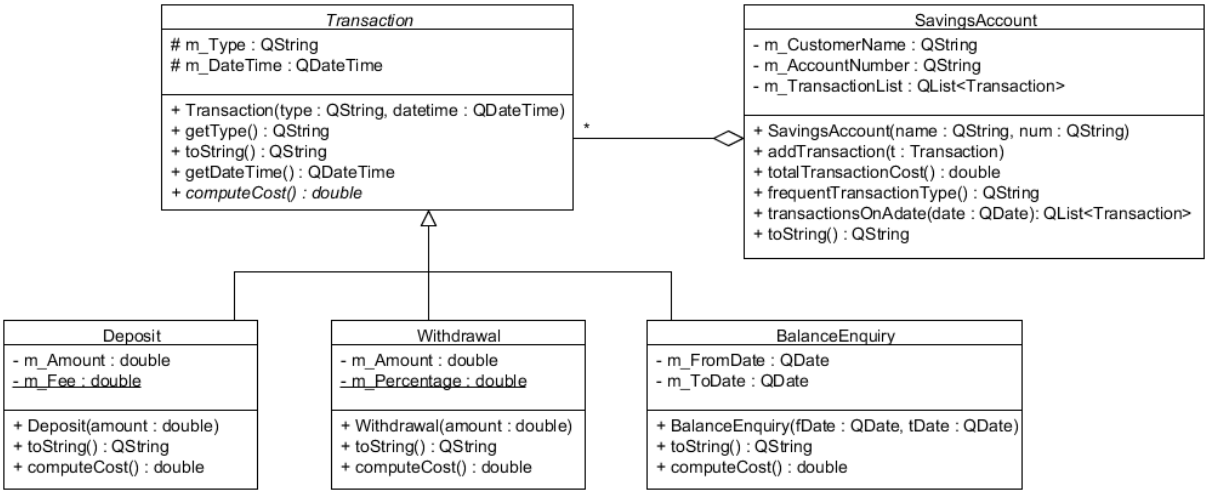
COS2614 Assessment 3

Part A – For Submission

Question 1

Consider the following UML class diagram modelling the classes for different types of transactions for a savings bank account:



A transaction can be of three types: deposit, withdrawal and balance enquiry, which is stored in m_Type, which is initialized in the constructor of the appropriate subclasses. The date and time of the transaction is stored in m_DateTime, which is also initialized in the constructor of the appropriate subclasses. getType() returns the type of the transaction. getDateTime() returns the date and time of the transaction. toString() returns a string representation of a transaction. computeCost() is a pure virtual function to calculate the cost of each transaction.

A deposit is charged a fee as specified in m_Fee. Hence invoking computeCost() on a Deposit object simply returns the value of m_Fee. Initialize m_Fee to a reasonable amount. However a withdrawal costs a percentage of the amount withdrawn. You can decide on this percentage and include this when calculating the cost of a withdrawal. A balance enquiry costs nothing to the customer. toString() functions in these subclasses should reuse the toString() in Transaction and should include values of all its data members in a readable format.

A SavingsAccount instance consists of the customer name, account number and a list of transactions. An existing Transaction object is added using addTransaction(). You should be able to obtain the combined cost of all the transactions in m_TransactionList using totalTransactionCost(). frequentTransactionType() should return the type of transaction that is done most frequently in a savings account. transactionsOnADate() returns a list of transactions carried out on a given date. toString() returns a string representation of SavingsAccount.

Implement all these classes, and write a short program to test all the member functions of the SavingsAccount class. It should add at least 6 transactions (2 of each type) to the account.

Note that you should write a console application, not a GUI app.

Question 2

Write a GUI application to simulate a traffic light (robot) with three possible lights: green, yellow and red. The application should satisfy the following conditions:

- The user is allowed to specify the time interval once on the GUI to be used by the traffic light for changing colours. Once the time interval is set the user should not be allowed to change the time during the execution of the application.
- A QTimer is used to manage the time between different colours of the traffic light

- After the user has specified the time, the application should first display the green light, followed by yellow light, then red light, then again green light, and so on by keeping the uniform time intervals between different colours.
- Traffic lights are displayed on the same GUI as the widget for specifying the time interval.
- Assume that the three different colours of the traffic light are stored in three image files green.jpg, yellow.jpg and red.jpg and you are expected to display these files on the GUI to represent different colours of the traffic light. Note that these three files will be made available on *my*Unisa under Additional Resources.

Question 3

Write a GUI application that records items of stock for a business. A barcode, description, stock level and price are recorded for each item. New items must be able to be added, and existing items must be able to be changed or deleted.

The main window should have a menu bar, with menus File and Edit. The File menu should have options Open, Save and Exit, and the Edit menu should have options to Add, Change and Remove. The main window should also have a toolbar (directly under the menu bar) with a toolbar button for each of these options. See Figure 1.

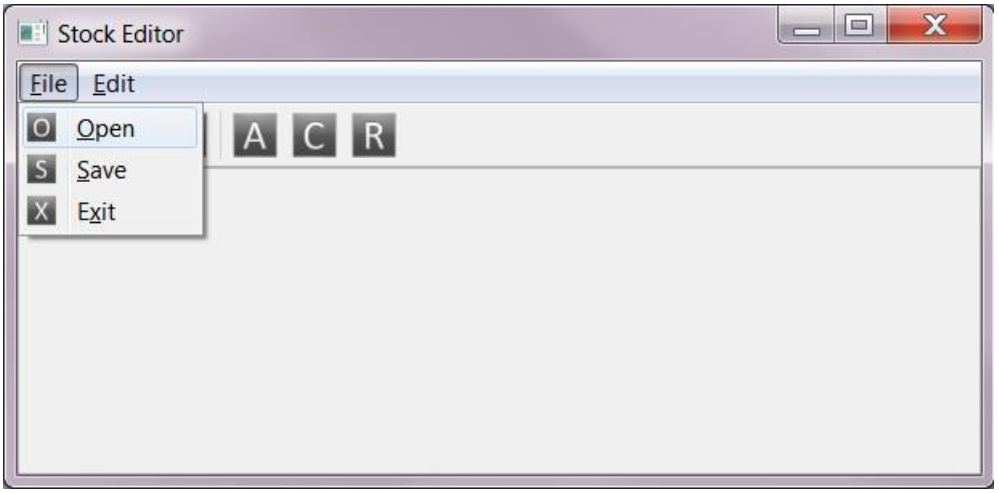


Figure 1

The following should occur when the user chooses each of the options:

Open: A file dialog should be displayed, allowing the user to select the name of a text file containing all the items captured previously.

Save: A file dialog should be displayed, allowing the user to select the name of the text file in which the items captured thus far should be saved.

Exit: The program should exit. If the data has been changed since it was last saved, the program should warn the user and allow the data to be saved if required.

Add: A dialog should be displayed consisting of a form that allows the user to specify the barcode, description, stock level and price of an item. If the user attempts to add an item that has been added previously, the program should tell the user that an item with this barcode already exists, and ask whether the stock should be increased.

Change: A dialog should be displayed allowing the user to specify a barcode. Another dialog must then be displayed with a form already populated by the data for that item. The user should be allowed to change any of the fields except the barcode.

Remove: A dialog should be displayed allowing the user to specify a barcode. If an item with the barcode exists, the user should be asked for confirmation before it is deleted.

In the centre of the window, a list of all the items should be displayed, sorted in order of their barcodes, and should reflect any changes made by performing the options. See Figure 2.

Figure 2

Implementation requirements:

- You may not use Qt Designer to create the user interface. You must code the GUI manually.
- You should adhere to good design principles and your program should provide user-friendly feedback to the user.
- The program must separate the model from the view. The model consists of the non-visual classes used to create, edit and store the items in stock. The view consists of the graphical user interface including the main window with the menus, toolbar and edit box, and the other dialog boxes that are displayed.
- You must use a QMapto store the items in stock in the order of their barcodes. Note that this is part of the model, so the QMapshould not be stored in, or be directly accessed by, the view. All access to it should be through the operations provided by the model.
- Opening a file to load the items in stock into the model, and saving the items in stock in a file, are part of the model. The view code should therefore not do any file handling – this should all be managed by the model.
- We have provided Item, ItemReaderand ItemWriterclasses (in Additional Resources) for part of the model. You must use these classes without changing them.

Hints
:

You can use a static member function of QFileDialogto allow the user to choose a file to open or to save.

Remember that you must submit a program that compiles correctly and displays something on the screen, to obtain marks for this question. (See Section 5.2 of this tutorial letter.) This is a complex application to complete, so we recommend that you build it incrementally, making sure that each new feature compiles and works correctly before adding the next feature.

Approximately a third of the marks will be allocated to the model. Another third will be allocated to the view, and the remaining marks will be allocated to the functionality of linking the model to the view.

Setting up the view is not difficult. The tricky bit is to get the toolbar buttons to perform the same actions as the options on the menus. You should be able to design and implement the model fairly easily with the knowledge you have gained from the first two assignments. The trickiest part is linking the model and view together, so that any actions on the view change the model, and that any changes to the model are reflected in the view.

Questions 2, 3 and 4 of Part B of this assignment develop parts of the solution to this question. We suggest you try them before tackling this one.

Part B (Self-assessment: Do not submit)

Question 1

Write a console app that uses the Item, ItemReader and ItemWriter classes, as provided in Additional Resources. It should simply input an item from a text file, display it, and write it back to a file.

As you will see when you examine these classes, when an instance of the ItemReader class is constructed, it opens a text file (which should contain data about items – one item per line with its values separated by tab characters). The read()function inputs data for one item from the file, and constructs and returns an Itemfrom it. The ItemWriterclass also opens a text file for the purpose of writing the data of a number of items to it. The write()function takes an Iteminstance as parameter, and writes the values of its data members to a line of the file, separated by tab characters.

Once you are sure you understand how the member functions of these classes work, define and implement an ItemListclass for storing a collection of items in sorted order. Either inherit from or use a QMapas a data member for this purpose.

Once you have a program that tests the ItemListclass, add member functions to read a collection of items into it from a text file, and to write all items in it to a text file, using the ItemReader and ItemWriterclasses. Fix your program to test them.

Question 2

Define and implement a class called ItemDialog. It should have two static member functions, ItemDialog::addItem()and ItemDialog::changeItem()(similar to the static member functions of QMessageBoxand QInputDialog) to input a new item, and to change an existing item, respectively. Both functions should return a pointer to the item representing the result of the operation. If the user closes the dialog without pressing the OK button, the function should return the null pointer. Write a program to test them. See Figure 3.

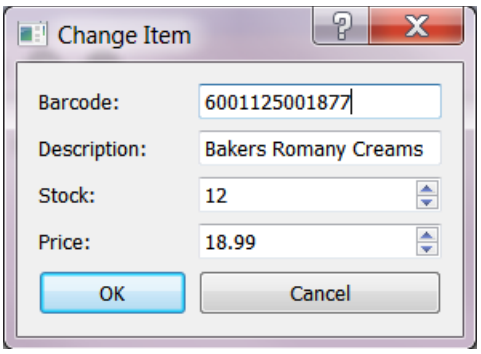


Figure 3

Question 3

Define and implement a class that inherits from QMainWindowto produce the user interface illustrated in Figure 1 above. First make sure that the menus and toolbar buttons are displayed correctly without any actions for the options. Then implement and test one option at a time.