COIT20245

Assignment 2

# Introduction

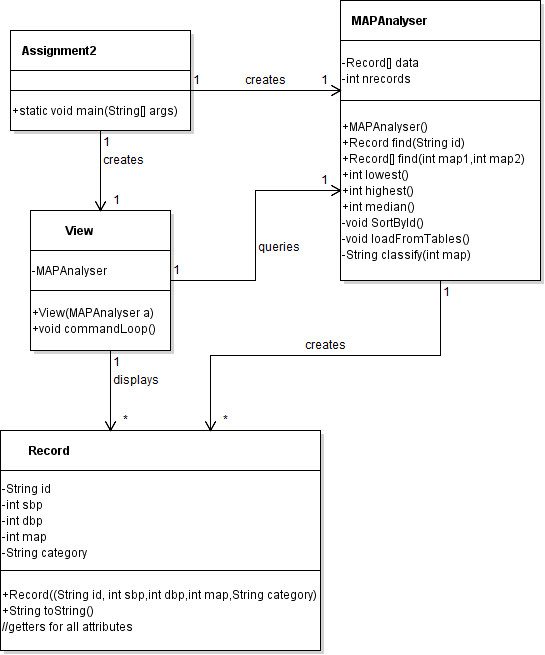
In this assignment, you are to implement a console application that supports simple MAP (Mean Arterial Pressure) analysis functionality. A phased implementation approach is recommended and an outline for such an approach is provided in Appendix 1. The application is to be conformant with the class diagram provided in Figure 1.

# The Application

The raw data to be analysed is to be stored in 3 arrays. Use array initialisers to specify initial values of your choosing. The 3 arrays are to contain person identifiers (string values), systolic blood pressure (SBP) and diastolic blood pressure (DSP). Both SBP and DSP are to be specified as int values. MAP is determined from an individual’s blood pressure using the following formula:

MAP = 1.0/3.0 \* SBP + 2.0/3.0 \* DBP

The raw data is to be used to create objects of type Record, which are to be stored in an array of type Record. The Record class is to conform to the class shown in the UML Diagram below:



**Figure 1. Class Diagram**

Note that in Figure 1, the visibility of class methods and attributes are designated as + (public) and – (private). Associations are annotated with a label (for readability) and multiplicity. The multiplicity captures the number of object instances that can be involved in the association. In this case, only two values apply – 1 and \*, which means many.

While map is calculated as a floating point value, it is to be stored as an int value. In addition to the map attribute, Record also has a category attribute, which has values of “high”, “medium” and “low”. A normal mean arterial pressure is a value in the range 70 and 100 mmHg inclusive.

Category determination is to be provided as a separate (private) method called classify(). Creation of the Record array is to occur when the MAPAnalyser object is created, using a private method called loadFromTables(). This method will iterate through three separate arrays containing ids, systolic blood pressures and diastolic blood pressures, construct Record objects and add them to the data array. Populate these arrays using array initializers. In order to facilitate search, records are to be maintained in ascending order of person id. Sorting is to be done after record creation, using a private method called sortById(). This method must implement the selection sort algorithm. Both loadFromTables() and sortById() are to be invoked from the constructor for the MAPAnalyser class.

The application’s View class is to execute (using a switch statement) the following command options:

1. Display the record for a specified person
2. Display records for all persons with MAP values within a specified range.
3. Display the minimum, maximum and median MAP values
4. Exit the application

As it is a console application, the user will need to be prompted to enter a command and its arguments (if any). My personal preference is for a minimal interaction scheme, as shown below:

run:

The following commands are recognised

Display this message > 0

Display a specific subject record: > 1 id

Display records for all subject records within a range > 2 map1 map2

Display statistics (minimum, maximum and median) > 3

Exit the application > 9

> 1 S01

<S01,120,81,94.0, normal>

>

Feel free to adopt the above scheme or if you prefer, implement a more verbose interaction scheme.

Note that

1. Each command is designated a number
2. Command 1 requires a single argument – the subject id
3. Command 2 requires 2 arguments – the lower and upper bounds of a range of MAP values.
4. I have added a “help” command (command 0)
5. The command options are displayed at the start of the application and whenever a “help” command is entered, rather than after each command.
6. Records are displayed (with no explanation of the fields) using Record.toString()
7. For a sorted list of values, if the number of values is odd, the median is the middle value. If the number of values is even, the median is the average of the two middle values.

For the commands that require arguments note that

1. For commands 1 and 2, basic error checking is expected. For command 1, an error message is to be displayed if an id does not exist. For command 2, an error message is to be displayed if a member of the range is < 0 or > 200 or if the second member of a range is less than the first member.
2. For command 2, the range is inclusive of the values specified. Consequently, having both members of a range equal is valid.
3. For command 1, binary search is to be used.
4. For command 2, the results are to be stored in an array for display. The length of the array will correspond to the number of records found.

The application **must** conform to the class diagram in Appendix 1, although additional **private** members and methods are permitted.

# Submission

You are to submit two separate files:

1. Your zipped NetBeans project folder. Details of how to do this are available in the NetBeans FAQ on the unit website.
2. Report.docx. This file is to contain the following Sections:
3. Limitations
4. Test plan
5. Test results

Student name, student ID number, unit name and unit code are to be included on the title page.

The limitations section is to specify any limitations that your program has in terms of calculations and data validation.

The test plan is to contain a comprehensive list of program functionality to be tested, the input values to be used to test each item of functionality, the expected output from the test and the actual output from the test.

The test results section is to contain screenshots to demonstrate that the program generates the actual outputs shown in the test plan.

# Marking Criteria

|  |  |  |
| --- | --- | --- |
|  | **Criteria** | **Marks Allocated** |
| **1** | **Functionality** | /10 |
|  | Command loop |  |
|  | Sorting |  |
|  | Searching |  |
|  | Statistics (maximum, minimum, median) |  |
|  | Conformance to class diagram |  |
| **2** | **Language use** | /8 |
|  | Variable declarations |  |
|  | Class definition/object creation |  |
|  | Loop control |  |
|  | Selection statement use |  |
|  | Method definition and use |  |
|  | Reading input |  |
|  | Display of results |  |
|  | Unnecessary/very inefficient code |  |
| **3** | **Layout and documentation** |  |
|  | Conventions (Spacing, indentation, naming) | /1 |
|  | Comments | /3 |
| 4 | **Report** |  |
|  | Limitations | /2 |
|  | Test Plan and Test Results | /6 |
|  | **Sub-Total** | /30 |
|  | **Penalties** |  |
|  | Does not compile: up to 30 marks |  |
|  | Partial implementation: up to 30 marks |  |
|  | Late submission : 5% (1.5 mark) / day or part of a day |  |
|  | **Total** | /30 |

For Language Use and Functionality, a checklist is given, but not a detailed marks breakdown. For checklist item non-compliance, you will be penalised 0.5 – 2.5 marks, depending on the degree of non-compliance.

Note that it is your responsibility to ensure that source code files are included in your submission as well as report.docx. If either are missing, the assignment will not be marked.

# Appendix 1

It is best practice to develop an application incrementally. The idea is that the work is broken up into phases, with each phase producing a working system. Furthermore, if a phase involves the provision of multiple functionalities, as in Phase 3 below, these should also be tackled incrementally. The advantage of this approach is that you have a series of working systems that you can demonstrate to the customer. In your case, it means that you can always submit a working system, albeit one perhaps with limited functionality. For this to work properly, you need to save copies of your working versions as you go. This way of doing things is such a good idea (especially for team development) that software support is readily available in the form of version control systems such as SVN, GIT, CVS etc. However, we don’t introduce you to version control until later in your course.

In terms of how you might phase this assignment, below is how I would go about it, given that I would be working from a class diagram.

## Phase 0

Read Section 2 **very** carefully and seek clarification from your tutor (or me) about anything that is unclear.

## Phase 1.

1. Create all classes with members and dummy methods as per the class diagram. Where constructors set member variables via parameters, add code to do this. Leave void methods with an empty body. Where methods require a return value, just return a sensible type-consistent default value, as shown below for the MAPAnalyser public methods:

public MAPAnalyser() {

nrecords = 0;

data = new Record[nrecords];

}

public Record find( String id ) {

return null;

}

public int lowest() {

return 0;

}

public int highest() {

return 0;

}

public int median() {

return 0;

}

public Record[] find( int map1, int map2 ) {

// no records found

Record[] r = new Record[0];

return r;

}

Note what is returned in the case of the two find methods. With find(String), null is returned, which is a sensible return value if the specified id is not found. For find(int,int), an array of length 0 is returned. Note that the number of records found is not returned this is the length of the array. Also note that with find(int,int), you will need to construct two arrays – one to collect the matching values and one to return the matching values.

1. In the main() method, add the following code:

MAPAnalyser a = new MAPAnalyser();

View v = new View( a );

v.commandLoop(); // does nothing

Do not proceed to Phase 2 until this code compiles. You now have an application that compiles but when it runs, no output will be produced. Also, we have not created any records. However, we do have an empty data array, which is an important test case

## Phase 2.

Get the command loop working with the dummy methods. The find methods will be unsuccessful; you will need to explicitly test the return values and display an appropriate message.

The following commands are recognised

Display this message > 0

Display a specific subject record: > 1 id

Display records for all subject records within a range > 2 map1 map2

Display statistics (minimum, maximum and median) > 3

Exit the application > 9

> 1 S01

S01 not found

> 2 70 100

No records in this range

> 3

Lowest MAP is 0

Highest MAP is 0

Median MAP is 0

> 9

My preference is for minimalist interaction. You may prefer to display the menu before every command and explicitly prompt for all inputs – either style is fine.

## Phase 3

1. We now need to populate the data array. Note that if we present the data in the arrays in sorted order, we don’t need to write our sorting method just yet. Once the array is populated, you will need to test that it has been set up correctly by displaying its contents. You can do this using the getter methods or with the toString() method. The convention in Java is to make the return value of toString() minimalist, as in <S01,90,120,110,high> - there is no naming of values.

1. Now that we have a sorted data array to work with, we can implement find(String). This will require you to code the binary search algorithm – use the implementation in Deitel & Deitel as a starting point, but add comments to indicate that you have done this. We can also implement find(int,int), lowest() and highest(). The order in which you tackle these is up to you; I would do lowest() and highest() first.

## Phase 4

Finally, we need to do sorting. First, get sorting by id working. As with binary search, use the implementation in Deitel & Deitel as a starting point, but add comments to indicate that you have done this. Note that while sorting an already sorted list is a sensible test case, you will need to test your code with other data sets. A convenient way to do this is to use the approach employed in the Week 6 Laboratory exercise.

Once you have sorting by id working, then move onto implementing median(). The median is the “middle” value of a sorted list of values, such as house prices. To calculate the median, you will need to sort the MAP values and find the middle value. If there are n values in the array and n is odd then there is 1 middle value, which is the median; its index is n/2. If n is even, there are two middle values and the median is the average of the two values; their indices are n/2 and n/2-1.

Note that sorting by id is to be done once per session, as find(String) may be invoked multiple times. With sorting by MAP value, the median is likely to be invoked only once per session when the application is in production (as opposed to testing) so whether you embed sorting into the median() method or do it independently of the method is up to you. Also, for sorting by MAP value, no algorithm is specified – you can use either selection sort, insertion sort or Arrays.sort().