SE2832 Lab 1 Triangle Trouble

1 Introduction

Telly loves triangles. At least he does on Sesame Street. However, have you ever thought of how to go about determining if a set of numbers defines a valid triangle?

In this exercise, you are going to work with a lab partner to test and verify a piece of software. The software reads in a file which defines triangles and prints out whether the line defines a valid triangle or not and what type of triangle it is.

2 Lab Objectives

- Generate test cases for a problem.
- Write accurate and complete defect reports
- Enter defect reports into a mock defect tracking tool
- Define test cases that verify that a defect has been fixed.
- Correct defects in software

3 Lab Overview

This lab will involve working with a single lab partner. One partner is to define test cases and test the software, finding defects within the software. The other partner is to fix the software so that failures do not repeat again.

The lab begins with the team downloading the triangle analysis program from the blackboard repository. The program itself has the user enter a filename. The file itself contains a set of definitions for triangles that are to follow the format defined below. The program reads in each line and processes it, determining if the triangle is valid. If the definition is valid then an appropriate analysis (scalene, isosceles, or equilateral) will be printed out to the console. If it is invalid, an appropriate message will be printed out. For example, the input line:

```
3.0 3.0 3.0 # This line demonstrates a basic equilateral triangle
```

should print out

```
The triangle 3.0 3.0 3.0 is a valid triangle. The triangle is an equilateral
triangle.
```

The input line :

```
10.0 3.0 4.0 # This line represents what should be an invalid triangle.
```

Should print out

```
The triangle 10.0 3.0 4.0 is not a valid triangle.
```

A valid line in the system starts with the definition of the three sides (a,b,c). Each side may be either an integer or a floating point number, and the two types may be combined together. The three entries may then be followed by a comment. Comments start with the character #. Anything after the # is to be ignored.

A valid triangle meets the following criteria:

$a + b > c$

$a + c > b$

$b + c > a$

$a > 0$

$b > 0$

$c > 0$

A **scalene** triangle is defined as a triangle in which
$a \neq b \neq c$

An **isosceles** triangle is defined as a triangle in which
$a = b \neq c \cup a \neq b = c \cup a = c \neq b$

An **equilateral** triangle is defined as a triangle in which
$a = b = c$

The program itself should not crash, and more so, should indicate lines that are incorrectly formed do not match the input criteria for the program.

4 Lab Specifics

Once you determine your lab partner, you are to partition the work. One of you will draft a set of test cases for the program. The test cases represent inputs to the program. For example, the following shows two lines that might be present in the test cases:

```
3.0 3.0 3.0 # This line demonstrates a basic equilateral triangle
2.0 3.0 4.0 # This line demonstrates a basic scalene triangle.
```

As a tester, you will develop test inputs and execute the tests. If a problem develops, you will enter a defect report for the developer to fix.

The second lab partner represents the developer. As the developer, you are to receive the defect reports back from the tester and are to use them to try and fix the software so that it doesn't break when the program executes. You may want to create a git project to do this. Once you, as the developer, have fixed all the bugs submitted by the tester, you should turn the final version back over to the tester who will then execute the test cases against the final program. If all bugs have been fixed, the program should run without an

issue. If further bugs are found, then the tester will file a bug report and the process will continue until the tester no longer uncovers and bugs with the submitted software.

5 Deliverables / Submission

Now that you have completed your lab assignment, submit the following lab report detailing your experiences. The lab report should be submitted electronically through the course upload page. One lab report should be submitted per pair of students.

1. Introduction
   a. What are you trying to accomplish with this lab? This section shall be written IN YOUR OWN WORDS. DO NOT copy directly from the assignment.
2. Test Developer
   a. How did you go about determining your test cases?
   b. How did you organize your test cases?
   c. How many defect reports did you file?
3. Test Cases
   a. What were the test cases you used to test the software? (Simple copy and paste the text file which contains your test cases.)
4. Bug Fixer
   a. How many bugs did you end up fixing?
   b. Based on the descriptions provided by the tester, how easy was it to understand the root causes for the test problems?
5. Things gone right / Things gone wrong
   a. This section shall discuss the things which went correctly with this experiment as well as the things which posed problems during this lab.
6. Conclusions
   a. What have you learned with this experience?
   b. How has this lab experience changed your attitude toward testing and when you should work on testing your projects?

In addition to this, you should upload into blackboard a zip file with the fixed source code so that the instructor can test against a set of “golden” tests.

If you have any questions, consult your instructor.

Appendix A. Writing Defect Reports

Defect reports are technical documents files by testers and other software users when a program appears to behave incorrectly. They provide tangible and traceable artifacts that document areas where software is performing in either an incorrect or unanticipated manner.

On any large scale project, thousands of defect reports may be received. Many will be of high quality and useful to the developers for quality improvement. But, many reports will also be bad. Instances of these types of reports may include:

- Reports that say nothing ("It doesn't work!")
- Reports that make no sense
- Reports that don't give enough information
- Reports that give wrong information.

Proper defect reporting requires practice, and it is a skill that is perfected not instantaneously obtained. While the specifics of defect reporting may vary given the environment and the uncovered defect, there are few things in common with all defect reports that can be used to make them better.

The goal of a defect report is to enable the development team to reproduce the failure which has been uncovered. Being concise yet detailed is important. It is important that the report clearly indicate the facts of the situation ("The output was 5 when it should have been 7"). Unless you are a member of the development team and have intimate knowledge about the system, do not speculate. Speculation simply increases the defect report and may send the developer down the wrong track. Defect reports should never be hostile or place blame on the development team, but rather should focus on conveying the information necessary to reproduce the defect in the development setting.

Each and every defect filed should have an accurate title which summarizes the defect in a few words. These words should completely and succinctly describe the issue. In some cases, a slightly longer title can add tremendously to the comprehension of the defect which needs to be fixed. For example, the title "Search Selected Text" conveys significantly less information than "Find and Replace dialog should default to last-searched term".

Once the title has been determined, a good defect report describes specifically what failed. Was an error code generated? If so, which one. Did the program crash? If it did, how did it crash, and did any diagnostic info get written to the screen. This can help tremendously when debugging a defect for the developer to ensure that what they are seeing matches what you saw during testing or program usage. If you can capture a screen shot, that is great, but do not simply file the screen shot without explanation. Same goes for a core dump on a UNIX system.

Next, a good defect report describes specifically what was occurring when the defect was found. This should be done in a clear yet minimalistic manner. Do not go back to turning on the PC and tell everything that was done, but highlight the key steps that you did to cause the problem to occur. If you can reproduce the defect, be sure that is mentioned as well. A defect that is extremely reproducible if one follows a set of steps is significantly easier to resolve than one which is not reproducible. When writing the steps, repeat them through after you have written to make certain you have not left out a step.

After documenting the process to create the defect, it is important to describe to the best of your ability the system configuration upon which the defect occurred. What version of software is being used and what OS? What libraries are present that may affect the behavior? Have you installed add ins? Did you

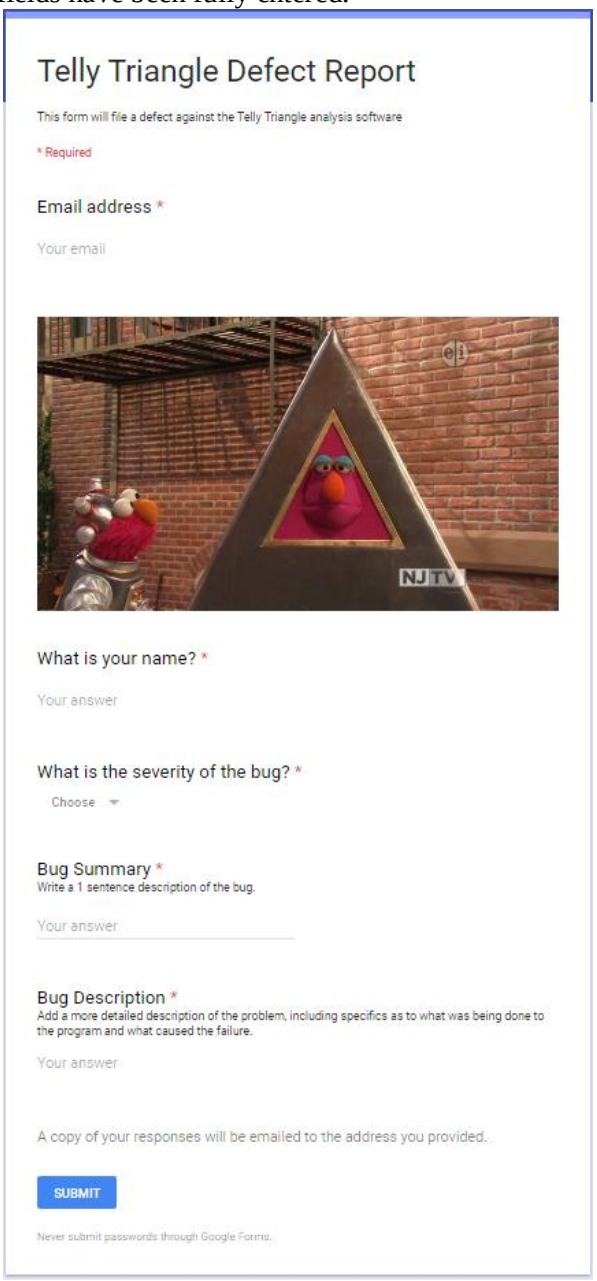
recompile the kernel with a patch from another vendor in order to add new functionality underneath the program?

A good defect report only addresses a single issue. If in testing you uncovered five problems, a total of five defect reports that should have been filed. Do not try to be efficient and combine defect reports, for this makes it difficult to assess and fix. When using a traceable process such as Unified Change Management (this is a topic for SE3800), each defect report will be linked with a defect fix and tracked to closure. If multiple defects are submitted on a report, this important traceability step cannot be completed.

Appendix B: Defect Report Filing

To file defect reports in this lab, you will be using a Google form to simulate a defect tracking system. A link to the form is available in Blackboard. While this is not a full defect tracking system, it includes the important features necessary to enter defects in a defect tracking system.

The figure below shows a defect report template. The template will not allow you to submit the defect report unless the important fields have been fully entered.



Overall, the fields are relatively self-explanatory. Your name and email are used to track the originator of the defect.

The defect summary is a one sentence summary of the defect. This should include only the most important aspects of the defect. The bug description is a larger field which you can use to enter the details of the defect: what specifically you did to cause the defect, and exactly what happened when the defect was encountered.

The severity field indicates how severe the defect is to the operation of the software. Definitions for this category are provided in the table below.

Severity	Description
Blocker	Reserved for catastrophic failures - exceptions, crashes, corrupt data, etc. that (a) prevent somebody from completing their task, and (b) have no workaround. These should be extremely rare. They must be fixed immediately (same-day) and deployed as hotfixes.
Critical	These may refer to unhandled exceptions or to other "serious" bugs that only happen under certain specific conditions (i.e. a practical workaround is available). No hard limit for resolution time, but should be fixed within the week (hotfix) and must be fixed by next release. They key distinction between (1) and (2) is not the severity or impact but *the existence of a workaround*.
Major	Usually reserved for performance issues. Anything that seriously hampers productivity but doesn't actually prevent work from being done. Fix by next release.
Minor	These are "nuisance" bugs. A default setting not being applied, a read-only field showing as editable (or vice-versa), a race condition in the UI, a misleading error message, etc. Fix for this release if there are no higher-priority issues, otherwise the following release.
Trivial	Cosmetic issues. Scroll bars appearing where they shouldn't, window doesn't remember saved size/location, typos, last character of a label being cut off, that sort of thing. They'll get fixed if the fix only takes a few minutes and somebody's working on the same screen/feature at the same time, otherwise, maybe never. No guarantee is attached to these.