**CIS 3309**

**Component-Based Software Design**

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

**FALL Semester 2020 (ver 8.0, November 6, 2020)**

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

**Introduction:**

The goal of the Owl Community 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 Graduate Student, Undergraduate Student, Faculty, or Chairperson. These objects will be inserted into an Owl Member list. 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 OwlMember 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 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 mechanisms for handling data entry are acceptable. 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.]

This document is designed to help you get started on the Final Project. Except for the information on databases, the document is self-contained. 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.
* 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.

**You do not have to use this code (except for the Sfile class code) 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) and it must support the same inheritance structure as shown later in this document. You should feel free to use multiple forms if you wish and not use the Form Controller Class.**

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. Note that 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.

* 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.)

**Once again: You do not have to use the code we provide (except for the serializable file /persistent object class) if you do not want to. But your project must reflect a similar structure to the class hierarchy and database relationships that will be shown in the next section.**

In closing this section, we note that more information on the use of inheritance, databases, serializable files, and regular expressions may be obtained through many sources, including those listed below.

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

Inheritance:

Discussed in detail in [Appendix E](http://cis-iis1.temple.edu/cis3309/Lecture%20Set%20ZZZ%20Appendix%20E%20New%20-%20Fundamentals%20of%20Inheritance%20v19%20CSharp%20(11-1-2020)%20Fall%202020.doc) on this website.

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 Set 14A New Abridged- Database Review and Connection Introduction CSharp.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**

The project organization revolves around four data entities: undergraduate students, graduate students, faculty, and department chairs (also faculty). The entities have unique attributes, but some fields are common to two or more entities. A good object model arranges data in a similar manner to what is done in a normalized database. Consider the data entities:

Name  
TUID  
Birth Date  
Tuition  
Year Credits  
Major GPA

Graduate

Name  
TUID  
Birth Date  
Stipend  
Degree Program  
Major

Name  
TUID  
Birth Date  
Department  
Rank

Faculty

UnderGraduate

Chairperson

Name  
TUID  
Birth Date  
Department

Rank  
Stipend

**Figure1a: The four problem domain entities to be modeled**

Note that all four entities have the attributes Name, TUID, and Birth Date in common. To eliminate this duplication, we factor out these commonalities and place them in a new entity, called an OwlMember. The resulting five entities can be related in a tree structure shown in Figure 1b with OwlMember at the root node.

Name  
TUID  
Birth Date  
Department  
Rank

OwlMember

Degree Program  
Stipend  
Major GPA

Faculty

Chairperson

UnderGraduate

Graduate

Tuition  
Year Credits  
Major GPA

Department  
Rank   
Stipend

Department  
Rank

**Figure1b: The four problem domain entities with factored out commonalities**

We also see that both the Undergraduate and Graduate Student entities have the attribute Major in common, but Faculty and Chairperson do not. So, we factor out the Major attribute and create a sixth entity, Student (see the figure below). Undergraduate and Graduate Students are considered as *specializations* on the Student entity. The relationship between Faculty and Chairperson is a little different. Chairperson shares two attributes in common with Faculty (as well as the three OwlMember attributes, so we will treat Chairperson as a specialization of Faculty. Note that there is no student attribute per se in our problem domain, as all students are either undergraduates or graduates. We therefore treat Student as an abstract class. However, there are many faculty members who are not chairpersons. So, both Faculty and Chairperson are treated as concrete classes. The six entities relate to one another as shown next.

Name  
TUID  
Birth Date  
Department  
Rank

OwlMember

Major   
GPA

Student

Faculty

Department  
Rank

Chairperson

Stipend

Graduate

Undergraduate

Stipend  
Degree Program  
EDG

Tuition  
Year  
Credits

The OwlMember class hierarchy shown above has six nodes: one *root node*, two *inner nodes* and three *leaf nodes*. The “is-a” test can be used to check object hierarchies: An Undergraduate is a Student and a Student is an OwlMember. Replace “is a(n)” in the previous statements with inherits and Undergraduate inherits Student and Student inherits OwlMember. This latter relation can be expressed in C# as shown toward the bottom of the next page. The use of the colon : between the entities Student and OwlMember creates the inheritance relationship between the Student and the OwlMember classes. The “<Serializable()>” tag in the first line will force .NET to include functions that allow objects to be saved to an XML-coded (very binary looking) file. (This tag should be removed from your code for Stage II of the project.) The “abstract” modifier disallows the direct instantiation of a Student. OwlMember is also an abstract class. The use of the keyword abstract means that the programmer cannot instantiate objects of these classes but may instantiate objects that are derived from these classes. In this case, the classes for which objects will be created are Undergraduate, Graduate, Chairperson, and Faculty).

We use an abstract class in this way for two reasons. First, it is used in building a representative model for the problem domain entities to be manipulated. In addition, while we have no need to manipulate an OwlMember or a Student, these two classes enable us to define a template for attributes and methods that are common to their subclasses. The OwlMember and Student 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 for completeness that in an *Interface class* (not used for this project), 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 class.

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

// Student inherits the data and methods in OwlMember

// Note that before this parameterless Student constructor executes

// a parameterless constructor for the OwlMember class must execute

[Serializable()] public abstract class Student : OwlMember

{

private string hiddenStudentMajor;

private decimal hiddenStudentGPA;

// Default constructor

public Student ()

{

hiddenStudentMajor = "";

hiddenStudentGPA = 0;

} // end Student parameterless constructor

. . .

} // end Student Class

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

The remaining tasks are to code and test the class hierarchy. To the six classes in the hierarchy, we add an OwlMemberList 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 an Sfile made up entirely of entries in the OwlMemberList.

Each node (class) in the Owl Member hierarchy includes the following:

* The attributes (all private) of the class
* Parameterless and parameterized constructors (there should be at least one parameterless constructor that you write)
* Properties that allow you to access the private data of the class (if you want to use C# Properties)
* Methods to save data from the form into an appropriately type object which is then added to the list and another method to display data in the list on the form
* An overridden toString method to convert data in any of the instantiated objects to a string

Next, 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.

**Temple Owl Community Member** (completed on the next page)

* Name (string)
* TUID (5 digit integer or string)
* Birth Date (mm/dd/yyyy)

**Student**

* Major Department (any combination of letters and blanks – such as CIS, Math, Psychology, Music, Political Science - etc)
* GPA (Decimal with a maximum of two digits to the right of the decimal point, 0.0<= GPA <= 4.00)

**Undergraduate Student**

* Tuition (Decimal with a maximum of two digits to the right of the decimal point, greater than or equal to 0.00)
* Year (Fr, So, Jr, Sr)
* Credits (to date) (int >= 0)

**Graduate Student**

* Stipend (Decimal with a maximum of two digits to the right of the decimal point, greater than or equal to 0.00)
  + - Degree Program (MA, MS, MBA, EMBA, PhDBA, MFA, MEd, MD, PhD – add in a few more that you can find in a catalog)  
        
       (completed on next page) 🡪

**Faculty**

* Primary Department (any combination of letters and blanks)
* Rank (Assistant Professor (AstP), Associate Professor (AscP), Full Professor (Prof), Lecturer (Lect), Instructor (Inst), Emeritus (Emer)

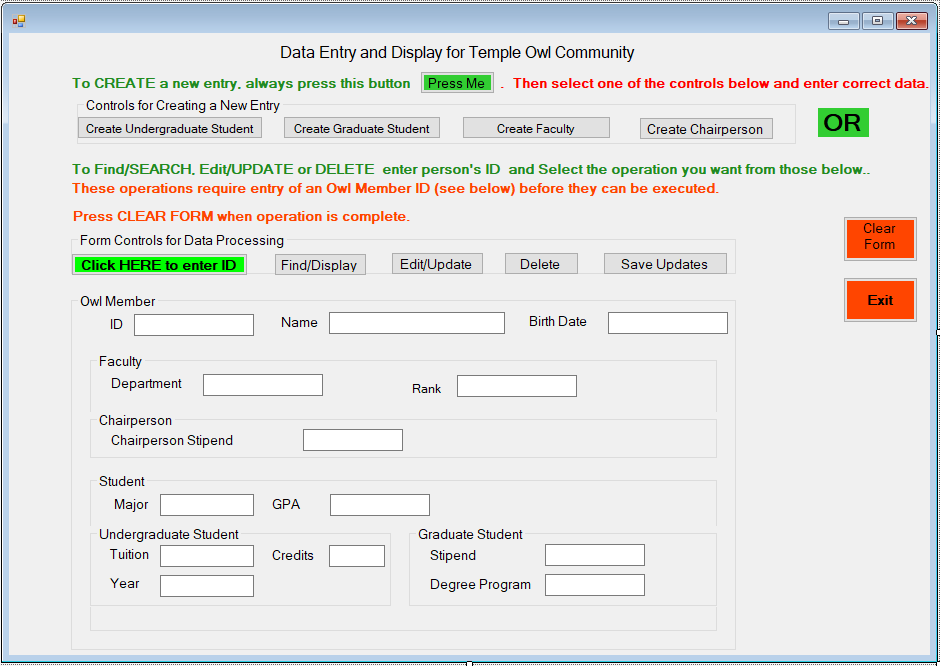
**Chairperson**

* Stipend (Decimal with a maximum of two digits to the right of the decimal point, greater than or equal to 1000.00)

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

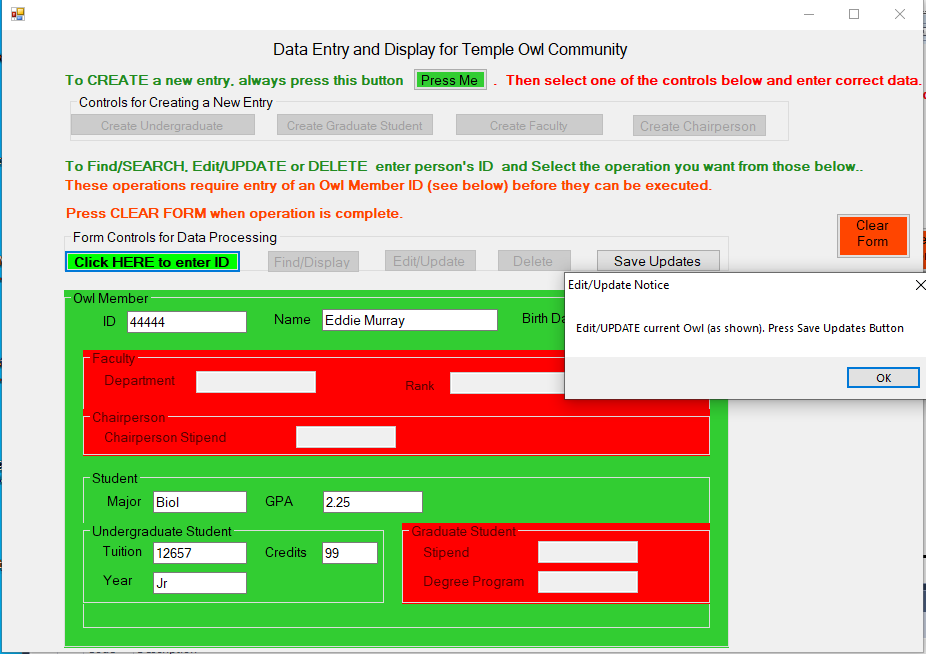
**The Form**

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 (and the database model) just described. The textboxes for input related to the OwlMember class can be found at the top of the form, while those for the Student and Faculty classes are next. Under the Student data entry group box are the textboxes the Undergraduate and Graduate Student classes, while the text box for the Chairperson data entry is under the Faculty group box (See next page).



**Figure 4: The Design of a Possible Version of the OwlMember Input Form**

An execution-time version of the form, colored to illustrate the portions to be used for data entry for an Undergraduate Student, is shown next in Figure 5. Textboxes relevant to Undergraduate Student data entry appear in the green portion of the form. The rest of the form related to data entry for a Graduate Student, Faculty, or Chairperson is shown in red. Data entry related to instances of these classes is not permitted during Undergraduate Student data entry. We have attempted to illustrate a form that reflects as well as possible the inheritance hierarchy formed by the 5 classes just described earlier. You may wish to use this same form to save time and energy in designing and building your project. Or, you can design your own form(s) for data entry.



**Figure 5: One Possible Input Form – Set for Data Entry for a Undergraduate Student Object**

**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 Owl Member and Undergraduate Student classes, as well as Form Controller and Serializable File classes. The FormController class contains methods to activate and deactivate parts of the form based on the designated current object. The coloring of the group boxes and the enabling of the relevant textboxes can be seen in the sample form shown in Figure 5.

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 Student, Graduate Student, Faculty and Chairperson classes
* The Owl Member List class
* A Database class
* A small Globals class (only if you need it)
* 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 Six Classes in the Owl Member Inheritance Hierarchy (see HWA #08) -** In this stage, you are responsible for implementing the Owl Member list-serializable file project. For this HW you need to **fully document** (attributes and methods, etc.) each ofthe six classes in the inheritance hierarchy described above and document the OwlMemberList class. Then you will want to begin the development of the code behind for your form. This code-behind should be built upon the OwlMember, Student, Faculty, UndergraduateStudent, GraduateStudent and Chairperson classes 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, we will email you two of the classes you need:

Undergraduate Student Class

Owl Member Class

To assist you developing the code behind the form, we will also email you the Form Controller Class

which you can use to support the things you want or need to do related to 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 mirror the class hierarchy and having it clearly show, 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 Owl Member 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 possible of ensuring that every piece of data entered into your form by a user is valid. We do not want invalid information anywhere in our data stores. The details of your validation code are best hidden in a separate static class.

**C. Data Storage** - We will utilize three storage mechanisms in this project, two of which are persistent (can be saved across executions of your project) and one which is not. A list of Owl Members 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 (Sfile) and a database. For Stage I, you will enter and modify data for Undergraduate and Graduate students, Faculty, and Chairpersons in the OwlMember List and at the end of each processing session, you will write this list to the Sfile. For later processing sessions, you will read the Sfile data back into the OwlMember List, update this list, and then rewrite back to the Sfile. [An Sfile 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 six classes in your hierarchy, and the list of Owl Members class all as serializable, and with one simple call to an Sfile write function we will be able to write out the entire list contents as a binary file. You will be able to examine the contents of the Sfile directly in your project. It will not be quite as easy to read as a text file, yet easy enough to decipher to see if your software is working. (You can search for things in this file, too).

A few other suggestions.

1. When you *create* an object of type Undergraduate Student, Graduate Student, Faculty, or Chairperson for entry in the OwlMember 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 the 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 OwlMember 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.

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

1. Part 1: Make sure you are able to create (Undergraduate Student, Graduate Student, Faculty, and Chairperson) 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 OwlMember 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 OwlMember list information in the serializable file upon clicking btnExit. If you are not able to get this task done, you cannot do much else.
2. Part 2: Display the information that we have stored about data entered for Undergraduate, Graduate, Faculty and Chairperson objects that we have in our OwlMember list based on the user's ID entry. 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.
3. Part 3: Now, once Parts 1 and 2 have been completed, work on the updates and deletes for an object in the OwlMember list (a specific Undergraduate Student, Graduate Student, Faculty or Chairperson) 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 OwlMember 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 OwlMember.

**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 OwlMember.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.

Questions? ASK!

**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 propensity for redundancy is our 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 OwlMemberList entity (not shown) is a list of OwlMember objects. Because all the classes in this project (UndergraduateStudent, GraduateStudent, Faculty, and Chairperson) directly or indirectly inherit the OwlMember class, the OwlMemberList can contain instances of all objects instantiated from these four 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.