School of Engineering and Built Environment

Object Oriented Software Development

Level: 2

Module Code: M2G605207

Coursework

Issue: 7th November 2013
This coursework comprises 50% of the overall mark for the module.

Hand-in date: 13th December 2013

An average student should be able to complete
this assignment in about 20 hours of work.

Attention is drawn to the university regulations on plagiarism. Whilst discussion of the coursework between individual students is encouraged, the actual work has to be undertaken individually. Collusion may result in a zero mark being recorded for the coursework for all concerned and may result in further action being taken.

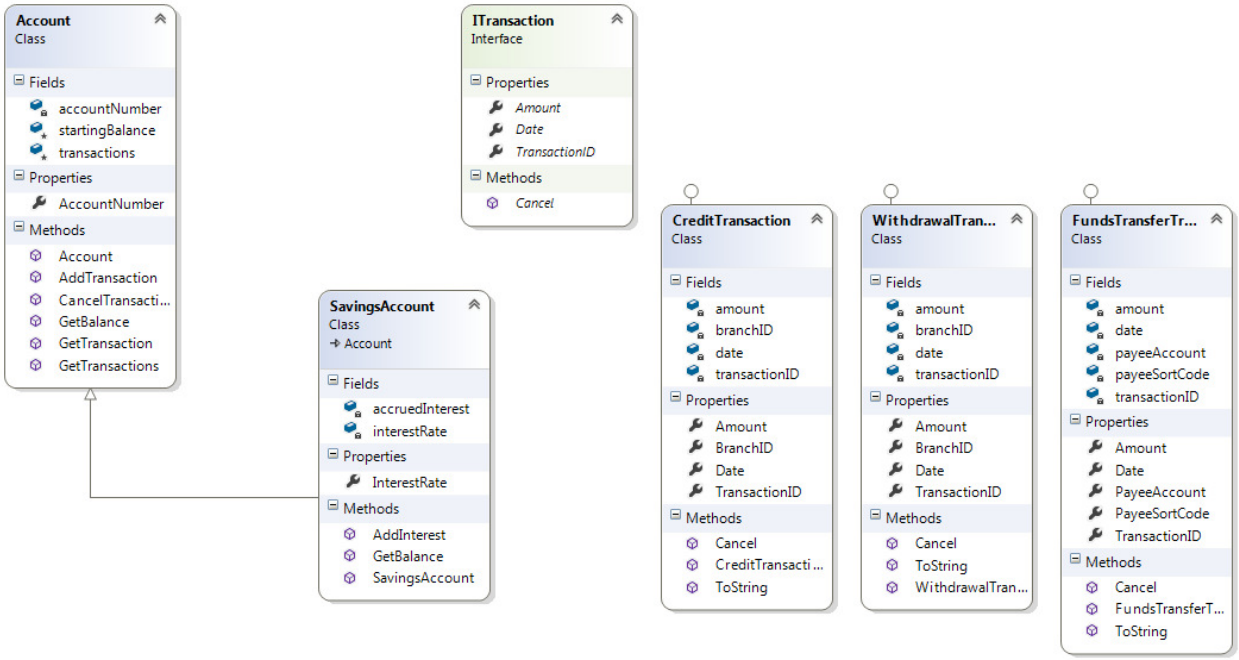
GCU Bank System

GCU Bank requires a new software system to manage accounts and transactions. Some initial analysis and design has been done and set of C# classes partially implemented to represent accounts and transactions. One part of the system will be a screen which will be used by staff in bank branches to record cash credits and withdrawals. This screen has has been partially implemented as a WPF application.

Your assignment requires you to complete the C# classes, complete the WPF window, and to test the application.

Class Diagram

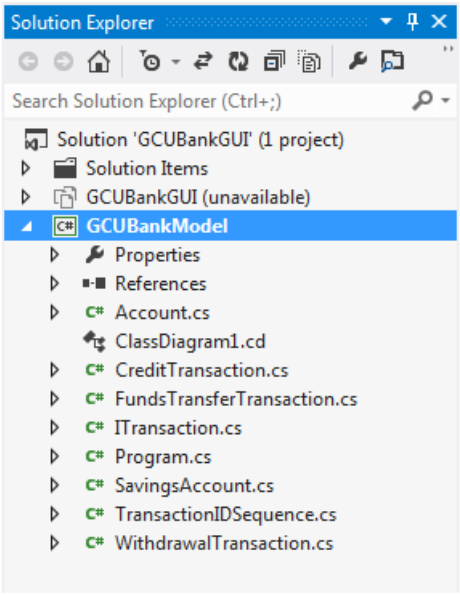
The following class diagram was created in Visual Studio, and shows the required classes and interface, and shows the fields (instance variables), properties and methods of each of these.



Starter Code

You should download *GCUBankGUI.zip* from GCULearn, and extract the contents. The contents consist of a Visual Studio solution which contains two projects, as shown. The *GCUBankModel* project contains the model classes for the system, while the *GCUBankGUI* project contains the

user interface. The *GCUBankGUI* project is initially set to be unavailable as you will not need it right away.



You should complete the development of the application in **four stages** according to the instructions on the following pages.

Stage 1 – Transactions

Tasks for this stage:

ITransaction interface (will be implemented by all classes which represent transactions) – **complete code** following TODO instructions in code comments

CreditTransaction class (represents a cash credit transaction) – **complete code** following TODO instructions in code comments

WithdrawalTransaction class (represents a cash withdrawal transaction) – **complete code** following TODO instructions in code comments

FundsTransferTransaction class (represents a transfer of funds to a different account) – **complete code** following TODO instructions in code comments

Documentation – provide **XML comments** for all public members of your classes and their return types and parameters.

Make sure you read all the TODO instructions in each class

Testing this stage:

It is important at this stage that you check that you have completed these tasks correctly, and some test code has been provided to help with this. Running the test code will give you **feedback** on your progress on Stage 1. It is important that your code first of **compiles** (or builds) with no errors and that it gives the **correct output** when it runs.

Open *Program .cs.* Find the following lines at the top of the file:

```
//#define TEST_TRANSACTIONS
//#define TEST_ACCOUNTS
```

Uncomment the first line

```
#define TEST_TRANSACTIONS
//#define TEST_ACCOUNTS
```

Build the project and run it. Check that you get the output as shown below - all numeric values should be exactly the same as those shown

```
TRANSACTION TEST:
FT T002:11/09/2013  -£200.00  99-99-00:990011
WD T003:11/09/2013  -£50.00  88-88-00
CR T005:11/09/2013  £200.00  88-88-00
Cancelled - FT T002:11/09/2013  -£1.00  99-99-00
Cancelled - WD T003:11/09/2013  £0.00  88-88-00
```

You can also see the expected output in the code in the *Main* method of *Program.cs*, in the section marked with the comment `// TESTING TRANSACTION CLASSES`

Stage 2 – Accounts

Tasks for this stage:

Account class (represents a bank account which contains a collection of transactions) – **complete code** following TODO instructions in code comments

SavingsAccount (represents a specific type of account which pays interest on the balance) – **complete code** following TODO instructions in code comments

Documentation – provide **XML comments** for all public members of your classes and their return types and parameters.

Make sure you read all the TODO instructions in each class

Testing this stage:

Open *Program .cs.* Find the following lines at the top of the file:

```
//#define TEST_TRANSACTIONS
//#define TEST_ACCOUNTS
```

Uncomment the second line:

```
//#define TEST_TRANSACTIONS
#define TEST_ACCOUNTS
```

Build the project and run it. Check that you get the output as shown below - all numeric values should be exactly the same as those shown

```
ACCOUNT TEST:
Account Balance=£250.00
SavingsAccount Balance=£275.00
Cancelled Transaction - Account Balance=£349.00
Cancelled Transaction - SavingsAccount Balance=£350.00
Cancelled Non-existent Transaction - SavingsAccount
Balance=£350.00
Added Interest - SavingsAccount Balance=£385.00
Account transactions:
FT T001:11/09/2013  -£1.00  99-99-00:990011
WD T003:11/09/2013  -£50.00  88-88-00
CR T005:11/09/2013  £200.00  88-88-02
SavingsAccount transactions:
FT T002:11/09/2013  -£200.00  44-44-00:440011
WD T004:11/09/2013  £0.00  88-88-01
CR T006:11/09/2013  £50.00  88-88-03
```

You can also see the expected output in the code in the *Main* method of *Program.cs*, in the section marked with the comment `// TESTING ACCOUNT CLASSES`

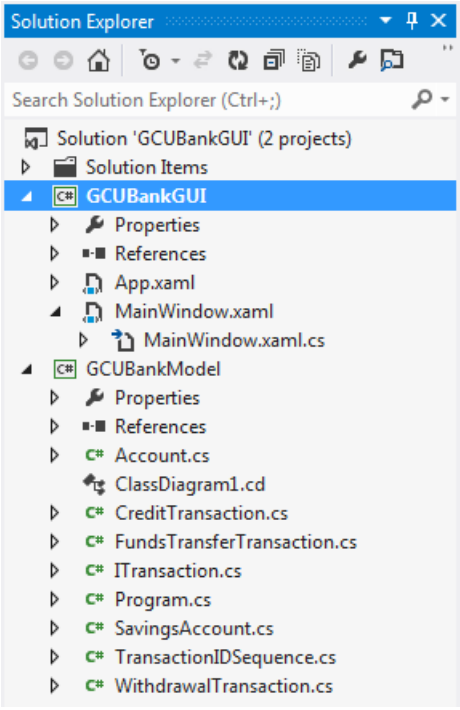
Stage 3 – UI and system testing

You will now complete the screen which is to be used for recording credits and withdrawals on an account. This screen must meet the following **functional requirements**:

1. When the screen loads the current account number should be shown
2. When the screen loads the current account balance should be shown
3. A list of transactions saved during the current session should be displayed – when the screen loads this list should be empty
4. The user should be able to choose Credit or Withdrawal transaction type
5. When the user enters a transaction amount and saves the transaction:
   - a string representation of the transaction should be added to the list of transactions
   - the saved transaction should have the specified amount and type
   - the updated account balance should be displayed
6. If the transaction is a debit (withdrawal or funds transfer) and the resulting balance would be negative, an error message should be shown and the transaction should not be added
7. If the transaction is a withdrawal and the specified amount is > £350, an error message should be shown and the transaction should not be added
8. If the amount entered by the user is not in a valid format (not a number, in other words) an error message should be shown

Some of these requirements are already satisfied by the code you are given, others will be satisfied when you have completed the tasks correctly. You must complete the program and test all the requirements.

To get started with this stage you need to load the user interface project. Right-click on the *GCUBankGUI* project in Solution Explorer and select **Reload Project**. You should now be able to see the files in that project. Right-click again on the *GCUBankGUI* project and select **Set as StartUp Project**.



Tasks for this stage:

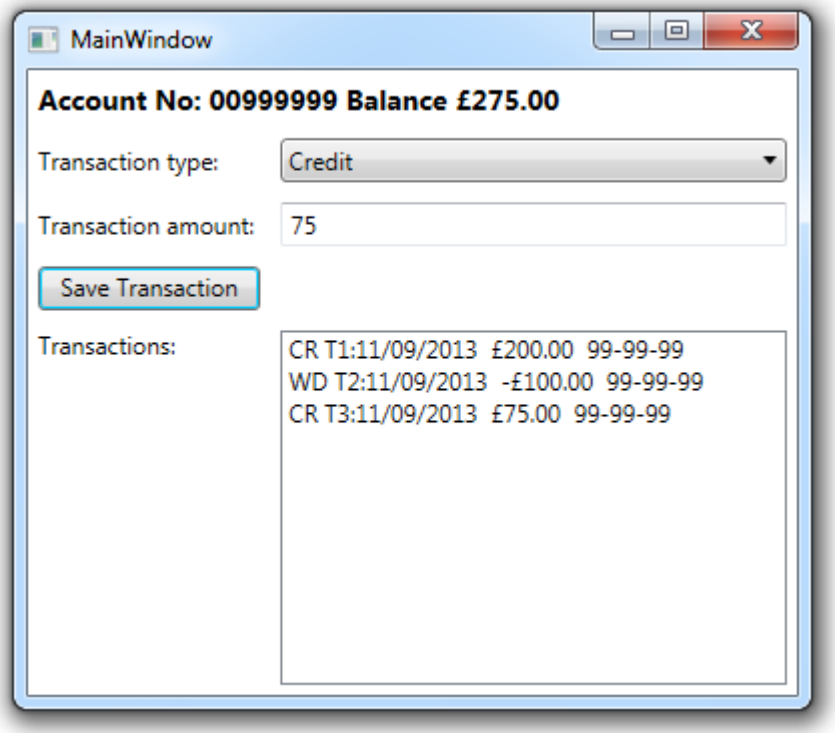
DebitException class – **add a new class:** this should be a custom exception class which has an instance variable *amount* which is initialised in the constructor, and encapsulated in a property *Amount*.

Account class – **modify the code** for the *AddTransaction* method so that if the transaction to be saved meets either of the conditions specified in requirements 6 and 7 above then an exception of type *DebitException* is thrown, encapsulating the amount of the attempted transaction, and the transaction is not saved

MainWindow.xaml.cs – **complete code** following TODO instructions in code comments in the button event handler method

MainWindow.xaml – **complete XAML code**:

- add a binding expression to the text box element *txtAccountNumber* so that it displays the *AccountNumber* property of the Account which is the data context for the window.
- all the controls in the window are currently enclosed in a *StackPanel* container. They are functional, but the layout is not very attractive. Replace this container with a *Grid*, add row and column definitions and add attributes to controls as required to position them on the grid so that the layout looks similar to this:



System testing - You should devise and implement a **system test plan** which tests that the completed program satisfies the requirements listed above. Document your test plan and test results in the format illustrated in lab 8. If any of your tests fail you should document the failure first of all (which will gain you credit for testing), and then attempt to fix the problem and re-test.

Stage 4– Unit testing

Add a **Visual Studio unit test project** which contains a test class for the *Account* class, and contains the following test methods. Each test method should provide a suitable test for the required functionality of the specified *Account* method

- a test method for the *GetBalance* method of *Account*.
- a test method for the *CancelTransaction* method of *Account*.

Assignment Deliverables

You should submit the following:

Implementation:

- This should be submitted as a Visual Studio solution. Make sure that your submission includes the whole solution folder, not just the solution (.sln) file.

Documentation:

- XML code comments, as specified in the task instructions, in your implementation
- Document containing test plans and results of system testing and unit testing. Unit test results may be illustrated with screenshots.

Submission:

- Your implementation and documentation should be submitted in a ZIP archive via GCULearn. Submission instructions will be available on GCULearn.

.

Marking Scheme

*ITransaction interface**	**4**
*Transaction classes**	**12**
*Account classes**	**12**
DebitException class	**2**
MainWindow code-behind	**2**
MainWindow XAML	**6**
System testing	**6**
Unit testing	**6**
Total	**50**

*note that the indicated marks include credit for XML comments, and full credit will not be awarded if comments are missing or incomplete