010-2 Component Application Programming Report

George Pake | SID2109071

Table of Contents

[1 Introduction 1](#_Toc255593646)

[2 Application Model 2](#_Toc778612644)

[2.1 Methodology 2](#_Toc1765552577)

[2.2 Design Patterns 4](#_Toc1642217446)

[3 Implementation 5](#_Toc821266796)

[4 Testing and Results 6](#_Toc244161023)

[4.1 Functional Testing 6](#_Toc405623600)

[4.2 Unit Testing 9](#_Toc1781588983)

[4.2.1 Round 1 9](#_Toc1350826309)

[4.2.2 Round 2 10](#_Toc1175610936)

[5 Conclusion 11](#_Toc1297394030)

# 1 Introduction

The aim of my application is to analyze student results from a text file. I have used the IntelliJ IDE to process my application in Java. After reading in the file containing the relevant information, the application includes a menu which takes user input. The following features have been implemented under each menu option:

* Printing out a results list including every student. The lists contain the students' first name and surname, along with their final result. This is the core functionality of the application.
* Printing out results for one year.
* Printing out results for one subject.
* Printing out first and last names of students who have failed an individual module.

For each of the options above, I have included each student’s grades, which are based on their final result.

In later sections I have explained how I structured the entire application and produced each individual output above. I have also rigorously tested the application to make sure that each output is expected for whatever is inputted. Object oriented design principles such as encapsulation and inheritance have been used in the application and referenced in the report.

# 2 Application Model

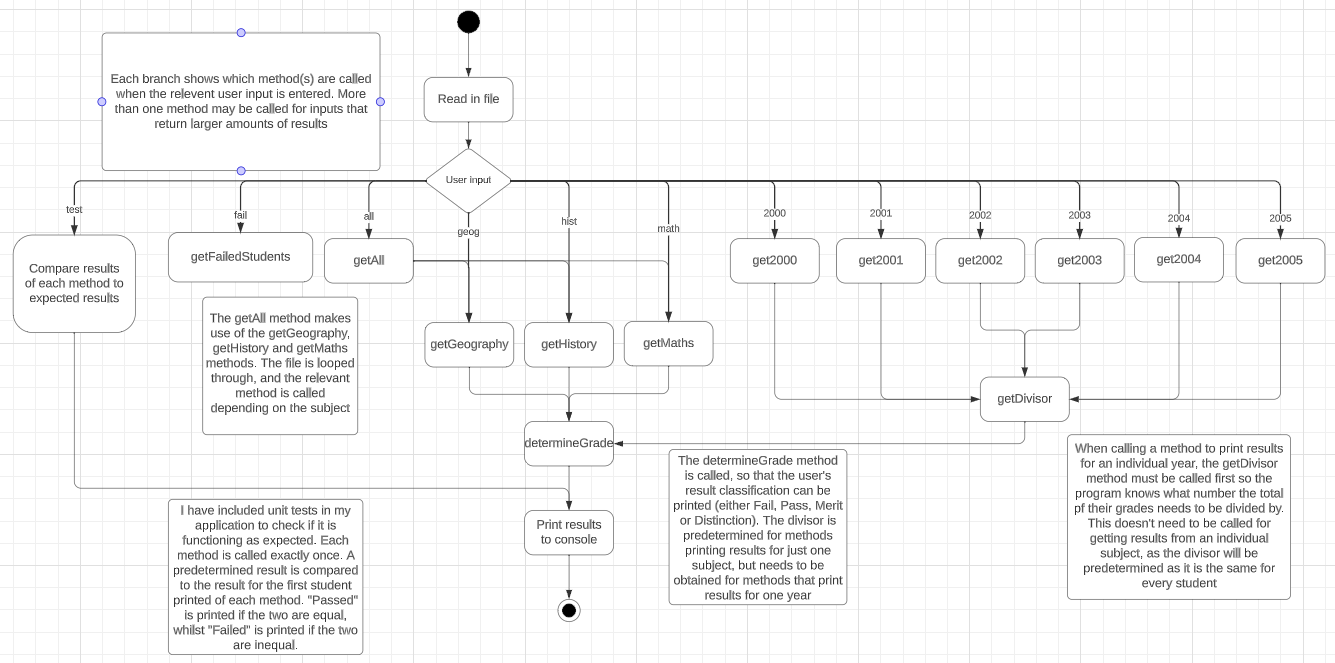
## 2.1 Methodology

Table 1 below describes some of the methods and variables used in the application. Before the system logic, a BufferedReader object is created which is needed to read in the file. The methods are accessed as the file is read line by line.

|  |  |  |
| --- | --- | --- |
| Method | Parameters | Description |
| getFailedStudents() | None (return type void) | This method prints the firstnames and surnames of the students who have failed an individual module (grade<40). |
| getAll() | Geography geogObj  History histObj  Maths mathsObj  float total  String moduleResult  Return type String | This method prints the year, firstname, surname, subject, result and classification of every student. It calls the methods of getting results from an individual subject for each line of the file. |
| get2000() | Student myStudent  String moduleResult  Geography geogObj  float total  Return type String | This method prints the year, firstname, surname, subject, moduleResult and classification of every student who graduated in the year 2000. It uses the getDivisor method which depends on the subject, and determines the moduleResult (result classification) by using the divisor |
| get2001() | “” | “” year 2001 |
| get2002() | “” | “” year 2002 |
| get2003() | “” | “” year 2003 |
| get2004() | “” | “” year 2004 |
| get2005() | “” | “” year 2005 |
| getDivisor() | String subject | This method is used to know what number the total of the grades is to be divided by. If the subject is Geography, the student’s grade total is divided by 8 to calculate their final result and so on. |
| getGeography() | String moduleResult  float total  Geography geogObj | This method prints the year, firstname, surname, subject, moduleResult and classification of every student who is studying Geography. The divisor for the determineGrade function is set to 8 (they take 8 modules |
| getHistory() | “” | “” History, divisor 10 |
| getMaths() | “” | “” Maths, divisor 12 |
| determineGrade() | int divisor  float total | This method is used to determine the students result classification. The grade boundaries are:  <40% Fail,  40%-59% Pass  60%-69% Merit  >69% Distinction  The divisor passed into this method is pre calculated depending on the subject. The total will be divided by this. The moduleResult is set to whichever boundary the final result falls into, and is returned and displayed alongside other student information depending on user input. |

*Table 1: Table of methods used in the application*

The activity diagram in Figure 1 shows how a user could interact with the program in order. This shows how each part of the program functions and also how I have implemented testing inside the program.

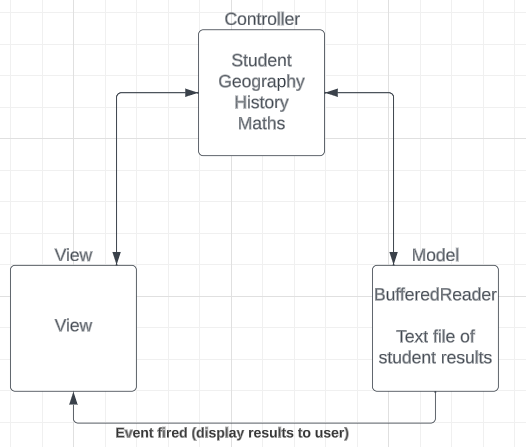
*Figure 1: Activity diagram for my application*

Before the user runs the program, a pre-condition is that they make sure that they are using the correct file name and path. If the user enters an alternate input to the ones listed above, no results will be printed to the console and the program will be terminated. Upon a successful running of the program, the BufferedReader will be closed, and the file will remain intact (no data is written to the file inside my application), and the correct results will be printed to the console. If the user wants to retrieve more data, they will have to run the program again. I am assuming for the testing option, that every result printed to the console is correct. I have only tested the first result for each option, as testing all would be a long and time-consuming process.

My final design made use of Object-Oriented coding principles. This included accessing and mutating variables, which is an example of encapsulation. This means that data objects inside a class cannot me directly accessed. I made sure to include methods inside each class, so that I was not repeating large chunks of code, and making the program easier to read and understand myself. I inherited from the Student class for each type of student. This was useful so that I would not have to reuse the attributes and methods from inside the Student class, making it more efficient and the program was better in structure

## 2.2 Design Patterns

I have used the Model View Controller (MVC) design pattern for this application. In my application as shown below in Figure 2, the model is represented by the data file, which includes the student records to be processed. This is read whilst the user inputs an option of what data they want to be processed (contained in the View class). The controller is represented by the other classes in the program, including Student and its subclasses of Geography, History and Maths. Each class contains methods which display a result to the View class. This is a suitable choice of model for my application because it is a structured approach that separates code efficiently. If I was improving or making changes to the application, the controller and view could be changed without modifying the data processing logic in the model. Having a clear separation between each part of the model made it easier for me to manage and extend my application.



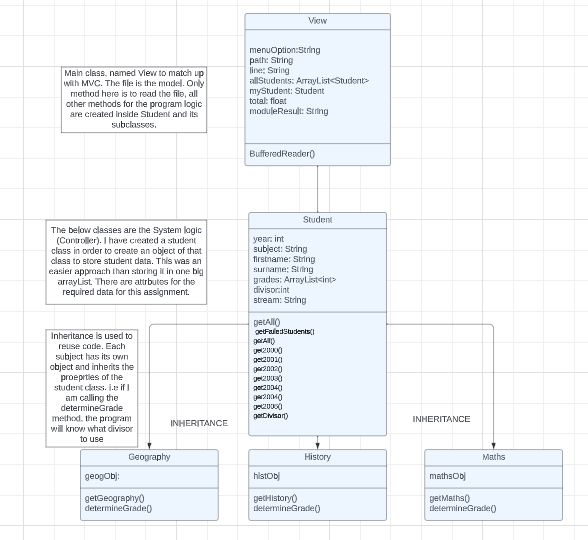
*Figure 2: Diagram of how my application follows the MVC design pattern*

# 3 Implementation

When first developing the application, I had planned to read the file first and store each student’s data in a multi-dimensional array list. I encountered many issues with this. Firstly, it was difficult to append a multi-dimensional array list and loop through it. Secondly, it was a time-consuming process to add 1000 student records with long lengths to one array list. Looping through this took a while and needed substantial amounts of code to run effectively. I had planned to implement the code procedurally at first, and then once my application met functional standards, I would sort the code into classes.

I then changed the application format, so that the data was printed out as the file was read. This meant that data could be stored into a 1d array list (allStudents). It was easier to perform operations on this, including iterating through each line of the file printing grades, and using conditionals to compare one student field to a user input e.g. getting the year of each student and comparing that to what the user enters and wants to retrieve.

I used a Student object to create a student with their first name, surname, year, subject, and grades. Any methods that were used were created inside the student class unless they were related to a specific subject e.g. calculating the moduleResult is different for each type of student, because they all take a different number of modules. The code was more readable compared to my first implementation, where I thought that some sections of the code were jumbled and unnecessarily repeated in parts. Below shows the final implmentation of the application in Figure 3.



*Figure 3: Final Class Diagram*

# 4 Testing and Results

As previously mentioned, my application:

* Reads in a file of text
* Includes a menu with options to perform the below
* Prints out the results of all students
* Prints out results for one subject
* Prints out results for one year
* Prints out the names of students who have failed an individual module
* Includes the students’ grades e.g. <40 Fail 40-59 Pass 60-69 Merit 70+ Distinction

There is also an option to run a series of automated tests to check if each method is printing the expected output

## 4.1 Functional Testing

Every functional component of my application has been tested, apart from reading in the file and accessing and mutating properties in the Student Class. For functions where there are many outputs, I have used the first available student record for each example. It would be very tedious to test up to 1000 records depending on the function, so I am assuming that by testing 12 methods to retrieve a student record and verifying that each is correct, that there aren’t any further errors in the code. The results are shown in Table X below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method number + name | Desc. | Test Data | Expected Result | Actual Result | Action + Retest number |
| **1) getFailedStudents()** | Check if correct students are printed | fail | Alayna Owens  Anatole Thornton  Ellwood Sorenson |  |  |
| **2)**  **getAll()** | Check if correct student records are printed | all | 2000 History Braulio Valentine 44.0 Pass |  |  |
| **3)**  **get2000()** | Check if correct student records are printed and are for the correct year | 2000 | 2000 History Braulio Valentine 44.0 Pass |  |  |
| **4)**  **get2001()** | “” | 2001 | 2001 Geography Alysia Barrett 45.0 Pass |  |  |
| **5)**  **get2002()** | “” | 2002 | 2002 History Holmes Forte 56.0 Pass |  |  |
| **6)**  **get2003()** | “” | 2003 | 2003 History Zettie Wittman 66.0 Merit |  |  |
| **7)**  **get2004()** | “” | 2004 | 2004 History Derald Shipley 68.0 Merit |  | Modify code- accidental error: if year==2003  get2004() |
| **8)**  **get2004()** | “” | 2004 | 2004 History Derald Shipley 68.0 Merit |  |  |
| **9)**  **get2005()** | “” | 2005 | 2005 Maths Gretchen Bustos 61.0 Merit |  |  |
| **10)**  **getDivisor()** | Check if the divisor (for use later) is correct | 2000 | 10 (first 2000 student takes history) |  |  |
| **11)**  **getGeography()** | Check if correct student records are printed and are for the correct subject | geog | 2001 Geography Alysia Barrett 45.0 Pass |  |  |
| **12)**  **getHistory()** | “” | hist | 2000 History Braulio Valentine 44.0 Pass |  |  |
| **13)**  **getMaths()** | “” | math | 2005 Maths Gretchen Bustos 61.0 Merit |  |  |
| **14)**  **determineGrade()** | Check if the correct grades are printed for each student | geog | Pass (the first geography student’s mark was 45, which falls in the pass bracket |  |  |

*Table 2: Table of functional tests for the application*

I used every possible user input that was available to test the above methods. For the getDivisor() method, I added an extra line of code which printed the divisor to the console. This is now commented out after testing. I only need to store the divisor and use it as a parameter for the moduleResult method- there is no need to print it for every student. The methods that were tested all produced the expected functionality, apart from one where there was a minor error in spelling a variable. Some other errors that I encountered whilst functional testing were small things like forgetting to close braces at the end of a function. This required minimal effort to fix.

## 

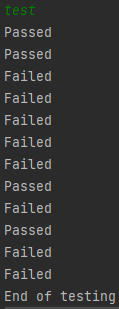
## 4.2 Unit Testing

For unit testing, I wrote automated tests for every method above that returned something. I used the exact same input data as I did for functional testing. Table 3 below shows the results for the 1st round of testing. I would access the tests in my program via the user entering “test” for the user input. This would automatically print the results of the 12 tests listed in Table 3 in order. If the input data matches the expected result, “Passed” is printed to the console. If not, “Failed” is printed instead.

### 4.2.1 Round 1

|  |  |  |
| --- | --- | --- |
| Method name | Test data | Result (Pass/Fail) |
| getAll() | 2000 History Braulio Valentine 44.0 Pass | Pass |
| get2000() | 2000 History Braulio Valentine 44.0 Pass | Pass |
| get2001() | 2001 Geography Alysia Barrett 45.0 Pass | Fail |
| get2002() | 2002 History Holmes Forte 56.0 Pass | Fail |
| get2003() | 2003 History Zettie Wittman 66.0 Merit | Fail |
| get2004() | 2004 History Derald Shipley 68.0 Merit | Fail |
| get2005() | 2005 Maths Gretchen Bustos 61.0 Merit | Fail |
| getDivisor() | 10 | Pass |
| getGeography() | 2001 Geography Alysia Barrett 45.0 Pass | Fail |
| getHistory() | 2000 History Braulio Valentine 44.0 Pass | Pass |
| getMaths() | 2005 Maths Gretchen Bustos 61.0 Merit | Fail |
| determineGrade() | 2001 Geography Alysia Barrett 45.0 Pass | Fail |

*Table 3: 1st round results of unit testing*



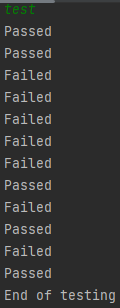
*Figure 4: Console results of automating tests in table 3*

The results above are not what I expected, with only 4/12 cases passing. Upon having a closer look, I noticed that the cases that passed had one thing in common. The data *“2000 History Braulio Valentine 44.0 Pass”* is the first record in the file, and the divisor *“10” i*s acquired by retrieving the first record in the file. The rest of the test data for the 8 failed tests is stored further down into the file. I retested the data after making some changes to the code, the results are shown below in Table 4.

### 4.2.2 Round 2

|  |  |  |
| --- | --- | --- |
| Method name | Test data | Result (Pass/Fail) |
| getAll() | 2000 History Braulio Valentine 44.0 Pass | Pass |
| get2000() | 2000 History Braulio Valentine 44.0 Pass | Pass |
| get2001() | 2001 Geography Alysia Barrett 45.0 Pass | Fail |
| get2002() | 2002 History Holmes Forte 56.0 Pass | Fail |
| get2003() | 2003 History Zettie Wittman 66.0 Merit | Fail |
| get2004() | 2004 History Derald Shipley 68.0 Merit | Fail |
| get2005() | 2005 Maths Gretchen Bustos 61.0 Merit | Fail |
| getDivisor() | 10 | Pass |
| getGeography() | 2001 Geography Alysia Barrett 45.0 Pass | Fail |
| getHistory() | 2000 History Braulio Valentine 44.0 Pass | Pass |
| getMaths() | 2005 Maths Gretchen Bustos 61.0 Merit | Fail |
| determineGrade() | Pass | Pass |

*Table 4: 2nd round results of unit testing*



*Figure 5: Console results of automating tests in table X*

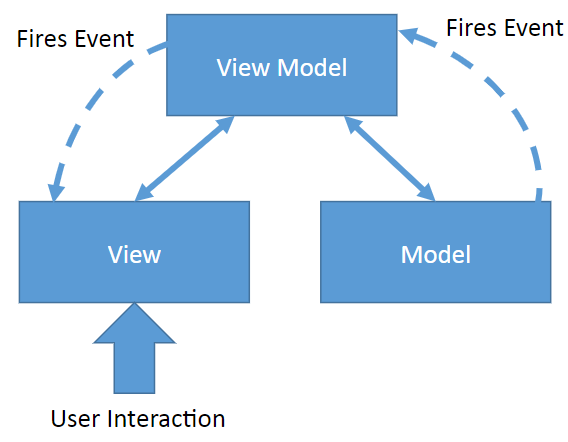
Table 4 shows that 5/12 test cases passed this round. A quick error I fixed was that the wrong test data was entered for determineGrade(). The test passed when this was fixed- but the first Geography student attained a “Pass” mark, as they scored 45. I knew that I also needed to implement the code so that any results that didn’t match the test data were skipped over, when it wasn’t the first instance in the file. Unfortunately, I did not manage to implement such a method.

# 5 Conclusion

In conclusion, I felt like I have met most of the assignment requirements including the basic application functionality and some of the additional functions. I produced an application which was easy to interact with in the end, but I could have improved it by sorting each output list into a logical order and added some additional functionality outside of the assignment requirements. I felt that I had implemented OOP (Object Oriented Programming) principles well, including inheritance and encapsulation as I have splitted my code up into relevant classes/subclasses.

For my application to be professionally used in a university computer. I would need to consider two important things: using a GUI and a database. As some staff would be unfamiliar with command line interfaces, I would implement a GUI on a web-based application. This would be the most user-friendly approach. A database would also be required as large amounts of data would need to be queried, and it is impractical to read in a text file. The database would need to be regularly updated with new student records. This is more efficient than writing one new line of data into a text file and reading the whole thing again.

The interface should closely follow the MVVM (Model View ViewModel) design pattern shown below in Figure 6. By designing the application this way, the which overall quality of the application would be improved, as this pattern is typically used for large applications.



*Figure 6: Illustration of MVVM design pattern*

Separating the View from the logic (stored in the ViewModel) helps with multiple people developing the system. Teams could work on different components simultaneously. It is easy to test, as the ViewModel and Model are separated from the View, meaning that tests can be written without having to use the View. Testing the UI is the hardest part sio this is important.

Separation between the components makes the code simpler and cleaner. This results in a more understandable application (thus easier to maintain). It should be straightforward to implement new features and connect them to the existing pattern.