Tutorial Letter 202/2/2018

Advanced Programming
COS3711

Semester 2

School of Computing

Discussion of Assignment 2

BARCODE

Define tomorrow.

UNISA university of south africa

CONTENTS

Dear Student

1 INTRODUCTION

This discussion, and the solutions, are only made available electronically.

The solution to assignment 2 is placed under Additional Resources of COS3711 on *my*Unisa. A separate folder is created for each question, in which you will find all the relevant files for its solution. Please note that the solutions provided on *my*Unisa are only suggested solutions and they are not the best or only solutions.

This tutorial letter contains a short discussion of the solution to Assignment 2 of COS3711 made available on *my*Unisa. Hence, this tutorial letter should be used in conjunction with the solutions uploaded on *my*Unisa.

The marking rubric used for marking of Assignment 2 is also included in this tutorial letter. It is impossible to follow a marking rubric strictly for a programming assignment. Hence the given marking rubric should be used only as a rough guideline.

2 TUTORIAL MATTER

The following are important documents that you need to consult. Please download them from *my*Unisa if you do not already have them.

COS3711/101/3/2018	First tutorial letter
COS3711/MO001/3/2018	The contents of the *my*Unisa site for COS3711
COS3711/102/3/2018	Practical study guide
COS3711/103/3/2018	Networking and the Web – Additional notes
COS3711/201/2/2018	Discussion of assignment 1

3 COPYING OF ASSIGNMENTS

Assignments that contained solutions copied from other students' assignments were penalised.

4 DISCUSSION OF SOLUTIONS TO ASSIGNMENT 2

4.1 Question 1

This is a fairly simple task of creating a GUI (which in the sample solution is done using Qt Designer, although you are obviously free to do it manually as well), and then linking the Add button (via the `clicked()` signal to `on_addButton_clicked()` slot – again, done via Qt Designer, although you could write the connect code yourself – make sure that you do know how to do this).

The error checking is the part where you may have needed to think. You will see that input masks have been used to ensure that a user can only enter certain data:

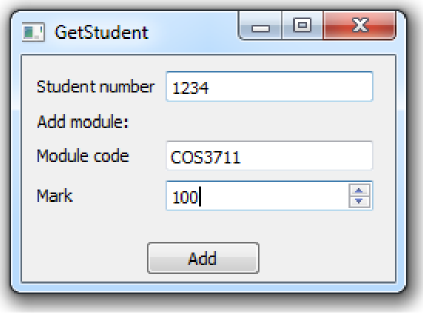
- `9999`: a `9` allows any ASCII digit, and so 4 `9`s means that the user can enter 4 digits.
- `>AAA999N`: `>` ensures that all alphabetic characters entered are converted to uppercase; `999` allows 3 digits to be entered, and the final `N` allows any alphanumeric character to be entered.

Also, maximum and minimum values are set for the mark field.

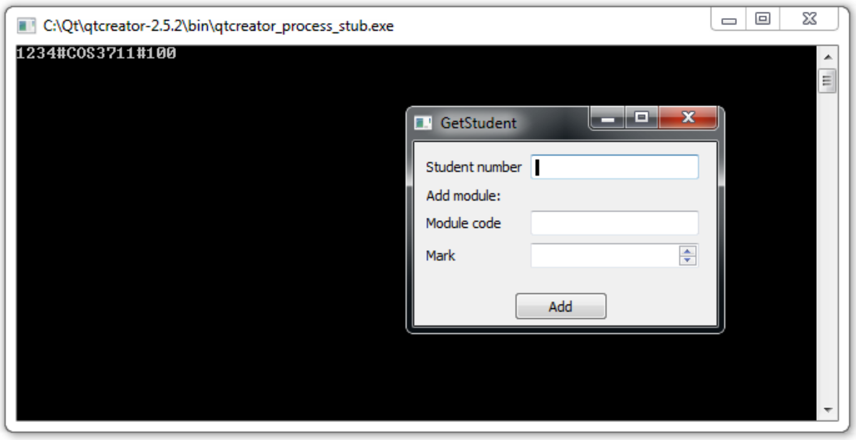
However, an input mask is not enough to ensure that data is correctly entered. An input mask can check what is entered, but not that the correct number of digits is entered for a student number, or that the 4th character of the module code is a 1, 2, or 3. For this you need to use regular expressions. So, in the slot, the user's input is checked for exact matches to the required pattern for the student number and module code. Where there is an error, the user is informed via a message box.

When data is correctly entered, the text is output to the console and the input fields are cleared. Note that the 3 fields are separated from each other by including a # between them – this is just to simplify the next step in question 2 when this string must be read by another program and deciphered.

Below is an example of the running program.



Student number 1234
Module code COS3711
Mark 100



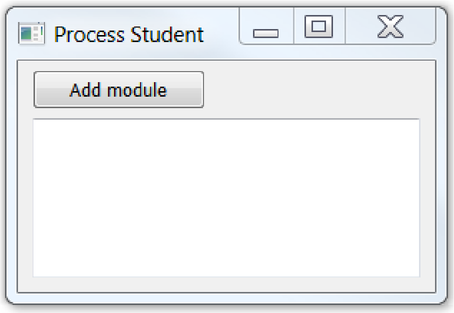
```
1234#COS3711#100
```

4.2 Question 2

You will firstly need to create a GUI that has

- a button used for adding a student, and
- an area for displaying data.

Also, the button will need to be connected via the signal/slot system to a function that will collect the student module information.

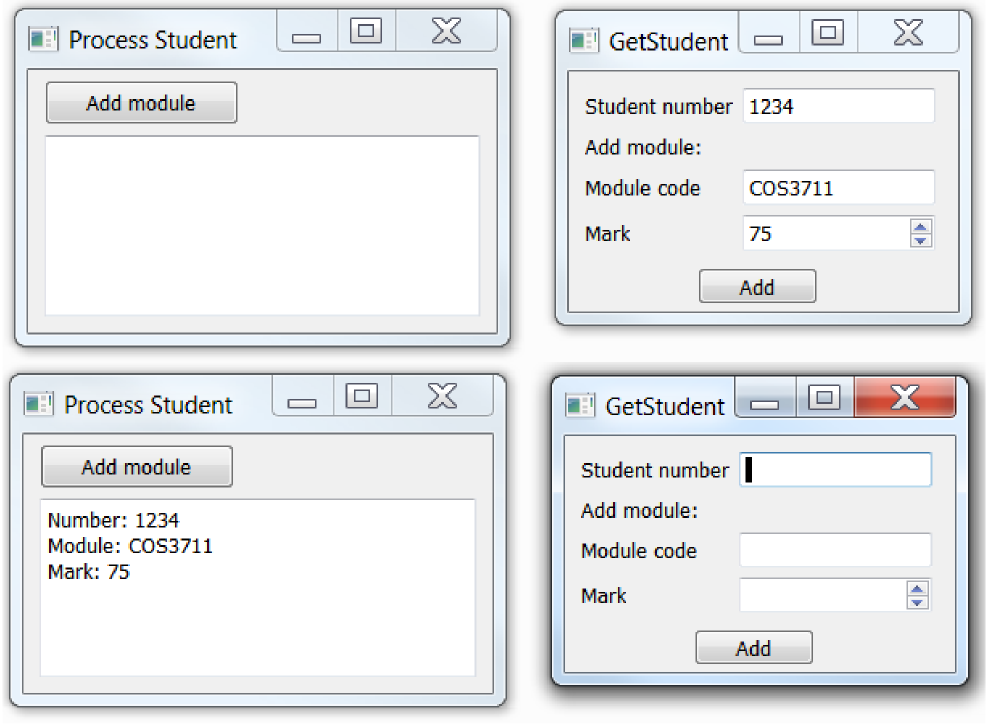


As the program that collects the data (which you created in question 1) needs to be used, you create a new process and then run (using `start()`) question 1's `.exe` file – remember to put this `.exe` file in the current project's Debug folder, or no program will be run and nothing will happen. You now need to listen for this process to return information (via the `readyReadStandardOutput()` signal as we sent the output of question 1 to the console, the standard output) – connect this signal to a slot that can display the data in the GUI. Also, listen for the process's `finished()` signal so that you can handle this as well.

The information is received from the process as a `QByteArray`, and you access the data using the `readAllStandardOutput()` function. As the data there was created as a single string with the data fields separated by #s, it is now simply a matter of converting the `QBytesArray` to a `QString`, and then splitting it to create a `QStringList`. This data can then be displayed on the GUI.

When the process sends the `finished()` signal, you can `close()` the process (which closes all communication with the process and kills it), call `deleteLater()` to schedule the process for deletion when control returns to the main event loop, and then let it point to 0 (so that the pointer does not point to a process that no longer exists). You should be able to end the process and then restart it again without the main program crashing when you close it.

Below is an example of the program running before and after the Add button was clicked.


Process Student
Add module

GetStudent
Student number 1234
Add module:
Module code COS3711
Mark 75
Add

Process Student
Add module
Number: 1234
Module: COS3711
Mark: 75

GetStudent
Student number
Add module:
Module code
Mark
Add

4.3 Question 3

This program is a continuation from Question 2. Creating the `Student` class should be simple enough. Remember to include `const` for functions that do not make changes to a class's data. The implementation of the various functions should also be fairly simple. With `average()`, remember that integer division gives an integer answer, and that you want an average that has a decimal value.

Create the `StudentList` class following the class requirements from the question. This class is designed using the Singleton design pattern to ensure that only one instance is created, so

- the constructor is made `private` so that no instances can be created without going through the `getInstance()` function (which returns a pointer to the `StudentList` instance); and
- there is a `static StudentList` pointer (named `onlyInstance`) that will be used to point to the only instance of `StudentList` that is allowed to be instantiated.

`onlyInstance` is set to point to `NULL` at the top of the `.cpp` file – it is a `static` data member, and so must be initialised. Memory is only allocated to this pointer when a user calls the `getInstance()` function (which checks whether an instance has been created yet or not), and a pointer to the only instance of the `StudentList` is returned. The `StudentList` constructor simply allocates memory for the `QList` that will be used to store the list of pointers to `Student` objects that this list maintains. Note that the destructor deletes all the items that are pointed to in the `studentList` `QList` before deleting the `QList` itself. The rest of the class implementation should be simple enough.

The GUI should be extended from Question 2, and so the functionality to add a module needs to be extended. You would need to check whether the student number exists in the list already (in which case you have to locate the `Student` object in the list and add this new module to that particular student); otherwise a new `Student` object is created, its data members set, and then it is added to the list of students.

When displaying a student, you do need to check that the student number exists in the list first. Note that separate functions have been used to check for the existence of a number, and to display the student detail, to reduce redundancy. The buttons to get the average and whether the student graduates also check first whether the student exists before processing the request. It is then a simple matter of calling the appropriate function in `Student` to get the required information.

4.4 Question 4

To ensure that the `studentList` is automatically written to file, the code for this is placed in the `closeEvent()` function. The tasks covered are:

- Check that the list has at least one record.
- Open a file for writing. Continue only if this opens successfully; provide an error message if not.
- Get a copy of the `studentList`.
- Create an instance of the `StudentSerializer` class.
- Iterate though `studentList` adding `Student` objects one at a time to the `StudentSerializer` object.
- Get the DOM document from the `StudentSerializer` object.
- Write the document to file (using its `toString()` function).
- Close the file.

The real work is done in the `StudentSerializer` class. Here there are 2 data members: a `QDomDocument` (`doc`), and a `QDomElement` (the `rootElement`). In the constructor, the `rootElement` is created (`"StudentList"`) and appended to `doc`. You now have the following:

```
<StudentList>
</StudentList>
```

As `Student` items are added one by one, you need a `studentElement` (`"student"`) that is made a child of the `rootElement`.

```
<StudentList>
  <student>
  </student>
</StudentList>
```

The process of adding data for the `Student` follows a similar pattern:

- Create a `QDomElement` instance – this is for the main tag.
- Create a `QDomText` instance – this is for the text that will go into the tag.
- Append the tag to the tag above it.
- Append the text to the tag.

You now have

```
<StudentList>
  <student>
    <number>1234</number>
  </student>
</StudentList>
```

The modules are added in much the same way (except that it is necessary to iterate through the container of modules).

4.5 Question 5

The question gives a fairly detailed description of what needs to be done to meet the requirements of the question. Firstly, you need to change the `GetStudent` application so that it is possible to indicate degree/diploma student status. This new piece of information also needed to be added to the data that is displayed in the console so that it can be picked up by another application that is listening to the standard output.

The `Student` class is changed so that most of the calculations that are similar to both degree and diploma students are implemented in the base class, and `graduate()` is made `virtual` and should be implemented in the derived classes `DegreeStudent` and `DiplomaStudent` in their specific ways. This, then, is the only function that needs to be implemented in these two derived classes. Implementing these two derived classes via inheritance should pose no problem. Note that the data members in Student should be changed to `protected` to allow them to be inherited.

The factory method design pattern in the example solution does not have an abstract base class, although this can certainly be done. Note that the `createStudent()` function receives a `QString` which it uses to determine what sort of `Student` to create (`DiplomaStudent` or a `DegreeStudent`) and it returns a `Student` pointer (so that it can point to either of the derived classes). The `createStudent()` function itself simply checks what type of `Student` is required, and returns a pointer to a newly created object of the required type. Note that it returns `0` if it does not know the type of `Student` to create. This can then be handled in the calling function.

As the list of students that is maintained by the application is a list `QList` of `Student` pointers, there is no need to make any changes here. When a new student has to be created, the `acceptNewStudent()` slot creates and initialises a pointer to `FactoryMethod`, and then uses the `createStudent()` function (passing the type of student required by the user) to get a pointer to the particular object. This object is then filled with student number and module values, and added to the `studentList`. It is here that a returned `0` is handled.

Displaying the object takes a little more work. As the list of students is a list of pointers to the base class, we need to check what sort of object is stored in the list before its detail is displayed in the GUI. To do this, you use the `QMetaObject`'s `className()` function, and depending on this value, you display an appropriate message on the GUI.

The `StudentSerializer::addStudent()` function also had to be extended. Using the meta-object (as above), an attribute is added to the `<student>` tag indicating the type of `Student`.

The rest of the program remains unchanged from previous questions.

4.6 Question 6

To use SAX parsing to read in a file when the program starts, you would add the code into the constructor of the GUI. Note that you should check first whether the file exists before trying to read it.

The sample solution has used a new handler class to parse the file – `SAXReader`. Once the file has been parsed, you simply have to get the list that the parser created and add each student to the `studentList`.

The real work lies in the `SAXReader` implementation. This handler class contains all the usual functions that are overridden from `QXmlDefaultHandler`: `startDocument()`, `startElement()`, `characters()`, `endElement()`, and `endDocument()`. The class also has a `QList` of `Student` pointers (`studentList`) so that student objects can be saved till needed, and a `QStringList` (`holdData`) to hold data until it can be used to create a `Student` object.

`startDocument()`: As you will want to read the characters that are given in the tags in the XML document, you need some way of knowing when you are in one of those tags (`number`, `code`, `mark`) that contain data. So in this function, `wantChars`, which will be used to indicate this, is set to `false`.

`startElement()`: If a `student` element found, then record the `type` attribute. However, if the element that has been found is one that contains data between the opening and closing tags, set `wantChars` to `true`.

`characters()`: if the `wantChars` flag is set to `true`, then the data between the opening and closing tags is added to the `holdData` `QStringList` for use later. The flag is also reset back to `false`.

`endElement()`: This function handles what happens when the end of the `modules` tags is reached – which means that a full student record has been read. Now is it a simple matter of using the data stored in `holdData` to construct an appropriate `Student` object using `FactoryMethod`, and add it to the `QList studentList` which is holding all `Student` objects found.

This process then repeats itself, parsing all the tags until the end of the document is found. As there is no special work to be done here, there is just the `return true` statement in the `endDocument()` function.

4.7 Question 7

In this solution, we have added `TextAndStyle` to represent the text and the formatting style applied to the text. The `TextAndStyleMemento` class is used to represent a memento of `TextAndStyle`. `MyNotepad` acts as the caretaker and thus requests mementos, stores up to five mementos, as well as using these mementos to perform undo operations.

We have implemented `TextAndStyleMemento` slightly differently from what you may be familiar with. We haven't used a `QString` or a `QStringList` to represent the state of a `TextAndStyle` object. Instead we used `TextAndStyle` as a data member of `TextAndStyleMemento`. This technique eliminates the process of creating a string equivalent of the state of an object as well as string processing to set the state of an object.

Note that `QTextEdit` can be used as an advanced document editor. It has built-in undo facilities. Hence this exercise should be seen just as an exercise to implement the Memento pattern.

It is also arguable whether or not this is a good project in which to implement the Memento pattern. Firstly, `MyNotepad` can work independently without the `TextAndStyle` class. We only use `TextAndStyle` when we want to create a backup. Secondly, the Memento pattern is usually applied when the Originator has strict encapsulation requirements where no getters and setters are provided for this class. However, `TextAndStyle` needs to have getters so that `MyNotepad` can access the saved text and format of the text.

4.8 Question 8

`UDPSend`: The sender's constructor initialises `socket`, sets up a list of `insults` in a `QStringList`, and sets up the GUI interface (with the `connect` statement to begin the transmission). The `startSending()` function creates a socket, sets up the `timer` to produce random timeouts so that the `sendMessage()` function will be called at random intervals. The `sendMessage()` function does most of the work: a random message is written to a `QDataStream` that operates on a `QByteArray`, and this is then broadcast. The `timer`'s interval is changed again to ensure another random wait before the next message is sent.

`UDPListen`: The constructor simply sets up the GUI and `connect`s the listen button's `clicked()` signal to the `startListening()` function/slot. The `startListening()` function creates a `QUdpSocket`, binds it to the port that will be used, and then listens for incoming datagrams. When one is detected, it is signalled via the `readyRead()` signal, which is connected to the `processPendingDatagrams()` slot. The incoming datagram is read into the `QByteArray buffer`, and then used by the `QDataStream` object to read the data into a `QString` and display it in the listener's `QTextEdit` display window. As datagrams are received, they are then displayed in the window.

Remember to add `QT += network` to the `.pro` files for both applications.

5 MARKING RUBRIC

Please note that Questions 1 and 2 were not marked in detail. We just checked that the code compiled and executed correctly.

Question 1 [15]	**Marks**
Get student info • GUI created to allow data entry • Input masks used • Regular expressions used • Data displayed on console in any format	
Program builds and runs, providing some functionality	15
TOTAL	**15**

Question 2 [15]	**Marks**
Process student info • GUI created for initiating and displaying data • Program started as a process • Data from process displayed on GUI • Process properly deleted	
Program builds and runs, providing some functionality	15
TOTAL	**15**

Question 3 [40]		**Marks**
`Student` • Appropriate container for modules • Getters and setters implemented • Correct logic for average and graduate • `const` used as necessary	 2 2 2 1	7
`StudentList` • Implemented as a singleton • `const` used as necessary • Pointer to list and list of pointers • All memory properly deleted	 6 1 4 4	15
GUI • Uses new process to gather student data • Uses student list singleton correctly • Extra functionality added: display, average, grad? • Data correctly displayed to GUI	 2 1 2 3	8
Program builds (6) and runs, providing some functionality (6)		10
TOTAL		**40**

Question 4 [30]		**Marks**
GUI • File written automatically on program close • Checks: list, file open • Data written to file • File closed	 2 2 2 1	7

Serializer		13
• Handled as a separate class	3	
• DOM used to serialise list	8	
• All students and modules included	2	
Program builds (4) and runs, providing some functionality (6)		10
TOTAL		**30**

© UNISA
2018