Test Suite report by Kent Spence

#3.1

A component within this system can be identified as any separate file containing either data that will be used in another file, or methods and data that will be used in a separate file. The components that currently interact with each other:

CourseInfo.java

Database.java

GPAcalculator.java

GPAconverter.java

Parser.java

TermInfo.java

TranscriptReader.java

WordExtractor.java

Messages received from other components and how to test them

#1: TranscriptReader messages to Parser

Line 54 -> parser.readTranscriptFile()

Test that if given an invalid file, check if the database returned from the read transcript file is null.

#2: TranscriptReader messages to Database

Line 57 -> db.getStudentName() and db.getStudentID()

Check that the student name and the one given in the file are the same, same for student id.

#3: Line 59, 60 -> db.getNumberOfTerms()

Check if the number of terms in the file matches the one stored in the file.

#4: Line 61 -> db.getTermName()

Check if the term name matches what is in the database.

#5: Line 62 -> db.getNumberOfCourses()

Check if the number of courses matches what is in the file.

#6: Line 63, 67 -> db.hasCourseGradePercentageWithinTerm()

Run it with a file containing a grade for that term, and check if the function returns true. Then run it with a term in the file that has no grade and check if the function returns false.

#7: Line 65, 70 -> db.getCourseNameWithinTerm()

Check if the course name in the file and the course name in the database match for that particular course code.

#8: Line 66, 71 -> db.getCourseGradeLetterWithinTerm()

Check if that particular course grade matches the course grade in the file.

#9: Line 74 -> db.getTermGPA()

No use to test this, if it does not return the correct GPA, that would be the fault of calculator, since this function just returns the result of calculator.getOverallGPA().

#10: Line 77 -> db.getOverallGPA()

No use to test this, if it does not return the correct GPA, that would be the fault of calculator,

since this function just returns the result of calculator.getOverallGPA().

Parser messages to Database

#11: Line 63 -> db.addTerm()

Check if the calculator.addLetterGrade returns -1 and if that term was still added, even though it returned failure.

Parser messages to WordExtractor

#12: Line 66, 89 -> tokenizer.isAtEOF()

Check if this returns true after the parsing is finished.

#13: Line 212 -> tokenizer.getNextWord()

Check if the word that has been returned is null, or if it matches what the next word should be in the file.

Parser messages to GPAconverter

#14: Line 152 -> GPAconverter.getLetterForNumericGrade()

Check if it returns null, and if it matches the numeric grade expected for the given letter grade.

Database messages to GPAcalculator

#15: Line 163 -> calculator.getTermGPA()

Check if the calculated Term GPA matches what would be calculated based on the contents of the file for that given term.

#16: Line 171 -> calculator.getOverallGPA()

Check if the calculated overall term GPA, matches what would be calculated based on the contents of the file for that given term.

#17: Line 182 -> calculator.addLetterGrade()

Check if calculator.addLetterGrade() added the grades, by checking if it returns -1 on failure when the database adds these grades.

GPAcalculator messages to GPAconverter

#18: Line 100 -> GPAconverter.getLetterForNumericGrade()

Check if it does not return null, and if it matches the numeric grade expected for the given letter grade.

#19: Line 132,137 -> GPAconverter.getGPAforLetterGrade()

Check if if it returns a grade under 100, and if it matches the GPA expected for the given letter grade.

How to test these messages:

#3.1 A decision table would help as it can be used to evaluate the truth values of test cases should one condition change, which can be very helpful for a test suite where various conditions are being checked, with each entry existing as one test. With the number of components and the number of interactions present, this will simplify test cases greatly.

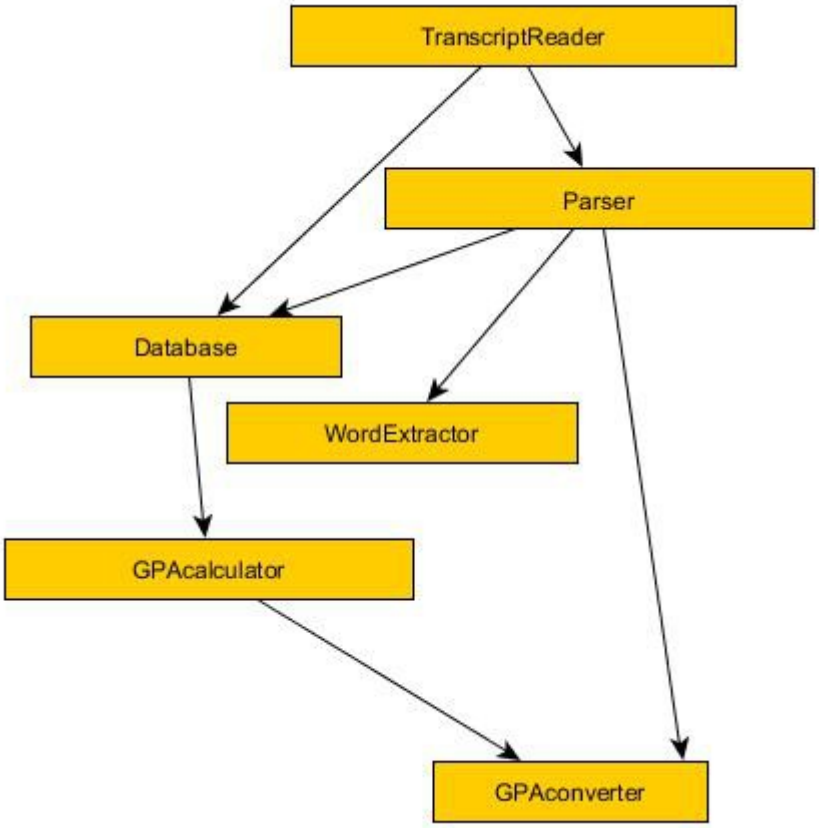
Boundary value testing is likely insufficient, as there are many values that are just checked if they return null or not null, rather than a numeric value. Since the majority of input values are not necessarily numerical values or can be evaluated numerically, boundary value testing is not very

helpful in general.

Equivalence classes are also relevant, since some messages in the database are just function calls to calculator to do the exact function it needs. Through this, by testing the database and the calculator function, you are testing the same thing, since all that database function does in this interaction is “return calculator.nameOfThisFunction();”, so using equivalence classes would eliminate the need to

3.1.1 In order to do all of this testing, I will using a call-graph decomposition strategy where I look at the paths taken from calling different components, in order to set up stubs where interfaces would be necessary.

In order to use these stubs, I will create a java file, or a series of java files, which will be added to these files, in order to test them. I will likely use a scripting language to take each function body, store it in a file called original_ then concatenated by the name of the function being tested, and then I will replace the code of the body with the stub.



3.1.2 Testcases I will require:

I would need around 34 test cases, since I would need a stub designed to give an invalid return value and a stub designed to give a proper return value, so 17 component interactions to test * 2 stubs each = 34 testcases, with 1 stub per testcase.

The 5 testcases I will be implementing are these:

#1: a parser.readTranscriptFile() stub which returns null, to see what happens if there is nothing to parse. Success if it returns that there is a parsing error.

#2: a parser.readTranscriptFile which returns a db file with a student name and id, but which has nothing stored in it. Success if it it returns an error, and not a standard java exception to notify the user what has happened.

#3: tokenizer.isAtEOF() except it returns true. Success if it closes the file successfully.

#4: tokenizer.isAtEOF() except it only returns false. Success if it still closes the file successfully.

#5: stub for calculator.addLetterGrade except it returns -1. Success if it does not add those grades, since this would signify an error with adding the grades as letters, like if the letters are not stored as letters currently.

In order to implement these test cases, I will make a shell script for every test case, where the stub will be added to the java file, and it will be ran with that stubbed java file. The expected and received outputs will then be compared by the script, using the cmp command. If they match, then it will have passed the test, otherwise it will have failed.

3.2.1

Interfaces that require testing with high level use cases:

The interface between the user and the program (the command line)

ID: HLUC1

Name: valid file is given

Description: A valid transcript file is given to the program.

ID: TC-1

Name: File with one course in one term

Description: An input file is given, where there is one input file with one term and one course in that term.

Preconditions: The input file exists, and obeys the syntax required.

Input events:

1. TranscriptReader.java is executed with the input file given as an argument.
2. Contents of the file are displayed to the screen.

Post-conditions: The input file is printed to the terminal screen, with the

GPA calculated, should not be positive infinity.

Test Execution result: Pass/fail

ID: TC-2

Name: File with two courses in one term

Description: An input file is given, where there are two courses in a single term.

Preconditions: The input file exists, and obeys the syntax required.

Input events:

1. TranscriptReader.java is executed with the input file given as an argument.
2. 2. Contents of the file are displayed to the screen.

Post-conditions: The input file is printed to the terminal screen, with the correct GPA calculated, should not be positive infinity.

Test Execution result: Pass/fail

ID: TC-3

Name: File with three terms with two courses each.

Description: An input file is given, where there are two courses for three terms.

Preconditions: The input file exists, and obeys the syntax required.

Input events:

1. TranscriptReader.java is executed with the input file given as an argument.
2. Contents of the file are displayed to the screen.

Postconditions: The input file is printed to the terminal screen, with the correct GPA calculated, should not be positive infinity.

Test Execution result: Pass/fail

ID: HLUC2

Name: invalid file is given

Description: An invalid transcript file is given.

ID: TC-4

Name: File is missing the word “term”.

Description: An input file is given, where there are two courses in a single term. However, the file does not have the word “term”.

Preconditions: The input file exists, and obeys the syntax required in all other ways.

Input events:

1. TranscriptReader.java is executed with the input file given as an argument.
2. Contents of the file are displayed to the screen.

Post-conditions: The input file is printed to the terminal screen, with the correct GPA calculated, should not be positive infinity.

Test Execution result: Pass/fail

ID: TC-5

Name: File has no courses

Description: An input file is given where the two terms both have no courses.

Preconditions: The input file exists, and obeys the syntax required in all other ways.

Input events:

1. TranscriptReader.java is executed with the input file given as an argument.
2. Contents of the file are displayed to the screen.

Post-conditions: The input file is printed to the terminal screen, where the GPA should be positive infinity for both terms.

Test Execution Result: Pass/Fail

ID: TC-6

Name: File does not exist

Description: No input file is given.

Input events:

1. TranscriptReader.java is executed
2. Displays error to show that there is no file.

Post-conditions: Error is printed to show it could not find the specified file.

Test Execution Result: Pass/Fail

These test cases would all be implemented by creating a file with the specified changes, and then running the program with that changed file on the command line using a shell script. Each test case will need a modified input file, its own shell script, as well as a file containing its expected output to compare with the received output. The only exception to this is test case #6 which has no text file, but this test case would still be ran using a shell script and its output would be compared to a file I have made containing the expected output.

References:

I worked with Julian Tang 0887663 on this assignment, as he was my lab partner on lab4, and he helped me with determining a testing plan for this assignment, particularly for system level testing.