2.4 Character String Test Generation . . . 1
1.2.5 Run away program testing . . . 1
1.2.6 Handling Presentation Errors . . . 2
1.2.7 Menu - driven Testing . . . 2
1.2.8 Code Coverage . . . 2
1.2.9 Code Performance . . . 2
1.3 Systems Goals . . . 2
1.4 System Overview and Diagram . . . 2
1.5 Technologies Overview . . . 2

2 Project Overview **5**
2.1 Team Members and Roles . . . 5
2.1.1 Sprint 1 with Latex Samurai . . . 5
2.1.2 Sprint 2 with Kernel_Panic . . . 5
2.1.3 Sprint 3 with The Adventure Line . . . 5
2.2 Project Management Approach . . . 5
2.3 Phase Overview . . . 5

3 User Stories, Backlog and Requirements **7**
3.1 Overview . . . 7
3.1.1 Scope . . . 7
3.1.2 Purpose of the System . . . 7
3.2 Stakeholder Information . . . 7
3.2.1 Customer or End User (Product Owner) . . . 7
3.2.2 Management or Instructor (Scrum Master) . . . 7
3.2.3 Developers –Testers . . . 7
3.3 Business Need . . . 8
3.4 Requirements and Design Constraints . . . 8
3.4.1 System Requirements . . . 8
3.4.2 Network Requirements . . . 8
3.4.3 Development Environment Requirements . . . 8
3.4.4 Project Management Methodology . . . 8
3.5 User Stories . . . 8

3.5.1 User Story #1 . . . 8
3.5.2 User Story #2 . . . 9
3.5.3 User Story #3 . . . 9
3.5.4 User Story #4 . . . 9
3.5.5 User Story #5 . . . 9
3.5.6 User Story #6 . . . 9
3.5.7 User Story #7 . . . 9
3.5.8 User Story #8 . . . 10
3.5.9 User Story #9 . . . 10
3.5.10 User Story #10 . . . 10

4 Design and Implementation **11**
4.1 Traversing Subdirectories . . . 12
4.1.1 Technologies Used . . . 12
4.1.2 Design Details . . . 12
4.2 Running the Program Using Test Cases . . . 12
4.2.1 Technologies Used . . . 12
4.2.2 Design Details . . . 12

5 System and Unit Testing **15**
5.1 Overview . . . 15
5.2 Dependencies . . . 15
5.3 Test Setup and Execution . . . 15
5.3.1 Student Grading . . . 15

6 Development Environment **17**
6.1 Development IDE and Tools . . . 17
6.2 Source Control . . . 17
6.3 Dependencies . . . 17
6.4 Build Environment . . . 17

7 Release – Setup – Deployment **19**
7.1 Deployment Information and Dependencies . . . 19
7.2 Setup Information . . . 19
7.3 System Versioning Information . . . 19

8 User Documentation **21**
8.1 User Guide . . . 21
8.1.1 Test Case Generation . . . 21

9 Class Index **23**

10 Class Documentation **25**

Acknowledgement **27**

Supporting Materials **29**

Sprint Reports **31**
10.1 Sprint Report #1 . . . 31
10.2 Sprint Report #2 . . . 31
10.3 Sprint Reports for Team #3 - The Adventure Line by Jonathan Tomes . . . 31
10.3.1 Sprint 1 . . . 31
10.3.2 Sprint 2 . . . 31
10.3.3 Sprint 3 . . . 31

Industrial Experience **33**

Appendix **35**
10.1 Introduction 35
10.2 Build process 35

List of Figures

1.1 System Diagram 3

List of Tables

List of Algorithms

Mission

Sprint 2 from Kernel_panic:

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

Sprint 3 from The Adventure Line:

The mission of the adventure line is to imrpove upon the code from sprint 2, adding the new features desired.

Document Preparation and Updates

Current Version [3.0.0]

Prepared By:
Hafiza Farzami
Ben Sheerman
James Tillma
Anthony Morast
Erik Hattervig
Andrew Koc
Jonathan Tomes

Revision History

Date	***Author***	***Version***	***Comments***
2/17/14	*Hafiza Farzami*	*1.0.0*	*Initial version*
3/21/12	*Ben Sherman*	*1.0.1*	*Updated for new features*
3/22/12	*Anthony Morast*	*1.1.0*	*Updated testing and user sections*
3/23/12	*James Tillma*	*1.2.0*	*Updated early portions of document*
4/27/14	*Jonathan Tomes*	*2.33.0*	*Updated Sections 1 and Sprints*
4/30/14	*Jonathan Tomes*	*2.34.0*	*Updated Sprint Retrospective*

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 many students' `.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 containing the "test" keyword.

1.2.2 Running the Program Using Test Cases

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

1.2.3 Test Case Generation

A major update in sprint 2 is test case generation. This allows the user to actually generate psuedo-random test cases.

1.2.4 Character String Test Generation

A large edition to test generation is the creation of tests for a single string of lower case characters up to or exactly a length specified by the user.

1.2.5 Run away program testing

A major edition of imparting a time limit for the test programs to run, and if they exceed this time limit on a test, they will fail that test.

1.2.6 Handling Presentation Errors

The program will now examine the test results of each out put program. It will mark a test passed if a few small mistakes in presentation are present:

1. extra spaces
2. misspelled words (i.e. all letters, not right order; first and last letters are correct)
3. decimal - not enough places is marked as incorrect, but too many is marked as correct if it rounds to the correct value.

1.2.7 Menu - driven Testing

The program will now generate test cases for programs that run off a menu. The specifications for how the menu is set up in the program are stored in filename.spec

1.2.8 Code Coverage

The program will keep track of how many lines of the tested code were covered by the tests it ran.

1.2.9 Code Performance

The program will also keep track of which functions in the tested program use 10% or more of the time.

1.3 Systems Goals

The goal of this system is to grade students' `.cpp` file(s) just by typing `grade <filename>.cpp`. The product is built to test the `.cpp` file(s) 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:

1.5 Technologies Overview

- Notepad - Andrew's preferred coding method
- Notepad++ - Erik's Preferred coding method
- Netbeans - Jonathan's preferred coding ide
- gcc - The school's preferred code compiler for c and c++ programs.
- trello - used to help manage sprint tasks `https://trello.com`
- github - used as a code repository and management system. `https://github.com`
- LaTex - Used to create documentation

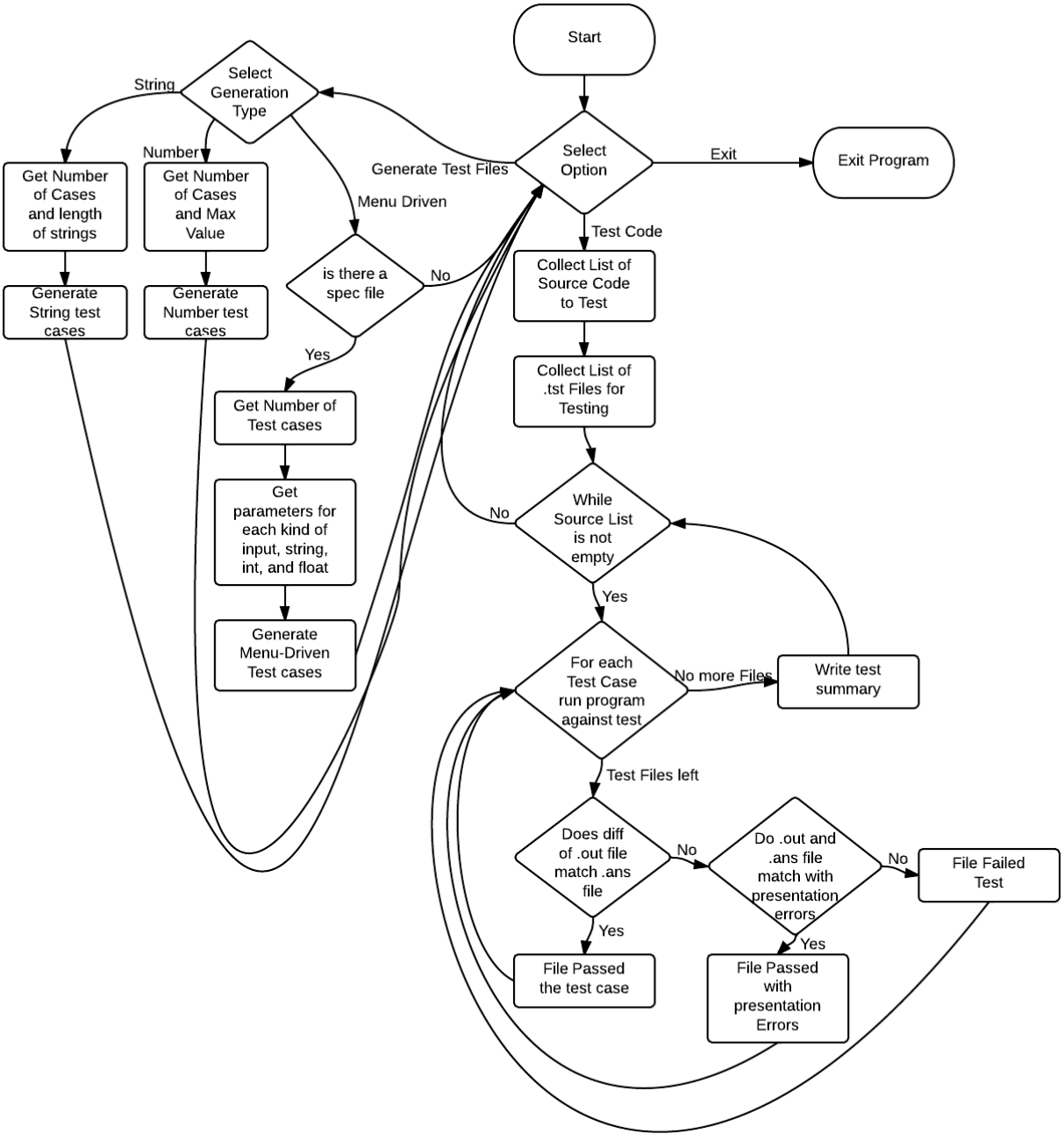

Figure 1.1: System Diagram

2

Project Overview

2.1 Team Members and Roles

2.1.1 Sprint 1 with Latex Samurai

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

2.1.2 Sprint 2 with Kernel_Panic

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

2.1.3 Sprint 3 with The Adventure Line

- Erik Hattervig - Product Owner
- Andrew Koc - Technical Lead
- Jonathan Tomes - Scrum Master

2.2 Project Management Approach

The approach taken to manage this project is `scrum`. The project is broken into tasks to be completed over two-week sprints. The tasks are listed in `Spring Backlog` in Trello. During the sprint, the team meets for ten to twenty-minute scrum meetings to explain their progress, next steps, and impediments. After each meeting, the whiteboard notes are posted as images to the appropriate Trello board.

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 tested, it is stamped as `Complete`. Then the member moves to the next task. After each sprint the tasks are moved from complete to `Product Backlog`

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 many students' `<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)

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

For sprint 3 of this project Erik Hattervig is the product owner, he kept track of the product backlog and communicated with that scrum master and technical lead to determine the specifications of the project's sprint 3.

3.2.2 Management or Instructor (Scrum Master)

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

For sprint 3 of this project Jon Tomes is the scrum master, He managed the sprint back log, breaking the project into smaller tasks, and was in charge of determining what tasks were assigned to members of the team.

3.2.3 Developers –Testers

Anthony Morast is the technical lead for Sprint 1, and is in contact with both Tillma and Sherman regarding the requirements during scrum meetings and through Trello notes.

For sprint 3 Andrew Koc was the technical lead.He was in charge of managing how the project was implemented in the code.

3.3 Business Need

This product is essential for grading computer science programs focused on numerics. All the user has 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 the Linux machines in the Opp Lab.

3.4.2 Network Requirements

This software does not require an internet connection of any sort.

3.4.3 Development Environment Requirements

There are not any development environment requirements except that there must be a C++ compiler.

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 #1

: As a professor I want to be able to automatically grade a student's source code so that I can save time.

3.5.1.a User Story #1 Breakdown

The highlight of this story is "automatically". The professor does not want to have to interact with the grading tool after it is run.

3.5.2 User Story #2

As a professor I want to be able to provide test cases with answers so that I can grade different program assignments.

3.5.2.a User Story #2 Breakdown

This story refers to assembling test cases and running the student code on those test cases

3.5.3 User Story #3

As a professor I want to be able to regrade a students source code without loosing the data from a previous grading so that I can see what changes upgrading test cases have.

3.5.3.a User Story #3 Breakdown

This story refers to saving data. Data from an old runtime should not be removed by a new runtime. Data should be appended to a neatly formatted file for viewing.

3.5.4 User Story #4

I want to be able to run this program on an entire class of students.

3.5.4.a User Story #4 Breakdown

The highlight of this story is that the user wants to be able to run the program one time for an entire class of students. This will require more directory crawling.

3.5.5 User Story #5

I as a user want to be able to generate test cases within a certain range instead of being able to only use existing test cases.

3.5.5.a User Story #5 Breakdown

The important part of this user story is generating test cases. The user will need to provide a min/max, a type of int/float, a number of cases to generate per file, and a number of files to generate. This will be the only portion requiring user input beyond command arguments.

3.5.6 User Story #6

I as a user want to be able to be able to make a certain test case critical. Meaning, that test case will make the student's program a complete failure if their program does not process it properly.

3.5.6.a User Story #6 Breakdown

The highlight of this story is the potential for a student to "critically fail" a program. That is, it my no longer matter how their code does on other test cases because if it fails a critical one, it instantly fails.

3.5.7 User Story #7

I as a user want to be able to terminate programs that run too long.

3.5.7.a User Story #7 Breakdown

This story refers to student programs that are stuck in an infinite loop or are very inefficient. This user story will be implemented by terminating student programs that run for more than 60 seconds and giving the student a failing grade in the log.

3.5.8 User Story #8

I as a user want to be able to generate test cases for menu based programs based on a template that I will provide.

3.5.8.a User Story #8 Breakdown

This story tells us that we must create another method of generating test cases for menu based programs. We will accomplish this by having the user create a .spec file that will contain the template for the menu. The test cases that it provides should do bounding checking on the menu as well.

3.5.9 User Story #9

I as a user want to be able to have the program ignore presentation errors in the student's output.U

3.5.9.a User Story #9 Breakdown

This user story tells us that we need to count programs that output presentation errors as a correct answer. Presentation errors include errors such as misplace decimal places and spelling mistakes in outputted words. For spelling mistakes we will check if the first and last letter of a word are correct and if the number of letters in the word is correct for it to be considered a spelling mistake.Us

3.5.10 User Story #10

I as a user want to be able to view the code coverage for each student's program to see if any of there code was not run and how many times parts of it were run.

3.5.10.a User Story #10 Breakdown

This user story tells us that we need to generate code coverage statistics for each of the student's programs using an external tool that we will then use to create logs of how many times each line of code in the student's program was run. This will allow the user to easily create and access this information.

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. Note that the code to generate test cases is only run if there is a -g cammond line switch is specified after the directory to test is entered.

Here is an overview of the algorithm:

- Determine if there is a .cpp file in the root directory, if so it is used to generate test cases
- Create a vector containing every subdirectory in program folder
- Change to directory where student subdirectories are located
- Step through each directory in the directory vector and determine if it contains tests or student source code
- Create a vector containing the name each student's source and one containing the path to their source code
- Create a vector containing all of the test cases
- While the source code vector is not empty
- Compile the program
- Determine if the student passes the critical test cases, if not stop testing
- Run code against tests in the test vecotor
- Count whether the program passed or failed test case
- If it failed with a normal diff, test if it passes with presentation errors
- Change back to home directory
- Write log file containing percentage of tests passed and final grade
- Write students results to summary file

4.1 Traversing Subdirectories (Sprint 2)

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 (Sprint 2)

4.2.1 Technologies Used

The software was designed in the Linux Environment.

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);
}
```

4.3 Testing to Allow for Presentation Errors

4.3.1 Technologies Used

This set of function relies on the string, cmath, and fstream libraries to test if two files match, allowing for presentation errors.

4.3.2 Design Details

This software checks if two files match, allowing for presentation errors. These presentation errors include, ignoring capitilization errors, if two words start and end with the same letters then they are counted as matching, if two words have the same letters but the wrong order they are counted as matching, if a number would round to the correct value it is considered correct. Below is the code for comparing two files.

```
bool cmpFiles(string s1, string s2)
{
    ifstream file1, file2;
    string in1, in2;

    file1.open(s1.c_str());
    file2.open(s2.c_str());
    if( !file1 || !file2 )
    {
        cout << "Files to be compared could not be opened" << endl;
        file1.close();
        file2.close();
        return false;
    }

    while (file1 >> in1)
    {
        if (file2 >> in2)
        {
            if(in1 != in2)
            {
                if(isNumber(in1))
                {
                    if(isNumber(in2))
                    {
                        //both inputs are numbers
                        //if in2 doesn't round to in1 file is wrong
                        if (!cmpNum(in1,in2))
                        {
                            //file failed
                            file1.close();
                            file2.close();
                            return false;
                        }
                    }
                    else
                    {
                        //in1 is number, in2 is not, file is wrong
                        //file failed
                        file1.close();
                        file2.close();
                        return false;
```

```
                    }
                }
                else
                {
                    if(isNumber(in2))
                    {
                        //in1 is a string in2 is a number, file is wrong
                        //file failed
                        file1.close();
                        file2.close();
                        return false;
                    }
                    else
                    {
                        //in1 and in2 are strings
                        //if in1 and in2 don't match with presentation errors
                        if (!cmpString(in1,in2))
                        {
                            //file failed
                            file1.close();
                            file2.close();
                            return false;
                        }
                    }
                }
            }
        }
        else
        {
            //file is missing information
            //file failed
            file1.close();
            file2.close();
            return false;
        }
    }

    //if the second file has additonal information
    if (file2 >> in2)
    {
        //file failed
        file1.close();
        file2.close();
        return false;
    }
    file1.close();
    file2.close();
    return true;
}
```

4.4 System Diagram

The following are system diagrams for first, the overall program, and the second a more detailed one of how to determine if two files match with presentation errors.

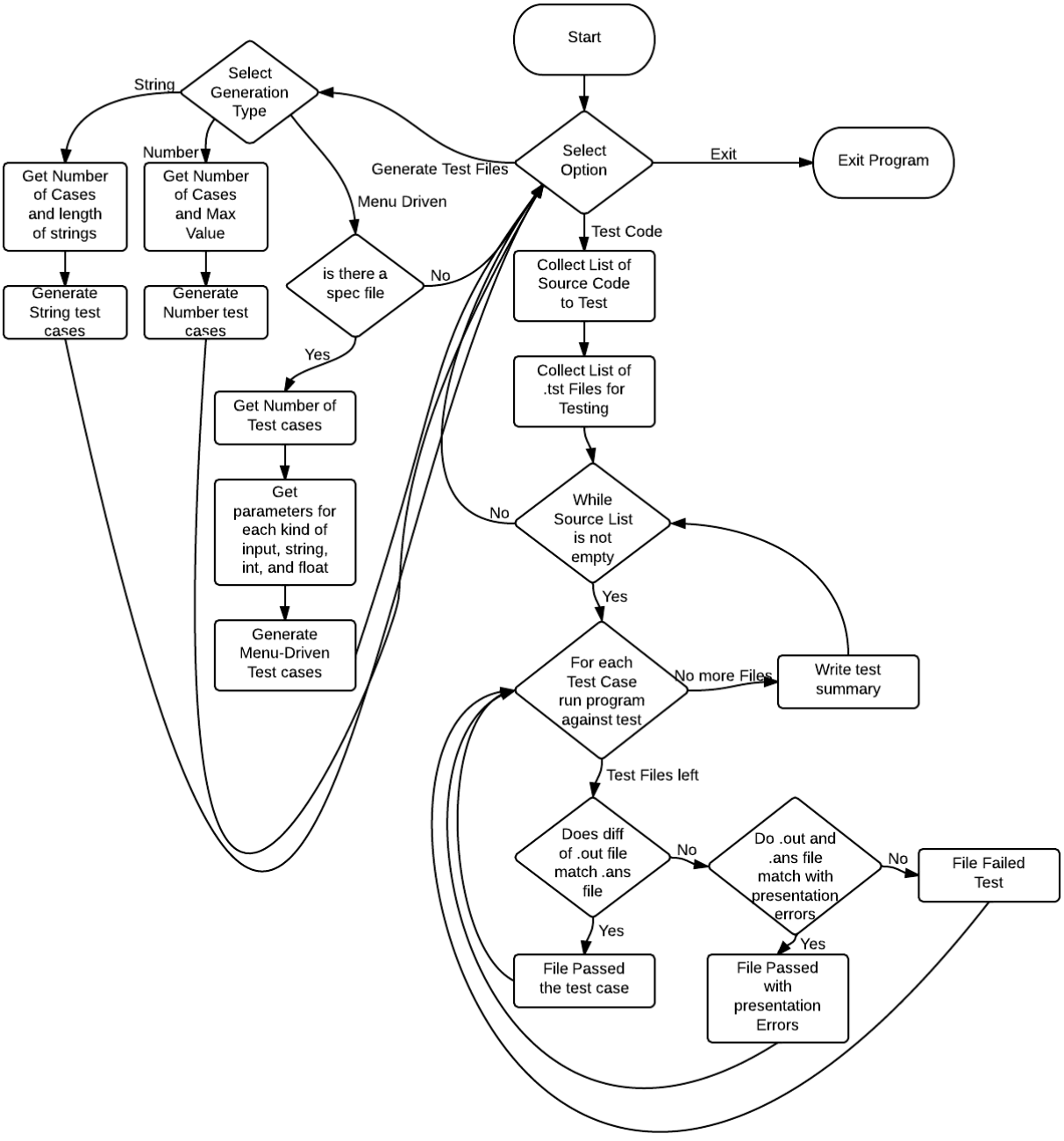

Figure 4.1: System Diagram

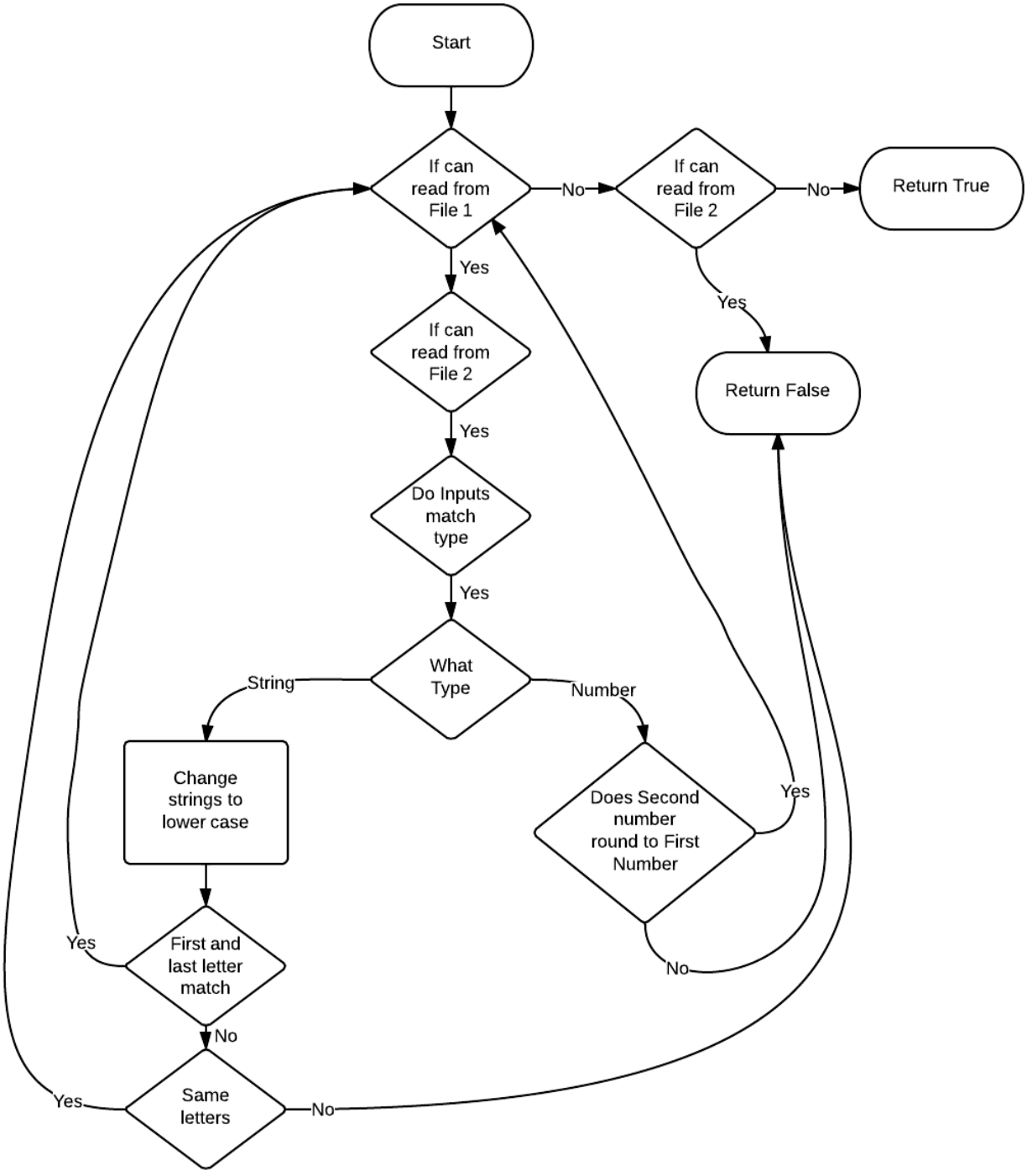

Figure 4.2: System Diagram for Testing to Allow for Presentation Errors

5

System and Unit Testing

Testing was first done on individual function, or processes if a process required more than one function, and then on larger parts of the program as our individual peices were integrated.

5.1 Overview

In general, we began testing on our directory traversal and determining which type of subdirectory we had encountered. Once we could filter out the source code and test/answer files we tested our ability to change into these directories and perform the proper actions, compilation for source code and testing the student's program for test files. After we were able to do all this we began testing our log files and summary file and adjusted the foramtting as needed. Finally, once the program was working, we implemented and tested running the test generation code and critical test cases as a part of the program, rather than individually.

5.2 Dependencies

No dependencies other than a traversable directory be specified on the command line.

5.3 Test Setup and Execution

5.3.1 Student Grading

The student grading section involves

- Crawling through student directories.
- Running each students source code on the test cases.
- Generating test cases.

The product was tested on the customer supplied example class directory and class directories where created to for further testing.

Below is a list of the individual directories that were used in testing as well as the difference between them that tested aspects of the product.

Customer Test CSC_150 Class Directory This product test demonstrates the products ability to test a class without any critical test cases and multiple directories. It also demonstrates automatic test case generation.

```
Customer Test CSC_150 Class Directory:
  Students:
    bad_student_1 Fails some test cases
    bad_student_2 Fails some test cases
    student_3 Passes all test cases
    student_4 Passes all test cases

  Multiple directories containing tests
  No critical test cases
  Golden Source Code: average.cpp
  directories containing useless files
```

Test Directory 1 This created test demonstrates the products ability to fail a student if they do not pass a critical test case. It also demonstrates automatic test case generation.

```
Test Directory 1:
  Students:
    Alex_Johnson
    Bob
    Francis
    John

  Critical and normal test cases
  Golden Source Code: max_3.cpp
```

Test Directory 2 This created test demonstrates the programs ability fail a student if they do not pass a critical test case; it demonstrates the ability of the product to find test cases in subdirectories; finally it demonstrates the products ability to allow test case generation only when a golden source code is available.

```
Test Directory 2:
  Students:
    student_1 Fails critical
    student_2 Fails critical
    student_3 Passes all test cases
    student_4 Passes all test cases
    student_5 Passes all test cases
    student_6 Passes all test cases
    student_7 Passes all test cases

  Test case subdirectories
  Critical and normal test cases
  Golden Source Code: none
```

Test Generation Function Multiple test cases were run against the function that generates the random values for the generated .tst files. Some of these test scenarios are:

- Passing in different values when either "int" or "float" are selected initially (typical testing)
- Passing in floating minimum and maximum values to integer generation.
- Intentional use of improper values.
- Generating test cases in a directory where they had previously been stored (will not overwright old files).

Although we have extensively tested this code, there are a few errors that were not handled (explained in this function's code documentation). These errors were not handled as they are "special cases" and would sacrifice the readibility of the function.

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

This program was developed in a Linux environment. The g++ compiler was used for compilation and generic text editors were used to write the code (QT,gedit,etc.).

6.2 Source Control

Git was used for source control in the project. Our repository is setup on the GitHub website. However, it is private and can only be accessed once a developer is deemed a collaborator.

6.3 Dependencies

There are no atypical dependencies to compile and run this program.

6.4 Build Environment

The program may be built using the g++ compiler on linux. There is also a `Makefile` which allows the user to type `make` to compile the program.

7

Release – Setup – Deployment

7.1 Deployment Information and Dependencies

This program has no dependiencies other than it needs to be compiled and run within a linux environment.

7.2 Setup Information

The program is built with the g++ compiler on linux. The program can also be built by typing `make`.

7.3 System Versioning Information

The program will be versioned depending on the current Agile sprint. The current sprint version is 2.0.

8

User Documentation

8.1 User Guide

This product is to make it easy for you to grade multiple computational computer program written in C++ language. In order to benefit from this product, the student directories must all be contained within the same "root" directory. Within each student's directory there must be a .cpp file that has the same name as the student's directory (omit the extension). That directory should also contain the test cases the user wishes to run against each student's program. Finally, if the user wishes to have test cases generated, the root directory should contain a .cpp file which will be compiled and used to generate answers to the generated test cases. In summary the root directory should contain:

- The subdirectories containing student source code
- Test files or subdirectories containing test files (with `.tst` extensions)
- Corresponding solution files (with `.ans` extensions)
- A .cpp to generate answers to generated test cases.

As a user, you must complie this program in a Linux environment. To run this program the first command line argument must be the name of the dirctory you wish to test (note this program must be executed in a directory "one step back" from the directory you wish to test. If you want to generate test cases, a second command line argument "-g" must be specified when executing the program.

8.1.1 Test Case Generation

If a "-g" is the second command line arugment and a .cpp file exists within the directory being traversed prompts will be displayed to determine what type of test cases you want to generate. You may generate floting point or integer values, decide in what range you want the values generated, decide how many values to generate in each .tst file, and choose how many .tst files you want to generate. After all these values are specified, the program will create each .tst file, compile the "golden.cpp" file, run each test file through the golden.cpp executable, and output to individual answer (.ans) files.

9

Class Index

This section is intentionally left blank as there are no classes in this project.

10

Class Documentation

This section is intentionally left blank.

Acknowledgement

Special thanks goes to LaTex Samurai for their well documented student testing system.

Supporting Materials

This section is intentionally left blank.

Sprint Reports

10.1 Sprint Report #1

Intentionally left blank (there was no sprint report by Latex Samurai).

10.2 Sprint Report #2

At this point there is only one sprint. So far, our product seems to be functioning to customer specifications. There are no known bugs in our project related to Assignment Grader. It meets each of the requirements outlined by the user stories. It does do "good practice" error checking on user input. For a second release version, this meets everything that it is required by the customer, Dr. Logar.

This sprint was differnent in that most of the time was spent in the testing phase. We believe this was due to the heightend complexity of the product. Also a lot of time was lost on consolidation of each team members code in to one working product. In the interest of the future of the product, the members of Kernel Panic will remain in their current positions for the next sprint.

10.3 Sprint Reports for Team #3 - The Adventure Line by Jonathan Tomes

10.3.1 Sprint 1

Forgot to do this back on sprint 1, so I'll do a retrospective on sprint 1 now at the end of sprint 3.

Overall sprint 1 went off with out a hitch. Everyone did their parts and contributed well to the project.

10.3.2 Sprint 2

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

10.3.3 Sprint 3

This sprint was a complete reversal of sprint 2's code. This time there were FAR too many functions in the code that we got. It was extremely difficult to work out how the original code worked. Modifying the existing code was extremely difficult. I feel that myself and the other members were kind of drained after the second sprint, along with all the other projects in our other classes. There were several problems with the code that we received. Poor in-line commenting and explanation of logic; poor naming conventions in both function names and variables; functions being called with out any explanation as to why it was being called or what it was expecting to get back; etc. Several example functions of such are: usage, err_usage, generateFiles, isGolden, run_file, test_loop, test_code.

Sadly because of these difficulties, we were unable to get all of the desired functionality working with sprint 3. I'm a little disappointed in my self for not starting sooner to find these problems sooner.. but I'm unsure how much that would have really helped. There would have been no way to integrate our code from sprint 2 into the code we received. So short of having deciding to scrap the code we got... I do not know if there is anything that could have been gained by starting sooner. As for why we did not scrap the code that we got, we were unsure if that was even a valid option. For if this was supposed to be based on working in a company, if we scraped the code we would have received from the company we would have had to start over from scratch.

In short, this sprint could have gone a lot better, but we learned a few things from it.

Industrial Experience

This section is intentionally left blank.

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 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 Latex Samurai for a well developed product and Dr. Logar for being an enthusiastic involved customer!

Bibliography