**CIS 3309**

**Component-Based Software Design**

**The Suggestion Box –**

**Suggestions on the Design of Your ATM Project (Project II)**

**Fall SeMESTER 2020**

Project Goals/Objectives

This is the second of two CIS 3309 Lab Projects (Project II) intended to orient your programming skills toward the basic principles of object-oriented design and programming. Our objectives are to reinforce what was learned in Project I about information hiding, limiting the need to know, encapsulation, class cohesion and coupling, and to ensure that everyone has mastered C# programming techniques as described in Chapters 1-12 in the text, especially related to array list manipulation, the use of text files and streams (Chapter 21), form input validation, debugging, writing loops (such as sentinel-controlled while loops), the design and coding of multi-form projects. We will

* Put into practice the basic ideas underlying MVC by constructing 1) the Models of the key data structures required for your project, 2) the Views of what you want the user to see (and where appropriate, the data structures behind these views), and 3) the user interactive Controllers). (For our purposes, the Views and Controllers will most often be the same.)
* Practice incremental implementation, starting with the core part of your project and adding and testing new features step-by-step.
* Learn more advanced programming techniques in C# .NET.
* Learn more about how to design windows forms that meet the needs of your project and make sense to the user.

Project Introduction:

This Project II involves writing a simulation for a simplified ATM machine. This may be a bit overwhelming at first. Perhaps the best approach is to get started on the project now, by reading over the material in this document and ASKING QUESTIONS.

PLEASE NOTE: There are many good approaches to this project. At best, this might just be one of them. COME TO CLASS WITH QUESTIONS!!

Project Fundamentals:

Three aspects of this project should be developed in parallel and in some detail.

1. A detailed map of the behavior of your ATM system (a Behavior Diagram) is needed. There should be a separate diagram for each form you design. As you work on the behavior diagrams, all the problem domain entities to be manipulated should become clearer, and other entities, sometimes referred to as purely fabricated classes, needed to solve a problem but not directly related to a problem domain entity.
2. A model (class description) is needed for every entity (problem domain or fabricated) that requires processing as specified in the behavior diagram. Each field and property of the class should be described, along with each method (and its return values and arguments). The exceptions handled should also be documented. The attributes, properties, methods, and exceptions should be described in separate sections for each class. The method descriptions, return values, and arguments should also be described in separate sections for each method. For example:

**Class x – statement of responsibility or purpose**

 Attributes – (list of data stores and description of each)

 Properties – (list of properties and descriptions – if you use any Properties)

 Methods –

 Method1 – description (purpose or responsibility)

 Return Value (if any) - describe

 Arguments – (list and description of each)

 Method2 – etc.

 Exceptions to be Handled – (list and description of each exception you handled)

1. A reasonable set of forms should be developed and the flow among must be clear from the behavior diagram. I suggest four forms, which I will be happy to discuss in class.

The Project (Statement of Required Work – with hints)

1. To simplify your behavior diagrams, separate them according to the forms you will use – one set of diag- rams for each form. I expect a neat, uncluttered, and detailed diagram at the time the project is due.
2. The Underlying Layer of Classes – As summarized below, these classes must contain every piece of data and every method required to make your system work properly. The only way to figure out what goes in these classes is to maintain a list of attributes, properties, and methods that are needed based on the details of your behavior diagram. I will map out a partial outline of what is required here. You will need to do the rest, including completing the descriptions of the attributes, properties, and methods I have listed here (or those that you used). When you write your code, I strongly urge you to use Message Boxes in every possible place where you think you will want to know whether or not your code is working.
3. The ATMBank Class – This is a relatively simple class.

Attributes – I try to prefix the names of attributes that have related properties with the word hidden

Account length (I fixed this at 5 characters, all digits) (hiddenAccountLength)

 Max number of user tries (3) to enter information before being referred to the manager

 Maximum allowed daily withdrawal amount ($300)

 Current text file – sequential file containing bank customer records before any transactions

 Updated text file - sequential file containing bank customer records after any transactions

Properties – (You do not need properties. But without them you need additional methods)

 One for each field – accountLength for example

Methods – I treated the collections of customers (the current and updated files) as belonging to the bank (along with the attributes and associated properties)

 findCustomer – calls customer.customerMatch to read records and look for an account
 as entered by a user

 copyRemainingRecords

1. The Customer Class –

Attributes –

customer – a string used to store all the information related to ONE customer record (the string split method is used to split out the five components of the records, which once split into an array list of substrings can be stored in the attributes indicated below.

recordAccountNumber -

recordName - string name of the acocunt holder

recordPin – pin fo the account holder (4 digit integer with no leading zeros; value > 1000)

recordChkBalance – decimal > 0 and with a maximum of two digits to the right of the decimal point

recordSavBalance – decimal > 0 and with a maximum of two digits to the right of the decimal point

Properties (none)

Methods –

customerMatch – given a customer account (entered by the customer) and a customer record (read from the currentFile) checks to see if the user account matches the record account (the first field of the record). Returns true or false.

verifyNameAndPin – given the name and pin entered by the customer, checks to see if they match the account number entered earlier. Returns true or false.

modifyCustomerRecord – modifies customer record based on requested transaction and returns the modified record. Input arguments involve a transaction code (1-8, or an element of an enumeration) and the amount involved (an amount of 0 if a balance is requested)

getCheckBalance – returns the current state of the customer’s checking account balance

getSavingsBalance - returns the current state of the customer’s savings account balance

1. The currentFile Class – (to be given in class)
Attributes –
 string currentFilePath – file path

 System.IO.StreamReader currentFileSR – stream reader object

 integer recordReadCount – tracks number of records read

Properties – (none)

Methods –

(constructor) – sets count of records read to 0 and opens the current file

getNextRecord – gets and returns the next record from the current file (also returns a Boolean flag indicating if the end of file was encountered)

getRecordsReadCount – returns number of records read

closeFile – closes the file

rewindFile – (not yet written)

1. The updatedFile Class –

(very similar to the currentFile class)

1. This is a multi-form project. I strongly urge you to limit the number of forms to 4 or 5.
	1. Declare and instantiate the classes and objects you need as public data accessible to the rest of your system. You might wish to declare and instantiate a couple of the forms (that have multiple uses) in a separate GlobalDataClass that it accessible to all the other classes. This will simplify life a bit. In our case, once you have the declarations of the objects we need to have accessible our system, all we need to do is create an instance of your first form in the module (I chose a splash screen form). This will cause the form to appear dynamically, during the execution of your program. As a suggestion, consider declaring four forms. The first is the Splash Screen, which could be used to request the user’s account.
	2. UserAccountEntry Form – Welcomes the user and asks for the user’s account number. I used the code behind this form to verify that the account was properly structured (a five digit integer) and then to search the input file to find the record for this account. This search is complicated and if the account number is not found, the required steps are alo a bit complicated. If the account number sought was found, you can then turn control over to the …
	3. UserNameAndPin Form – Use this form to ask the user (the customer) to enter his/her name and pin number. Check the user name and pin against the record for the account. Give the user three chances to get the name and pin correct before telling them to see the bank manager and ending the program (and closing all the files). Once a matching name and pin was entered, you can turn control over to the …
	4. TransactionEntry Form – This was the most complicated of the forms. You might want to consider displaying its controls in three stages. For example, start by putting eight buttons on the form ((1) Savings Deposit, (2) Savings W/D, (3) Checking Deposit, (4) Checking W/D, (5) Savings to Checking Transfer, (6) Checking to Savings Transfer, (7) Savings Balance, (8) Checking Balance. Six of these transactions require the user to enter an amount (and press OK).

The controls for entering the amount and pressing OK could be made visible only after a transaction button was selected. Once OK was pressed, you can make more controls visible for the purpose of asking the user if this was the desired transaction and to giving the user Yes and No buttons. If No is selected, reset the form controls and ask the user to reenter the transaction information (button and amount). If Yes is selected, go on to process the transaction, write out the updated record, and then turned control over to the …

* 1. TransactionCompleteForm – Used to display the balances on both accounts and ask the user if he or she wants another transaction (back to the TransactionEntry form) or not. If not (if the user is done), copy all the remaining records from the currentDataFile to the updatedDateFile, display the total number of records processed, close the files, and then close the forms.

Control flows from one form to the next with a loop going back to the TransactionEntry form in case there are multiple transactions for the same user.

The above comments should provide you with some idea of the large-grained behavior of the ATM system. You will also note that, as is the case with almost all such systems, the ATM project is layered. That is, the code behind these four forms involves the manipulation of four key entities or objects, the *Bank* object itself, a *Customer* object, and two objects that are really collections of customer objects, the *currentDataFile* (input) and the *updatedDataFile* (output). The classes that model these objects make up the layer of code between the forms and the CLR. Virtually every line of code you write as the code behind these forms will involve access to the methods and properties of these four classes. This is the way it is supposed to be.