**CIS 3309**

**Component-Based Software Design**

**The Suggestion Box –**

**Suggestions on the Design of Your**

**BOOKSTORE INVENTORY Project (PROJECT II)**

**Ver 6 - Spring SeMESTER 2019**

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 the maintenance of a bookstore inventory (a collection of books) and a collection of bookstore employees. 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 Bookstore Inventory 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 –

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.

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 diagrams for each form. Please try to keep your diagrams neat, uncluttered, but as detailed as needed to complete the project.
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 an 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. They can be a great debugging tool in tracking the behavior of program. You can always comment some of them out if they become too annoying.
3. The Employee Class – Provides a model for a bookstore Employee entity. Note that the information about each Employee initially will be stored in a text file, one record per Employee. Each record contains five substrings each separated from the other by an asterisk and one or more blanks. We use this file only to validate an Employee’s AccessID and Pin and to update the last date of Bookstore Inventory access for this Employee. The Employee is not allowed to change any of this information on his/her own. (See also Item B that comes next.)

Attributes: I try to prefix the names of attributes in the actual code for a class with the word hidden. You need to fill in the description of these attributes, what they represent and how they are used. Examples: hiddenName; hiddenAccessID

Access ID (a 5 digit integer) --

Name (text) --

Pin (4 digit integer) --

Annual Pay (decimal) --

Last date of access to inventory (date) --

\*\*\* Employee Data String – a string containing the above information as text (read from the file and to be split in this class, with the information stored in the above 5 attributes). You also need a list of strings for split to store the results of its work, before you place these results in the above attributes. The string should be an argument to the createAndPopulateEmployeeObject method and the list of strings should be local to this method. They should not be considered attributes of the Employee Class. (Why not?)

\*\*\* Valid AccessID Length and Valid Pin Length – constants for validating lengths of input

Properties: (You define the get and set properties as you see fit.)

Methods: (Mostly public methods. Others may be needed. Arguments to be added)

(constructor) – used to perform any required initialization for the Employee class.

checkEmployeeID – Compares the Employee Access ID entered by the user to the Access ID in a list element. If no match, ask for re-entry. If there is a match, we can then move on to …

createAndPopulateEmployeeObject – Takes a string from the Employee Text File, splits it into 5 components, creates an Employee object from this string and adds it to the EmployeeList Class (described next). All 5 elements from the Text File must be validated here.

createStringToWrite – converts Employee attributes to a string suitable for display in a Message Box

updateEmployeeTransactionDate – updates the Last Date of Access for an Employee to the current date

verifyPin – given the Pin entered by the Employee, checks to see if it matches the Pin corresponding to the Access ID entered by the user.

Exceptions: Handle cases in which the Employee Access ID cannot be found and/or Pin in Employee Object does not match the Access ID. In addition, handle cases of failed validations (checked as records are read and stored in an Employee Object) of

Access ID (must be a 5 digit integer),

Pin (must be a 4 digit integer),

Name (cannot be blank),

Annual Pay (must be a decimal),

Last Date of Access to the Inventory (DateTime format)

1. The Employee List Class – Provides a model for an internal array list used to store the information about all of the employees working for the Bookstore. This list initially will be populated from the information stored on the Current Employee Text File, and at the end of execution of your program, it will be written out to an Updated Employee Text File.

Attributes:

InternalList – an array list containing data on all Employees

index – holds index in List where current logged in Employee is stored.

Properties: (You define the get and set properties as you see fit.)

Methods: (Some of the public methods. Others may be needed. Arguments to be added)

(constructor) – used to perform any required initialization for this class

findEmployeeInList – Searches the Employee List to find an object with an Access ID that matches the ID entered by the Employee. (Saves and return employee index in the list

getCount – Returns the Count attribute of the InternalList

initializeEntireList – Reads each record in sequence from the currentEmployeeFile object, converts the string read to valid Employee object attributes and adds the object to the list. Continues to read (using a sentinel-controlled while loop) while there is still data in the file. (Note that in order to actually create an object to be added to the list we have to pass each string to the Employee class so it can split the string and store the substrings (create an Employee object to be inserted).

updateEmployeeObject – Updates last date of access to today’s date (NOW).

verifyPin – calls verifyPin in in Employee class, passing the Pin.

writeEntireListToFile – Writes the contents of the entire EmployeeList object to the updatedEmployeeFile object.

Exceptions: Handle case in which Employee file object is empty or cannot otherwise be read.

1. The Book Class – Provides a model for a Book in the Bookstore inventory

Attributes –

Note: This class is similar to the Employee class. The information about the books in the Store inventory is stored in a text file, one record per book. Each record contains seven substrings each separated from the other by a dollar sign and one or more blanks. [The main difference between how we handle Books as opposed to Employees is that the collection of Employees to be manipulated is stored in an internal storage array list to be filled from the Employee file when the program begins, and written back out to an updated file when the program is ready to terminate. The Books collection is never stored internally. Rather information on each book is read and processed one record at a time from a text file (of Current File type) and when processing is complete, updated information is stored in another text file (of Updated File type). Only one record at a time is stored internally. (Note that the Bookstore might have a few employees, but tens of thousands of books, which could take up considerable internal storage). In addition, we need some exercise in manipulating array lists and additional work in manipulating external text files.)

ISBN (string: 3 digits, a hyphen, and 3 more digits: no spaces)

Title (string)

Author (string)

Price (decimal)

Number on hand (integer)

Date of last tranaction (date)

\*\*\* Book Information String – a string containing the above information as text (to be split in this class with the information stored in these 6 attributes). Again, you also need a list of strings for split to store the results of its work, before you place these results in the above attributes. NOTE: This string in this case can be an attribute of the Book class as it is used by more than one method in the class.

Properties (none unless you want some)

Methods –

bookMatch – given a book ISBN (as entered by the user) and a record from the current Book File, checks for a match. (Must split and save the record contents in Book’s attributes for future reference, in addition to checking for an ISBN match.)

displayBookRecord – creates a nicely formatted string to display in a Message Box. (Converts content (attributes) of a Book object to string to a format suitable for display in a Message Box.)

modifyBookRecord – (not used – modifications to a book record are written directly to the Updated Book File)

Exceptions: Handle case in which Employee ISBN cannot be found and all cases in which any data involved in a Book transaction are not in proper format or values make no sense (such as a negative price, etc.)

1. The BookStore Class – This is a relatively simple class. Note that the currentEmployeeFile and currentBookFile objects belong to this class. So does the employeeList object. Employees and Books are, in a very real sense, part of the Bookstore. Thus, this model of the entire store must contain the collections of Employees and Books. For this project, these Book and Employee collections are stored in external text files. But the Employee collection is small enough that we decided to temporarily store the information nternally in an array list.

Attributes –

Book object

Employee List object

Current Book File object and path

Current Employee File object and path

Updated Book File object and path

Updated Employee File object and path

\*\*\* Also used to store system constants as defined by the Bookstore owner. For example – information about restrictions on lengths of some of the Employee and Book attributes.

Properties – (Whatever you think you need or want. I used a bunch of them in this class)

Methods –

checkForDuplicateRecord – checks for a duplicate record when user attempts to add

new book

closeFiles – closes all external files

copyRemainingRecords – after an employee’s changes are processed, copies all

remaining records in the currentEmployeeFile to the updatedEmployeeFile.

findAndSaveBook – given a book ISBN (entered by the user) and a book record (read from the currentBookFile) checks to see if the user ISBN matches the record ISBN (the first field of the record). If there is a match, this method splits the record into its 6 component parts and saves them.

findEmployee – given an employee’s Access ID uses a sentinel-controlled while loop to find the correct employee record in the EmployeeList (calls findEmployeeInList in the employee List class).

rewindFiles – rewinds all external files prior to closing

writeEntireEmployeeListToFile – calls writeEmployeeListToFile in the Employee list class to write all Employee records in the List back out to the updatedEmployeeFile.

writeOneRecord – writes the contents of one Book record to the updatedBookFile.

1. The CurrentFile 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 – rewinds a file

1. The UpdatedFile Class – (very similar to the currentFile class)
2. This is a multi-form project. I strongly urge you to limit the number of forms to 4 or 5.
   1. You may wish to declare and instantiate in a separate GlobalsClass any objects (if there are any) you need to be publicly accessible to the rest of your project. You might also wish to declare and instantiate in the GlobalsClass a couple of the forms that have multiple uses (if there are any) and need to be accessed in several different forms or classes. This will simplify life a bit.
   2. Remember that your first form, the EmployeeAccessIDEntry Form, needs to be designated as your startup form.
   3. EmployeeAccessIDEntry Form – Welcomes the user and asks for the user’s account number. Note that a user of this system needs to be an Employee of the Bookstore and needs to have a valid Access ID (and Pin). Information about all employees (fewer than 10 of them) will be stored in a text file which we will model with the CurrentFile class (discussed earlier). I suggest you use the code behind the EmployeeAccessIDEntry Form to verify that the Employee Access ID was properly structured (a five digit integer) and then search the Employee file object to find the record for this ID. This search is complicated and if the AccessID number is not found, the required steps to give the user another chance at valid data entry are also a bit complicated. If the AccessID number sought was found, you can then turn control over to the …
   4. EmployeePinEntry Form – Use this form to ask the user (the Employee) to enter his/her pin number. Check the user pin against the record for the account. Give the user three chances to get the pin correct before telling them to see the BookStore manager and ending the program (and closing all the files). Once a matching pin is entered, you can allow the Employee to choose a Book transaction by turning control over to the …
   5. TransactionSelect Form – This is the most complicated of the forms. The transactions to be implemented include (1) Add New Book to Inventory, (2) Update a Book in Inventory, (3) Delete Existing Book, (4) Processing Complete (Done). In all cases except (4), the user should be required to enter the ISBN Number for the book involved in the transaction. The user should be given the option to indicate whether the transaction option and ISBN number selected is the one they want (Yes or No). If No is selected, reset the form controls and ask the user to select the desired transaction and ISBN. If Yes is selected, you can allow the user to enter data relevant to the update or add transactions and then complete processing. Again, displaying the results of the transaction and the updated files is important after which files and forms can be closed. Use Message Boxes for this – use them a lot so you can see what is working and what is not. You will also find the debugger really helpful.

The above comments should provide you with some idea of the large-grained behavior of the Bookstore Inventory Project. You will also note that, as is the case with almost all such systems, the Bookstore project is layered. That is, the code behind these three forms involves the manipulation of 8 key entities or objects, the *Bookstore* object itself, *Book, Employee* and *Employee List* objects, and four text based file objects, currentBookFile and currentEmployeeFile (instances of the CurrentFile class), and updatedBookFile and updatedEmployeeFile (instances of the UpdatedFile class). 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 your forms will involve access to the methods and properties of these four classes.

Stage I Requirements for the Bookstore Project:

Write the code behind for the first three forms.  Test it, including validating forms input **an**d the input from the current Book File as it is read, record for record with each record stored in an employee object. I can show samples of these forms. They are relatively simple, and some of the code behind is simple,

**NOTE:** The handling of the entry of a Book ISBN Number that cannot be found in the current Book File is complicated, so the sooner you start thinking about this the better. Hint: You will need Sentinel Controlled While loop

Stage II Requirements for the Bookstore Project: (The final, completed code for the project, as described above))