COMPSCI280 – Assignment 2, 2013

Due: Monday 14 October at 10 am via the assignment dropbox: https://adb.auckland.ac.nz

Help is available from:

- Your tutor
- You lecturer, Jim Warren (jim@cs.auckland.ac.nz)

Introduction

From the 1960s until recently, a very common enterprise computing task would be the 'batch' processing of a collection of transaction records against a master file or database. The transaction information would be collected manually, or at any rate on systems lacking a live connection to the central database, at a local site (e.g. the point of sale). Periodically (maybe nightly or weekly) a batch of transactions would be processed – entered into the computer if necessary, electronically or physically transported to the main data processing centre and there interpreted for updates of the master database. If you (or your parents) have ever written out a cheque, this is basically the kind of processing that's done with it – indeed there could be even a lot more intermediate handling between the point of sale, the vendor's bank and bank the cheque was drawn against.

In the world of 2013 with online purchasing and cloud computing, batch processing is on the decline – then again, if your dear lecturer wants to make an expense claim to the University, it's still only processed at a specific time each week. And there are still many other reasons that you may need to write code to process a file of transaction records. Due to acquisitions or mergers, the point-of-sale system and central database may not integrate directly, requiring a program to patch them together. Or even when using an online transaction processing system (OLTP), transaction records may be periodically processed to prepare the data for use with analytical tools – perhaps for 'data mining' to better understand consumer needs, or as an internal or external audit for fraud detection.

For this assignment you will be working with a file of credit (essentially deposit) and debit (essentially withdrawal) transactions. The file will have several classes of detectable errors in it. Your task is to sort the transactions into three classes:

- A file of credit transactions
- A file of debit transactions
- A file of transactions with detected errors

For Part 2 you'll develop a graphical interface for review of the transactions and interactive editing of the transactions with errors.

Part 1.

Create a new Visual C# Console Application. It must process a transaction file (Transactions.txt, available from the Assignments page of the course website) with the following fields:

Field	Format	Value restrictions
Client Number	9-digit number; may have leading zeroes	Final digit must be a valid check digit*
Transaction Type	2-characters	'Cr' for a Credit transaction or 'Dr' for a Debit transaction
Transaction Date	dd/mm/yyyy	Transaction date must be in 2011 or 2012
Transaction Amount	1-4 digits, a decimal place, and two digits	Must be >0 and <5000

* Using the Luhn algorithm: see http://en.wikipedia.org/wiki/Luhn_algorithm

The transaction file includes the above fields in order, separated by commas. You must work with the transaction file exactly as provided. The file has errors in terms of violations of the above formatting and expected values. Use **try/catch** statements to capture errors due to data that is missing or not compatible with its expected data type. Generate custom errors (use the **throw** statement) where you detect invalid values. You must successfully detect and process the following types of errors.

Field	Error
All	Truncated record (i.e. having less than four fields)
Client Number	Invalid check digit
Transaction Type	Invalid type code (anything but 'Cr' or 'Dr')
Transaction Date	Missing
	Invalid
	Out of range (before 1 January 2011 or after 31 December 2012)
Transaction Amount	Missing
	Non-numeric
	Negative
	Zero
	≥5000

You will produce three output files: CreditFile.txt, DebitFile.txt and ErrorFile.txt. Any transaction with a detectable error must be written to ErrorFile.txt. All 'clean' transactions go to CreditFile.txt or DebitFile.txt depending on their transaction type code. The format of the transaction records should be exactly as it was on input (e.g. comma delimited) – write transactions with errors to ErrorFile.txt exactly as they appeared in Transactions.txt.

Write your solution to read the transaction file from, and to write the output files to, the folder at the same level as the solution project directory (i.e. to the parent folder of the Solution file – if your file Part1.sln is in C:\CS280\A2\Part1\ then read and write .txt files in C:\CS280\A2\).

Your program should write to the console the following information:

- One line of descriptive output for each error transaction found including the transaction number, counting from 1 for the first record (e.g., "Record 25: ERROR – Transaction Amount Non-numeric").
- Final output summarising how many records read and how many written to each file.

Use the ReadLine method so the console stays open for the user to review the output until they press Enter.

Part 2.

While the batch processing in Part 1 was perfectly suitable to a console application, most users nowadays expect a GUI WIMP interface (graphical user interface, with windows, icons, menus and pointing devices). And such an interface is not bad for reviewing the results and manually correcting some of the errors.

Create a new Visual C# Windows Forms Application. The start-up form should have a control at the top of the window (can use a MenuStrip or ToolStrip as you wish) that pops up a modal dialog that allows the user to select the three input files for Credit Transactions, Debit Transactions and Error Transactions, respectively – these should default to the three files output from your Part 1 application (and, as with Part 1, at the level of the parent folder of the Solution file). When the user selects 'Load' from the modal dialog (they should also have the option to 'Cancel'), the modal dialog will close and a tab control on the main form should be populated with the contents of each file visible in a data grid (i.e. three data grids, one per file, on three tabs).

On the tab showing the Error Transactions, the user should be able to select any one of the transactions. When they press a 'Correct…' button a modal dialog should pop up that displays the current values of the four fields for the selected transaction (including "[Missing]" if the value is missing/null due to being blank in the transaction file or due to its record being truncated), and that allows the user to enter new values for each field. The dialog should also show at least one current error in the transaction's data (e.g. if it has a missing field, or an invalid check digit etc.). The dialog should have three buttons:

- Validate: to re-check the transaction data as entered by the user and display at least one error, if any is present, or "Free from errors" if it's OK, on the modal dialog form
- Save: if the transaction still has errors, the data is written back to the record on the data grid in the Error Transactions tab; if the transaction is free from errors, the old record is removed from the Error Transactions data grid and added to the end of the data grid of transactions on the Credit Transactions or Debit Transactions tabs, depending on its type. Either way, the modal dialog closes and the user is returned to the main form with the Error Transactions tab showing.
- Cancel: close the modal dialog without updating the transaction and return to the main form with the Error Transactions tab showing.

The main form should also have a control that pops up a modal dialog for saving the updated transaction files. It should default to the same files as were used for input (i.e. CreditFile.txt, etc.). The user should be able to change the file names, but there shouldn't be an error if they wish to overwrite the old ones. The dialog needs buttons for Save and Cancel.

Hints

The lecture slides and lab sheets cover the basic programming techniques that you'll need for this assignment.

For reading the transaction file, use of the **split** method is recommended: see http://msdn.microsoft.com/en-us/library/tabh47cf.aspx.

For Part 2, don't feel that you have to do everything programmatically. For instance, to create, position and label Button controls, go ahead and do that visually in Design mode; double-clicking on the image of the Button adds a click event handler and takes you to its code.

It's handy to modify the constructor for modal forms that you design. In particular, you can pass a reference to the main ('parent') form to the modal form when you create an instance of it. That way the modal form has handy access to any public variables or properties of the parent. This is a good way to exchange information such as file paths, StreamReaders or any other data. For instance, if the main form is the default Form1 and the modal dialog form is open_dia (a dialog for getting the transaction files to open), open_dia.cs could be edited with the three commented lines below:

```
    public partial class open_dia : Form
    {
        Form1 parform; //this variable is visible to components on the open_dia form

        public open_dia(Form1 mypar) //pass in the identify of the form that makes me
        {
            InitializeComponent();
            parform = mypar;  //share ref to parent form to other methods in the class
```

You'd pop up the modal form like this (passing the reference to the Form1 instance via the constructor):

```
            open_dia op = new open_dia(this);
            op.ShowDialog();
```

Use OpenFileDialog controls (see http://msdn.microsoft.com/en-us/library/system.windows.forms.openfiledialog.aspx) to allow the user to select file names for loading, and SaveFileDialog controls for saving. Note that you can use the FileName property of the OpenFileDialog (or SaveFileDialog) control to set a default file name (e.g. "CreditFile.txt" etc.).

Do not use absolute directory paths in your code (your directory structure probably isn't identical to the marker's directory structure). The following code sets the OpenFileDialog control to open onto the directory four levels up from the running .exe file (the required directory for your transaction files):

```
            string binpath = Path.GetDirectoryName(Application.ExecutablePath);
            DirectoryInfo d = Directory.GetParent(binpath).Parent.Parent.Parent;
            openFileDialog1.InitialDirectory = d.FullName;
```

For the data grids in Part 2, use DataGridView controls; see http://msdn.microsoft.com/en-us/library/system.windows.forms.datagridview.aspx – as in that example, leave the DataSource property as '(none)', set the ColumnCount property (to 4 in this case), and use the control's Rows.Add method to add the fields from the transaction file (Credit, Debit or Error) to the appropriate data grid.

Submission

Zip each project up using WinZip or 7-Zip as a .zip file and submit using the assignment dropbox. You will have two files for submission: A2Part1.zip and A2Part2.zip.

Do not submit the transaction (input) file. The marker will run your code against the original version of the file.

Notice

The usual university policies for academic integrity in coursework apply. See http://www.auckland.ac.nz/uoa/home/about/teaching-learning/academic-integrity/tl-uni-regs-statutes-guidelines. In particular, you need to avoid copying work and presenting it as your own (plagiarising), or enabling others to plagiarise from you. Don't share source code files relevant to the assignment (by thumb drive, forums, email attachments or other means), don't borrow or lend (or steal) code printouts, and don't lose track of printouts you make. Discussion with fellow students of general approaches, including mention of individual classes, methods and statements, and exchange of pointers to relevant third party examples (e.g. on MSDN or StackOverflow), is permissible. But if multiple students submit work with stretches of code that the lecturers deem similar beyond chance, expect Academic Misconduct proceedings.

Marking scheme

Part 1

Max points	Description	Details
2	It runs!	Program loads the transaction file and produces output without unhandled exceptions (even when running it a second time)
4	Correct counts	Correct number of records read, and sent to each file (minus half a point if off by one on a count)
4	Correct transaction errors	The console output re the transactions with errors describes one applicable error for each transaction with errors (and for no other transactions!)
3	Output files sound	Output files formatted correctly
2	Error handling	Use of **try/catch** and **throws** in the code
15	**Total**	

Note: As much as 10 points may be deducted if the code has any 'case-specific' logic (i.e. based on specific record numbers or specific erroneous data values particular to the sample transaction file).

Part 2

Max points	Description	Details
2	It runs!	Program launches, can be navigated and loads/displays file data without unhandled exceptions
3	Good file load dialog	Provides default files to load and allows changes, with error handling
3	Tabbed display	Shows loaded transactions in three tabs, each with a readable tabular display that scrolls
5	Validation dialog	Brings up details of selected error transaction, including an error description; allows edit, validation and save (that refreshes the tabbed displays)
2	File save dialog	Writes back files, allowing change of name or overwrite
15	**Total**	