**CIS 3309**

**Component-Based Software Design**

**Suggestions on the Design of Your FINAL Project**

**SPRING Semester 2021 (ver 11.0, March 29, 2021)**

**(This document contains references and links to other reading that might be helpful)**

**Introduction:**

The goal of the Employee/Manager/Client (EmpMan for short) Project is to illustrate the use of 1) inheritance and 2) databases in modeling the data for a small, simple organization having graduate students, undergraduate students, faculty, and chairpersons. The Project involves the use of one form (shown later in this document) and requires 3) the *validation of all data* entered in the form to prevent bad data from getting into the database. We have added the use of 4) a serializable file (*Sfile for short)* to illustrate some issues related to these files, what they look like, and how to read and write them. You may also find it helpful to learn just a little bit about 5) *regular expressions* to help in the validation process. The focus is on building 6) a hierarchical class structure and 7) a relational database that realistically and efficiently models the relevant entities in the problem domain.

For Stage I of this project, data will be entered into the form and used to create and initialize objects of type Client, Worker, and Manager. These objects will be inserted into an PersonList object. We will also want to be able to delete, update and search this list. Each time the Project executes (and some set of these operations are performed on the list), the list then will be written (dumped byte for byte) to the Sfile. Note that the Person list exists only during the execution of your program. The serializable file, on the other hand, is a persistent file, stored on a disk. It does not go away when program execution terminates.

Once your Stage I work is complete and functioning correctly, we will leave the use of the Serializable File and the Person List behind and embark on Stage II of the project, repeating a process similar to that of Stage I, but using a simple relational database instead of the list and the Sfile. All references to the serializable file should be removed from your program once you start on this Stage II of your code. References to the list may be removed, except possibly for inserting objects into the list, and displaying the contents of the list at exit. In Stage II, we will want to develop Insert, Delete, Update, and Find (SELECT) capabilities for items in the database.

[Note: You are free to use the suggested form(s) or make up your own. Some students like these forms; some are amused; some toss them away and devise their own. Anyone of these approaches is acceptable to me as long as the functionality of the project is not altered.

This document is designed to help you get started on the Final Project. The document is self-contained in that it specifies all the references to any other information you may need. Once again, the devil really is in the details. We will give you the code for some of the components required for the project -- specifically:

* A Form Controller Class – This class manages a number of aspects of the manipulation of the single form for this project. Some of the functions related to the form will be contained in the form class itself. This Controller class is intended to unclutter the form class (already large enough) by placing some of the functionality required for the desired form manipulation in a separate class. This is a static class. It has no attributes.
* A serializable (Sfile) Class – This class is responsible for the functionality related to the manipulation of the Sfile (reading from and writing to the file). You should use this class as the methods in it do all the work required to manage an Sfile.
* The Manager Class – This class is responsible for all functionality related to the manipulation of the data stores related to a Manager entity.
* The Person Class -- This class is responsible for all functionality related to the manipulation of the data stores required for a Person entity.

**Once again: You do not have to use this code (except for the serializable file/persistent object class) if you do not want to. But your project must reflect a similar structure of the form (even if you do not use the Form Controller Class – there are lots of other ways to do this) and it must reflect the same inheritance structure as shown later in this document.** Whatever you do, user friendliness (a beaten-to-death term, I know) will be important here, just as it surely will be in CIS 3342, and your capstone work.] PS – I think that studying and using the Form Controller class can provide a useful lesson in off loading form code behind work to a separate class consisting of methods but no attributes.

Some systems analysis materials will be provided to aid you in completing the project. You will need to produce:

* Complete descriptions of your classes, including a brief statement as to the purpose of the class, an annotated list of all the attributes required for the class, and list with descriptions of purpose for each the operations (methods) on the attributes. (See HWA #8 in the Stage I A description at the end of this document. Most of the information you need for HWA #8 may be found later in this document.)
* Behavior diagrams for your system. Please do not complicate your diagrams. Use separate diagrams for each of the major tasks to be performed. If you are not sure what this means, ask. (I will not collect these, but working without them will likely be like flying in the dark without radar.)

* Data validation details for all data to be entered in both Stage I and Stage II. The validation work you do for Stage I should be completely reusable in Stage II. (See HWA #9 in the Stage I B description at the end of this document. Most of what you need to complete this HWA may also be found later in this document.)

More information on the use of inheritance, databases, serializable files, and regular expressions may be obtained through many sources, including those listed below.

Inheritance:

[Discussed in detail in Appendix E on this website.](http://cis-iis1.temple.edu/cis3309/Lecture%20Set%20ZZZ%20Appendix%20E%20New%20-%20Fundamentals%20of%20Inheritance%20v20%20CSharp%20%2803-29-2021%29%20Spring%202021.doc)

Database Processing:

Discussed in detail in the Chapter 14 material included on this website.

[Lecture Set 14A New Abridged - Database Review and Connection Introduction CSharp](http://cis-iis1.temple.edu/cis3309/Lecture%20Set%2014A%20New%20Abridged-%20Database%20Review%20and%20Connection%20Introduction%20CSharp.pptx)

[Lecture Set 14B New Abridged - Database Processing with Data Readers CSharp](http://cis-iis1.temple.edu/cis3309/Lecture%20Set%2014B%20New%20Abridged%20-%20Database%20Processing%20with%20Data%20Readers%20CSharp.pptx?web=1)

Serializable Data:
Go to the link

<http://msdn.microsoft.com/en-us/library/vstudio/ms233843.aspx>

Regular Expressions:

See Jake Bricker's [Regex Document](http://cis-iis1.temple.edu/cis3309/ATM%20Regex%20Documentation.docx). Or, you can try this link to [Regex Buddy](https://www.regexbuddy.com/create.html). Another option is this Youtube tutorial.

<https://youtu.be/sa-TUpSx1JA>

Need help? Ask the lab assistants or those of your classmates who already have some experience with Regex. You do not need to use Regex, but it might help in your data validation efforts.

**The Problem (Specification)**

**Modeling the Data**

This project’s organization revolves around three data entities: managers, workers, and clients. The entities have unique attributes, but some fields are common to two or more entities. A good object model arranges data in the same way as is done in a normalized database. Consider the data entities:



 **Figure 1a: The three problem domain entities to be modeled**

Notice that all three entities have the attributes Name, Birth Date, and ID in common. To reflect this attribute repetition, we factor out these commonalities and place them in a new entity, which we call a Person. The three entities now can be related in a tree structure as shown below with Person at the root node:



 **Figure 1b: The three problem domain entities with factored out commonalities**

We also see that both the Manager and Worker have the attribute {Job Title} in common, but Client does not. So we factor out the Job Title attribute and create a fifth entity, Employee. The five entities relate to one another as shown next. We consider the Client, Manager, and Worker entities as "leaf nodes" in our inheritance hierarchy." These three leaf nodes have no remaining common data. In fact, none of the nodes in the above tree has data in common with any of the others.



We will use an *abstract class* to model the Person and Employee entities. We do this for two reasons. First, it is useful in building a *representative model* for the problem domain entities to be manipulated. In addition, while we have no need to manipulate a Person or an Employee, these two classes enable us to define a template for attributes and methods that are common to their subclasses. The Person and Employee classes encapsulate the common functionality for their subclasses. This functionality can therefore be used by the subclasses without duplication. When writing an abstract class, we have a choice to implement (or not) its methods depending upon any code we might right is relevant to the subclasses. (Think about what you might do in designing and programming a Property hierarchy for the game of Monopoly. How would you handle computing the rent for a property? This method is considerably different for “Streets” than for “Railroads” and “Utilities.”)

[We note in passing that in an Interface class, all members are unimplementable. They simply define a contract for the subclasses. If you want to change this contract you need to develop an entirely new one.]

An example of the stub (well, a little more than a stub) for the serializable Employee class is shown on the next page.

 // Employee inherits the data and methods in Person

 [Serializable()] public abstract class Employee : Person

 {

 private string HiddenEmployeeJobTitle; // Client Type

 // Default constructor

 public Employee ()

 {

 HiddenEmployeeJobTitle = "";

 } // end Employee default constructor

 . . .

 } // end Employee Class

**Figure 2: C# Code Illustrating Part of a Employee Class as Inherted from the Person Class**

The remaining tasks are to code and test the class hierarchy. To the five resulting classes, we add a Person List Class. This is done to illustrate the use of such lists with entries that are part of a class hierarchy and to enable the creation of a serializable file made up entirely of entries in the Person List.

Each node (class) in the Person hierarchy includes the following:

* The attributes (all private) of the class
* Non-parameterized and parameterized constructors
* Properties to access the private data (if you choose to use C# Properties)
* Methods to save data from the form to the list and display data in the list back to the form
* An overridden toString method to convert the data in the instantiated objects of the class to a string

The tree on the next page shows the complete Person class hierarchy. It is rendered automatically by VB .NET 2015 by right clicking on the project in the Solution Explorer and selecting “View Class Diagram”. (You may choose to implement the validation code in the Form code-behind rather than in these classes.)

The PersonList entity (not shown) is a list of Person objects. Because the Manager, Worker and Client classes directly or indirectly inherit the Person class, the PersonList can contain instances of all three objects. The processing of the objects is handled through late or dynamic binding (to be discussed in more detail in the document on inheritance (referenced above).

The use of the <Serializable()> tag facilitates the conversion of an instantiated object to a file. This is referred to as a persistent object. This object can also be read from a file back to an instance in memory. More about Serializable (persistent) objects may be found online or in the text.



The FormController class (also not shown) contains methods to activate and deactivate parts of the form based on the current object. The figure on the page 9 shows the form as statically designed (before execution). We have attempted to illustrate a form that reflects as well as possible the inheritance hierarchy formed by the 5 classes just described. You may wish to use this same form to save time and energy in designing and building your project.

Finally, we provide a more detailed textual description of the class hierarchy and the attributes of the classes for our project. (No blank entries allowed. All words are strings of letters.

**Person**

* Name (string- at least two names, first letter capitalized, all letters unless you want more complexity)
* TUID (5 digit integer or string – all digits (hence > 0, perhaps >10000)
* Birth Date (mm/dd/yyyy -use date-time picker?))

 **Employee**

* Anything allowed here (but cannot be blank)

 **Manager**

* Salary (Decimal with a maximum of two digits to the right of the decimal point, greater than or equal to 0.00)
* Bonus (Decimal with a maximum of two digits to the right of the decimal point, greater than or equal to 0.00)

 **Worker**

* Hourly Pay (Decimal with a maximum of two digits to the right of the decimal point, greater than or equal to 0.00)

 **Client**

* Type (Must be a string of lower-case letters and blanks)

 **Figure 3: Classes, Attributes, and Attribute Formats**

You are free to make changes to these descriptions, but not to appreciably change the required validations. Strongly urge you to write a separate Validators class – a static class, with no attributes.

**The Form** (to use or not to use?)

The form we will use is shown prior to the start of execution on the next page. The goal in designing the form is to make it appear to have the same structure as the class model just described (and the database). Shown here is a version of the form, which you are welcome to use, as it was manually created. The textboxes for input related to the Person class can be found at the top of the form, while those for the Employee and Client classes are next. Under the Employee data entry group box are the textboxes the Manager and Worker classes.



Shown on the next page is version of the form as it would appear during execution after the user clicked on the Create Client button in the original form. What has happened as a result of this click is that methods in the FormController class have been called that allow the user to enter data for a Client entity. The textboxes relevant to this data have been enabled and the group boxes containing these textboxes are colored green. Other methods in FormController were called to prevent the user from entering any data not related to a Client entity. These methods disable non-Client related textboxes and color their groupings in red to indicate that the user cannot enter data in these controls. While you are not required to use the form we designed, you might find it interesting to try and to learn a little about how the FormController class works.



**What is to be Given and What is to be Done by You**

As indicated earlier, we will give you the code for some of the components required for the project, specifically the Form Controller class, the Manager class, a Serializable File Class (aka the Persistent Object Class) and the Person Class.

The rest of this project, including a substantial amount of data entry validation is up to you. You will need to design and implement:

* The main form class, which serves of the driver for this project.
* The Employee class
* The Worker, and Client classes
* A Person List class
* A Database class
* A small Globals class (if you need it)
* A Validators class
* Any other classes you feel you need

If there is other code you wish to see, all you need to do is ask (no promises, but if you ask for help, you may well get it). We do not pretend to have the best way of doing anything, but at least we can suggest one way to do the things you need to do.

In completing this project, we strongly urge you to do the work in stages, as described earlier and amplified upon next.

 **Stage I:**

1. **Full Documentation of the Five Classes in the EmpMan Hierarchy (see HWA #08) -** In this stage, you are responsible for **fully documenting** (attributes, methods, important exceptions, etc.) each of **the five classes in the EmpMan project hierarchy**. Then you will want to begin the development of the code behind for your form. This code-behind should be built upon the Person, Client, Employee, Manager, and Worker class hierarchy described in this document. The hierarchy MUST be adhered to as you develop your code. To assist you in the development of this class hierarchy I will be emailing you two of the five classes you need:

 Manager Class

 Person Class

To assist you developing the code behind the form, I will also email you a

 Form Controller Class

which you can use to support the things you want or need to do for the form. You do not need to do all the things shown in the form, but whatever you do, you need to keep the functionality shown, including having the form(s) mirror the class hierarchy clearly showing, for any operation, which controls are relevant and which are not. (In the past, some students did a masterful job of this with multiple, smaller, and simpler forms). You will want to be sure that users cannot click on any form control until all required preceding work has been handled. For example, users should not be allowed to click the Update button until the ID of the Person to be updated is entered on the form.

**B. Validation** **(see HWA #09)** - You are also responsible **for building a validation infrastructure** in which you do the most thorough job practical of ensuring that every piece of data entered into your form by a user is as valid as we know how to make it. We do not want invalid information anywhere in our data stores. As part of the validation process, you will need to use C#.NET ToolTips to allow the user to hover over each Textbox requiring data entry and provide details as to what a valid entry should look like.

1. **Data Storage** - To varying degrees we will utilize three storage mechanisms, two of which are persistent (can be saved across executions of your project) and one is not. A list of Persons will be used to store information during the execution of your project code. This list is not persistent and will disappear once program execution terminates. The persistent storage we will use includes a serializable file and a database. For stage one, you are responsible for writing data from the Person List to the serializable file (and back again). A serializable file is a stream of bytes that can be used to store entire objects or even hierarchies of objects without translation to text. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization. You will designate the five classes in your hierarchy and the list of Persons class all as serializable, and with one simple call to a serializable file write function we will be able to write out the entire list contents as a binary file. You can find out more about these files online. You will be able to examine the contents of the serializable file directly in your project. It will not be quite as easy to read as a text file, but yet easy enough to decipher to see if your software is working. The code for this stage is to be submitted on Canvas two weeks before the submittal of Stage II (see below).

**A few suggestions as to how to proceed incrementally:**

Part 1: Make sure you are able to create (Manager, Worker, Client) objects and you are able to SAVE the entered data in these objects (use the Save method in the relevant class). Then, you need to save each created and populated object to the Person list. Be sure to validate each data entry item BEFORE placing the data in the object you create. Use the Validators class we discussed earlier (but that you need to write). You also need to store the Person list information in the serializable file upon clicking btnExit. If you are not able to get this task done, you cannot do much else.

Part 2: Display the information that we have stored about data entered for Managers, Workers and Clients that we have in our Person list. You will want to display the entire list, and then be able to search this list for a particular object and Display it using the Display method in the relevant class.

Part 3: Now, once Parts 1 and 2 have been completed, work on the updates and deletes for an object in the Person list (a specific Manager, Worker, or Client) given a user-entered ID. For an update operation, first display the specified object and allow the user to confirm that this is the object to update. Then validate all entered data and allow the user to save the new data in a new object which then can be entered into the Person list. The old object should be deleted from the list as no duplicates are allowed. For deletes, allow the user to delete any object we have based on the ID that the user enters, but double check that they want to delete them by confirming. Once confirmed then delete the specified Person.

**Stage II (see HWA #10)**

For Stage II, we will perform some simple database processing using the Data Reader/Writer abstraction provided by the .Net FCL. You will be able to do this using the Microsoft Access DB engine or any other DB engine of your choosing, provided you know how to connect from your C# program to the database you choose. For this stage should first save a zipped version of your Stage I code. (I assume this was done prior to submitting Stage I.) Then using this code as a starting point, keep the front-end part of the code (all the forms code-behind, all the validation, etc.). Remove the serializable file processing and the array list maintenance, and replace it with the use of a database engine handling: Inserts, Deletes, Selects, and Updates to an Employee-Manager database. We will provide you with a skeleton of the database using Microsoft Access with just a few entries, and with examples of the code needed to do the database processing, both from the forms side of your code and the EmpManDB class which actually performs the database manipulation (and which you have to write).

A few other suggestions.

1. When you *create* an object of type Manager, Worker or Client to be entered in the Person List, be sure to first check for duplicates. If the ID of the object is already in the list, generate an error message and ask the user to pick a unique (not used) ID. You should not allow the user to add duplicates to the list. Be sure to advise the user when a create and list add is complete.
2. When you *delete* a subtype object be sure the item is found in the list and display its contents in a form. If the item is found, ask the user to confirm that this is the object to be deleted. If the ID is not found, generate an error message. Then ask the user for a different ID.
3. If you attempt an *Edit/Update* or a *Find/Display* operation again allow for the case that the ID involved is not Person list. If the object is in the list, display its contents in a form. If the user is requesting an update, first ask for a confirmation that you have the correct object to update.

**Stage II (see HWA #10 –** if I assign it -- ask)

For Stage II we will perform some simple database processing using the Data Reader/Writer abstraction provided by the .Net FCL. You will be able to do this using the Microsoft Access DB engine or any other DB system of your choosing, provided you know how to connect from your C# program to the database you choose, and that you have a good understanding of the syntax for whatever SQL code you need to write.

When you are ready to start Stage II BE SURE TO SAVE YOUR COMPLETED STAGE I Project to submit, and to use in case you need it as a backup.

Note: Once you have everything working in Stage I, you will want undo a portion of this code dealing with the Serializable file and some work with the list, and start working on Stage II, the database code. Create a separate database class (such as PersonDB.cs, for example) with methods for database select, insert, update, and delete processing. Also add calls to these methods in the appropriate places in your existing code. You should not need to make many (if any) changes in your form(s). Note that suggestions 1. – 3. shown for Stage I apply equally as well to the Stage II database processing.

**Why Use Inheritance?**

The use of inheritance as shown provides us with a tool that more accurately reflects the relationships among the various entities modeled in a software system. It also enables programmers to reduce the redundancy in their code. Redundant (common) attributes across entities are factored out and placed in a separate entity. In so doing, we also enable the programmer to factor out the methods that operate on these attributes. So, both shared attributes and methods need be written only once and accessed at lower levels in the hierarchy as will be illustrated as we implement our Final Project.

The PersonList entity (not shown) is a list of Person type (actually, objects of subtypes of Person). Because all the classes in this project (Worker, Manager, and Client) directly or indirectly inherit the Person class, the PersonList can contain instances of all objects instantiated from these three classes. The processing of these objects is handled through late or dynamic binding.

The use of the <Serializable()> tag facilitates the copying of an instantiated object to a binary file. This file is referred to as a persistent object. There is no need to convert the data in such an object to text (or vice versa). The data in the object is simply “dumped” in binary form to the file. This object can also be read from the file without conversion and dumped to an object instance in memory. More about Serializable (persistent) objects may be found online or in the text.