ICT283 Topic 5 (not assessed in this lab but must be completed and submitted)

Objectives:

- Preparation for assignment 1 – Exercises/questions 1 **and** 5 must be completed before attempting the assignment. **These two questions are part of Assignment 1**. Assessment of these questions is in the assignment. **Question 5 needs to be submitted** to access the full assignment specification when the assignment becomes available.

- To learn
  - OO Design and appreciate the value of doing this design properly
  - practical implications of cohesion and coupling
  - to do refactoring and code maintenance
  - to do testing
  - to pay attention to detail

Prerequisite:

Lab 4 must be completed

It is very important not to fall behind with these exercises.

You should note that even though an exercise is not assessed, not attempting the exercise would make it very difficult for you to understand subsequent material.

If you want to work on you own computer, install graphviz first, then install doxygen.

Reading list is for Topic 5. Some of the chapters are repeat readings from previous topics. See "*Reading List for ICT283*". The textbook chapter on "*Overloading and Templates*" is particularly important – needed for the assignment.

Exercise

Do **not** start coding until you have worked out exactly what is required. Do this on paper. Draw a UML diagram illustrating how the classes are connected and being used. You need to read all the questions (sub tasks) below first to work out what is needed and then plan the best way to tackle each of the sub tasks listed. You may need to read the specifications more than once to understand what is really needed. Make sure you document all code using doxygen style comments.

Think about the test data you will use to demonstrate that your program works. You can assume that the input data file contains data in the correct format. This means that you must create the data file correctly.

Exercise/question 1 shown below requires you to follow the specifications as provided. When you finish question 1, you would have an appreciation of how C++ classes work and are ready to do your own design and implementation.

From exercise/question 2 onwards, you have to do your own design. So, reflect on all the work you have done so far and re-design and re-implement all the classes that have been created in the previous exercises. **The aim is to create classes that can be re-used in many contexts without re-coding of the classes. Aim for high cohesion and low coupling in your design**.

Just getting code to work is not going to be sufficient to pass. Design is very important.

1. In the previous lab exercises, the Registration class had a raw array of results. This array itself was not encapsulated inside a class to provide controlled access to the array.

For this and later exercises, a data structure needs to be created that encapsulates a raw array. For our purposes, we will call this data structure a **Vector**. **Vector** is a templated data structure that encapsulates a raw **dynamic** **array** (uses "new" to create memory on the heap[1]).

As Vector is templated, it can store anything, including itself.

The closest analogy would be the Java *ArrayList* which is also templated. You would have used *ArrayList* in the prerequisite unit. The raw array is a private member of the ***template*** **class** ***Vector***. You write the *Vector* class and provide controlled access to the dynamic raw array via the *Vector's* public methods. Your Vector class is **not** the same as the STL vector, so please do not attempt to duplicate the STL vector. You will get a bloated result and your work will be penalised. The Vector class needs to be **minimal but complete**.

Do not attempt to the mimic the STL vector (std::vector). Methods or operators should not have duplicated functionality. **See *Rule 33 Prefer minimal classes to monolithic classes*** in the unit's reference book: *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices* by Sutter and Alexandrescu. ebook is in the library if the following link does not work.
http://lib.myilibrary.com.libproxy.murdoch.edu.au/Open.aspx?id=291948 (may need sign in)

Do **not** use the STL vector class as a service data structure to your Vector (i.e. you must **not** have *#include <vector> in any part of your code for this lab or assignment 1*).

The ***template Vector*** class that you have to write is an example of a linear data structure. The entire (interface/declaration and implementation) template Vector class is written in a file called *Vector.h* because it is a template class. The textbook explains schemes for having *.cpp* files for template classes. There is **no** *Vector.cpp* for the template Vector class in this lab (and assignment). There is no need to complicate the issue by having a .cpp file for template classes.

It is absolutely essential that you separate [2]the interface of Vector and its implementation even though only one file *Vector.h* exists. **Not making this separation will result in a mark of 0 in the assignment and exam.**

Vector.h contains the following:

```
#ifndef VECTOR_H
#define VECTOR_H

// The class interface/declaration must have doxygen comments – put these
in
// Follow the style in modelfile.h

template <class T>
class Vector   // This is NOT the std::vector.
{
      public:
            // class declaration
            // you fill in the rest and include doxygen comments
      private:
            // the encapsulated dynamic array
            // fill in ..
};

// class implementation follows with normal comments
      // you fill in the rest
```

[1] As memory is created on the heap, a number of methods become mandatory. One of these methods is the destructor. See Lect-13.ppt

[2] See topic 1. Separation of declaration/interface from implementation is mandatory

```
#endif
```

How big is the *Vector* going to be? Once you create the internal array on the heap using “new”, the size is fixed. What happens if there is more data to be added? The Vector would need to resize itself (grow) once full. The user of Vector should not be bothered with this detail as the Vector would take care of this behind the scenes. Naturally, you as the designer of Vector would have to write the code to do the resize. The amount to resize is typically 1.5 to 2 times current size.

The required Vector class is a container class. It must not violate the Law of Demeter. Find out what this law is. Wikipedia has an understandable explanation. Did you keep or violate this law in Lab 4?

You will need to **unit test** this Vector **before** using it. Discuss your ideas with your tutor. You will need this *template Vector* class for the assignment.

In the previous labs, you were using an array of Result objects. Assuming you had the following in Registration.h:

`Result results[MaxResults];`// you may have been using different names for array and max size of array, so amend this example accordingly.

Change the above to:

`Vector<Result> results(MaxResults);` // use the same name for the Vector object as your actual array name.

You will need to *#include "Vector.h"* in *Registration.h*. Build your Registration program. Provide the required operator [] methods in Vector if you have not provided them.

After the above change, was there any need to change any other code? Should there be a need to change anything else?

Did you make your *Vector* class dependant on any part of application? Is any part of the registration data a part of the member data of *Vector*? If it is, the *Vector* can’t be used correctly for question 5 of this lab and the assignment: your *Vector* is done incorrectly.

Vector must **not** depend on any part of the application. *Vector* is just an encapsulated array that can store any other data type, including *Vector* itself. To be able to have this flexibility, ***Vector*** must **not know** about the data type that it is storing.

Can your Vector class be used in exercise 7 of Lab 4 that you submitted to access this topic? You need working Lab 4 exercise 7 code before you proceed. Your declaration in exercise 7 of lab 4 would have been:

*float *dataArray;*

In exercise 7 of lab 4, replace the line above with the following:

Vector<float> dataArray(N); // You will need to *#include "Vector.h"* at the top of the file for exercise 7 to be able to use the Vector class. **N** is the value you used when you used new to create the dynamic array in the original exercise 7.

Do not change anything else in exercise 7 of lab 4. Build and run the program. Did the program build and run correctly? Were changes needed? If so, redesign the Vector. Check the Vector again with Vector of Result in the Registration lab exercise.

2. Consider a change in requirements in the Registration lab work from previous weeks: each unit has a unit coordinator and the unit coordinator's details include: coordinator's name, room number, telephone number and email.

   Reflect on the re-design you would need to do to cater for new requirements for the Registration codebase. How would you "future-proof"[3] your design so that any new code that you write will introduce additional functionality to cater for new requirements and not keep on modifying existing code? (Open-Closed principle) Note that in the past, you were given an example of what output the program should do. What if the output format was changed? Would you have to go back and change existing classes?

   Ideally, to change the output format should only require a changed "main program" (client code) and **not** a change to the data classes. What about a change in the input format? For example, how would you cater for the input format shown at the end of the document? You will be using the first version of the input format for the rest of ict283.

   You have a free hand in re-designing all the classes in the Registration codebase to attempt to future-proof them. Later, you will be learning from your successes and mistakes when you discover the amount of re-designing and coding that you must do to cater for expanded/changing requirements if you do not design well and implement that chosen design carefully.

   It is a common mistake to spend little or no time in design and spend a lot of time trying to make the code "work". For example, assignment 2 makes use of earlier work. If the earlier work is not designed (and implemented) right, assignment 2 will require more work (and time) to complete when you have very limited time available. Not thinking about design issues can result in having to complete all of assignment 2 in a two week time frame. This will be stressful.

   You should use UML for the design.

   For this particular exercise/question, would you have to change the *Vector* class from question 1 so that it will work in your new design in this question 2? If you did, then you had designed your *Vector* class badly in question 1.

3. Implement your new design for the Registration codebase. The new design has additional information related to Unit as shown in question 2. Demonstrate how flexible your design is so that different output formats can be catered for by writing different main programs. Note that just getting different output formats this way is not sufficient to indicate good design – but it is a start.

   See sample input formats are at the end of this list of questions. Use the first version of the input format (CSV) for this question.

4. Run doxygen on your codebase and examine the output to see that your initial design matches the design as shown in doxygen.

5. This exercise is part of assignment 1. You need to complete this exercise first before attempting the rest of assignment 1 when it is available. Access to assignment 1 will require the submission of this exercise in LMS.

   Use the text editor **Notepad++** (https://notepad-plus-plus.org/) to examine the data file data/*MetData-31-3.csv*. This data was obtained from http://wwwmet.murdoch.edu.au/. This online weather station was originally created by one of our final year Murdoch Computer Science project teams. You can view the site if the current administration has it online.

   Open the same file (*data/MetData-31-3.csv*) using a spreadsheet like **Microsoft Excel**. Compare the views shown in **Excel** and **notepad++**. This is the type of data that will be used for assignment 1. The **notepad++**

[3] It may not be possible to get a perfect design which will cater for all future requirements. What you may want to hope for is minimal change to existing code to cater for new/changing requirements.

version is what your program will see. To see what your program will see: in notepad++, select the View Menu/Show Symbol/Show All Characters.

The first row shows the sensor codes. See the file *SensorCodes.rtf* for their meaning.

The first field contains both the date and time. You will need to split the date and time values. The date can be stored in the *Date* class from a previous exercise. You need to create, **and** unit test a ***Time*** class to store time values.

For this question, we are interested only in the first field marked "**WAST**" and the field marked "**S**". The data in the **S** field is the wind speed. You should assume that the speed recorded in the data file is in **m/s**. You can download your own data files to check the units. The units are shown in the raw data files you download.

For lab 5 you will use the following data structure:

```
// #include anything else you need
#include "Date.h"  // your Date class from a previous lab
#include "Time.h"  // your Time class from this lab
#include "Vector.h" // your Template Vector class from this lab

typedef struct {
    Date d;
    Time t;
    float speed;
} WindLogType;

Vector<WindLogType> windlog; // Vector<WindlogType> is a realised Vector.
                             // This cannot be declared global

See lecture notes. What would the UML diagram look like?
```

The Date class was created, and unit tested in lab 4. The Time and Vector classes were created, and unit tested as part of this lab 5 (this lab). Note that as you are creating classes, the classes get reused later in other contexts, so make sure you design for reuse. Remember to keep the classes simple. The more you add to classes beyond the bare minimum, their reusability goes down. Apply the principles of low coupling and high cohesion. ***See Rule 33 Prefer minimal classes to monolithic classes*** *in the unit's reference book: C++ Coding Standards: 101 Rules, Guidelines, and Best Practices by Alexandrescu et al. Ebook is in the library if the following link does not work.* *http://lib.myilibrary.com.libproxy.murdoch.edu.au/Open.aspx?id=291948*

Write a C++ program to read data from the file *data/MetData-31-3.csv* into *windlog*.

Once the data is read into *windlog*, your program will use *windlog* to work out the **average wind speed in km/h and the sample standard deviation (SD).** The numbers are printed to the screen on a new line with appropriate text information to show what the numbers represent.

The program will go through *windlog* **and when a value of speed is found that matches the average speed that was printed out, the date and time stored (in *windlog*) for this speed is printed on a new line**.

Some things to consider:

- Is it possible that the average speed would not be in *windlog*?
- What does it mean, when you need to "match" and the values being matched are floating point values?
- What values could be in the data file if a sensor is offline? What would happen to your program if, say, a value of N/A (or some other) value is there for an offline sensor.
- Should any offline sensor value take part in the average and SD calculation?

Once you have the correct results, check your program on the larger data file, *data/MetData_Mar01-2014-Mar01-2015-ALL.csv*. This data file is used for assignment 1.

Exercise/question 5 and Assignment 1 needs to work with only 1 data file at a time.

Although your program has to work with just one file, your program needs to handle data with **different column arrangements/order**. There is no change to column/field names, like WAST, S, T, etc. The number of columns/fields also remain the same.

The example data files in the data folder has different arrangements. Other arrangements are possible, and the **assignment marker would use their own arrangement of columns/fields to test your program**.

Some essential things to consider:

- A number of fields that are not of interest would be read and discarded for question 5. Other fields will be used later (Ambient Air Temperature **T**, Solar Radiation **SR**). Future work, including assignments would not have columns that are in the same order. The field names (column names) would be similar, but column arrangement can be in a different order. How would you future-proof your read routine?
- How do you know that the data is being read in correctly? Hint: before you attempt to read the data into the data structure, you need to read, split the data, and print it out on screen or a file for debugging. Then manually check the output. Your program doesn't have to do this for the entire data file - just the first few lines will be sufficient to check. Once this limited manual check is successful, check that you can read the whole file correctly. You do this check by reading *MetData-31-3.csv* and printing the date, time, and wind speed into an output file (*testoutput.csv*). Commas separate the date, time, and wind speed, with 3 data values per line.

  Open *MetData-31-3.csv* and *testoutput.csv* in a text editor and spreadsheet, and then **manually** check that the values are correct in *testoutput.csv*. Once you can **visually** confirm that the values are correct, you start on this question and use the *windlog* data structure.

  Test your program by printing out all the data from *windlog* into the output file called *testoutput.csv*. The three values in the output file must be separated by commas. Do a visual check to see that the data in *testoutput.csv* is correct. Once you can confirm that the output is correct, do the average speed calculation and output date/time on screen. Make sure that you check that the screen output is correct by viewing the input data file in a spreadsheet (application test). Use the spreadsheet's average formula calculation to verify your program's calculation. Repeat this process for the sample standard deviation.
- Can you think of any good reason why *WindLogType* should be a class and not a struct? When would you use a class and when would you use a struct? Think very carefully about this. Readings from this topic and earlier topics would guide you. The assignments do require (**mandatory**[4]) you to provide rationale (reasons why) for data structure approaches you take.

Sample input formats for questions:

There are two formats to experiment with. The first one is easier than the second. Attempt the second version only if you can complete the first version. The first version below is needed for question 5 (assignment 1 preparation).

Get your design and program to work with the same information that was used in lab 4's registration exercise. Once you have a working design and program, think of how you could incorporate additional details (as in question 2 above) into the input data. Would you use a separate data file to store unit coordinator information?

First version of the input data format (CSV): (All assignments will use this input format)

[4] **The assignment must meet the mandatory requirements. So attention to detail is critical.**

This format is commonly called the CSV (comma-separated value) format. You would have used this format in the prerequisite/revision exercise from ICT159. This format is usually used when the number of fields appearing on a line is fixed. Data may be missing. Use this format for question 5 above and the assignments.

All data is separated by commas until the end of line. Each row (line) is called a record. Each piece of data is called a field.

ID,Semester,Unit ID, Unit Name, Credit, Marks, Date

For question 5, the data will come from the data file(s) that have been provided.

Second version of the data format: (not for the Assignments – for additional practice)

Attempt this version only if you can get the first version to work, have completed question 5 above and are looking for something more "challenging". This format is not needed for Assignment 1.

All data for one student id is on one line. Major separator symbol is the semicolon (**;**) and minor separators are commas (**,**). Each line is called a record.

*ID***;***Semester,Unit ID, Unit Name, Credit, Marks, Date, Unit ID, Unit Name, Credit, Marks, Date***;***Semester,* .. etc

From the above, you will notice that within a semester, there can be more than one unit. Any item in between a separator is called a "field". So, Unit Name is a field, so is ID, Semester, Credit, ... etc.

The date is the when the marks were entered into the system. The semester is when the unit was studied.

Notice that the record (line) is identified by the student ID. So each student will have all his/her details on one line. After the student ID, the semi-colon separates units completed according to semester.

Once you have a working design and program using the above data format, think of how you could incorporate additional details into the input data.

Notes:

From the above, you will note that a record contains one or more fields and that each record appears on a line.

Think about how you might handle an alternative Date format that has the date and time in the one field as in question 5's data file. This means that the date and time is only separated by one or more spaces.

When thinking about how to resolve the above Date issue, would you make the Date class handle all these possibilities? If so, what happens when the input format is changed again to another specification?

You need to complete exercise 5 now. Once you have completed the preparatory labs, assignment 1 is not that complicated. This same approach also applies to assignment 2. The assignments are mostly "packaging the lab work into a larger application".

To access Assignment 1, submit the solution to exercise 5.