Software Engineering Assignment SDD

Trenton MacNinch, Michael Lin, and Ying Gao

2024/12/03

Contents

1 Introduction

1.1 Date of Issue

2024/12/03

1.2 Authorship

Trenton MacNinch, Michael Lin, and Ying Gao

1.3 Summary

The system architecture will be a monolithic design running on Linux, with interfaces for reading from the provided input files, and to write to the output file.

2 Software Architecture

2.1 Overview

Given the requirements for our application (Object oriented features, compiled language, file handling, etc.), we have chosen to create a C++ program running on a Linux system. Thus, a monolithic architecture is used, placing all functionality within the single executable.

2.2 System Design Concerns

A monolithic design is chosen as it is the simplest design to solve the given problem. As the scope is not large, there are few concerns about scalability and constant usage. The lack of data transfer between programs can be ignored as all data handling can be safely handled within the single program.

2.3 Usage Case View

The program is used by calling a command with the following syntax:

```
calcgrades [name file] [course file] [output file]
```

The application reads two input text files and outputs one file with specific formatting. The input files are given by the user, formatted as the following:

```
[name file]
Student ID, Student Name
100000001, John Hay
100000002.  Mary Smith
...

[course file]
Student ID, Course Code, Test 1, Test 2, Test 3, Final exam
100000001, CP317, 75.3, 80.4, 60.3, 70.5
100000002, CP414, 80.2, 90.5, 50.4, 75.6
100000001, CP414, 60.5, 70.6, 80.6, 75.6
...
```

The output file is formatted as the following, with special notice given to student IDs being in ascending order:

```
[output file]
Student ID, Student Name, Course Code, Final grade
1000000001, John Hay, CP317, 66.7
1000000001, John Hay, CP414, 72.6
1000000002, Mary Smith, CP414, 74.8
...
```

2.4 Component View

As this is a monolithic design, there are no components on the high-level design level to view. Deeper components will be documented in the Detailed Design section of the document.

3 Detailed Design

3.1 Overview

The application loads all student information into a map, then loads all course information into a vector of custom course objects, and then sorts the vector of course objects. The program will then proceed to iterate through all stored course information and attach student names based on IDs while placing it in the designated output file. If the program encounters an error, it will print the encountered error.

3.2 Student Information

The program opens the name file as a normal C++ file handle. The program then calls a function that generates a map. This function reads the info from a line in the name file, validates the given information, and adds the student info to the map if the information is acceptable. Within this map, the Student Information is stored with the Student ID as an integer and the Student Name as a C++ string.

3.3 Course Information

The program opens the course file as a file handle, and uses a function to generate a vector of Course objects. Each Course object consists of a Student ID stored as an integer, a Course Code stored as a C++ string, and all grades stored as floats. The function reads the information from a line in the course file, validates the given course information, and rejects any invalid courses. The program then sorts the vector of course information by student ID to enable a more efficient outputting phase.

3.4 Creating Output

The program takes the information stored in the student id map and the course object vector, and combines it according to the output format while pushing it to the file handle for the output file. This output is printed in the following format, Student ID as an integer, Student Name as a C++ string, Course Code as a C++ string, and Final Grade as a float truncated to one decimal point (nn.n). The program then closes.

3.5 Error Handling

Anytime a new entry is read an about to be added to the data structures, is it first parsed through a validator check. This check first confirms the the given Student ID is a valid nine digit number, then if object being validated is a Student, the Student Name is automatically assumed to be valid (who knows what parents are naming their children nowadays, and yes the input is sanitized). In the case of Course Information, a check is conducted to ensure that the Course Code is a valid (following the XXnnn format), and that all grades are positive (we are assuming that bonus marks are possbile). If any of these checks are invalid, the error handling code will print the first invalid input, stating both the invalidity type, and what was invalid.

4 Glossary

- CourseFile.txt - Provided input file of all Course information, including Student IDs, Course Codes, and all Grades.
- NameFile.txt - Provided input file of all Student information, including Student IDs, and Student Names.
- OutputFile.txt - Generated output file in desired formatting, including Student IDs, Student Names, Course Codes, and Final Grades.

5 Change History

- Nov 14, Added subsections to Introduction
- Nov 14, Made date of issue use system current date
- Nov 19, Added text to Software Architecture Overview
- Nov 19, Itemized list in Software Architecture Overview
- Nov 19, Added example formatting for input and output
- Nov 19, Added .txt to output file filename
- Nov 19, Added command usage and added sections to Detailed Design
- Nov 19, Elaborated upon information and added text to Creating Output subsection
- Nov 20, Changed input/output file formatting section to stress sorted IDs in output file
- Nov 22, Added subsection Error Handling
- Nov 22, Added change history
- Dec 3, Re-organized contents, added addtional information to Software Architecture and Detailed Design